CrossLink-NX VGA MobileNet Human Counting on VVML Board

Reference Design

FPGA-RD-02231-1.0

June 2021

Disclaimers

Lattice makes no warranty, representation, or guarantee regarding the accuracy of information contained in this document or the suitability of its products for any particular purpose. All information herein is provided AS IS and with all faults, and all risk associated with such information is entirely with Buyer. Buyer shall not rely on any data and performance specifications or parameters provided herein. Products sold by Lattice have been subject to limited testing and it is the Buyer's responsibility to independently determine the suitability of any products and to test and verify the same. No Lattice products should be used in conjunction with mission- or safety-critical or any other application in which the failure of Lattice's product could create a situation where personal injury, death, severe property or environmental damage may occur. The information provided in this document is proprietary to Lattice Semiconductor, and Lattice reserves the right to make any changes to the information in this document or to any products at any time without notice.

Contents

Figures

Tables

Acronyms in This Document

A list of acronyms used in this document.

Acronym	Definition
CKPT	Checkpoint
CNN	Convolutional Neural Network
EVDK	Embedded Vision Development Kit
FPGA	Field-Programmable Gate Array
ML	Machine Learning
MLE	Machine Learning Engine
SPI	Serial Peripheral Interface
VIP	Video Interface Platform
VVML	Voice and Vision Machine Learning

1. Introduction

This document describes the Human Counting Design process using the CrossLink™-NX Voice and Vision Machine Learning (VVML) Board, Rev B platform. Human Counting is a subset of the generic Object Counting base design.

1.1. Design Process Overview

The design process involves the following steps:

1. Training Model
   - Setting up the basic environment
   - Preparing the dataset
   - Preparing Images
   - Labeling dataset of human bounding box
   - Training the machine
   - Training the machine and creating the checkpoint data
2. Neural Network Compiler
   - Creating Binary file with Lattice sensAI™ 4.0 program
3. FPGA Design
   - Creating FPGA Bitstream file
4. FPGA Bit stream and Quantized Weights and Instructions
   - Flashing Binary and Bitstream files
   - Binary File to Flash Memory on CrossLink-NX VVML Board, Rev B
   - Bit stream to Flash Memory on CrossLink-NX VVML Board, Rev B

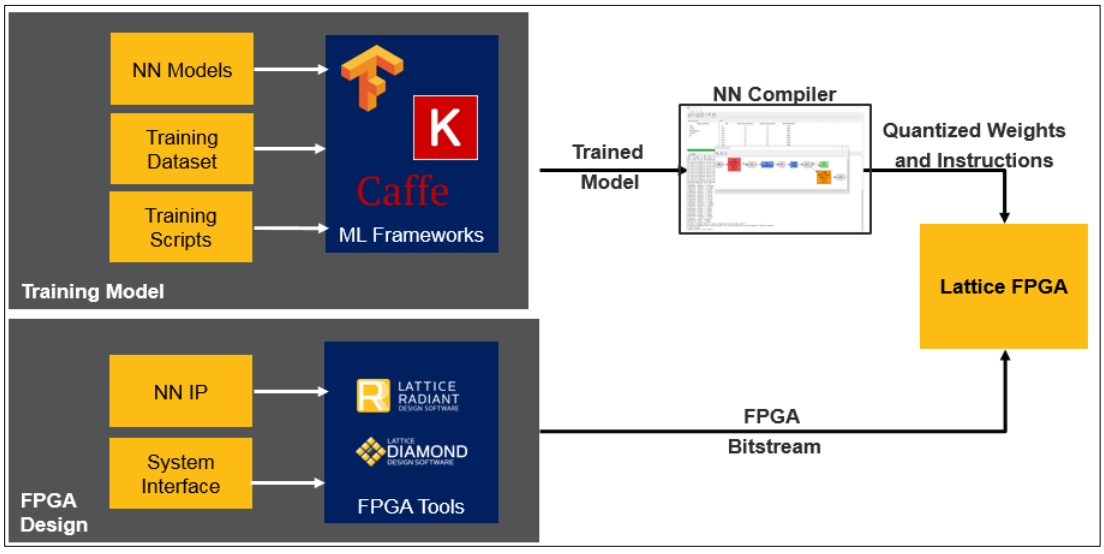

Figure 1.1. Lattice Machine Learning Design Flow

2. Setting up the Basic Environment

2.1. Software and Hardware Requirements

This section describes the required tools and environment setup for FPGA Bit stream and Flashing.

2.1.1. Lattice Software

- Lattice Radiant™ Tool v2.2 – Refer to http://www.latticesemi.com/LatticeRadiant.
- Lattice Radiant Programmer v2.2 – Refer to http://www.latticesemi.com/programmer.
- Lattice sensAI Compiler v4.0 – Refer to https://www.latticesemi.com/Products/DesignSoftwareAndIP/AIML/NeuralNetworkCompiler.

2.1.2. Hardware

CrossLink-NX Voice and Vision Machine Learning (VVML) Board

Refer to https://www.latticesemi.com/products/developmentboardsandkits/crosslink-nxvoiceandvisionmachinelearning.

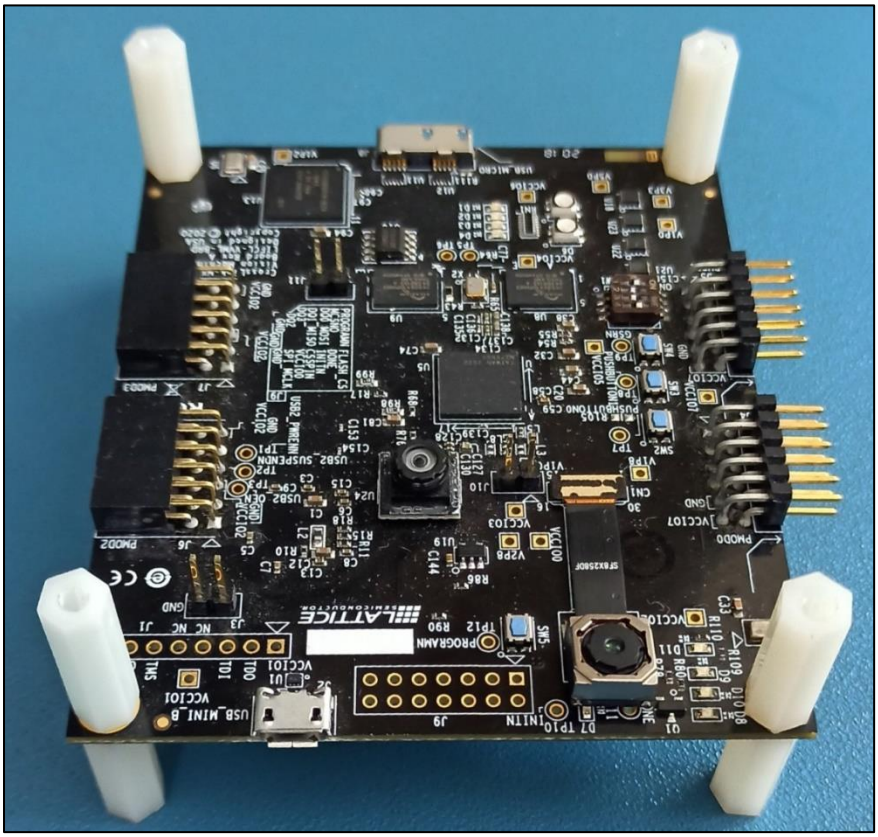

Figure 2.1. CrossLink-NX Voice and Vision Machine Learning Board, Rev B

2.2. Setting Up the Linux Environment for Machine Training

This section describes the steps for NVIDIA GPU drivers and/or libraries for 64-bit Ubuntu 16.04 OS.

Note: The NVIDIA library and the TensorFlow versions are dependent on the PC and the Ubuntu/Windows version.

