MASTER

Exploring the feasibility of high compute kernels of Kernelized Correlation Filter based object tracker on Intel's Image Processing Unit

Bhupesh

Award date:
2016

Link to publication
Exploring the feasibility of high compute kernels of Kernelized Correlation Filter based object tracker on Intel’s Image Processing Unit

Master thesis

submitted in partial fulfillment of the requirements for the degree of

Master of Science

in

Embedded Systems

by

Bhupesh (0923476)

August 17, 2017
Abstract

Intel Corporation is one of the largest and highest valued semiconductor chip maker [25]. It produces a vast majority of SoC for desktops, mobiles, servers and other devices. Today’s computing platforms are complex heterogeneous SoCs which are made to perform various kind of tasks. The image processing unit (IPU), a multicore & fixed functions Digital Signal Processing (DSP) platform, is an integrated part of Intel’s SoC which is designed to perform image processing tasks. To capture the increasing market of embedded computer vision applications, it is desirable to investigate the suitability of this IP for complex and compute intensive vision algorithms. This thesis work is targeted to explore the feasibility of object tracking algorithm on IPU where we mapped high compute kernels of the tracking algorithm in fixed point and exploited SIMD and VLIW architecture. In this thesis work, we explored the mapping of two main algorithms on Intel’s IPU platform, Felzenszwalb’ Histogram of Oriented Gradients (FHOG)[10] and Fast Fourier Transform(FFT), used in the object tracking algorithm Kernelized Correlation Filter based tracker (KCF)[5]. We evaluated the mapping of these algorithms both on speed (time) and quality (tracking accuracy) metrics. The implementation of FHOG on IPU shows ~6x gain in comparison to the state-of-the-art implementation of Dollar’s FHOG [14] and ~1.5x speed up in comparison to libHOG [13]; both implementation are designed for Intel x86 processors. As the tracking algorithm offers implicit data level parallelism for FFT computation, we can gain up to ~31x better performance with any FFT algorithm on IPU in comparison to a scalar DSP platform. We lose 4-5% of tracking accuracy with fixed point implementation of FHOG and FFT. We can gain overall good level of performance on Intel IPU by accelerating these kernels and further exploiting the data parallelism in the algorithm itself which shows that Intel IPU can be an attractive option to enable object tracking for a low cost and low power devices.

Committee members :

**Supervisor** : Prof.dr.ir.C.H. (Kees) van Berkel, Computer Science, TU/e

**Supervisor** : David van Kampen, Senior DSP HW architect, Intel, Eindhoven

**Member** : Prof.dr.Henk Corporaal, Electrical engineering, TU/e
# Table of Contents

List of Figures ................................................................................................................ iv  
List of Tables .................................................................................................................. v  
List of Acronyms ........................................................................................................ vi  
Acknowledgements ....................................................................................................... vii  

1. Introduction ............................................................................................................. 1  
   1.1 The Kernelized Correlation Filter based object tracker ............................... 2  
   1.2 Problem statement ......................................................................................... 4  
   1.3 Goal ............................................................................................................... 5  
   1.4 Structure ...................................................................................................... 5  

2. Intel IPU hardware architecture .......................................................................... 6  
   2.1 System Setup ............................................................................................. 6  
   2.2 Core of the IPU ......................................................................................... 7  
   2.3 Functional Units ...................................................................................... 7  
   2.4 Memory hierarchy .................................................................................... 8  
   2.5 Compiler ................................................................................................... 9  

3. Feature descriptor - FHOG ............................................................................ 10  
   3.1 FHOG - algorithm .................................................................................. 10  
      3.1.1 Pixel-level feature maps ................................................................. 11  
      3.1.2 Histogram ..................................................................................... 12  
      3.1.3 Normalization and truncation ...................................................... 14  
   3.2 FHOG mapping on vector processor .................................................... 16  
      3.2.1 Data access pattern ..................................................................... 16  
      3.2.2 Gradient ....................................................................................... 17  
      3.2.3 Gradient magnitude ................................................................. 17  
      3.2.4 Histogram generation ............................................................... 18  
      3.2.5 Bin index calculation ................................................................. 18  
      3.2.6 Spatial Binning .......................................................................... 20  
      3.2.7 Energy Calculation ................................................................... 23  
      3.2.8 Feature calculation .................................................................. 24  
   3.3 Fixed point FHOG design ....................................................................... 25
## List of Figures

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>KCF algorithm stages</td>
<td>2</td>
</tr>
<tr>
<td>2</td>
<td>KCF algorithm – pseudo code</td>
<td>3</td>
</tr>
<tr>
<td>3</td>
<td>System setup</td>
<td>6</td>
</tr>
<tr>
<td>4</td>
<td>Vector processor architecture at abstract level</td>
<td>7</td>
</tr>
<tr>
<td>5</td>
<td>Memory hierarchy in vector processor</td>
<td>8</td>
</tr>
<tr>
<td>6</td>
<td>FHOG algorithm</td>
<td>11</td>
</tr>
<tr>
<td>7</td>
<td>Contrast sensitive and insensitive bins</td>
<td>12</td>
</tr>
<tr>
<td>8</td>
<td>Spatial binning – pseudo code</td>
<td>13</td>
</tr>
<tr>
<td>9</td>
<td>64 pixels contributes the histogram of a cell</td>
<td>14</td>
</tr>
<tr>
<td>10</td>
<td>Normalization of a target cell</td>
<td>14</td>
</tr>
<tr>
<td>11</td>
<td>FHOG feature channels</td>
<td>16</td>
</tr>
<tr>
<td>12</td>
<td>Gradient calculation – pseudo vector code</td>
<td>17</td>
</tr>
<tr>
<td>13</td>
<td>Gradient magnitude- pseudo vector code</td>
<td>18</td>
</tr>
<tr>
<td>14</td>
<td>Bin index calculation - pseudo code</td>
<td>19</td>
</tr>
<tr>
<td>15</td>
<td>Selection mask generation – pseudo vector code</td>
<td>20</td>
</tr>
<tr>
<td>16</td>
<td>Static pattern for spatial binning</td>
<td>21</td>
</tr>
<tr>
<td>17</td>
<td>Processing of common pixel block between 4 cells</td>
<td>21</td>
</tr>
<tr>
<td>18</td>
<td>Histogram generation – pseudo vector code</td>
<td>22</td>
</tr>
<tr>
<td>19</td>
<td>Contrast sensitive histogram generation</td>
<td>23</td>
</tr>
<tr>
<td>20</td>
<td>Contrast insensitive histogram - pseudo vector code</td>
<td>23</td>
</tr>
<tr>
<td>21</td>
<td>Cell energy calculation - pseudo vector code</td>
<td>23</td>
</tr>
<tr>
<td>22</td>
<td>Normalization factor calculation - pseudo vector code</td>
<td>24</td>
</tr>
<tr>
<td>23</td>
<td>Feature calculation – pseudo vector code</td>
<td>24</td>
</tr>
<tr>
<td>24</td>
<td>FHOG - pseudo vector code</td>
<td>25</td>
</tr>
<tr>
<td>25</td>
<td>Histogram of cells serialize on Intra vector addition</td>
<td>30</td>
</tr>
<tr>
<td>26</td>
<td>INTRA_VEC_ADD extension proposal</td>
<td>31</td>
</tr>
<tr>
<td>27</td>
<td>SW pipeline-feature calculation</td>
<td>32</td>
</tr>
<tr>
<td>28</td>
<td>Radix-2, 8 points FFT</td>
<td>35</td>
</tr>
<tr>
<td>29</td>
<td>Butterfly FFT computation</td>
<td>36</td>
</tr>
<tr>
<td>30</td>
<td>FFT Pseudo code</td>
<td>37</td>
</tr>
<tr>
<td>31</td>
<td>FFT implementation with scalar radix-2 algorithm</td>
<td>37</td>
</tr>
<tr>
<td>32</td>
<td>FFT implementation on IPU</td>
<td>38</td>
</tr>
<tr>
<td>33</td>
<td>FFT implementation structure for vector processor</td>
<td>38</td>
</tr>
<tr>
<td>34</td>
<td>1D FFT performance comparison</td>
<td>40</td>
</tr>
<tr>
<td>35</td>
<td>2D-FFT performance comparison</td>
<td>41</td>
</tr>
<tr>
<td>36</td>
<td>Tracking evaluation setup for fixed point FHOG implementation</td>
<td>43</td>
</tr>
<tr>
<td>37</td>
<td>Tracking accuracy evaluation-introducing noise by quantizing FFT result</td>
<td>44</td>
</tr>
<tr>
<td>38</td>
<td>Tracking accuracy evaluation setup- fixed point FHOG &amp; QFFT</td>
<td>45</td>
</tr>
</tbody>
</table>
List of Tables

Table 1 FHOG parameters ........................................................................................................ 10
Table 2 Fixed point optimization-FHOG .................................................................................. 27
Table 3 FHOG performance benchmark result ........................................................................ 28
Table 4 Tracking evaluation results with fixed point FHOG ...................................................... 43
Table 5 Tracking accuracy result with quantized FFT ............................................................... 45
Table 6 Tracking accuracy result for fixed point FHOG & QFFT ............................................. 46
### List of Acronyms

<table>
<thead>
<tr>
<th>Acronym</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>BAMEM</td>
<td>Block Access Memory</td>
</tr>
<tr>
<td>CFT</td>
<td>Correlation Filter Tracker</td>
</tr>
<tr>
<td>DCF</td>
<td>Dual Correlation Filter</td>
</tr>
<tr>
<td>DMA</td>
<td>Direct Memory Access</td>
</tr>
<tr>
<td>DSP</td>
<td>Digital Signal Processor</td>
</tr>
<tr>
<td>FFT</td>
<td>Fast Fourier Transform</td>
</tr>
<tr>
<td>FHOG</td>
<td>Felzenszwalb’s Histogram of Oriented Gradients</td>
</tr>
<tr>
<td>FPS</td>
<td>Frame per Second</td>
</tr>
<tr>
<td>GOPS</td>
<td>Giga Operations per Second</td>
</tr>
<tr>
<td>GPU</td>
<td>Graphics Processing Unit</td>
</tr>
<tr>
<td>HIVECC</td>
<td>HIVEC Compiler</td>
</tr>
<tr>
<td>ILP</td>
<td>Instruction Level Parallelism</td>
</tr>
<tr>
<td>IPU</td>
<td>Image Processing Unit</td>
</tr>
<tr>
<td>IS</td>
<td>Issue Slot</td>
</tr>
<tr>
<td>KCF</td>
<td>Kernelized Correlation Filter</td>
</tr>
<tr>
<td>LUT</td>
<td>Look Up Table</td>
</tr>
<tr>
<td>OpenMP</td>
<td>Open Multi-Processing</td>
</tr>
<tr>
<td>SIMD</td>
<td>Single Instruction Multiple Data</td>
</tr>
<tr>
<td>SoC</td>
<td>Systems on Chip</td>
</tr>
<tr>
<td>SSE</td>
<td>Streaming SIMD extensions</td>
</tr>
<tr>
<td>SWP</td>
<td>Software Pipelining</td>
</tr>
<tr>
<td>VLIW</td>
<td>Very Long Instruction Word</td>
</tr>
<tr>
<td>VMEM</td>
<td>Vector Memory</td>
</tr>
<tr>
<td>VP</td>
<td>Vector Processor</td>
</tr>
</tbody>
</table>
Acknowledgements

I am grateful to Eindhoven University of technology (TU/e) for awarding me Amandus H. Lundqvist (ALSP) scholarship. Without this scholarship, it would have been very difficult for me to pursue master degree.

I would like to thank to prof.dr.ir. Kees Berkel for giving me an opportunity to carry out my thesis under his guidance, and for motivating me to get good results.

I would like to express my deepest gratitude to my supervisor at Intel, David van Kampen, for his excellent guidance, caring nature and for helping me to learn about Intel’s DSP architecture and its programming.

I would like to thank Mark Wezelenburg, senior application engineer, and Hyunjoon Lee, software engineer at Intel for sharing their expertise in signal processing and computer vision field.

Finally, I would like to thank my family and friends for always being with me.

Bhupesh
Eindhoven, the Netherlands
August 17, 2017
Chapter 1 Introduction

1. Introduction

Visual object tracking is one of the most challenging tasks within the field of computer vision. As an interdisciplinary technology, it is enabled by the expertise of image processing, pattern recognition, machine learning, and signal processing. It has a wide range of applications like human-machine interaction, surveillance, robotics, automotive, military etc. The advancement in the area of high quality videos cameras, computing engines, and the needs to automate a video analysis have led to an increasing interest of many researchers in this field.