2.2.1. Installing the NVIDIA CUDA and cuDNN Library for Machine Learning Training on GPU

2.2.1.1. Installing the NVIDIA CUDA Toolkit

To install the NVIDIA CUDA toolkit, run the commands below:

1. Download the NVIDIA CUDA toolkit.

```
$ curl -O
https://developer.download.nvidia.com/compute/cuda/repos/ubuntu1604/x86_64/cu
da-repo-ubuntu1604_10.1.105-1_amd64.deb
```

```
$ curl -O https://developer.download.nvidia.com/compute/cuda/repos/ubuntu1604/x86_64/cuda-repo-ubuntu1604_10.1.105-1_amd64.deb
  % Total    % Received % Xferd  Average Speed   Time    Time     Time  Current
                                 Dload  Upload   Total   Spent    Left  Speed
100  2832  100  2832    0     0   2204      0  0:00:01  0:00:01 --:--:--  2205
```

Figure 2.2. CUDA Repo Download

2. Install the deb package.

```
$ sudo dpkg -I ./cuda-repo-ubuntu1604_10.1.105-1_amd64.deb
```

```
$ sudo dpkg -i ./cuda-repo-ubuntu1604_10.1.105-1_amd64.deb
Selecting previously unselected package cuda-repo-ubuntu1604.
(Reading database ... 5287 files and directories currently installed.)
Preparing to unpack .../cuda-repo-ubuntu1604_10.1.105-1_amd64.deb ...
Unpacking cuda-repo-ubuntu1604 (10.1.105-1) ...
Setting up cuda-repo-ubuntu1604 (10.1.105-1) ...

The public CUDA GPG key does not appear to be installed.
To install the key, run this command:
sudo apt-key adv --fetch-keys http://developer.download.nvidia.com/compute/cuda/repos/ubuntu1604/x86_64/7fa2af80.pub
```

Figure 2.3. CUDA Repo Installation

3. Proceed with the installation.

```
$ sudo apt-key adv --fetch-keys
http://developer.download.nvidia.com/compute/cuda/repos/ubuntu1604/x86_64/7fa
2af80.pub
```

```
$ sudo apt-key adv --fetch-keys http://developer.download.nvidia.com/compute/cuda/repos/ubuntu1604/x86_64/7fa2af80.pub
Executing: /tmp/tmp.a2QZZnTMUX/gpg.1.sh --fetch-keys
http://developer.download.nvidia.com/compute/cuda/repos/ubuntu1604/x86_64/7fa2af80.pub
gpg: key 7FA2AF80: public key "cudatools <cudatools@nvidia.com>" imported
gpg: Total number processed: 1
gpg:               imported: 1  (RSA: 1)
```

Figure 2.4. Fetch Keys

```
$sudo apt-get update
```

```
$ sudo apt-get update
Ign:1 http://developer.download.nvidia.com/compute/cuda/repos/ubuntu1604/x86_64  InRelease
Hit:2 http://archive.ubuntu.com/ubuntu xenial InRelease
Get:3 http://developer.download.nvidia.com/compute/cuda/repos/ubuntu1604/x86_64  Release [697 B]
Get:4 http://developer.download.nvidia.com/compute/cuda/repos/ubuntu1604/x86_64  Release.gpg [836 B]
Hit:5 http://archive.ubuntu.com/ubuntu xenial-updates InRelease
Hit:6 http://security.ubuntu.com/ubuntu xenial-security InRelease
Hit:7 http://archive.ubuntu.com/ubuntu xenial-backports InRelease
Ign:8 http://developer.download.nvidia.com/compute/cuda/repos/ubuntu1604/x86_64  Packages
Get:8 http://developer.download.nvidia.com/compute/cuda/repos/ubuntu1604/x86_64  Packages [428 kB]
Fetched 429 kB in 1s (386 kB/s)
Reading package lists... Done
```

Figure 2.5. Updated Ubuntu Packages Repositories

```
$ sudo apt-get install cuda-9-0
```

```
$ sudo apt-get install cuda-9-0
Reading package lists... Done
Building dependency tree
Reading state information... Done
```

Figure 2.6. CUDA Installation Completed

2.2.1.2. Installing the cuDNN

To install the cuDNN:

1. Create your Nvidia developer account in https://developer.nvidia.com.
2. Download cuDNN library from https://developer.nvidia.com/compute/machine-learning/cudnn/secure/v7.1.4/prod/9.0_20180516/cudnn-9.0-linux-x64-v7.1.
3. Execute the commands below to install cuDNN.

```
$ tar xvfcudnn-9.0-linux-x64-v7.1.tgz
$ sudo cp cuda/include/cudnn.h/usr/local/cuda/include
$ sudo cp cuda/lib64/libcudnn* /usr/local/cuda/lib64
$ sudochmoda+r/usr/local/cuda/include/cudnn.h/usr/local/cuda/lib64/libcudnn*
```

```
$ tar xcf cudnn-9.0-linux-x64-v7.1.tgz
cuda/include/cudnn.h
cuda/NVIDIA_SLA_cuDNN_Support.txt
cuda/lib64/libcudnn.so
cuda/lib64/libcudnn.so.7
cuda/lib64/libcudnn.so.7.1.4
cuda/lib64/libcudnn_static.a

$ sudo cp cuda/include/cudnn.h /usr/local/cuda/include
$ sudo cp cuda/lib64/libcudnn* /usr/local/cuda/lib64
$ sudo chmod a+r /usr/local/cuda/include/cudnn.h /usr/local/cuda/lib64/libcudnn*
```

Figure 2.7. cuDNN Library Installation

2.2.2. Setting Up the Environment for Training, Freezing, and Pruning

This section describes the environment setup information for training and model freezing scripts for 64-bit Ubuntu 16.04. Anaconda provides one of the easiest ways to perform machine learning development and training on Linux.

2.2.2.1. Installing the Anaconda Python

To install the Anaconda Python 3:

1. Go to https://www.anaconda.com/products/individual#download.
2. Download the Python3 version of Anaconda for Linux.
3. Run the command below to install the Anaconda environment:

```
$  sh  Anaconda3-2019.03-Linux-x86_64.sh
```

Note: Anaconda3-<version>-Linux-x86_64.sh, version may vary based on the release

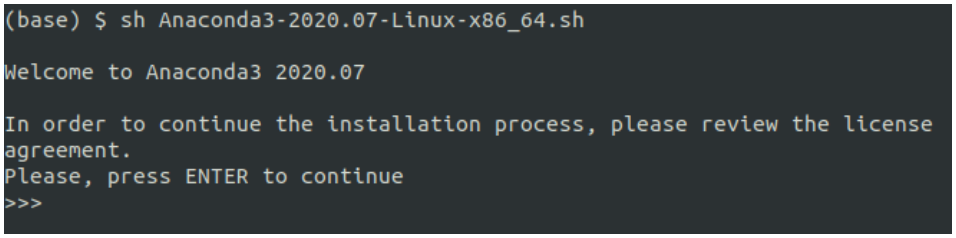

Figure 2.8. Anaconda Installation

4. Accept the license.

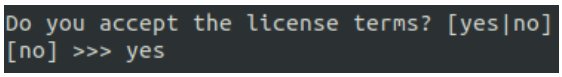

Figure 2.9. License Terms Acceptance

5. Confirm the installation path. Follow the instruction onscreen if you want to change the default path.

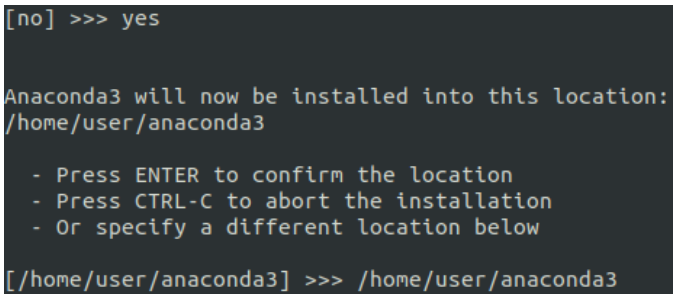

Figure 2.10. Installation Location

6. After installation, enter **No** as shown in Figure 2.11.

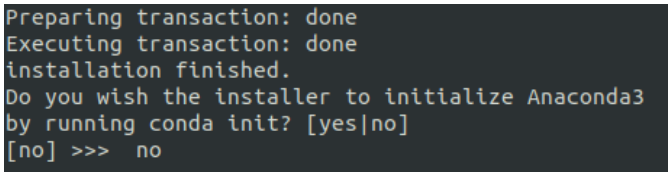

Figure 2.11. Launch/Initialization of Anaconda Environment

2.2.3. Installing TensorFlow version 1.14

To install TensorFlow version 1.14, run the commands below:

1. Activate the conda environment.

```
$ source <conda directory>/bin/activate
```

```
$ source anaconda3/bin/activate
(base) ~$
```

Figure 2.12. Anaconda Environment Activation

2. Install TensorFlow.

```
$ conda install tensorflow-gpu==1.14.0
```

```
$ conda install tensorflow-gpu==1.14.0
Collecting package metadata (current_repodata.json): done
Solving environment: failed with initial frozen solve. Retrying with flexible solve.
Collecting package metadata (repodata.json): done
Solving environment: done

## Package Plan ##

  environment location: /home/user/anaconda3/

  added / updated specs:
    - tensorflow-gpu==1.14.0
```

Figure 2.13. TensorFlow Installation

3. After installation, enter **Y** as shown in Figure 2.14.

```
  tensorboard         pkgs/main/noarch::tensorboard-1.14.0-pyhb230dea_0
  tensorflow          pkgs/main/linux-64::tensorflow-1.14.0-mkl_py36h4920b83_0
  tensorflow-base     pkgs/main/linux-64::tensorflow-base-1.14.0-mkl_py36he1670d9_0
  tensorflow-estima~ pkgs/main/noarch::tensorflow-estimator-1.14.1-pyh2649769_0
  termcolor           pkgs/main/linux-64::termcolor-1.1.0-py36h06a4308_1
  webencodings        pkgs/main/linux-64::webencodings-0.5.1-py36_1
  werkzeug            pkgs/main/noarch::werkzeug-0.16.1-py_0
  wrapt               pkgs/main/linux-64::wrapt-1.12.1-py36h7b6447c_1
  zipp                pkgs/main/noarch::zipp-3.4.0-pyhd3eb1b0_0


Proceed ([y]/n)? y
```

Figure 2.14. TensorFlow Installation Confirmation

Figure 2.15 shows that the TensorFlow installation is completed.

```
Preparing transaction: done
Verifying transaction: done
Executing transaction: done
```

Figure 2.15.TensorFlow Installation Completed

2.2.4. Installing the Python Package

To install the Python package, run the commands below:

1. Install Easydict.

```
$ conda install -c conda-forge easydict
```

```
(base) $ conda install -c conda-forge easydict
Solving environment: done
Collecting package metadata (repodata.json): done
Solving environment: done

## Package Plan ##

  environment location: /home/user/anaconda3

  added / updated specs:
    - easydict
```

Figure 2.16. Easydict Installation

2. Install OpenCV.

```
$ conda install opencv
```

```
(base) $ conda install opencv
Collecting package metadata (repodata.json): done
Solving environment: done

## Package Plan ##

  environment location: /home/user/anaconda3

  added / updated specs:
    - opencv
```

Figure 2.17. OpenCV Installation

2.2.5. Setting Up the Pruning Environment

The *setup_env.py* script is located under the *setup_optimization_env* directory with the VGA Code as shown in Figure 2.18.

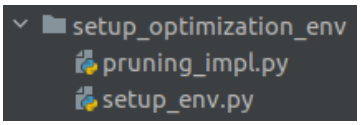

Figure 2.18. setup_optimization_env Directory Structure

To set up the pruning environment, run the command below.

```
$ python setup_env.py
```

```
setup_optimization_env$ python setup_env.py
Installing tensorflow_model_optimization
Collecting tensorflow_model_optimization
  Using cached tensorflow_model_optimization-0.5.0-py2.py3-none-any.whl (172 kB)
Collecting dm-tree~=0.1.1
  Downloading dm_tree-0.1.5-cp37-cp37m-manylinux1_x86_64.whl (294 kB)
     |████████████████████████████████| 294 kB 206 kB/s
Requirement already satisfied: numpy~=1.14 in /home/user/anaconda3/lib/python3.7/site-packages (from tensorflow_model_optimization) (1.18.5)
Requirement already satisfied: six~=1.10 in /home/user/anaconda3/lib/python3.7/site-packages (from tensorflow_model_optimization) (1.15.0)
Installing collected packages: dm-tree, tensorflow-model-optimization
Successfully installed dm-tree-0.1.5 tensorflow-model-optimization-0.5.0
/home/user/anaconda3/lib/python3.7/site-packages/tensorflow_model_optimization/python/core/sparsity/keras/pruning_impl.py
```

Figure 2.19. Setup Optimization Environment

3. Preparing the Dataset

This chapter describes how to create a dataset using examples from Google Open Image Dataset.

The Google Open Image Dataset version 4 (https://storage.googleapis.com/openimages/web/index.html) features more than 600 classes of images. The Person class of images include human annotated and machine annotated labels and bounding box. Annotations are licensed by Google Inc. under CC BY 4.0 and images are licensed under CC BY 2.0.

3.1. Downloading the Dataset

To download the dataset:

1. Clone the OIDv4_Toolkit repository by running the command below.

```
$ git clone https://github.com/EscVM/OIDv4_ToolKit.git
$ cd OIDv4_ToolKit
```

```
(base) k$ git clone https://github.com/EscVM/OIDv4_ToolKit.git
Cloning into 'OIDv4_ToolKit'...
remote: Enumerating objects: 25, done.
remote: Counting objects: 100% (25/25), done.
remote: Compressing objects: 100% (24/24), done.
remote: Total 382 (delta 3), reused 14 (delta 1), pack-reused 357
Receiving objects: 100% (382/382), 34.06 MiB | 752.00 KiB/s, done.
Resolving deltas: 100% (111/111), done.
(base) k$
```

Figure 3.1. Open Source Dataset Repository Cloning

Figure 3.2 shows the OIDv4 directory structure.

```
OIDv4_ToolKit/
├── classes.txt
├── images
│   ├── classes.png
│   ├── rectangle.png
│   └── visualizer_example.gif
├── LICENSE
├── main.py
├── modules
│   ├── bounding_boxes.py
│   ├── csv_downloader.py
│   ├── downloader.py
│   ├── image_level.py
│   ├── parser.py
│   ├── show.py
│   ├── test.py
│   └── utils.py
├── README.md
└── requirements.txt
```

Figure 3.2. OIDv4_Toolkit Directory Structure

View the OIDv4 Toolkit Help menu by running the command below.

```
$ python3 main.py -h
```

```
(base) k$ python3 main.py -h
usage: main.py [-h] [--Dataset /path/to/OID/csv/]
               [--classes list of classes [list of classes ...]]
               [--type_csv 'train' or 'validation' or 'test' or 'all']
               [--sub Subset of human verified images or machine generated h or m)]
               [--image_IsOccluded 1 or 0] [--image_IsTruncated 1 or 0]
               [--image_IsGroupOf 1 or 0] [--image_IsDepiction 1 or 0]
               [--image_IsInside 1 or 0] [--multiclasses 0 (default or 1]
               [--n_threads [default 20]] [--noLabels]
               [--limit integer number]
               <command> 'downloader', 'visualizer' or 'ill_downloader'.

Open Image Dataset Downloader

positional arguments:
  <command> 'downloader', 'visualizer' or 'ill_downloader'.
                        'downloader', 'visualizer' or 'ill_downloader'.
```

Figure 3.3. Dataset Script Option/Help

2. Use the OIDv4 Toolkit to download dataset. Download Person class images by running the command below.

```
$ python3 main.py downloader --classes Person --type_csv validation
```

```
(base) k$ python3 main.py downloader --classes Person --type_csv validation --limit 200
```

Figure 3.4. Dataset Downloading Logs

Figure 3.5 shows the downloaded dataset directory structure.

```
OID
├── csv_folder
│   ├── class-descriptions-boxable.csv
│   └── validation-annotations-bbox.csv
└── Dataset
    └── validation
        └── Person
            ├── ff7b4cc8ca9b6592.jpg
            └── Label
                └── ff7b4cc8ca9b6592.txt
```

Figure 3.5. Downloaded Dataset Directory Structure

3. Lattice training code uses KITTI (.txt) format. The downloaded dataset is not in the required KITTI format. Convert the annotation to KITTI format.

```
$ sed -i -- 's/Person/Person 0 0 0/g' OID/Dataset/validation/Person/Label/*
$ sed -i -- 's/Person/Person 0 0 0/g' OID/Dataset/train/Person/Label/*
$ sed -i -- 's/Person/Person 0 0 0/g' OID/Dataset/test/Person/Label/*
```

```
(base) k$ cat OID/Dataset/validation/Person/Label/ff7b4cc8ca9b6592.txt
Person 324.614144 69.905733 814.569472 681.9072
(base) k$ sed -i -- 's/Person/Person 0 0 0/g' OID/Dataset/validation/Person/Label/*
(base) k$ cat OID/Dataset/validation/Person/Label/ff7b4cc8ca9b6592.txt
Person 0 0 0 324.614144 69.905733 814.569472 681.9072
(base) k$
```

Figure 3.6. OIDv4 Label to KITTI Format Conversion

Note: KITTI Format: Person 0 0 0 324.61 69.90 814.56 681.90. It has class ID followed by truncated, occluded, alpha, Xmin, Ymin, Xmax, Ymax. The code converts Xmin, Ymin, Xmax, Ymax into x, y, w, h while training as *bounding box rectangle coordinates*.