There are many challenges [1] which makes the task of object tracking difficult such as loss of information in projecting the 3D world on a 2D image, noise in image processing, cluttered-background, complex object motion, partial or full occlusion, illumination changes etc. Besides that, it is also important for an object tracker to be fast enough so that it can be deployed in real time systems.

The basic task of an object tracking algorithm is to predict the position of the target in a video frame. It is quite difficult to design a robust tracker and fulfill the real time requirements due to variations in illumination, rotation, and occlusion and other factors. Numerous algorithms have been proposed meet these challenges [2] but we focus only on the recent tracking algorithm in this work especially on KCF because it produces better tracking results with good real time performance on x86 machine [5]. Interested readers are referred to the surveys [2] and [3] which outline different object tracking algorithms.

When we look at object tracking algorithms, the basic question arises that how these algorithms are able to identify the target in sequential video frames and track them. Most of the algorithms integrate concepts from machine learning to identify the object. These algorithms can be put into two categories, one which uses a generative model and the other which uses discriminative model for detection. Generative trackers perform tracking by searching the best match for the target object. Discriminative trackers learn to discriminate the target object from the background. It has been found that background knowledge (information besides the target in an image) is helpful for better performance. These algorithms are also known as tracking-by-detection algorithms.

As mentioned in the survey by et.al Z. Chen [3], all the state-of-the-art algorithms of object tracking are based on the Correlation Filter based Tracker (CFT) which uses discriminative model. In the recent years, correlation filter based trackers are getting huge attention because of good real time performance with high tracking results. The survey also describes the training, detection and updating scheme of all the state-of-art algorithms along with the shortcoming of these algorithms. According to this survey, there are two most important aspects of a tracking algorithm. First, the training schemes are extremely crucial for CFTs. Since the target may change its appearance continuously, correlation filters should be adaptively trained and updated on-the-fly to adapt to the new appearances of the target. Second, the feature representing the target object also greatly influence tracking results. Although raw pixels can be directly used for detection, the tracker may be affected by various noises like illumination change and motion blurs. More powerful features
are supposed to be helpful. In the next section, we describe the general framework of the correlation filter based tracking algorithm, and briefly explain the algorithmic pipeline of Kernelized Correlation Filter (KCF) algorithm. Curious readers are referred to [4] and [5] for detailed description.

1.1 The Kernelized Correlation Filter based object tracker

Before tracking an object, the object first needs to be detected in a video frame. However, the object tracking algorithms only focus on the tracking part, and assume the position of target object is already known in the first frame.

The intuitive working of correlation filter-based tracking algorithm is explained in [3]. The tracking algorithms can work with raw image data but suffers from low tracking results. To get better tracking results, an image data is converted into robust features using feature descriptor algorithm such as HOG and FHOG [5]. A cosine window is applied on the extracted feature for smoothing the boundary effects. Then, the obtained result is analyzed in the frequency domain by applying Fast Fourier Transform (FFT).

The position of the target object is known in the first frame. The correlation filter is trained with the cropped image around the target object after preprocessing.

In the subsequent frames, a correlation operation is performed with the pre-processed image patch at the previously predicted position. Following the correlation procedure, a spatial confidence map, or response map is obtained using Inverse Fast Fourier Transform (IFFT). The position with maximum value in this map then used to predict the new position of the target for the next frame, but appearance at the estimated position is extracted for training and updating the correlation filter in the same frame. This process continues for all successive frames.

The KCF algorithm exploits the structure of circulant matrices to enhance the discriminative ability of track-by-detection algorithm. The high level algorithmic flow of KCF is shown in figure 1.

![Figure 1 KCF algorithm stages](image-url)
Chapter 1 Introduction

For the given position of the target center and size, an image patch is created. The size of this image patch is larger than the target size. This is done to collect some background information as the algorithm follows the principle of tracking-by-detection. Then, feature descriptor algorithm is applied on this image patch. In case of KCF algorithm, FHOG is used as the feature descriptor. FHOG produces 31 channels per cell. Each channel is weighted by a cosine window to remove boundary affects \[5\].  A circulant matrix, \([4] \& [5]\), is used to learn all the possible shifts of the target from the base sample. The coefficient \(\alpha_f\) encodes all the training samples, consisting of all shifts of a base sample in the frequency domain. The fast learning equation is expressed as:

\[
\alpha_f = \frac{y_f}{K_f^{xxr} + \lambda}
\]  

where \(y_f\) is the training label matrix, a Gaussian function that smoothly decays from the value of one for the centered target to zero for other shifts, in frequency domain. \(\lambda\) is the regularization parameter and \(K_f^{xxr}\) denotes the DFT of \(K_{xxr}^{f}\) the kernel correlation function between signal \(x\) and \(x'\).

```
function feature  = Pre_process(img, pos, size)  
    patch         = region(image, pos, size);  
    feature       = FHOG(patch) .* cosine_window; 

function pos'   = Detection(frame, img, pos, size, x_f', alpha_f') 
    if(frame>1) 
        z_f   = FFT(Pre_process(img, pos, size, x_f', alpha_f')) 
        K_f^{xxr} = Linear_correlation(z_f, x_f'); 
        Pos'   = pos + max(IFT(K_f^{xxr} .* alpha_f')); 
    end; 

function [alpha_f, x_f] = Train(img, pos', size)  
    x_f = FFT(Pre_process(img, pos', size)); 
    K_f^{xx} = Linear_correlation(x_f, x_f); 
    alpha_f = y_f./(K_f^{xx} + lambda); 

function [alpha_f', x_f'] = Model_update(alpha_f, x_f) 
    if(frame>1) 
        f   = 1; 
    else 
        f   = interpolation_factor; 
    end; 
    alpha_f'   = f * alpha_f + (1-f) * (alpha_f'); 
    x_f'       = f * x_f + (1-f) * ( x_f'); 
```

The KCF algorithm works with both polynomial & linear kernel correlation functions. The displacement between the current (x) and next image patch (z) is the spatial index with maximum

Figure 2 KCF algorithm – pseudo code
response in $F^{-1}(K_{f}^{xz}.*\alpha'_{f})$. The learned model and alpha i.e. $x'_{f}$ and $\alpha'_{f}$ are linear interpolation of the $x_{f}$ and $\alpha_{f}$ where interpolation factor $\epsilon$ [0,1].

We present the pseudo code of KCF in figure 2 where variables with subscript $f$ represent the frequency domain component. In this thesis work, the focus is on two sub algorithms, FHOG & FFT, which form the front end of the KCF algorithm. These two algorithms takes ~70% (40% FHOG & 30% FFT) of the total computation time of the algorithm which make them good candidates to explore on the target platform first. Therefore, we restrict here to only a brief description of KCF algorithm. Interested readers are referred to [4] and [5] for more detail about KCF algorithm.

1.2 Problem statement

With the recent advancement in the Computer Vision field, vision algorithms have become robust in tracking/detecting objects. Despite being robust, most of the computer vision algorithms suffer from low real time performance. We can get reasonable real time performance with high end platforms e.g. x86 processors, but these processors are meant for general purpose usage and consume a lot of power. Due to high power consumption, these platforms are not the best option for embedded computer vision applications. To get balance between performance, power and cost, researchers and industries have started exploring other platforms such as NVIDIA’s GPU [28], CEVA’s DSP-XM4 [29], Tensilica’s IVP DSP cores [30] etc. which are specifically designed for imaging and computer vision tasks.

The Intel image processing unit (IPU), a multicore & fixed functions DSP platform, is designed to perform complex image processing tasks. Due to its high efficiency and low power consumption, it has been used in many products which are powered by Intel’s SoCs such as smartphones, tablets etc. The usage of this platform for processing certain computer vision applications on low power devices can be an ideal option. To determine the feasibility of Intel IPU for object tracking application, we will be exploring the mapping of an object tracking algorithm which is Kernelized Correlation Filter based tracker (KCF) [5]. Specifically, the focus will be on two main parts (sub algorithms), FHOG and FFT, as these algorithms takes ~70% (~40% FHOG & ~30% FFT) of the total computation time, when mapped on an Intel x86 processor.

The computer vision algorithms use advanced mathematical models to solve tracking/detection problem which often requires data to be in floating point format for high precision. On the other hand, the IPU platform is a fixed point machine. Therefore, it is necessary to design the fixed point implementation of the target algorithms and evaluate its effects on overall tracking performance.
Chapter 1 Introduction

1.3 Goal

The first goal is to design the fixed point implementation of FHOG and FFT. There is loss of precision as data type changes from floating point to integer data type, and it adversely affect the tracking results. The aim is to achieve tracking results as close as possible with original floating point implementation, but tracking results must be within 7-8% loss limit. To see how fixed point implementations affect the performance of tracking algorithm, we will focus on evaluating the quality of tracking results. With the results of the above step, we will reflect on the precision requirements for processing each of these kernels on IPU.

The second goal is to efficiently implement these algorithms on IPU to get the best performance that this platform can offer. By implementing these kernels on the target platform, we will see how it performs in comparison to other processors. Based on the result of this step, we will do design space exploration at architecture level and propose appropriate changes to improve future generations of the Intel’s IPU. In other words, it will lead us to make a decision on the need of developing new hardware accelerators or extending the instruction set so that this platform can become an attractive option for low power embedded vision devices.

1.4 Structure

The thesis report is structured in this way. Chapter 2 is dedicated to high level description of Intel Image Processing Unit (IPU) especially the core, vector processor, which performs all the image processing tasks.

Chapter 3 gives the detailed description on FHOG algorithm, its mapping on vector processor, fixed point design, performance comparison and architecture exploration.

Chapter 4 is dedicated to FFT algorithm, its implementation on vector processor, fixed point design and performance comparison.

The main focus of chapter 5 is to explain the meaning of tracking accuracy in case of KCF based object tracker. It also detailed out the tracking evaluation with three cases along with the evaluation setup; tracking accuracy evaluation with fixed point FHOG only, with fixed point FFT only, and with both fixed point FHOG and FFT.

Finally, chapter 6 sums up the thesis work with some suggestions for future work.
Chapter 2 Intel IPU hardware architecture

2. Intel IPU hardware architecture

2.1 System Setup

In this section, the detail of the system and architecture of the vector processor, on an abstract level, is presented.

IPU is quite complex computing platform which contains a number of vector processors (Vector processor offers VLIW and SIMD parallelism), scalar processors, DMAs, DDR, an interface to connect with camera sensors etc.

To map the algorithms, we used a simulator environment where a host directly communicates with the core of the IPU, the vector processor, which does all the signal processing tasks. Figure 3 represents system setup at an abstract level. Host transfers the data to the vector processor. Once the vector processor finishes processing image data, host can read its data for further processing.
2.2 Core of the IPU

A vector processor (VP) has scalar issue slots which perform scalar operations such as arithmetic, logical etc. It also has SIMD issue slots which perform operation on 512 bits long vector in general. Figure 4 represents the architecture of the vector processor at abstract level.

A vector contains 32 words each of 16 bit. Some functional units also perform operations with a logical double vector known as twide-vector (each word is 32 bits long) to support higher precision.

The operations are scheduled as VLIW instructions on the vector processor to exploit the instructions level parallelism available in the application.

2.3 Functional Units

Each issue slot contains dedicated Functional Units (FUs) supporting a variety of instructions. While some are present in most issue slots since the functionality they perform is very common (for example addition), others perform only specialized instruction and are fewer in number.
Chapter 2 Intel IPU hardware architecture

A loose categorization of these units would be the following:

**Load/store units** which are used to transfer information from vector /scalar memories to registers.

**Register transfer units** that are used to transfer information from one register to other registers. Required in order to pass information between issue slots and avoid storing intermediary data to the memories.

**Arithmetic and logic units** are responsible for performing arithmetic operations. Most of these units are small and fast.

**Intra-sum** units take the elements of a vector, add them together and results in a scalar value.

**Shuffle** units take the two vector and generate a new vector defined by an index vector.

**Accelerators** are much larger units and are responsible for doing more complex operations. These units are made to execute high computational task in few cycles.

Operations are performed in a pipelined manner by every functional unit. Even though the latency for some units is quite large yet we can gain performance by exploiting the data and instruction level parallelism in the application.

### 2.4 Memory hierarchy

The IPU has three type of storage as shown in figure 5. There are two types of register files – scalar register file can store the data of 32 bit longs and vector register files can hold 512 bits of data.

![Memory hierarchy in vector processor](image)

There are two types of local memories- scalar data memory, and vector memory. Scalar memory is used to store the scalar data of 32 bits long.