3.2. Visualizing and Tuning/Cleaning Up the Dataset

To visualize and annotate the dataset, run the commands below:

1. Visualize the labelled images.

```
$ python3 main.py visualizer
```

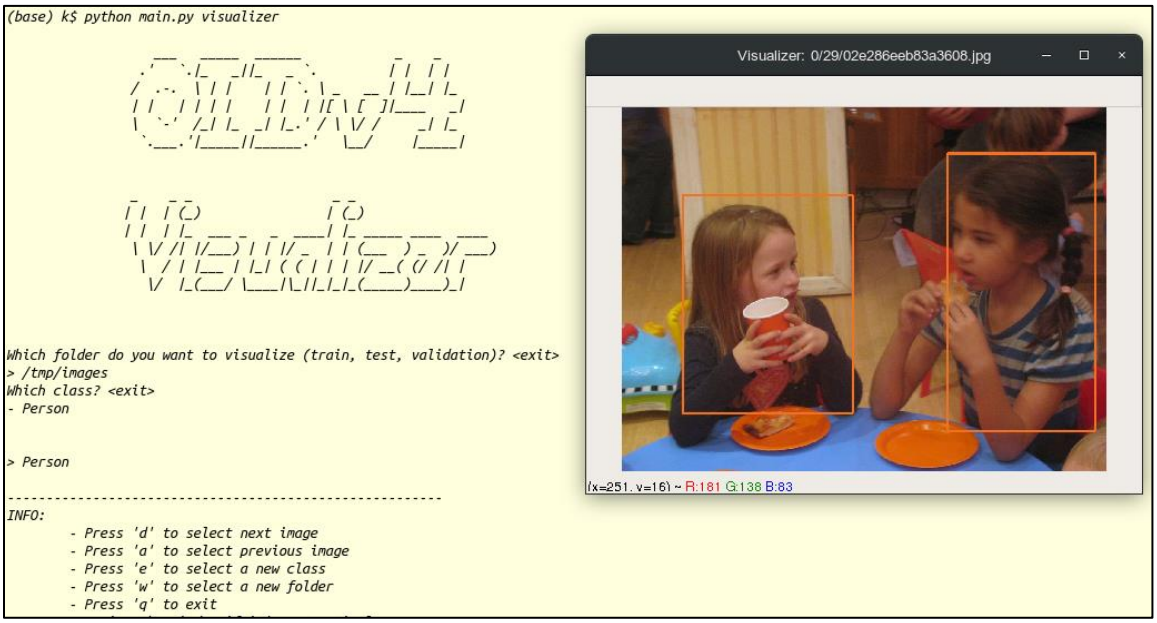

Figure 3.7. Toolkit Visualizer

2. Clone the manual annotation tool from the GitHub repository.

```
$ git clone https://github.com/SaiPrajwal95/annotate-to-KITTI.git
```

```
(base) k$ git clone https://github.com/SaiPrajwal95/annotate-to-KITTI.git
Cloning into 'annotate-to-KITTI'...
remote: Enumerating objects: 27, done.
remote: Total 27 (delta 0), reused 0 (delta 0), pack-reused 27
Unpacking objects: 100% (27/27), done.
(base) k$ _
```

Figure 3.8. Manual Annotation Tool – Cloning

3. Go to the *annotate-to-KITTI* directory.

```
$ cd annotate-to-KITTI
$ ls
```

```
annotate-to-KITTI/
├── annotate-folder.py
└── README.md
```

Figure 3.9. Manual Annotation Tool – Directory Structure

4. Install the dependencies (OpenCV 2.4).

```
$ sudo apt-get install python-opencv
```

5. Launch the utility.

```
$ python3 annotate-folder.py
```

6. Set the dataset path and default object label.

```
(base) k$ python3 annotate-folder.py
Enter the path to dataset: /tmp/images
Enter default object label: Person
[{'label': 'Person', 'bbox': {'xmin': 443, 'ymin': 48, 'xmax': 811, 'ymax': 683}}]
(base) k$ _
```

Figure 3.10. Manual Annotation Tool – Launch

7. For annotation, run the script provided in the website below.
   https://github.com/SaiPrajwal95/annotate-to-KITTI

3.3. Data Augmentation

Deep networks need a large amount of training data to achieve good performance. To train a neural network using minimal training data, image augmentation is usually required to boost the performance. Image augmentation creates training images through different processes such as random rotation, shifts, shears and flips, and others. Combinations of multiple processes may also be used.

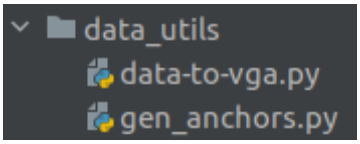

Figure 3.11. Augmentation Directory Structure

- data-to-vga.py: It is augmentations operations script.
- gen_anchors.py: It runs k-means on width and height of dataset boxes to find optimal anchor boxes.

3.3.1. Running the Augmentation

Run the augmentation using the following command:

```
$ python data-to-vga.py  --input <input_dataset_path>  --output
<output_dataset_path> --type kitti --output_dimension 640,480 --canvas_shift  --
brightness  --contrast
```

```
data_utils$ python data-to-vga.py --input ~/datasets/person_dataset --output ~/datasets/vga-data  --canvas_shift  --brightness  --contrast
Creating Directory: /home/user/datasets/vga-data
Creating Directory: /home/user/datasets/vga-data/ImageSets
Creating Directory: /home/user/datasets/vga-data/training
Creating Directory: /home/user/datasets/vga-data/training/images
Creating Directory: /home/user/datasets/vga-data/training/labels
Creating Directory: /home/user/datasets/vga-data/testing
Creating Directory: /home/user/datasets/vga-data/testing/images
Creating Directory: /home/user/datasets/vga-data/testing/labels
100%|                                                                                                                                  | 30078/30078 [21:21<00:00, 23.47it/s]
Time Taken : 1281.9826560020447
============================================================================================================
Augmentation operation finished

Total Discarded invalid boxes :4
============================================================================================================
```

Figure 3.12. Running the Augmentation

The *data-to-vga,py* contains additional optional flags as described below.

Flags explanation:

- --input : Input Dataset Path
- --output : Output Dataset Path
- --canvas_shift : Flag to add Canvas shifting augmentation
- --brightness : Flag to add brightness augmentation
- --contrast : Flag to add contrast augmentation
- --pixel_shift : Number of pixel shift in canvas shift augmentation
- --type : (kitti/pascal) type of input dataset (default kitti)
- --output_dimension : Expected output dimension in form of x, y (Default 640, 480)
- --visualize : Flag that saves images in /tmp/visualize with drawn box (Optional)
- --canvas_shift_percentage : Percentage of dataset to apply canvas shift augmentation
- --brightness_percentage : Percentage of dataset to apply brightness augmentation
- --contrast_percentage: Percentage of dataset to apply contrast augmentation
- --apply_multiple_aug: flag to apply another augmentation on already augmented image

3.3.2. Generating Anchors from Dataset (Optional)

To run k-means on the dataset box width and height, execute the command below.

```
$ python gen_anchors.py –labels <label directory path> -output_dir <output path to save anchors> -num_cluster 7
```

Arguments information:

- -labels: Label directory path
- -output_dir: Output path for generated anchors
- -num_cluster: Number of anchor boxes the script should generate, which is seven by default.

```
python gen_anchors.py -labels vga-dataset/training/labels/ -output_dir vga-dataset/   -num_clusters 7
iter 1: dists = 486090.2306730367
iter 2: dists = 36250.608193920634
iter 3: dists = 15238.320192258758
...
...
...
iter 62: dists = 182.10860154061328
iter 63: dists = 124.73942131292999
iter 64: dists = 80.19427102517825
iter 65: dists = 27.26131817124239
Centroids =  [[284.92431861 281.3585364 ]
 [107.91106147 143.30844663]
 [144.554071   177.49012884]
 [195.07541761 219.99530481]
 [159.9246952   89.24887741]
 [ 61.58436364  73.53124943]
 [ 82.3695388  107.13941642]]
(7, 2)

centroids.shape (7, 2)
```

Figure 3.13. Running the Script to Generate Anchors

At the end of script, optimal centroids that can be used as anchor boxes are printed.

Based on the above example, the following seven anchors are created:

Note: It is a good practice to sort the anchors by size.

- [285, 281]
- [195, 220]
- [145, 177]
- [108, 143]
- [160, 89]
- [82, 107]
- [62, 74]

4. Training the Machine

4.1. Training Code Directory Structure

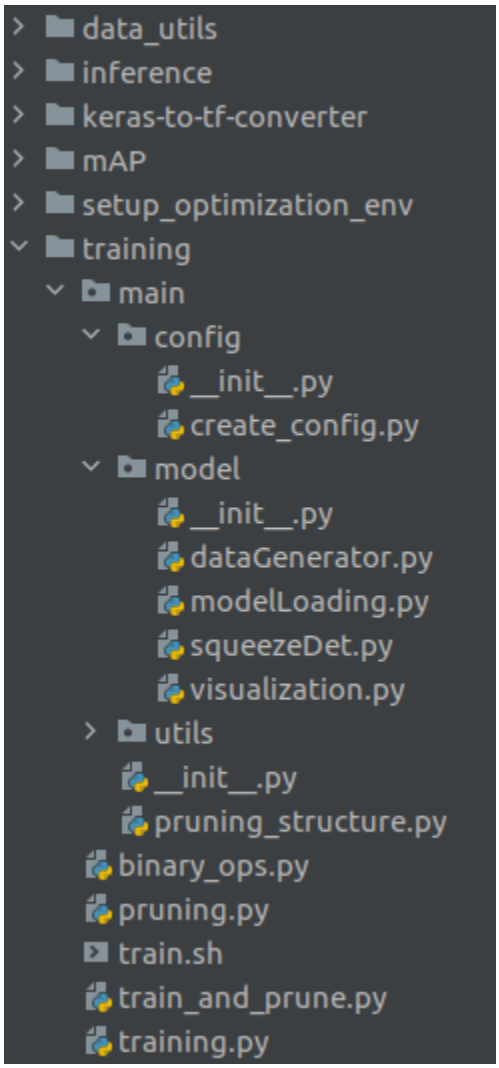

Figure 4.1. Training Code Directory Structure

4.2. Neural Network Architecture

4.2.1. Human Count Training Network Layers

This section provides information on the Convolution Network Configuration of the Human Count Detection design. The Neural Network model of the Human Count Detection design uses MobileNetV1 NN base model and the detection layer of SqueezeDet model.

Table 4.1. Convolution Network Configuration of Human Count Detection Design

Image Input (640 × 480 × 1)		
Fire 1	DWConv3 – 32	Conv3 - # where: • Conv3 = 3 × 3 Convolution filter Kernel size • # = The number of filter DWConv3 - 32- # where: • DWConv3 = Depth wise convolution filter with 3 × 3 size • # = The number of filter Conv1 - 32- # where: • Conv1 = 1 × 1 Convolution filter Kernel size • # = The number of filter For example, Conv3 - 16 = 16 3 × 3 convolution filters BN – Batch Normalization
	BN	
	Relu	
	Maxpool	
	Conv1 – 32	
	BN	
	Relu	
	Maxpool	
Fire 2	DWConv3 – 32	
	BN	
	Relu	
	Maxpool	
	Conv1 – 32	
	BN	
	Relu	
	Maxpool	
Fire 3	DWConv3 – 64	
	BN	
	Relu	
	Maxpool	
	Conv1 – 64	
	BN	
	Relu	
Fire 4	DWConv3 – 64	
	BN	
	Relu	
	Conv1 – 64	
	BN	
	Relu	
Fire 5	DWConv3 – 88	
	BN	
	Relu	
	Conv1 – 88	
	BN	
	Relu	

Fire 6	DWConv3 – 128	
	BN	
	Relu	
	Conv1 – 128	
	BN	
	Relu	
Conv12	Conv3 – 42	

- Human Count Network structure consists of six fire layers followed by one convolution layer. A fire layer contains convolution, depth wise convolution, batch normalization and relu layers with pooling layer only in fire1, fire 2, and fire 3. Layers fire 4, fire 5, and fire 6 do not contain pooling.
- Note that fire 1 and fire 2 contain two maxpooling operations to reduce calculation complexity and model size.
- Table 4.1 details the contents of the fire layers: convolution (conv), batch normalization (bn), and relu.
- Layer information:
  - Convolutional Layer
    In general, the first layer in a CNN is always a convolutional layer. Each layer consists of number of filters (sometimes referred as kernels), which convolves with input layer/image and generates activation map (such as feature map). This filter is an array of numbers (the numbers are called weights or parameters). Each of these filters can be thought of as feature identifiers, like straight edges, simple colors, curves, and other high-level features. For example, the filters on the first layer convolve around the input image and *activate* (or compute high values) when the specific feature (such as curve) it is looking for is in the input volume.
  - Relu (Activation layer)
    After each conv layer, it is convention to apply a nonlinear layer (or activation layer) immediately afterward. The purpose of this layer is to introduce nonlinearity to a system that basically has just been computing linear operations during the conv layers (just element wise multiplications and summations).In the past, nonlinear functions like tanh and sigmoid were used, but researchers found out that ReLU layers work far better because the network is able to train a lot faster (because of the computational efficiency) without making a significant difference to the accuracy. The ReLU layer applies the function f(x) = max (0, x) to all of the values in the input volume. In basic terms, this layer just changes all the negative activations to 0.This layer increases the nonlinear properties of the model and the overall network without affecting the receptive fields of the conv layer.
  - Pooling Layer
    After some ReLU layers, programmers may choose to apply a pooling layer. It is also referred to as a down sampling layer. In this category, there are also several layer options, with maxpooling being the most popular. This basically takes a filter (normally of size 2 × 2) and a stride of the same length. It then applies it to the input volume and outputs the maximum number in every sub region that the filter convolves around.
    The intuitive reasoning behind this layer is that once it is known that a specific feature is in the original input volume (there is a high activation value), its exact location is not as important as its relative location to the other features. As you can imagine, this layer drastically reduces the spatial dimension (the length and the width change but not the depth) of the input volume. This serves two main purposes. The first is that the number of parameters or weight is reduced by 75%, thus lessening the computation cost. Second, itl controls over fitting. This term refers to when a model is so tuned to the training examples that it is not able to generalize well for the validation and test sets. A symptom of over fitting is having a model that gets 100% or 99% on the training set, but only 50% on the test data.
  - BatchNorm
    Batch normalization layer reduces the internal covariance shift. To train a neural network, perform some preprocessing to the input data. For example, all data can be normalize so that it resembles a normal distribution (that means, zero mean and a unitary variance). Reason being preventing the early saturation of non-linear activation functions like the sigmoid function, assuring that all input data is in the same range of values, and others.
    But the problem appears in the intermediate layers because the distribution of the activations is constantly changing during training. This slows down the training process because each layer must learn to adapt themselves to a new distribution in every training step. This problem is known as internal covariate shift.

Batch normalization layer forces the input of every layer to have approximately the same distribution in every training step by following below process during training time:
Calculate the mean and variance of the layers input.
Normalize the layer inputs using the previously calculated batch statistics.
Scale Layer scales and shifts to obtain the output of the layer.
This makes the learning of layers in the network more independent of each other and allows you to be care free about weight initialization, works as regularization in place of dropout and other regularization techniques.

- Depth wise Convolution and 1 × 1 Convolution Layer
  Depth wise convolutions are used to apply a single filter per each input channel (input depth). Pointwise convolution, a simple 1 × 1 convolution, is then used to create a linear combination of the output of the depth wise layer.
  Depth wise convolution is extremely efficient relative to standard convolution. However it only filters input channels, it does not combine them to create new features. So an additional layer that computes a linear combination of the output of depth wise convolution through 1 × 1 convolution is needed to generate these new features.
  A 1 × 1 convolutional layer that compresses an input tensor with large channel size to one with the same batch and spatial dimension, but smaller channel size. Given a 4D input tensor and a filter tensorshape [filter_height, filter_width, in_channels, channel_multiplier] containing in_channels convolutional filters of depth 1, depthwise_conv2d applies a different filter to each input channel, then concatenates the results together. The output has in_channels * channel_multiplier channels.

The above architecture provide nonlinearities and preservation of dimension that helps to improve the robustness of the network and control over fitting.

4.2.2. Human Count Detection Network Output

From the input image model first extracts feature maps, overlays them with a W × H grid and at each cell computes K pre-computed bounding boxes called anchors. Each bounding box has the following:

- Four scalars (x, y, w, h)
  - A confidence score (Pr(Object) ×IOU)
  - C conditional class probability
- The current model architecture has a fixed output of *W×H×K(4+1+C)*. Where,
  - W, H = Grid size
  - K = Number of anchor boxes
  - C = Number of classes for which detection is required
- Based on the above description, the model has a total of 12600 output values. It is derived from following:
  - 20 × 15 grid with 7 anchor boxes per grid

    6 values per anchor box. It consists of:
    - 4 bounding box coordinates (x, y, w, h)
    - 1 class probability
    - 1 confidence score

  As a result, there is a total of 15 × 20 × 7 × 6 = 12600 output values.