Vector memory is divided into two memories based upon the access pattern - VMEM and Block Access memory (BAMEM). VMEM is vector aligned memory i.e. consecutive location in VMEM...
are separated by 512 bits. This type of memory can be found on the core (VMEM) or outside the core and is accessed and loaded into the register using dedicated load store units. Transferring data between VMEM and BAMEM can be done only via registers.

BAMEM is a more advanced type of vector memory that supports fine access granularity of one element of a vector for both loads and stores. This means that the access address is element aligned not vector aligned as comparison to VMEM. It can also be configured as a lookup table (LUT). The main benefit of this memory to have 2-D block access as a vector in various format such as 1x32, 4x8, 8x4, 16x2 etc.

DRAM is outside the vector processor. In real system, IPU communicates to other processing node via this memory.

2.5 Compiler

The architecture is supported by an intelligent compiler known as HIVECC. Besides basic optimization, it also supports loop unrolling and software pipelining. To use these functionality, user has to use a keyword which is “pragma” preceded by #.

Loop unrolling is the loop transformation technique for the reduction of control instructions the loop at the expense of increased program memory.

Software pipelining is the technique where compiler schedules the instructions of next iteration with current iteration so that the instructions of next iteration can be executed in parallel with current iteration. It can only be used when the instructions of the loop are independent on the previous iteration.
Chapter 3 Feature descriptor – FHOG

3. Feature descriptor - FHOG

In computer vision and image processing, feature descriptors are the algorithms which transform an image into useful information for solving complex detection/tracking tasks. The object tracking algorithm can work with raw pixels. However, in that case, it suffers from low tracking results as indicated in KCF algorithm [5] where tracking results reduced from 73% to 56% when using raw pixels. To get better tracking results, sophisticated feature extraction algorithms are used. Histogram of Oriented Gradients (HOG) descriptor [9] is a widely used feature descriptor. Felzenszwalb’s HOG (FHOG) is a variant of HOG algorithm which was first used in object recognition algorithm [10]. Due to better performance [11], FHOG has been used extensively in computer vision application. The authors, Felzenszwalb et al [10], released the C implementation of FHOG as an open source. We used this implementation as a reference. In the literature, we found two notable implementations which are accelerated versions of FHOG and can be used directly with tracking algorithm. First, Dollar’s FHOG which is a part of the pdollar MATLAB tool box [14] that exploits SIMD parallelism by using SSE extension of the Intel processors. Second, libHOG [13] which exploits both SIMD and thread level parallelism by using SSE and OpenMP directives (to use multicore) on Intel’s x86 multicore platform. This section is divided into five subsections; the FHOG algorithm (3.1), FHOG mapping on vector processor (3.2), fixed point design (3.3), performance comparison with existing implementations in section (3.4) and, finally, architecture level exploration in section (3.5).

3.1 FHOG - algorithm

FHOG algorithm works on different level of data abstractions such as pixels, cells and blocks to extract the useful feature of an image where a cell consists of 4x4 pixels and a block is formed form 2x2 cells or 16x16 pixels. Table 1 represents the parameters used in context with this feature descriptor.

<table>
<thead>
<tr>
<th>Parameters</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cell size</td>
<td>4x4 pixels</td>
</tr>
<tr>
<td>Block size</td>
<td>2x2 cells</td>
</tr>
<tr>
<td>Contrast sensitive bins</td>
<td>18</td>
</tr>
<tr>
<td>Contrast insensitive bins</td>
<td>9</td>
</tr>
<tr>
<td>Histogram vector length</td>
<td>27</td>
</tr>
<tr>
<td>FHOG Feature size</td>
<td>31 channels</td>
</tr>
</tbody>
</table>

The algorithmic steps of the FHOG descriptor can be elaborated in the following figure 6. The first step is to obtain a pixel level feature map by calculating the gradient and orientation of pixels. The
second step is to collect the pixel level feature map into a cell level feature map, specifically a histogram, by collecting weighted gradients magnitude into histogram bins. The bin index is decided based on the orientation of a pixel gradient and weights are calculated based on the relative position of the pixel with respect to cells center. The final step is to normalize and truncate histogram of a cell with respect to the block energy to obtain the feature of each cells.

![Figure 6 FHOG algorithm](image)

### 3.1.1 Pixel-level feature maps

For each pixel in the image, x and y gradients are calculated by applying filter with response $[-1 \ 0 \ 1]$ and $[-1 \ 0 \ 1]^T$ on the image.

\[
\text{Gradient in x direction} \quad (dx) = f(x+1,y) - f(x-1,y)
\]

\[
\text{Gradient in y direction} \quad (dy) = f(x,y+1) - f(x,y-1)
\]

After that, the magnitude and orientation of the gradients is calculated:

\[
\text{Gradient magnitude} \quad (m(x,y)) = \sqrt{dx^2 + dy^2}
\]

\[
\text{Orientation of the gradient} \quad (\theta(x,y)) = \arctan\frac{dy}{dx}
\]

For the color image, the largest magnitude of either of the color components defines \(\theta(x,y)\) and \(m(x,y)\) of the pixel \((x,y)\).

The gradient orientation at each pixel is discretized into 2p contrast sensitive \((B_1)\) bins and also into \(p\) contrast insensitive \((B2)\) bins, where bin indices are defined as:

\[
B_1(x,y) = \text{round}\left(\frac{p\theta(x,y)}{2\pi}\right) \mod p
\]
Each pixel contributes to two bins; one is between 0° to 360°, and second is between 0° to 180°. The pixels which belong to the contrast sensitive bin in the 4th quadrant also belong to contrast insensitive bins in 2nd quadrant, and pixels which belongs to contrast sensitive bins in 3rd quadrant also belongs to contrast insensitive bins in 1st quadrant as shown in figure 7.

Figure 7 Contrast sensitive and insensitive bins

Pixel-level feature map specifies a sparse histogram of gradient magnitude at each pixel. Let $b \in [0, \ldots, 2p - 1]$ ranges over the orientation bins. The feature vector at $(x, y)$ is:

$$F(x, y)_b = \begin{cases} m(x, y) & \text{if } b = B(x, y) \\ 0 & \text{otherwise} \end{cases}$$

The above equation is valid for $B1$ and $B2$

For each pixel, we select a channel by discretizing the gradient orientation. The gradient magnitude can be seen as a measure of the edge strength.

### 3.1.2 Histogram

To provide some invariance to small deformation and for reducing the size of the feature map, the pixel-level features are aggregated to obtain a cell-based feature map. The cell is a dense grid of pixels block. The cell level feature map is collected into two feature maps based on the orientations of pixels in a cell, contrast sensitive feature map (C) and contrast insensitive feature map (D). The cell-based feature map, C, is computed with $2p$ quantized contrast sensitive orientation and D is computed with $p$ quantized contrast insensitive orientation. Let F be a pixel-level feature map for $w \times h$ image and $k > 0$ be the parameter defines the length of the cell. The cell-based feature map, C and D, with feature vector $C(i, j) \& D(i, j)$ for total number of cells ($0 \leq i \leq \left\lfloor \frac{w-1}{k} \right\rfloor \times (0 \leq j \leq \left\lfloor \frac{h-1}{k} \right\rfloor)$.
Chapter 3 Feature descriptor – FHOG

\( j \leq \left\lfloor \frac{n-1}{k} \right\rfloor \). The basic scheme for aggregating features is to map each pixel \((x, y)\) into cell \( \left( \left\lfloor \frac{x}{k} \right\rfloor, \left\lfloor \frac{y}{k} \right\rfloor \right) \) and define the feature vector at a cell to be the sum of pixel-level features in that cell. But in FHOG algorithm each pixel contributes to the feature vectors of four neighboring cells. This techniques is known as spatial binning [10]. According to the concept of spatial binning, the pixel in a cell not only contributes to its own cell but also to neighboring cells histogram. The closer the pixel to the cell center higher its contribution. The contribution is weighted gradient magnitude, and weight factors are calculated based on the relative position of a pixel with respect to cell center of cells. To calculate the weight factors, the following algorithm is used as shown in figure 8.

```c
// spatial binning
for(y=0:h)
  for(x=0:w)
    //normalized position of a pixel in a cell
    float cid_x = x/CELL_SZ;
    float cid_y = y/CELL_SZ;
    //Cell center coordinate
    int cid_x0 = floor(cid_x);
    int cid_y0 = floor(cid_y);
    /* weight factor calculation (based on the distance of a pixel from the
       center of the cell*)/
    //near the pixel to the center of the cell higher its contribution
    float cw_x0 = 1-(cid_x - cid_x0);
    float cw_y0 = 1-(cid_y - cid_y0);
    // adjacent cells center are normalized unit distance away
    float cw_x1 = 1-cw_x0;
    float cw_y1 = 1-cw_y0;
    /*Weight factors*/
    float weight_1 = cw_x1 *cw_y1;
    float weight_2 = cw_x0 *cw_y;
    float weight_3 = cw_x1 *cw_y0;
    float weight_4 = cw_x0 *cw_y0;
```

Figure 8 Spatial binning – pseudo code

As the contribution of a pixel to a cell histogram depends on its relative position with respect to cells center so, the pixels, which lies within unit normalized distance from the cell center, only have significant contribution. Therefore, 64 pixels contribute to the histogram of each cell except to cells at the image boundary. This can also be seen in the figure 9.
3.1.3 Normalization and truncation

Gradient strengths vary over a wide range of owing to local variations in illumination and foreground-background contrast, so local contrast normalization is good for performance [10]. The target cell is normalized with the respect energy of 4 blocks containing the target cell. A block is composed from 2x2 cells shown in figure 10.

The normalization factors for target cell with cell index \((i,j)\) is computed from contrast sensitive vector \(C(i,j)\) and can be written as
Chapter 3 Feature descriptor – FHOG

\[ N_{\delta,\gamma}(i, j) with \delta, \gamma \in \{-1, 1\}, \]

\[ N_{\delta,\gamma}(i, j) = \left( ||C(i, j)||^2 + ||C(i + \delta, j)||^2 + ||C(i + \delta, j + \gamma)||^2 + ||C(i, j + \gamma)||^2 \right)^{1/2} \tag{9} \]

Each factor measures the “gradient energy” in a square block of four cells containing \((i, j)\). The contrast sensitive features (CS) of a cell \((i, j)\) is obtained by summing all the normalized contrast sensitive histogram or cell-based feature vector followed by truncation with factor \(\alpha=0.2\) as shown in equation 10, where \(T_\alpha(v)\) denotes the component-wise truncation of a vector \(v\) by \(\alpha\).

\[ CS(i, j)_{FHOG} = 0.5 \times \sum_{\gamma, \delta \in \{-1, 1\}} T_\alpha \left( \frac{C(i, j)}{N_{\gamma, \delta}(i, j)} \right) \tag{10} \]

Same mathematical steps are applied to collect contrast insensitive feature from contrast insensitive histogram of the target cell.

\[ ICS(i, j)_{FHOG} = 0.5 \times \sum_{\gamma, \delta \in \{-1, 1\}} T_\alpha \left( \frac{D(i, j)}{N_{\gamma, \delta}(i, j)} \right) \tag{11} \]

We obtained 4 additional features known as texture features for each cell by summing all \(2^p\) contrast sensitive feature vector after normalizing with four different normalized factor followed by truncation. This is further multiplied with a factor \(\beta = 0.2357\) as shown in the equation (12).

\[ Texture(z)_{FHOG} = \beta \times \sum_{p=0}^{2^p-1} T_\alpha \left( \frac{C(i, j)}{N_{\delta,\gamma}(i, j)} \right) \tag{12} \]

Where \(z\) is between \((1, 4)\) corresponds to 4 normalization factors.

The histogram vector consists both contrast sensitive and insensitive bin. Thus, its dimension is 27. After normalizing and truncating the histogram, we add 4 texture feature which leads to 31 dimensional feature descriptor as shown in figure 11.
3.2 FHOG mapping on vector processor

We coarsely profiled the reference FHOG implementation for determining the bottlenecks of the algorithm on Intel i5 processor. We found out that the histogram generation process (bin index calculation, spatial binning and setting the weighted gradients to particular cells histogram) takes 55% of total computation of FHOG calculation, followed by normalization factor & feature extraction part which takes 25% and gradient calculations which takes 20%. Thus, the main bottleneck of the algorithm is histogram generation process. We transformed the operations used in histogram process such that it can be accelerated on vector processor. In this section, we explain the design strategy of FHOG implementation for vector processor of IPU platform in detail.