If your images are smaller, it is recommended to stretch them to default size. You can also up-sample them beforehand. Smaller image size, due to its sparser grid, may not produce accurate detections.

If your images are bigger and you are not satisfied with the results of the default image size, you can try using a denser grid, as details might get lost during the downscaling.

.

4.2.3. Training Code Overview

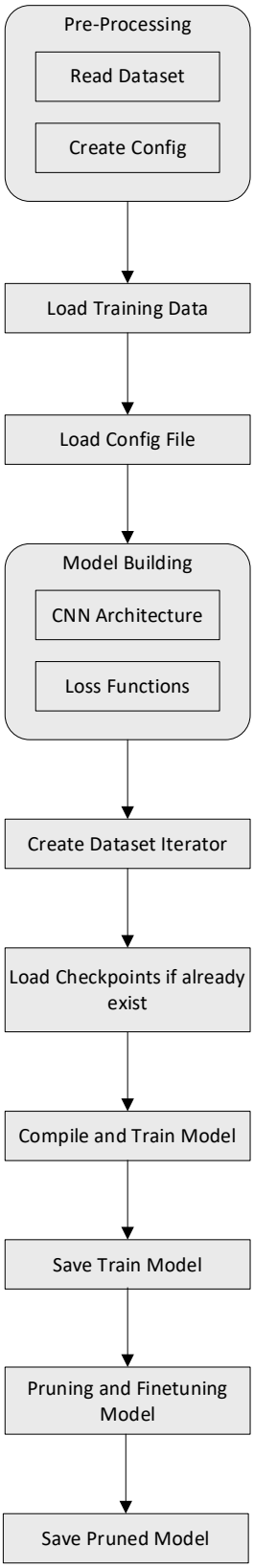

Figure 4.2. Training Code Flow Diagram

Training code can be divided into below parts:

- Model configuration
- Model building
- Model freezing
- Data preparation
- Training
- Pruning

Details of each can be found in subsequent sections.

4.2.3.1. Model Configuration

Demo uses Kitti dataset and SqueezeDet model. kitti_squeezeDet_config.py maintains all the configurable parameters for the model. Below is summary of configurable parameters:

- Training Object class
  - Configure class name here for which you want to train model.

```
cfg.CLASS_NAMES = ['person']
```

Figure 4.3. Code Snippet – Class Name

- Image size
  - Change cfg.IMAGE_WIDTH and cfg.IMAGE_HEIGHT to configure Image size (width and height) in main/config/create_config.py
  - You can also pass flag –gray to train model with 1 channel.

```
# image properties
cfg.IMAGE_WIDTH = 640
cfg.IMAGE_HEIGHT = 480
if gray:
    cfg.N_CHANNELS = 1
else:
    cfg.N_CHANNELS = 3
```

Figure 4.4. Code Snippet – Input Image Size Config

- Grid Size
  - Since there are 5 pooling layers grid dimension would be H = 15 and W = 20. Update it based on anchors per grid size changes

```
cfg.ANCHORS_HEIGHT = 15
cfg.ANCHORS_WIDTH = 20
```

Figure 4.5. Code Snippet – Anchors Per Grid Config #1 (grid sizes)

  - To run the network on your own dataset, adjust the anchor sizes. Anchors are kind of prior distribution over what shapes your boxes should have. The better these fit to the true distribution of boxes, the training becomes faster and easier.
  - To determine anchor shapes, first load all ground truth boxes and pictures. If your images are not of the same size, adjust their height and width by the images' height and width. All images are normalized before being fed to the network. Do the same to the bounding boxes and consequently, the anchors.
  - Second, perform a clustering on these normalized boxes. (I.e. you can just use k-means without feature whitening and determine the number of clusters either by eyeballing or by using the elbow method.)
  - Check for boxes that extend beyond the image or have a zero to negative width or height

```
div_scale_h = 2.0 * (224 / cfg.IMAGE_HEIGHT)
div_scale_w = 2.0 * (224 / cfg.IMAGE_WIDTH)

# cfg.ANCHOR_SEED = np.array([[int(368. / div_scale_w), int(368. / div_scale_h)],
#                             [int(276. / div_scale_w), int(276. / div_scale_h)],
#                             [int(184. / div_scale_w), int(184. / div_scale_h)],
#                             [int(138. / div_scale_w), int(138. / div_scale_h)],
#                             [int(92. / div_scale_w), int(92. / div_scale_h)],
#                             [int(69. / div_scale_w), int(69. / div_scale_h)],
#                             [int(46. / div_scale_w), int(46. / div_scale_h)]])

cfg.ANCHOR_SEED = np.array([[281, 278],
                            [194, 219],
                            [144, 178],
                            [108, 143],
                            [160, 90],
                            [82, 107],
                            [62, 74],
                            ])
```

Figure 4.6. Code Snippet – Anchors per Grid Config #3

- Code Snippet showed above is Configuring Anchor boxes as per input image size.
- If training with user data is not showing accurate results user can find out optimal Anchor Boxes Size for Respective dataset with steps mentioned in section **3.3.2**.

- Training parameters
  - Other training related parameters like Batch size, learning rate, loss parameters and different thresholds can be configured.

```
# batch sizes
cfg.BATCH_SIZE = 20
cfg.VISUALIZATION_BATCH_SIZE = 20

# SGD + Momentum parameters
cfg.WEIGHT_DECAY = 0.0001
cfg.LEARNING_RATE = 0.005
cfg.MAX_GRAD_NORM = 1.0
cfg.MOMENTUM = 0.9

# coefficients of loss function
cfg.LOSS_COEF_BBOX = 5.0
cfg.LOSS_COEF_CONF_POS = 75.0
cfg.LOSS_COEF_CONF_NEG = 100.0
cfg.LOSS_COEF_CLASS = 1.0

# thresholds for evaluation
cfg.NMS_THRESH = 0.4
cfg.PROB_THRESH = 0.005
cfg.TOP_N_DETECTION = 10
cfg.IOU_THRESHOLD = 0.5
cfg.FINAL_THRESHOLD = 0.0
```

Figure 4.7. Code Snippet – Training Parameters

4.2.3.2. Model Building

SqueezeDet class constructor builds model which can be divided in below sections:

- Forward graph :
  - File Path : main/model/SqueezeDet.py -> _create_model()
  - CNN architecture consist of Convolution, Batch normalization, Relu, Maxpool and 1x1 depthwise convolution layers
  - Default Forward graph consists of 6 fire layers as described in below image.
  - The length of network is generated based on argument depth, which consist number of filters for each layer.
    Note: Minimum length supported is 6, Maximum length supported is 10

```
self.input_layer = Input(shape=(self.config.IMAGE_HEIGHT, self.config.IMAGE_WIDTH, self.config.N_CHANNELS),
                         name="input")
prev_layer = self.input_layer
if self.useconv3:
    self.fire1 = self._fire_layer(name="fire1", input=self.input_layer, out_channels=self.depth[0], stdd=0.012,
                                  wt_quant=True, act_quant=True, bn_eps=1e-3, pool=self.pooling[0])
else:
    self.fire1 = self._mobile_layer(name="fire1", input=prev_layer, out_channels=self.depth[0], stdd=0.012,
                                    wt_quant=True, act_quant=True, bn_eps=1e-3, pool=self.pooling[0],
                                    pool2=True)
prev_layer = self.fire1
for layer_count in range(len(self.depth) - 1):
    if layer_count == 0:
        pool2 = True
    else:
        pool2 = False
    prev_layer = self._mobile_layer(name="fire{}".format(str(layer_count + 2)), input=prev_layer,
                                    out_channels=self.depth[layer_count + 1],
                                    stdd=0.012, wt_quant=True, act_quant=True,
                                    bn_eps=1e-3, pool=self.pooling[layer_count + 1], pool2=pool2)
```

Figure 4.8. Code Snippet – Forward Graph Fire Layers

Note that layers have depth wise 2D Convolution.

```
num_output = self.config.ANCHOR_PER_GRID * (self.config.CLASSES + 1 + 4)
self.preds = Conv2D(
    name='conv12', filters=num_output, kernel_size=(3, 3), strides=(1, 1), activation=None, padding="SAME",
    use_bias=False, kernel_initializer=TruncatedNormal(stddev=0.01),
    kernel_constraint=bo.MyConstraints('quant_' + 'conv12'))(prev_layer)
```

Figure 4.9. Code Snippet – Forward Graph Last Convolution Layer

- Interpretation graph
  This block interprets output from network and extracts predicted class probability, predicated confidence scores, and bounding box values.
  The output of the convnet is a 15 × 20 × 42 tensor - there are 42 channels of data for each of the cells in the grid that is overlaid on the image and contains the bounding boxes and class predictions. This means the 42 channels are not stored consecutively, but are scattered all over the place and needed to be sorted. Figure 4.10 and Figure 4.11 explain the details.

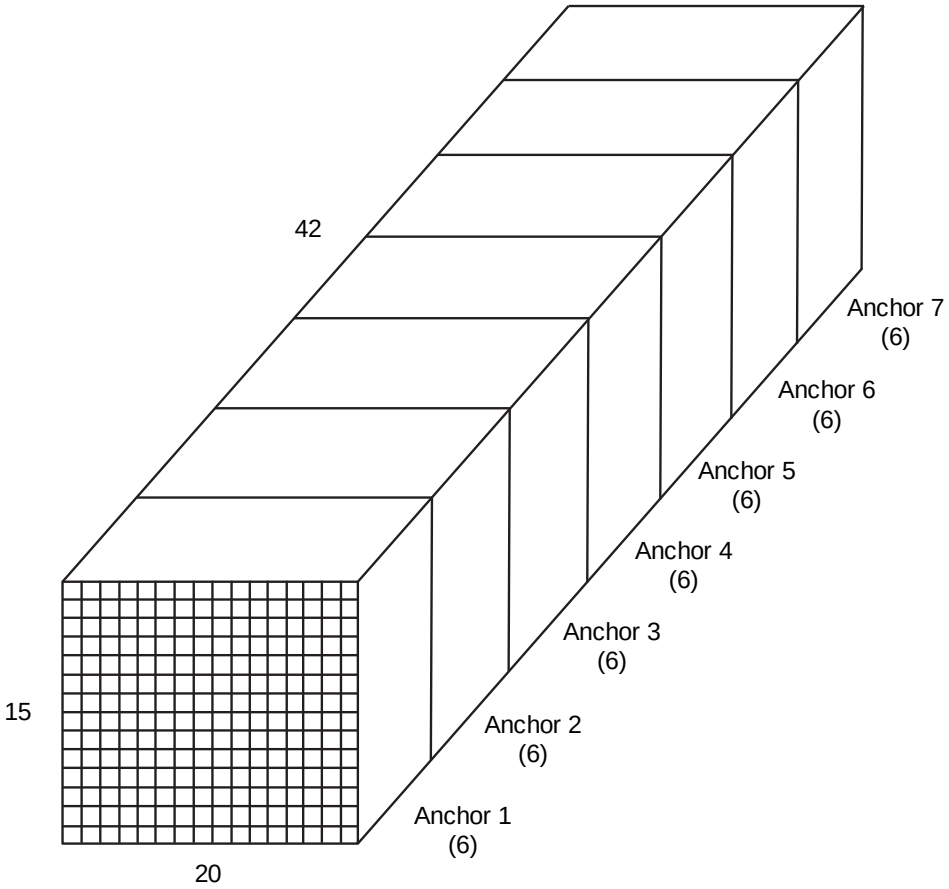

Figure 4.10. Code Snippet: Input Image Size Configuration

For each grid cell values are aligned like below diagram:

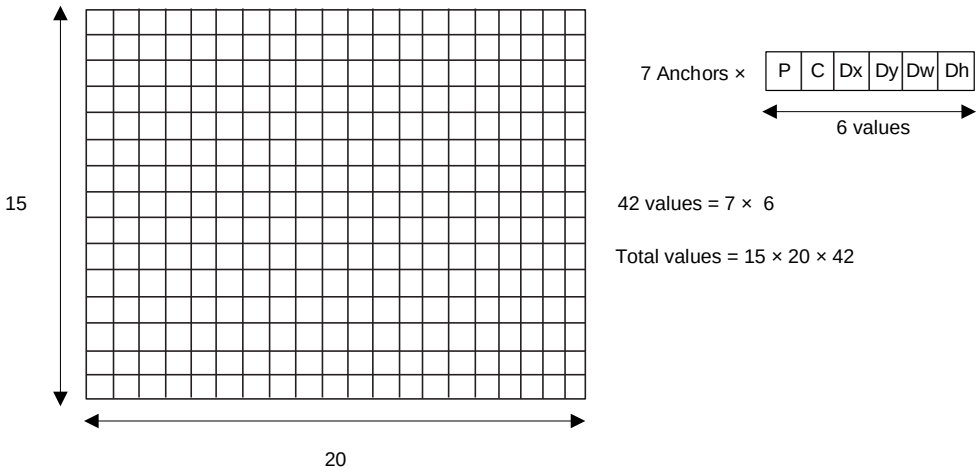

Figure 4.11. Grid Output Visualization #2

As shown in Figure 4.11, the output from conv12 layer (4D array of batch size × 15 × 20 × 42) needs to be sliced with proper index to get all values of probability, confidence, and coordinates.
The code snippet interpret output is main/utils/utils.py -> slice_predictions().

```
# slice pred tensor to extract class pred scores and then normalize them
pred_class_probs = K.reshape(
    K.softmax(
        K.reshape(
            y_pred[:, :, :, :num_class_probs],
            [-1, config.CLASSES]
        )
    ),
    [config.BATCH_SIZE, config.ANCHORS, config.CLASSES],
)
# number of confidence scores, one for each anchor + class probs
num_confidence_scores = config.ANCHOR_PER_GRID + num_class_probs

# slice the confidence scores and put them trough a sigmoid for probabilities
pred_conf = K.sigmoid(
    K.reshape(
        y_pred[:, :, :, num_class_probs:num_confidence_scores],
        [config.BATCH_SIZE, config.ANCHORS]
    )
)
# slice remaining bounding box_deltas
pred_box_delta = K.reshape(
    y_pred[:, :, :, num_confidence_scores:],
    [config.BATCH_SIZE, config.ANCHORS, 4]
)
```

Figure 4.12. Code Snippet – Interpret Output Graph

For confidence score, this must be a number between 0 and 1, so sigmoid is used.
For predicting the class probabilities, there is a vector of NUM_CLASS values at each bounding box. Apply a softmax to make it probability distribution.

- Bboxes – bboxes_from_deltas()
  This block calculates bounding boxes based on anchor box and predicated bounding boxes.
- IOU – tensor_iou()
  This block calculates Intersection over Union for detected bounding boxes and actual bounding boxes.

Loss graph

- File Path: main/model/SqueezeDet.py -> loss()
- This block calculates different types of losses which need to be minimized. To learn detection, localization and classification, model defines a multi-task loss function. There are three types of losses which are considered for calculation:
  - Bounding box

    This loss is regression of the scalars for the anchors

```
# bounding box loss
bbox_loss = (K.sum(mc.LOSS_COEF_BBOX * K.square(input_mask * (pred_box_delta - box_delta_input))) / num_objects)
```

Figure 4.13. Code Snippet – Bbox Loss

- Confidence score

To obtain a meaningful confidence score, each box's predicted value is regressed against the Intersection over Union of the real and the predicted box. During training, compare ground truth bounding boxes with all anchors and assign them to the anchors that have the largest overlap (IOU) with each of them.
The reason being, to select the *closest* anchor to match the ground truth box such that the transformation needed is reduced to minimum. Equation evaluates to 1 if the kth anchor at position-(i, j) has the largest overlap with a ground truth box, and to 0 if no ground truth is assigned to it. This way, you only include the loss generated by the *responsible* anchors.
As there can be multiple objects per image, normalize the loss by dividing it by the number of objects.

```
# confidence score loss
conf_loss = K.mean(
    K.sum(
        K.square((ious - pred_conf))
        * (input_mask * mc.LOSS_COEF_CONF_POS / num_objects
           + (1 - input_mask) * mc.LOSS_COEF_CONF_NEG / (mc.ANCHORS - num_objects)),
        axis=[1]
    ),
)
```

Figure 4.14. Code Snippet – Confidence Loss

- Class
  The last part of the loss function is just cross-entropy loss for classification for each box to do classification, as you would for image classification.

```
# compute class loss,add a small value into log to prevent blowing up
class_loss = K.sum(labels * (-K.log(pred_class_probs + mc.EPSILON))
                   + (1 - labels) * (-K.log(1 - pred_class_probs + mc.EPSILON))
                   * input_mask * mc.LOSS_COEF_CLASS) / num_objects
```

Figure 4.15. Code Snippet – Class Loss

In one model architecture, you obtain the bounding box prediction, the classification, as well as, the confidence score.