3.2.1 Data access pattern

The input is a gray scale image where pixel values ranges from [0,255]. The algorithm operates on different level of data abstractions such as pixels, cells and blocks to extract the feature of the image window. Using line based data access from VMEM increases the complexity of the implementation and number of operations to arrange data for different abstraction levels as comparison to block vector access from BAMEM. Initial analysis showed that line bases access of pixels heavily increases storage of intermediate data which directly increases in number of
LD\ST operations. The memory accesses are expensive both for performance and power consumption. With the block based vector access from BAMEM, we could keep the intermediate results into register files. As BAMEM supports access of 512 bits in various configuration as mentioned in section 2. We used 4x8 pixel blocks as a data accessing schemes. Thus, the outer loop iterates for \( \text{iter}_y = \text{round}(\frac{\text{HEIGHT}}{4}) \) and inner loop iterates for \( \text{iter}_x = \text{round}(\frac{\text{WIDTH}}{8}) \) these number of iterations to process pixels in vertical direction and horizontal direction respectively.

### 3.2.2 Gradient

To calculate the \( x \) \& \( y \) gradients of a pixel, we apply the filter \( G_x = [-1 \ 0 \ 1] \) and \( G_y = G_x^T \). This kernel is the perfect candidate for the SIMD parallelism. It requires 4 vectors access from the memory and 2 vector subtract operations to calculate both \( x \) \& \( y \) gradient of 32 pixels.

/*Gradient Calculation*/
for(i=0:iter_y)
    for(j=0:iter_x)
        GRAD_X_VEC[i][j] = BAMEM_LD(x+1, y) - BAMEM_LD(x-1, y);
        GRAD_Y_VEC[i][j] = BAMEM_LD(x, y+1) - BAMEM_LD(x, y-1);

Figure 12 Gradient calculation – pseudo vector code

### 3.2.3 Gradient magnitude

The magnitude of the gradient can be calculated as \( \sqrt{dx^2 + dy^2} \). The square root operation is not well suited for acceleration. We find in the literature that L1-norm can be used for calculating the magnitude of the gradient and it is claimed that it doesn’t affect the accuracy of the algorithm [13].

\[
\text{Grad}_\text{mag} = ||dx|| + ||dy|| \tag{13}
\]

There is another alternative which is widely used in DSP algorithm mentioned in [26].

\[
\text{Grad}_\text{mag} = \alpha \times \max(||dx||, ||dy||) + \beta \times \min(||dx||, ||dy||) \tag{14}
\]

where \( \alpha, \beta \) are constant.

The 1\textsuperscript{st} option requires less number of operations as compared to the 2\textsuperscript{nd} and it doesn’t affect the object tracking accuracy also [13]. Therefore, we chose equation 13 to calculate the magnitude of the gradient vectors. The pseudo code is presented in Figure 13.
/*Gradient Magnitude */
for(i=0:iter_y)
    for(j=0:iter_x)
/*L1-norm*/
    MAG_VEC[i][j] = ABS(GRAD_X_VEC[i][j]) + ABS(GRAD_Y_VEC[i][j]);

Figure 13 Gradient magnitude- pseudo vector code

3.2.4 Histogram generation

The image is divided into non overlapping cells of 4x4 pixels. Each pixel not only contributes to its own cell but also to neighboring cells histogram. This algorithm collects two types of histogram-(a) Contrast sensitive histogram (b) Contrast insensitive histogram. Contrast sensitive histogram results from the accumulation of the gradients with orientation from 0 to 2\pi and contrast insensitive is generated within 0 to \pi.

As mentioned in section (3.1.2), histogram generation is the combination of the quantizing the orientation of pixel into bin index and spatial binning.

3.2.5 Bin index calculation

There are two notable methods used in reference FHOG [10] and libHOG [13] implementations to calculate the bin index. The first method calculates the bin index based on the result of maximum dot product between gradient vector and bin edge vector. The bin vector has 2 components i.e. u and v. The bin vector’s component values are generated corresponds to the bin edges and stored in two tables ‘uu’ and ‘vv’. As mentioned in the snippet code figure 14, it requires 18 MUL operations and 9 if-then-else statements to calculate the bin index per pixel.

In libHOG [13], runtime look up table (LUT) is created based on the above method. As the gradient of a pixel is in the range of [-255, 255] so it requires a LUT of 512x512 bytes. Where each entry results one of the 18 quantized orientations. Further investigation shows that their program is serialized on bin index calculation i.e. it serially lookup for the bin index for each element in a SSE vector.

Both these methods are not good for the IPU as one requires branching statement and the other requires storage of 256KB of scalar data memory which cannot be fit in the data memory of IPU.
We developed a way to exploit both VLIW and SIMD parallelism without branching statements and a lookup table containing 5 tangents values. Our method collects the histogram of a cell without calculating the orientation of pixels. Instead of finding the orientation, we generate the selection mask for each bin. Then, we apply the selection mask to weighted gradient vector which results into the values belonging to a particular bin.

The bin range is $-10^0$ to $350^0$ with $20^0$ difference between each bin such as $-10^0$ to $10^0$, $10^0$ to $30^0$ etc. The approach is based on the exploiting the property of trigonometry and using the equation 15.

$$\tan(90^0 + \theta) = -\tan(\theta)$$
$$\tan(180^0 + \theta) = \tan(\theta)$$
$$\tan(270^0 + \theta) = -\tan(\theta)$$

where all angles are in degrees.

$$\tan(a_n) \times \text{ABS}(dx) \leq \text{ABS}(dy) < \tan(a_{n+1}) \times \text{ABS}(dx) \text{ where } a \text{ is an angle.} \quad (15)$$

The equation 15 checks whether a pixel with gradients (dx, dy) falls into bin bounded by angle $\tan(a_n)$ and $\tan(a_{n+1})$ in first quadrant. The value of $\tan(\theta)$ repeats in every quadrant except its sign which directly depends on the sign of gradients. If we have the knowledge about pixels quadrant and the result of above equation, we can easily generate a selection masks corresponds to number of bins.

As it can be seen from the pseudo code presented in figure 15 that it requires logical operations to generate the selection mask corresponds for each bin. These operations have latency of one cycle. Moreover, these operations are available on almost every issue slot of the vector processor. Thus, this implementation has potential to exploit both SIMD and VLIW parallelism.
Chapter 3 Feature descriptor – FHOG

/*Selection Mask */
for(i=0:iter_y)
 for(j=0:iter_x)
   FLAG_DX_GE = GRAD_X_VEC[i][j] ≥ 0;
   FLAG_DY_GE = GRAD_Y_VEC[i][j] ≥ 0;
 QUAD1[i][j] = FLAG_DX_GE & FLAG_DY_GE;
 QUAD2[i][j] = FLAG_DY_GE & (~FLAG_DX_GE);
 QUAD3[i][j] = (~FLAG_DX_GE) | (~FLAG_DY_GE);
 QUAD4[i][j] = FLAG_DX_GE & (~FLAG_DY_GE);

for(k=0:5)
   temp_VEC = ABS_GRAD_X_VEC[i][j] * tan_table[k]);
   FLAG_B[k] = ABS_GRAD_Y_VEC[i][j] ≥ temp_VEC;

for(l=1:5)
   BIN_FLAG[i][j][1] = (FLAG_B[l]) & (~FLAG_B[l+1]) & QUAD1[i][j];
   BIN_FLAG[i][j][9-1] = (FLAG_B[l]) & (~FLAG_B[l+1]) & QUAD2[i][j];
   BIN_FLAG[i][j][1+9] = (FLAG_B[l]) & (~FLAG_B[l+1]) & QUAD3[i][j];
   BIN_FLAG[i][j][18-1] = (FLAG_B[l]) & (~FLAG_B[l+1]) & QUAD4[i][j];

   BIN_FLAG[i][j][0] = ((FLAG_B[0]) & (~FLAG_B[1])) & QUAD1[i][j] & QUAD4[i][j];
   BIN_FLAG[i][j][9] = ((FLAG_B[0]) & (~FLAG_B[1])) & QUAD2[i][j] & QUAD3[i][j];

Figure 15 Selection mask generation – pseudo vector code

N.B - To generate the selection mask for bin 0 and bin 9, we need to consider two quadrant masks as their ranges lie in two quadrant i.e. for bin 0 it is −10° to 10° and for bin 9 it is 170° to 190°

3.2.6 Spatial Binning

Each pixel within a cell contributes to four cells histogram. The weight factors are calculated based on the relative position of a pixel with respect to cell centers. In the reference implementations, weight factors are calculated on-the-fly as shown in figure 8. Information of pixel position is required for calculating the weight factors on-the-fly.

One way to utilize the vector processor efficiently is by separating the processing pixel block from the weight factor calculation. The weight factors are static in nature. In figure 16, all black pixels have same relative position from its cell center and neighboring cells center. Thus, these pixels generate same weight factors for their corresponding neighboring cells. This pattern can be exploited by processing the common pixels between the cells and generating the repeating weights factors offline.

To explain the idea of our implementation, consider that we process a block of 4x4 pixels as shown in figures 16 and 17. Each pixel in the processing block contributes to the histogram of cell a, b, d and e, but with different weighted factors. By calculating the weight factors which correspond to this pixel block offline and multiplying the gradient magnitude, we can generate 4 weighted gradient magnitude vectors at same time.

Benefits of this methods are threefold. First, it reduces the number of operations. Second, we can generate partial histogram of 4 cells in parallel after calculating the selection masks. Third, it
Chapter 3 Feature descriptor – FHOG

requires, only 4 updates to generate complete histogram of a cell as comparison to serial process where it needs 64 updates to generate histogram of a cell.

N.B: We are in fact processing 4x8 pixels blocks. In each iteration, we get complete histogram of 2 cells and partial histogram of 4 cells. The only disadvantage of this method is that we have to process extra data to take care of the boundary pixels but the computation load doesn’t have any significant impact on the performance. In case of the KCF algorithm, this extra computation can be avoided as cosine window is applied on the feature of an image patch which takes cares of the boundary effects.

The next step is to combine both selection mask and spatial binning to generate the histogram of a cell. With the help of selection masks, we accumulate all the weighted gradients into corresponding contrast sensitive bins. This step requires an intra-vector addition as shown in the pseudo code, figure 18.
Figure 19 gives an overview of the histogram generation process for contrast sensitive bins where the whole process is divided into five stages. First, we generate the magnitude vector from x and y gradients. Second, we multiply it with bi-linear weights. Third, we apply the selection mask on the result which filters out the elements belonging to a particular bin. Then, we accumulate all those selected values and set the scalar result into a vector element of the histogram result vector corresponding to the to the desired bin index.

The contrast insensitive histogram is collected by quantizing the orientation of pixels into 9 bins which lie between 0° to 180°. On the other hand, contrast sensitive histogram is collected into 18 bins which lie between 0° to 360°. We can obtain contrast insensitive histogram from contrast sensitive histogram by mapping bins in 3rd & 4th quadrants to 1st & 2nd quadrants respectively as shown in figure 7. This can be achieved by histogram vector rotation, vector mux and vector addition operations as shown in the pseudo code figure 21.

/*Histogram generation-contrast sensitive bins*/
for(i=0:iter_y)
    for(j=0:iter_x)
        for(k=0:6) /*contribution to 6 cells Histogram */ /*Bilinear interpolation*/
            WEG_MAG_VEC[k] = WEGHT_VEC[k] * MAG_VEC[i][j];
        for(l=0:3)
            for(m=0:3)
                for(o=0:18)
                    temp_VEC   = MUX(WEG_MAG_VEC[k],0,BIN_FLAG[i][j][o]);
                    Scalar_sum = INTRA_VEC_ADD(temp_VEC);
                    temp_H_VEC[o] = Scalar_sum;
                    HISTOGRAM_VEC[i+l][j+m]+=HISTOGRAM_VEC[i+l][j+m]+temp_H_VEC;

Figure 18 Histogram generation – pseudo vector code
Chapter 3 Feature descriptor – FHOG

3.2.7 Energy Calculation

The energy of a cell is calculated by accumulating the square of contrast insensitive histogram bins i.e. \( \sum_{18}^{26} (\text{Histogram})^2 \). It can be calculated with three vector operations. The energy is calculated per cell, therefore, the iterators iterate for each cell.

```c
/*Energy Calculation*/
for(i=0; i<floor(HEIGHT/4); i++)
    for(j=0; j<floor(WIDTH/4); j++)
        Square_VEC = VEC_MUL(HISTOGRAM_VEC[i][j], HISTOGRAM_VEC[i][j]);
        temp_VEC = VEC_MUX(Square_VEC, 0, 0x7FC0000);
        ENERGY[i][j] = INTRA_VEC_ADD(temp_VEC);
```

Figure 21 Cell energy calculation - pseudo vector code
3.2.8 Feature calculation

The final step of the FHOG algorithm is to extract the feature of each cell. This first step in calculating the feature is to calculate the normalization factor. Each normalization factor is computed from the block energy. Each block consists of 4 cells as shown in figure 10.

```c
/*Normalization factor*/
for(i=1:floor(HEIGHT/4)-1)
 for(j=1:floor(WIDTH/4)-1)
 BLOCK_ENERGY = ENERGY[i][j]+ENERGY[i-1][j]+ENERGY[i][j-1]+ENERGY[i-1][j-1];
 temp_VEC = VEC_DIV(VEC_RES,VEC_SQRT(VEC_SET(BLOCK_ENERGY,0)));
 NORM[i-1][j-1] = VEC_GET(temp_VEC,0);
```

Figure 22 Normalization factor calculation - pseudo vector code

The computation of contrast sensitive/insensitive feature of target cell is divided into three steps:

1. Histogram of the target cell is normalized with energy of 4 blocks containing the target cell.
2. The result of step 1 is truncated with scaled $\alpha = 0.2$ on IPU.
3. Then, the addition of the normalized and truncated histogram results in the contrast sensitive/insensitive feature of the cell.

   The texture features are 4 scalar values which are obtained from the truncation of intra sum of 4 normalized and truncated contrast sensitive feature i.e. obtained in step 2. Here, the truncation factor is $\beta = 0.2537$ which we quantized to $\beta = 0.25$ because it can be replaced by right shifting the result by 2. Due to data dependency and long latency of operations, it is very hard to get instruction level parallelism, but this approach still manages to use SIMD parallelism.

```c
/*Feature Extraction*/
for(i=1:floor(HEIGHT/4)-1)
 for(j=1:floor(WIDTH/4)-1)
 Norm_HIST_VEC1 = MIN(HISTOGRAM_VEC[i][j]*NORM[i-1][j-1],0.2);
 Norm_HIST_VEC2 = MIN(HISTOGRAM_VEC[i][j]*NORM[i-1][j], 0.2);
 Norm_HIST_VEC3 = MIN(HISTOGRAM_VEC[i][j]*NORM[i][j-1], 0.2);
 Norm_HIST_VEC4 = MIN(HISTOGRAM_VEC[i][j]*NORM[i][j], 0.2);
```

/* contrast sensitive/insensitive feature of a cell */
 temp_feature=Norm_HIST_VEC1+Norm_HIST_VEC2+Norm_HIST_VEC3+Norm_HIST_VEC4;

```c
/* texture feature of a cell */
texture_feature_1=SHIFT_R(INTRA_VEC_ADD(MUX(Norm_HIST_VEC1,0,0x7FC0000)), 2);
texture_feature_2=SHIFT_R(INTRA_VEC_ADD(MUX(Norm_HIST_VEC2,0,0x7FC0000)), 2);
texture_feature_3=SHIFT_R(INTRA_VEC_ADD(MUX(Norm_HIST_VEC3,0,0x7FC0000)), 2);
texture_feature_4=SHIFT_R(INTRA_VEC_ADD(MUX(Norm_HIST_VEC4,0,0x7FC0000)), 2);
temp_feature[27] = temp_feature,texture_feature_1;
temp_feature[28] = temp_feature,texture_feature_2;
temp_feature[29] = temp_feature,texture_feature_3;
FEATURE[i-1][j-1] = temp_feature;
```

Figure 23 Feature calculation – pseudo vector code
The FHOG code structure on vector processor is represented in the pseudo code figure 24. By accessing data in form of block vectors from the BAMEM and transforming the histogram operations, we could fuse all the loops together for generating the partial histogram of 6 cells. Each iteration of the inner loop of the histogram generation produces complete histogram of two cells which helped to merge energy calculation with histogram generation loop as well.

```
FHOG(image, Height, Width)
{
    /* Histogram & cell energy calculation*/
    for(i=0:iter_y)
        for(j=0;iter_x)
            Cal_gradient();
            Cal_gradient_mag();
            Cal_selection_mask();
            Cal_weigted_gradient_mag();
            Cal_contrast_sensitive_hist();
            Cal_contrast_insensitive_hist();
            Cal_energy_two_cells();

    /*Block normalization & feature calculation*/
    for(i=1:floor(Height/cell_size)-1)
        for(j=1;floor(Width/cell_size)-1)
            for(k=0:4)
                Cal_block_energy();
                Cal_normalization_factor();
                Cal_contrast_sensitive_insensitive_feature();
                Cal_texture_feature();
}
```

3.3 Fixed point FHOG design

In this section, we present the fixed point design of FHOG. Table [2] summarize the bit requirements of each data component used in FHOG kernel.

- The input image is a gray scale image where each pixel ranges between [0,255]. Thus the format is U8.0.
- The gradient of pixels ranges from [-255, 255]. Thus the format is S8.0
- We use L1-norm to calculate the gradient magnitude of a pixel i.e. ||dx|| +|| dy|| which requires 9 integer bits only, i.e. U 9.0.
- To get the selection mask, we used the equation, where we multiply the absolute value of the gradient with the tangent value corresponds to the bin edges in the first quadrant.

\[
\tan(a_n) \times ABS(dx) \leq ABS(dy) < \tan(a_{n+1}) \times ABS(dx) \text{ where } a \text{ is angle.}
\]
Chapter 3 Feature descriptor – FHOG

- The bin range is $-10^0$ to $350^0$ with $20^0$ difference between each bin such as $-10^0$ to $10^0$, $10^0$ to $30^0$ etc. We only need to store bin angles in the first quadrant as scalar data i.e. $0^0$ to $70^0$. In that case, the tangent values lies between $[0, 2.74747]$ which corresponds to $\tan(0^0)$ and $\tan(70^0)$ respectively. To keep the product of tangent and absolute value of the gradient within 16 bits, we used a U2.6 bits format for representing the tangent values.
- The bilinear interpolation weights are calculated offline. The weight factors lies between (0, 1). Therefore, the format of bi-linear weights is U0.15. The weighted approximate gradient is in the format U9.15. To keep the result in 16 bits format and to avoid overflow in histogram generation process, the weighted gradient is shifted right by 5. Thus, its final format is U9.2.
- As mentioned in section 3.1.2, there are 64 pixels which contribute to the histogram of cell. In the worst case scenario, all 64 pixels fall into same bin with maximum gradient magnitude of 510. Therefore, it requires 6 more integer bits to accommodate the histogram. Thus, we needed U15.1 format for histogram. This case is too pessimistic as if all pixels belong to the same bin with maximum gradient magnitude their contribution would be less than the maximum gradient magnitude due to bilinear weights. We observed that format of U12.2 is sufficient for histogram. This format is applicable for both contrast sensitive and insensitive histogram.
- Energy of a cell is calculated from the contrast insensitive histogram $\Sigma_{i=0}^{26}(Histogram)^2$. The result of this equation cannot be accommodated in 16 bits so, we extended it to 32 bits format. In this case, integer part is more important than fractional part so, we dropped all the fractional bits and keep the format U30.0.
- Block energy is required for calculating the normalization factor. Block energy combination of 4 cells energy. Thus, it is format of U32.0.
- The normalization factor is calculated using equation 16 where $eps = 0.0001$, used to avoid division by zero.

$$norm\_factor = \frac{1}{\sqrt{(Block\ Energy_{B1} + eps)}} \quad (16)$$
- Ideally normalization factor values lies between (0, 10000] based on the above equation and requires U16.16 format. However, its value only contributes when it lies (0, 1). To understand this point, let’s jump to the feature calculation part. The contrast (in) sensitive feature value is directly proportional to the normalized histogram i.e. $Histogram \times norm\_factor$. The block energy is proportional to sum of square of 4 cells histogram i.e. $\Sigma_{i=0}^{26}(Histogram)^2$. If the block energy is zero that means the histogram values of 4 cells are zero which directly implies that feature vector will be zero. It results into zero feature irrespective of the normalization factor. Therefore, there is no point of extending the normalization factor into 32 bits so, we kept norm_factor in format U0.16.
- The contrast in/sensitive feature is obtained with the equation 17. The values never exceeds 0.4 Thus, the format U0.16.

$$C/IS\_F_{Target} = 0.5 \times \Sigma_{j=1}^{4} \min(Histogram_{(Target)} \times n_{b{j}}, 0.2) \quad \text{where} \quad i \in (0,17) \quad (17)$$
- As mentioned earlier, we collect 4 texture feature for a cell which is obtained from the equation 18.
Chapter 3 Feature descriptor – FHOG

\[ Texture \text{ } F(i)_{\text{Target}} = 0.2357 \times \sum_{i=0}^{i=17} \min \left( (\text{Histogram}_{(\text{Target})} \times n_{bj}), 0.2 \right) \]  

(18)

where \( i \in \{0,3\} \) and \( j \in \{0,3\} \)

The texture feature value never exceeds 0.8482 thus it can easily be represented with U0.16 format.

<table>
<thead>
<tr>
<th>Variables</th>
<th>Fixed point format</th>
</tr>
</thead>
<tbody>
<tr>
<td>Pixels</td>
<td>U0.8</td>
</tr>
<tr>
<td>Gradients</td>
<td>S8.0</td>
</tr>
<tr>
<td>Gradient magnitude</td>
<td>U9.0</td>
</tr>
<tr>
<td>Scale tangent</td>
<td>U2.6</td>
</tr>
<tr>
<td>Bilinear weights</td>
<td>U0.15</td>
</tr>
<tr>
<td>Histogram</td>
<td>U12.2</td>
</tr>
<tr>
<td>Energy</td>
<td>U30.0</td>
</tr>
<tr>
<td>Block Energy</td>
<td>U32.0</td>
</tr>
<tr>
<td>Normalization Histogram</td>
<td>U0.16</td>
</tr>
<tr>
<td>Contrast sensitive feature</td>
<td>U0.16</td>
</tr>
<tr>
<td>Contrast insensitive feature</td>
<td>U0.16</td>
</tr>
<tr>
<td>Texture feature</td>
<td>U0.16</td>
</tr>
</tbody>
</table>

Table 2 Fixed point optimization-FHOG

3.4 Performance benchmarking-FHOG

In this section, we compared our implementation with existing implementations that can be used with object tracking algorithm especially KCF algorithm. We found two implementations of FHOG which can be used directly without any major modifications. First, Dollar’s [14] SSE FHOG implementation. Second, libHOG [13] which uses both SSE and OpenMP parallel constructs to speed up the FHOG performance. libHOG implementation results in [13], are focused on object detection algorithm where the FHOG is run for a number of scales of a target image patch. We made appropriate changes to run for single scale of an image. Thereafter setting OpenMP threads to 6 on Intel Xenon X5670, we run it for several iteration to get average runtime. The detailed results are presented in the Table 3.

N.B- IPU is a complex multicore DSP platform which supports dynamic voltage frequency scaling (DVFS) to control the power based on the work load. For the confidentiality reason, we cannot expose the exact operating frequency ranges. For the purpose of benchmarking, the normal operating frequency of 500 MHz is considered.
Chapter 3 Feature descriptor – FHOG

Table 3 FHOG performance benchmark result

<table>
<thead>
<tr>
<th>Column 1</th>
<th>Dollar’s FHOG</th>
<th>libHOG</th>
<th>Our implementation</th>
<th>Our implementation</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Platform</strong></td>
<td>Core i5 (2415M).</td>
<td>Intel Xenon® X5670</td>
<td>IPU4</td>
<td>IPU5</td>
</tr>
<tr>
<td><strong>Frequency</strong></td>
<td>2.3GHz</td>
<td>2.93 GHz</td>
<td>~500 MHz</td>
<td>~500 MHz</td>
</tr>
<tr>
<td><strong>Memory</strong></td>
<td>L1, L2, L3 caches</td>
<td>L1, L2, L3 caches</td>
<td>Scratch pad memories</td>
<td>Scratch pad memories</td>
</tr>
<tr>
<td><strong>Vector length</strong></td>
<td>128 bits</td>
<td>128 bits</td>
<td>512 bits</td>
<td>512bits</td>
</tr>
<tr>
<td><strong>Data Type</strong></td>
<td>Float</td>
<td>Integer &amp; float</td>
<td>Integer fixed point</td>
<td>Integer fixed point</td>
</tr>
<tr>
<td><strong>No. of cores</strong></td>
<td>1</td>
<td>6</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

**Algorithmic difference**

- **Gradient magnitude**
  - L2-norm
  - L1-norm
  - L1-norm
  - L1-norm

- **Binning strategy**
  - Look up table
  - Look up table
  - Constant tangent table of 5 elements
  - Constant tangent table of 5 elements

- **Angular Binning**
  - Yes
  - No
  - No
  - No

**Parallel constructs**

- SIMD
- SIMD & OpenMP
- SIMD & VLIW
- SIMD & VLIW

**Results**

- **Time to process VGA frame**
  - 24 ms
  - ~6 ms
  - 4.24 ms
  - 3.97 ms

- **Accuracy**
  - ~73%
  - NA
  - ~71%
  - ~71%

The Dollar’s FHOG [14] implementation gives better tracking accuracy in comparison to others because it supports soft angular binning in addition to spatial binning. In soft angular binning, the weighted gradient magnitude is distributed into two neighboring bins besides its distribution it into four cell’s histogram.

The results show that FHOG implementation on single core of vector processor is ~6x and ~1.5x better performance as comparison to the Dollar’s SSE implementation and libHOG implementation respectively on intel x86 processor. We lose 2% of tracking accuracy for a ~6x performance gain with respect to SSE implementation. The tracking accuracy evaluation is explained in section 5.

IPU5 vector core has more issue slots as compared to the vector core of IPU4. Initial porting of FHOG algorithm from IPU4 to IPU5 resulted into 10% performance drop due to some internal
changes on vector issue slots. Later, we gained 6% better performance by optimizing the implementation of FHOG algorithm to exploit capabilities of vector core of IPU5.

To give some insight about the goodness of FHOG implementation on a vector core of Intel’s IPU without exposing the number of issue slots, we used normalized instruction level parallelism (ILP) parameter; the achieved instruction level parallelism (ILP) is divided by number of issue slots available on a vector core. We achieved an overall ILP of 0.44 and 0.4 on the vector core of IPU4 and IPU5 respectively. The FHOG implementation does not offer more parallelism to exploit the extra issues slots on IPU5. Therefore, the achieved ILP on IPU5 is less than the IPU4. In the histogram generation process, the loop length is determined by INTRA_VEC_ADD unit which is a critical resource. The loop length of feature calculation loop is determined by the VEC_SQRT and V_DIV operations as these operations are expensive (in terms of latency) and are on the same issue slot. We proposed appropriate improvements regarding these observations in the next section which can further increase the performance.

Vector processor is not designed to run a single kernel instead multiple kernels run in parallel. Therefore, FHOG performance is relatively good on vector processor. Moreover, the performance of FHOG on vector processor is quite good as compared to the other implementations on Intel x86 processor. This is an indication that IPU can be an attractive option for offloading such compute intensive work to it.

3.5 Architecture exploration

Although the benchmarking results show that the FHOG implementation on IPU5 has relatively good performance in comparison to x86 processor and IPU4 yet we think that it can be further improved. In this section, we detail out the concerns and propose appropriate improvements on the architecture level.

3.5.1 Observation 1

The histogram generation process is divided into five stages as shown in figure 19. As soon as we get a weighted gradient magnitude, we apply a VEC_MUX operation on it with a selection mask. Then, we accumulate the result by applying the INTRA_VEC_ADD which gives one or more scalar values. The scalar result of this operation is taken out of the vector register by applying a VEC_GET operation. Finally, we set this value to histogram vector by applying the VEC_SET operation. Thus, the generation of the histogram follows these operations VEC_MUX >> INTRA_VEC_ADD >> VEC_GET >> VEC_SET in a sequential order. The number of these operations linearly increase with the number of histogram bins.

All the operations, of the histogram generation process, are there on every other vector issue slot except for the INTRA_VEC_ADD operation. Therefore, histogram generation process serializes
on this unit. We are generating histograms of more than one cell simultaneously which helped to utilize INTRA unit efficiently as shown in figure 25. Even though we got good Instruction level Parallelism (ILP), but there are two negative effects on the schedule density of the VP. Firstly, resource utilization becomes ~60% due to VEC_MUX, VEC_GET and VEC_SET operations. Secondly, INTRA_VEC_ADD operations becomes the critical resource of the histogram generation process.

![Figure 25 Histogram of cells serialize on Intra vector addition](image)

3.5.1.1 Proposal

To resolve these effects, we propose to have extension to INTRA_VEC_ADD, and use two of them on one VP. Currently, INTRA_VEC_ADD unit supports two modes. In default mode, it takes all the elements of a vector then add them together to produce single scalar value. In second mode, it produces segmented intra sum of a vector. For example, in an N-way vector, N/m intra sums are produced where m is the number of elements per vector.

The high level view of the proposed extension is shown in figure 26. There is a merging of two operations in the intra sum unit. First improvement is to have the capability of selecting the elements for addition based on the selection mask. Second improvement is to have the capability of rotating the output of intra sum to the left by one position from the previous iteration and update the vector with scalar intra sum results. To maintain consistent behavior with previous unit, we pass a flag named ‘rot’. When it is set then the output after intra addition is given to MUX with previously generated output. It behaves as the vector is rotated by 1 position to the left. The previously generated output is passed to this unit via register b as shown in figure 26. Thus, it avoids the need of VEC_GET and VEC_SET operations. If ‘rot’ is not set then it behaves same as old unit.
3.5.1.2 Benefits

We performed scheduling experiments to find out the benefits of having two INTRA_VEC_ADD_EX on area and performance of FHOG. The benefits of having two of these units has following benefits:

- The overall performance of FHOG on vector increased by 14% as compared to the case having one INTRA_VEC_ADD units. The performance gain is not as per the expectation. Further investigation shows that the other operations (corresponding to histogram generations process) are scheduled on the same issue slot which limit the performance gain. To get better performance, the second unit must be on a different issue slot.
- Vector processor typically does not run single kernel at a time. Multiple kernels are run in parallel to get good throughput overall. Using two INTRA_VEC_ADD_EX units free up ~60% of resources which the HIVEC compiler can use to schedule other kernels in parallel. Thus, this solution will not only increase the performance of FHOG but also a promising solution to get better throughout overall.
- The extension increases the area of INTRA_VEC_ADD by 20%, and two of these units increase the total area of a core by 0.5%.

3.5.2 Observation 2

In the FHOG algorithm, we normalize the histogram a cell with respect to energy of four blocks as mentioned in the section 3 which requires a sequence of two operations i.e. division and square root. These two operations are relatively expensive in term of latency as compared to other arithmetic operations. Moreover, these functional units are on the same issue slots. In our case, these two operations determine the loop length of the feature extraction part. The latency of these operations can be partially/completely hidden with software pipelining. To take advantage of the vector sqrt and division operations, we manually unrolled the inner loop of feature calculation loop by loading more data (energy of cells) from memory then pipelined the loop. As the feature extraction part is quite data dependent so, the scheduler could only pipeline the load operations in
parallel with division and sqrt operations as shown in figure 27. This helps to produce more feature vectors per iteration of the outer loop. In our case one iteration of inner loop produces 4 feature vectors. Twelve load (LD) operations and VEC_SQRT+VEC_DIV take same cycles. In this particular case, we could hide the latency of sqrt and division operations with software pipelining only after manually unrolling which increased program size. We can make better use of these units than present.

3.5.2.1 Proposal & Benefits

We propose to have a unit which performs $1/\sqrt[n]{X}$ operation. We performed the scheduling experiment where we used only the SQRT operation to have the same timing behavior as of $1/\sqrt[n]{X}$ operation. For FHOG case, the total performance increased by 3%. The gain seems very small, however, this gain can be achieved without any cost on area as this functional unit uses same hardware blocks as used by SQRT and DIV operation.

With both proposals together, we gain 17% better performance than what we achieved on IPU5. The addition of functional units definitely cost area, but the performance benefits (17%) seem better than the incurred area cost (0.5%). Moreover, recent computer vision algorithms are frequently using HOG/FHOG as feature descriptor due its robust performance in tracking and detection algorithms. In that case, these proposals would definitely have a great impact on overall performance. Therefore, we recommend to support these changes for the next generation of Intel image processing units.
4. Fast Fourier Transform (FFT)

FFT is an algorithm for computing the Discrete Fourier Transform (DFT) of a signal efficiently. DFT is a computational intensive algorithm which has complexity $O(N^2)$ for calculating the DFT of N points. Cooley and Tukey [15] first introduced the concept of FFT which significantly reduced the computational complexity from $O(N^2)$ to $(N \log N)$. Since then plenty of algorithms have been introduced which further reduced the computational complexity of FFT such as radix-4, radix-8, split-radix [16], [17] etc. These algorithms recursively divide the computation into odd and even half parts and then extract as many common twiddle vectors as possible. Generally, the number of required multiplications and additions are used to compare the efficiency of different FFT algorithms. Therefore, the choice of FFT algorithm will have direct impact on the performance.

With reference to the target application, as shown in figure 11, the feature descriptor generates 31 channels/2D planes. For each plane, we have to apply 2D-FFT. This particular application offers implicit parallelism for the vector processor where we can process these 31 planes simultaneously with SIMD instructions. This case offers freedom to choose any scalar algorithm for the purpose of architecture exploration. In other words, with any FFT algorithm, we can gain ~31x performance on IPU with respect to a scalar FFT implementation on DSP platform. We chose radix-2 Cooley Tukey algorithm for this purpose, and algorithm is described in the section (4.1). In section (4.2), we will describe the mapping strategy of FFT algorithm on IPU. As mentioned earlier, Intel’s IPU supports only integer fixed point data so, we also performed fixed point analysis in section (4.3) by writing a fixed point FFT implementation in MATLAB and emulating the bit-true behavior of operations as used on vector core of IPU. After that, we will compare the performance vector processor FFT implementation with other platforms.

4.1 FFT algorithm

Before diving into the FFT algorithm, we first look at the DFT algorithm briefly. DFT is an algorithm to convert a time domain signal into frequency domain signal and vice versa. Given an input sequence $x(n)$, the N point DFT is defined as:

$$X(k) = \sum_{n=0}^{N-1} x(n). W_N^{nk}, k = 0, 1, 2, ..., N - 1$$

(19)

Where $n$ is the time index, $k$ is the frequency index, and the twiddle factor $W_N^{nk}$ is defined as:

$$W_N^{nk} = e^{(-2\pi jnk/N)} = \cos\left(\frac{2\pi nk}{N}\right) - j. \sin\left(\frac{2\pi nk}{N}\right)$$

(20)

Similarly, the inverse discrete Fourier transform (IDFT) can be expressed as:
Chapter 4 Fast Fourier Transform (FFT)

\[ x(n) = \frac{1}{N} \sum_{k=0}^{N-1} X(n) \cdot W_N^{-nk}, n = 0,1,2, ..., N - 1 \]  

(21)

According to above equations, we can see that \( N \) complex multiplication and \( N-1 \) complex addition operations are performed per sample. Thus, \( N^2 \) complex multiplication and \( N^2 - N \) complex addition operations are performed to complete the DFT/IDFT of \( N \) samples. This algorithms is computationally inefficient as the computation increases exponentially with number of samples.

4.2 Cooley-Tukey radix-2 algorithm

Cooley and Tukey algorithm [15] developed the efficient FFT algorithm in 1965. This algorithm recursively divides the transform of size \( N = N_1 N_2 \) into smaller DFTs of size \( N_1 \) and \( N_2 \) where \( N \) is highly composite. It reduces the computational complexity form \( O(N^2) \) to \( O(N \log N) \).

Radix-2 decimation-in-time (DIT) FFT is the simplest and most common form of the Cooley-Tukey algorithm where DFT of size \( N \) is divided into two interleaved DFTs of size \( N/2 \) in each recursive stage.

In this algorithm, equation (19), is decomposed into two halves i.e. sum over even indices \( n= 2m \) and sum over the odd indices \( n=2m+1 \):

\[ X(k) = \sum_{m=0}^{\frac{N}{2}-1} x(2m) \cdot W_N^{2mk} + \sum_{m=0}^{\frac{N}{2}-1} x(2m + 1) \cdot W_N^{(2m+1)k} \quad k = 0,1,2, ..., N - 1 \]  

(22)

In the above equation, the twiddle factor in 2\textsuperscript{nd} term can be expanded to \( W_N^{(2m)k} W_N^{k} \). Now, the term common between both sums can be simplified using the identity \( W_N^{mnk} = W_N^{nk/m} \).

\[ X(k) = \sum_{m=0}^{\frac{N}{2}-1} x(2m) \cdot W_{N/2}^{mk} + W_N^{k} \cdot \sum_{m=0}^{\frac{N}{2}-1} x(2m + 1) \cdot W_{N/2}^{mk} \quad k = 0,1,2, ..., N - 1 \]  

(23)

The two sums in above equations are now the DFT of the even indexed terms \( x(2m) \) and the odd indexed terms \( x(2m + 1) \), which are combined with twiddle factor \( W_N^{k} \).

In order to compute the transform more efficiently, the Cooley-Tukey algorithm divides \( X(k) \) into two halves, and the exploits the periodicity of sub-transform and symmetries in the trigonometric coefficients. The above equations can be rewritten as two halves with \( E(k) \) as the even sub transform, and \( O(k) \) substituted for the odd sub-transform:

\[ X(k) = E(k) + W_N^{k} O(k) \]  

(24)
Chapter 4 Fast Fourier Transform (FFT)

\[ X \left( k + \frac{N}{2} \right) = E \left( k + \frac{N}{2} \right) + W_N^{k + \frac{N}{2}} O \left( k + \frac{N}{2} \right) \]  

(25)

Where \( k = 0, 1, \ldots, \frac{N}{2} - 1 \).

The periodic property of DFT says:

\[ E \left( k + \frac{N}{2} \right) = E(k) \]  

(26)

\[ O \left( k + \frac{N}{2} \right) = O(k) \]  

(27)

By symmetry of exponential function, \( W_N^{k + \frac{N}{2}} = -W_N^k \), the equations (26) and (27), can be expressed as:

\[ X(k) = E(k) + W_N^k O(k) \]  

(28)

\[ X \left( k + \frac{N}{2} \right) = E(k) - W_N^k O(k) \]  

(29)

The original problem is solved by combining the above two equations. This makes clear that each of output shares common computation, approximately halving the number of arithmetic operations as compared to the DFT. In above equations, even and odd terms are itself DFTs which can also be computed with FFT algorithm. Thus, saving of computation compound with each stage of recursion. Let take an example of 8 points FFT. To calculate the 8 points FFT, first, the computation is divided in to two 4 points FFT followed by a combining stage consisting of many size-2 DFTs called butterfly operations as shown in figure 28.
Chapter 4 Fast Fourier Transform (FFT)

Each butterfly computation requires two complex/real data inputs. The butterfly computation structure represented in the figure 29 where P and Q are complex input, and W is a twiddle factor. It requires 2 complex load, 1 complex multiplication and 2 complex add/subtract operations which can be translated to total 4 LD, 4 MUL, 6 ADD/SUB and 4 ST operations.

\[
\begin{align*}
P & \rightarrow + \rightarrow P + (W \times Q) \\
Q & \rightarrow \times \rightarrow P - (W \times Q) \\
W & \rightarrow - \rightarrow P \div (W \times Q)
\end{align*}
\]

*Figure 29 Butterfly FFT computation*

4.3 FFT mapping on vector processor

To map the FFT, we choose radix-2, Decimation in Time, Cooley Tukey algorithm for the IPU implementation. As mentioned earlier 2D-FFT is applied all 31 channels/planes of the feature of an image patch, therefore, there exists data parallelism in the algorithm which can be exploited with SIMD instructions.

The size of FFT is determined by the size of target image patch. In the KCF’s MATLAB test bench, we found that 80% of the cases can be processed easily with 32x32 2D-FFT. Therefore, our prototype implementation is restricted to this size. The implementation of any algorithm on any platform has direct impact on the performance. As one of the purpose of this thesis work to find bottlenecks on the architecture level so we focused on getting the best performance on IPU. Some of the design choices are influenced by these factors.

The pseudo code of radix-2, Cooley-Tukey, FFT is represented in the figure 30. This implementation is not good for IPU implementation as after each stage of butterfly computations, the results are stored to the memory as shown in figure 31. Vector LD/ST operations from VMEM has same throughput as the MUL operation on vector processor. Each butterfly computation requires 2 times more load/store operations then multiply operations so, the performance is limited by load/store operations.

\[
\text{Number}_{-LD\_ST} \times \left(\frac{N}{2}\right) \times \log_2 N
\]

(30)

Where \( \log_2 N \) represents the number of stages in an N point radix-2FFT, \( \left(\frac{N}{2}\right) \) represents the butterfly computations per stage and \( \text{Number}_{-LD\_ST} \) represents the number of load/store operations per butterfly computation, being 8. The load/store operations dependency can be
avoided if we store the output of the each stages in the register files instead of transferring the data back to memory.

```plaintext
re_indexing_bitreversal(input, N);
stages = log(N)
n1=0;
n2=1;
for i =0:stages 
n1 =n2;
n2 =n2+n2;
index=0;
    for j=0:n1
        twiddle_factor = read_memory(index);
            for k=0:n2:N
                temp = complex_mul(input(k+n1), twiddle_factor)
                output(k+n1) = input(k) - temp;
                output(k) = input(k) + temp;
                index = index+offset;
            offset = offset_cal();
```

Figure 30 FFT Pseudo code

To get advantage of vector processor, we used register files to store the intermediate results between the stages as shown in figure 32. In this case, the lower bound of the performance (cycles) on IPU would be determined by MUL operations:

\[
Number_{MUL} \times \left( \frac{N}{2} \right) \times \log_2 N
\]  

(31)

Where \( \log_2 N \) represents the number of stages in an N point radix-2FFT, \( \left( \frac{N}{2} \right) \) represents the butterfly computations per stage and \( Number_{MUL} \) represents number of MUL operations per butterfly computation, being 4.
The structure of FFT implementation on vector processor is presented in figure 33. We divide the implementation into three layers: (a) Interface (b) Data handling (c) FFT computation layers.

**Interface layer** is an API layer from where host calls to IPU for FFT/IFFT computation based on the Flag. If Flag = 0 then it computes FFT else IFFT.

**Data handling layer** is responsible for loading and storing data to/from register files from/to memory. We used the static addressing scheme and fully unrolled the data access from memory. Therefore, we avoided the need of using bit reversal algorithm before FFT computation.

**FFT computation layer** is responsible for computing the FFT in recursive fashion where butter fly computation happens in $FFT_2$ function. All these functions in this layer are statically inline.
Chapter 4 Fast Fourier Transform (FFT)

Twiddle factors are not calculated on-the-fly instead generated offline and stored in the buffer with name ‘COS TABLE’. This manually unrolling of the algorithm with three layer structure and use of register files for storing intermediate results emulates the behavior of hardware FFT implementation. This helped to keep multiplier unit, a critical resource of the vector processor, busy every cycle. Moreover, the cycle counts are within the lower bound of performance for both vector core of IPU4 and IPU5.

4.4 Fixed point FFT design

In case of N-point FFT, the output of each point can grow up to N bits. The easy method to prevent overflow is to scale the input by 1/N before the FFT computation starts. When the input is scaled before the first stage of FFT then noise-to-signal ratio becomes proportional to $N^2$ [19].

To get the better signal-to-noise ratio (SNR), we follow the same scaling scheme as mentioned in the [18]. In this method, the input is scaled such that i.e. $|x|<1/2.4142$ before the first stage of FFT. Then, we scale the output by 2 every pair of stages to prevent overflow in the FFT computation. As we are working with fixed point machine so, instead of scaling with factor $1/2.4142$ we right shift the input data by 2 bits. It has some quantization effect that will be discussed later.

To calculate the SNR of IPU FFT implementation, first, we wrote a fixed point FFT implementation in MATLAB by reproducing the behavior of operations in IPU. We fed the uniformly distributed random numbers into the 2D fixed point implementation and MATLAB inbuilt FFT implementation. Then, we calculate the SNR by comparing the output power of these signals using the formula:

$$SNR\,(db) = 10 \log_{10} \frac{P_{output}}{P_{noise}} \quad (32)$$

where $P_{output}$ is an average output power of fixed point signal, $P_{noise}$ is the noise power computing from the difference of floating and fixed point FFT results.

As we used scaling factor of $1/4$ to write the FFT implementation on IPU so, we lose some information by quantizing the input. Therefore, we achieved $\sim 67\, \text{db}$ SNR as compared to $70.56\, \text{db}$ from the reference [18] for 32x32 FFT.

4.5 Performance benchmarking-FFT

To benchmark the performance of FFT on IPU, we chose the same method as used in FFTW benchmark [22]. It converted the execution time of FFT implementation on any platform into ‘mflops’ which is a scaled version of the throughput defined by:
Chapter 4 Fast Fourier Transform (FFT)

\[
mflops = \frac{5N \log_2 N}{(\text{time for one FFT in microseconds})}
\]  

(33)

Where \( N \) is the number of data points (the product of the FFT dimensions). This is not an actual flop count; it is simply a convenient scaling used in FFTW benchmark [24], based on the fact that the radix-2 Cooley-Tukey algorithm asymptotically requires \( 5N\log_2(N) \) floating-point operations.

We benchmarked our implementation with only the best FFT implementation, FFTW3 out-of-order from FFTW library. In case of KCF algorithm, 2D FFT is applied on 31 feature channels /planes separately. If we use the FFTW3 library for FFT computation on any processor then we have to run it 31 times to cover all feature channels. On the other hand, the FFT implementation on vector processor is specifically designed to exploit the data level parallelism in the KCF algorithm itself. Thus, it processes all 31 feature channels in parallel. To have fair comparison with scalar FFTW implementation, we divided the computation time of FFT on vector processor by 31.

The performance results of 1D FFT and 2D FFTs are presented in figures 34 and 35 respectively. The results show that the FFT implementation on a vector core of IPU can gain up to ~5x better performance than the FFTW on Intel Core Duo at 3 GHz frequency for 32 points FFT and ~7x gain compared to 32x32 points FFT.

N.B: In this particular algorithm we need to calculate the FFT of each channel/plane. This fact can also be exploited on the platform which support SIMD instructions or has a multicore architecture. The above benchmark represents the performance for the case where we use FFTW library for processing each feature channel in a sequential order.

![Figure 34 1D FFT performance comparison](image-url)
Chapter 4 Fast Fourier Transform (FFT)

Figure 35 2D-FFT performance comparison
5. Tracking accuracy evaluation

There are many methods to evaluate the tracking accuracy of an object tracker. A widely used method to determine the tracking accuracy is center location error. In this method, the average Euclidean distance between the center location of the tracked target and the manually labeled ground truth is calculated. The average center location error over all the frames of a video sequence summarizes the performance of tracker for that video sequence. However, when the tracker loses the target, the output location is random and the average error does not give correct indication of its tracking performance [23].

The Kernelized Correlation Filter (KCF) based tracker [5] uses precision plot to determine the tracking accuracy. A frame is considered correctly tracked if the predicted target center is within a distance threshold of a ground truth. Precision plot shows the percentage of correctly tracked frames for a range of distance thresholds. A higher precision at low threshold means the tracker is more accurate, while a lost target will prevent it from achieving perfect precision. As the representative precision score it uses the score for threshold of 20 pixels which is also used in other object tracking algorithms [23]. In other words, it calculates the percentage of correctly tracked frames within a threshold of 20 pixels of ground truth, and the result is termed as tracking accuracy.

We have implemented two major kernels, FHOG & FFT, of KCF algorithms in fixed point arithmetic. We lose precision as we transform the algorithm data type from float to integer fixed point. Therefore, it is necessary to determine the impact on tracking accuracy with fixed point implementations. To evaluate the tracking quality with fixed point implementations, we used the KCF MATLAB test-bench which is available as an open source [5]. This test bench contains a dataset of 50 videos targeted for unique objects and backgrounds. For each video, there is a text file containing ground truth with which the predicted position is compared to get a precision plot for threshold distances. The MATLAB implementation gives the average tracking accuracy achieved for the complete data set.

This section is divided into three subsections. In sub section (5.1), we see the impact of fixed point FHOG implementation, sub section (5.2) focuses on the effects of fixed point FFT and in subsection (5.3) we evaluate the tracking performance with both fixed point FFT and FHOG algorithm.

5.1 Tracking accuracy evaluation – Fixed point FHOG

First, we converted the reference C floating point FHOG implementation [10] into integer fixed point. Then, we wrapped the fixed point implementation into MEX file as shown in figure 24. A MEX file is a type of computer file that provides the interface between MATLAB and functions written in C/C++ or FORTRAN. It stands for “MATLAB executable” [31].
After integrating the, wrapped MEX, fixed point FHOG with MATLAB test bench, we run it with data set of 50 videos [5], to see the effects tracking result. The summary of results in presented into table 4.

**Table 4 Tracking evaluation results with fixed point FHOG**

<table>
<thead>
<tr>
<th>Sr. no.</th>
<th>FHOG-Version</th>
<th>Type</th>
<th>Binning strategy</th>
<th>Binning Type</th>
<th>Feature bits</th>
<th>Tracking accuracy (%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.</td>
<td>Dollar’s-FHOG</td>
<td>float</td>
<td>LUT based</td>
<td>float</td>
<td>Float</td>
<td>72.80</td>
</tr>
<tr>
<td>2.</td>
<td>Reference FHOG</td>
<td>float</td>
<td>Based on max dot product</td>
<td>float</td>
<td>Float</td>
<td>70.10</td>
</tr>
<tr>
<td>3.</td>
<td>Reference FHOG</td>
<td>float</td>
<td>Our strategy</td>
<td>float</td>
<td>Float</td>
<td>71.80</td>
</tr>
<tr>
<td>4.</td>
<td>Fixed point FHOG</td>
<td>fixed</td>
<td>Our strategy</td>
<td>fixed</td>
<td>16</td>
<td>71.70</td>
</tr>
<tr>
<td>5.</td>
<td>Fixed point FHOG</td>
<td>fixed</td>
<td>Our strategy</td>
<td>fixed</td>
<td>8</td>
<td>71.70</td>
</tr>
<tr>
<td>6.</td>
<td>Fixed point FHOG</td>
<td>fixed</td>
<td>Our strategy</td>
<td>fixed</td>
<td>4</td>
<td>71.50</td>
</tr>
<tr>
<td>7.</td>
<td>Fixed point FHOG</td>
<td>fixed</td>
<td>Our strategy</td>
<td>fixed</td>
<td>2</td>
<td>70.80</td>
</tr>
<tr>
<td>8.</td>
<td>Fixed point FHOG</td>
<td>fixed</td>
<td>Our strategy</td>
<td>fixed</td>
<td>1</td>
<td>70</td>
</tr>
<tr>
<td>9.</td>
<td>Fixed point FHOG</td>
<td>fixed</td>
<td>Our strategy</td>
<td>fixed</td>
<td>Binary output(either 1 or 0)</td>
<td>69</td>
</tr>
<tr>
<td>10.</td>
<td>Reference FHOG</td>
<td>float</td>
<td>Based on max dot product</td>
<td>float</td>
<td>1</td>
<td>70</td>
</tr>
</tbody>
</table>

We found some interesting results from tracking quality evaluation.

- With Dollar’s FHOG SSE [14] implementation KCF object tracker produces the best tracking accuracy results as shown in the table 4. The reason being that it supports soft-angular binning. In soft-angular binning, the weighted gradient of pixel is distributed into two neighboring bins.
- We achieved better results with our method of calculating bin index in comparison to the dot product based method. Our binning method uses sign information of the gradients which helped to distinctly put the pixel in right bins near the quadrant boundary.
Chapter 5 Tracking accuracy evaluation

- We kept intermediate results either in 16 or 32 bits except the final feature. We performed some experiments to determine how tracker behaves when we change the number of bits of final feature.
- The fixed point FHOG implementation reduced only ~1-2% of tracking accuracy depending on the final feature bits.
- We observed that this tracker works even with 1 bit of precision of the FHOG feature without showing any significant drop in tracking accuracy. The reason was not investigated further. But we verified the behavior with floating point reference implementation by quantizing the final output to one bit. The quantization is done according to equation (34).

\[
Float\left(\text{Right\_shift}\left(\text{int16}\left(FHOG_{\text{float}} \times \text{power}(2,16)\right),15\right)\right)/2
\]  

(34)

5.2 Tracking accuracy evaluation – FFT

In section 4, we explored the mapping of FFT radix-2 algorithm for 32x32 points on IPU. The size of the FFT is fixed which cannot cover all the cases available in the data set. We used MATLAB inbuilt FFT and introduced noise by quantizing the output of FFT with n number of bits as shown in figure 37. For quantizing the result, we used the equation 35:

\[
Float\left(\text{Right\_shift}\left(\text{int16}\left(MATLAB_{FFT} \times \text{power}(2,15)\right), 15 - n\right)\right)/\text{power}(2, n)
\]  

(35)

This method has three benefits. Firstly, by introducing noise with different number of bits will help to get the FFT SNR requirements for good tracking accuracy. As mentioned in section 4, we can gain 67 DB SNR on vector processor with 32x32 2D FFT implementation. With this experiment results, we can determine whether 67 DB SNR is good or bad. It also reflects on the bit requirement
Chapter 5 Tracking accuracy evaluation

of FFT. Thirdly, this experiment provides an early indication about the tracking accuracy before we have an end to end algorithm ported on the IPU, supporting all possible 2D FFT sizes.

<table>
<thead>
<tr>
<th>Sr. no.</th>
<th>Quantized of FFT/IFFT result with bits</th>
<th>Tracking accuracy (%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.</td>
<td>15</td>
<td>~69</td>
</tr>
<tr>
<td>2.</td>
<td>14</td>
<td>~69</td>
</tr>
<tr>
<td>3.</td>
<td>12</td>
<td>~69</td>
</tr>
<tr>
<td>4.</td>
<td>10</td>
<td>~67</td>
</tr>
<tr>
<td>5.</td>
<td>8</td>
<td>~66</td>
</tr>
</tbody>
</table>

In table 5 we presented the result of this experiment. We immediately lose 4% of tracking accuracy compared to the base case (~73%) by quantizing the output of MATLAB inbuilt FFT result with 15 bits. There is no further drop in performance until quantizing the FFT result with 10 bits where we lose another 2 percent of tracking accuracy.

We can conclude this experiment based on the fact that 1 bit provides 6 DB SNR. We get maximum tracking accuracy of 69% when FFT implementation has SNR between 90-72 DB and 67% when FFT SNR is 60 DB. Based on these results and achieved SNR of 67 DB on vector processor, we can say that the vector processor FFT implementation will give 68-69% of tracking accuracy.

5.3 Tracking accuracy evaluation – Fixed point FHOG and FFT

In this section, we present the evaluation of the effects of both fixed point FHOG and quantized FFT result with the help of KCF test-bench as shown in figure 38. As shown in the figure, the tracking results are presented with different number of feature bits and FFT.
Chapter 5 Tracking accuracy evaluation

The results show that fixed point FFT will have more adverse effects than fixed point FHOG implementation. The quantization of the floating point FFT result with 15 and 12 bits reduced the tracking accuracy by 4% except the case where fixed point FHOG feature is represented with 2 bits. Quantizing the FFT result further with 10 bits, the accuracy reduced further by 1%.

Table 6 Tracking accuracy result for fixed point FHOG & QFFT

<table>
<thead>
<tr>
<th>Sr. no.</th>
<th>Fixed point FHOG(feature bits)</th>
<th>FFT/IFFT quantization(bits)</th>
<th>Accuracy result (%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.</td>
<td>16</td>
<td>15</td>
<td>~69</td>
</tr>
<tr>
<td>2.</td>
<td>8</td>
<td>15</td>
<td>~69</td>
</tr>
<tr>
<td>3.</td>
<td>4</td>
<td>15</td>
<td>~69</td>
</tr>
<tr>
<td>4.</td>
<td>2</td>
<td>15</td>
<td>~69</td>
</tr>
<tr>
<td>5.</td>
<td>16</td>
<td>12</td>
<td>~69</td>
</tr>
<tr>
<td>6.</td>
<td>8</td>
<td>12</td>
<td>~69</td>
</tr>
<tr>
<td>7.</td>
<td>4</td>
<td>12</td>
<td>~69</td>
</tr>
<tr>
<td>8.</td>
<td>2</td>
<td>12</td>
<td>~67</td>
</tr>
<tr>
<td>9.</td>
<td>16</td>
<td>10</td>
<td>~68</td>
</tr>
<tr>
<td>10.</td>
<td>8</td>
<td>10</td>
<td>~68</td>
</tr>
<tr>
<td>11.</td>
<td>4</td>
<td>10</td>
<td>~68</td>
</tr>
<tr>
<td>12.</td>
<td>2</td>
<td>10</td>
<td>~66</td>
</tr>
</tbody>
</table>

5.4 Conclusion

- The tracking accuracy is decreased by 2% with fixed point FHOG implementation only.
- The tracking accuracy is decreased by 4% with quantizing the floating point FFT result only. Thus, fixed point FFT will have more impact than fixed point FHOG implementation on tracking accuracy.
- One bit increases the SNR by 6 DB. The quantization of floating point FFT with 10 bits (60 DB SNR) results into ~68% tracking accuracy and 12 bits (72 DB SNR) results into ~69% tracking accuracy. On a vector core of the Intel’s IPU, FFT implementation can achieve maximum of ~67 DB SNR with 16 bit implementation. Thus, achievable tracking accuracy on a vector core is between ~68 to ~69 %.
- Reducing the feature bits to less than 4 has direct impact on the tracking accuracy irrespective of FFT implementation.
6 Conclusion and future work

6.1 Conclusion

In this thesis work, we investigated the feasibility of Kernelized Correlation filter based object tracker. This had been done by efficiently mapping its two compute intensive kernels, FHOG and FFT, on a vector core of the Intel image processing unit (IPU) and evaluating the tracking accuracy of fixed point implementation of these kernels with the help of KCF’s MATLAB test bench [5].

Histogram generation process is the bottleneck of the FHOG algorithm. By transforming the conditional operations used in bin index calculation and spatial binning into logical and arithmetic operations, we efficiently used SIMD and VLIW parallelism. We achieved a good level of speed-up as compared to the implementations targeted for x86 architecture. With the current implementation of FHOG on IPU5, it takes ~4 ms to generate the feature of one VGA frame.

We propose two extensions to the existing instruction set, INTRA_VEC_ADD_EX and 1/SQRT(x). Scheduling experiments showed that these proposals can further increase the performance of FHOG implementation by 17% with an increase in overall area by 0.5%.

The feature descriptor, FHOG, generates 31 feature channels. The successive stages of KCF algorithm process all 31 feature channels. Thus, there exists data level parallelism which can be very well exploited on a vector core of the Intel’s IPU. For this particular case, FFT performance on vector processor gained ~5-7x speedup as compared to the best FFT implementation, FFTW3 out of order from FFTW library, on Intel dual core, 3 GHz processor. With respect to FFT, there is no bottleneck in the architecture, but the performance can be further increased by using advanced FFT algorithms.

The tracking accuracy results with fixed point implementation of these kernels shows that the fixed point FFT implementation reduced performance more than the fixed point FHOG implementation. We can achieve 68-69% of tracking accuracy overall by implementing these kernels on Intel image processing unit (IPU). We lose ~4-5% tracking accuracy as compared to floating point implementation which is within the margin.

The fixed point implementation of both kernels on a vector processor gives really good performance with maximum tracking loss of ~5% as compared to floating point implementation of these kernels. The overall performance of KCF algorithm can be further increased by offloading the complete algorithm to IPU. In preprocessing stage, each feature channel is weighted by a constant cosine window which can be processed with SIMD vector multiplication operation. In both detection and training stage of tracking algorithm, linear correlation is calculated between two variables which requires two operations i.e. multiplication and intra sum. As each variable has 31, 2D feature channels which makes the linear correlation a suitable candidate for SIMD operation. The processing of these two stages can exploit two multiplier units which are available on IPU5. Thus, we can also gain performance by implementing other algorithmic steps of KCF
algorithm besides FHOG and FFT on IPU. Therefore, we can say that the Intel’s IPU can be a decent option for low power vision devices to get good real time performance at the expense of losing some tracking accuracy.

6.2 Future Work

We achieved a good level of performance by accelerating the high compute kernels, FHOG & FFT, of KCF algorithm on the core, vector processor, of Intel’s IPU. There is still some scope of further improvements, and some further work is needed to port the complete pipeline of KCF algorithm on Intel’s IPU. A few possible directions are:

- To merge the histogram generation loop with feature calculation loop. After loop merging, we can software pipeline the inner loop which can provide higher instruction level parallelism. The loop merging requires some control code as both loops iterate for different times. It would be interesting to investigate the benefits of merging the two loops.

- Extend the FFT implementation on vector processor to support all the cases. As the multiplier unit is a critical resource for FFT implementation so, advanced FFT algorithms such as radix-8, split radix can be further mapped on the vector processor to get a better performance.

- To design fix point implementation of the functions such as cosine window & Gaussian function for IPU so, that these can be used to implement KCF algorithm on IPU.

- KCF algorithm has a limitation that it works with the fixed size of an object. Therefore, it is would be good idea to explore the feasibility of the object tracking which take care of the scale variation of a target object. In that case, it might be useful to have a multicore solution or to have an accelerator which produces various scale versions of a target object image.
References


[27]  https://en.wikipedia.org/wiki/Fast_Fourier_transform