4.2.3.3. Training

- Training Data Generator
  - *main/model/datageberator.py* takes care of reading dataset and creates iterator that feeds data to the model in batch size given.

```
# create train generator
train_generator = generator_from_data_path(self.img_names, self.gt_names, config=self.cfg)
val_generator = generator_from_data_path(self.val_img_names, self.val_gt_names, config=self.cfg)
```

Figure 4.16. Code Snippet – Dataset Iterator

  - Data generator scales image pixel values from [0, 255] to [0, 2] as shown in below code. Also, it converts image to gray scale if model is trained with –gray flag.
  - Current human count training code uses mean = 0 and scale = 1/128 (0.0078125) in pre-processing step.

```
if gray:
    img = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY)
    orig_h, orig_w = [float(v) for v in img.shape]
else:
    orig_h, orig_w, _ = [float(v) for v in img.shape]

img /= 128.0
```

Figure 4.17. Code Snippet – Scale Image

- Training Callbacks
  - Reduce learning Rate on Plateau:

```
# if flag was given, add reducelronplateu callback
if self.REDUCELRONPLATEAU:
    reduce_lr = ReduceLROnPlateau(monitor='loss', factor=0.1, verbose=1,
                                  patience=4, min_lr=0.0)
    cb.append(reduce_lr)
```

Figure 4.18. Code Snippet – Reduce Learning rate on plateau

  - Save Checkpoint

```
# add a checkpoint saver
ckp_saver = ModelCheckpoint(self.checkpoint_dir + "/model.{epoch:02d}-{loss:.2f}.hdf5", monitor='loss',
                            verbose=0,
                            save_best_only=False,
                            save_weights_only=True, mode='auto', period=1)
cb.append(ckp_saver)
```

Figure 4.19. Code Snippet – Save

4.2.3.4. Transfer Learning and Freezing some layers

- Transfer Learning

  You can pass the model checkpoint or saved keras model as argument in –init

  Note: The architecture of checkpoint and model should match.

  Checkpoints are also restored if you are using the log directory with existing training.

```
initial_epoch = 0
if self.init_file is not None:
    print("Weights initialized by name from {}".format(self.init_file))
    squeeze.model.load_weights(self.init_file)
    initial_epoch = int(os.path.basename(self.init_file).split('-')[0].split('.')[1])
elif len(os.listdir(self.checkpoint_dir)) > 0:
    ckpt_list = os.listdir(self.checkpoint_dir)
    sorted_list = sorted(ckpt_list, key=self.ckpt_sorting)
    ckpt_path = os.path.join(self.checkpoint_dir, sorted_list[-1])
    squeeze.model.load_weights(ckpt_path)
    print("Weights initialized by name from {}".format(ckpt_path))
    initial_epoch = self.ckpt_sorting(sorted_list[-1])
```

Figure 4.20. Code Snippet – Transfer Learning

- Freezing Layers

  If you are using pre-trained checkpoint and you want to freeze model up to some layer, you can provide flag freeze_landmark subset of the layer name. For example, *--freeze_landmark=fire5*.

```
if self.freeze_landmark is not None:
    # Freezing some of the layers as per user input
    model_layer_names = []
    max_matched_index = -1
    for layer in squeeze.model.layers:
        model_layer_names.append(layer.name)
        if self.freeze_landmark in model_layer_names[-1]:
            layerindex = squeeze.model.layers.index(layer)
            if layerindex > max_matched_index:
                max_matched_index = layerindex
    for layer in squeeze.model.layers[:max_matched_index]:
        layer.trainable = False
    print("layers frozen till {}".format(squeeze.model.layers[max_matched_index].name))
```

Figure 4.21. Code Snippet – Freeze Layers

4.3. Pruning

Pruning takes place in two phases:

- Pruning – As part of pruning, the code determines the channels with the lowest impact on accuracy and remove those channels.
- Fine Tuning – Create model with optimized number of channels, restore weights, and fine tune model.

```
for layer in squeeze.model.layers:
    if 'fire1' not in layer.name and re.match(r"fire(\d+)_PW", layer.name):
        sparsity_value = self.sparsity.pop(0) if len(self.sparsity) > 0 else 0.25
        pruning_schedule = sparsity.PolynomialDecay(
            initial_sparsity=0.01, final_sparsity=float(sparsity_value),
            begin_step=0, end_step=end_step, frequency=100)
        pruned_model.add(sparsity.prune_low_magnitude(
            layer,
            pruning_schedule,
            block_size=(1, 1)
        ))
    else:
        pruned_model.add(layer)
```

Figure 4.22. Code Snippet – Set Layer Sparsity

The code snippet shown in Figure 4.22 sets sparsity to prune the layer. Based on the sparsity you set, the pruning is implemented.

Note: There is no pruning in the first fire block, so sparsity is not set to first mobile block.

```
for i, w in enumerate(final_model.get_weights()):
    if 'fire1' in final_model.weights[i].name:
        continue
    if re.match(r".*\S(fire(\d+)_PW\Skernel)", final_model.weights[i].name):
        weights = np.transpose(w[0][0])
        non_pruned_channels = []
        for j in range(weights.shape[0]):
            if np.count_nonzero(weights[j]) != 0:
                non_pruned_channels.append(j)
        non_pruned_channels_length = len(non_pruned_channels)
        if non_pruned_channels_length % 4 != 0:
            non_pruned_channels_length += (4 - (non_pruned_channels_length % 4))
        non_pruned_depths.append(non_pruned_channels_length)
        l_name = final_model.weights[i].name.split('/')[0]
        non_pruned_layers[l_name] = non_pruned_channels
fine_model = transfer_weights(self.cfg, final_model, non_pruned_depths, self.EarlyPooling)
```

Figure 4.23. Code Snippet – Determine Pruned Channels

Figure 4.23 determines which channels to prune or not. Based on the number of non-pruned depths, the new model is created and weights are transferred.

Fine-tuning is performed on the new model, and the final model is saved.

4.4. Finding the Optimal Model

In the default code, the default depth of network is six. However, you can change the depth with respect to their use case.

For example, if you need faster inference, you can decrease depth of network. However, it drops the accuracy.

If you need accuracy, increase the depth. However, this increases inference time.

The relation between network depth, accuracy, and inference FPS is shown Figure 4.24.

Note: The models have first layer as normal convolution and rest of the layers as depthwise and 1 × 1 convolution.

Table 4.2. Model Performance Data with Models with Dw 1 × 1 Conv

Layer Length	Map	FPS	Count
6 Fire Layer	58%	31.5	4907286
7 Fire Layer	63%	28	5200822

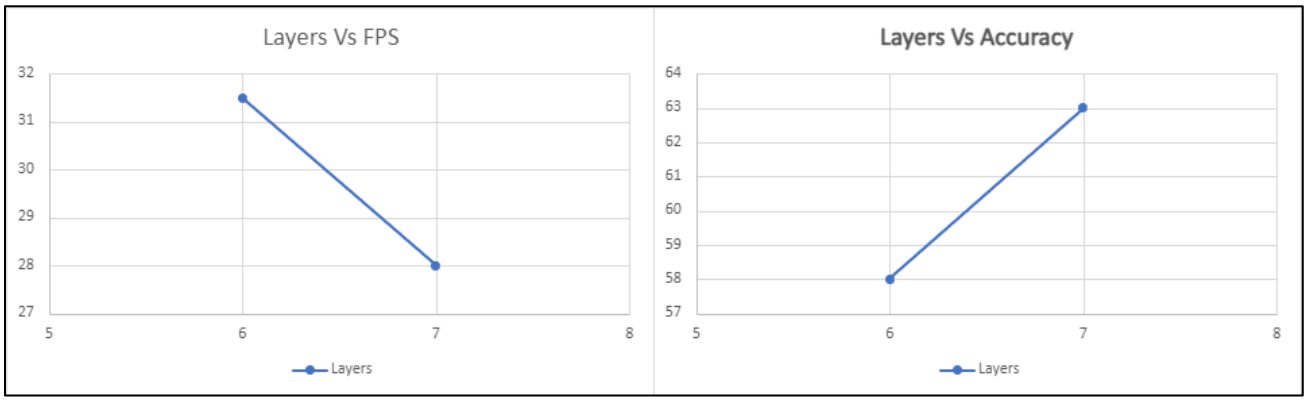

Figure 4.24. Relation Between # Layers Versus Accuracy Versus FPS

Table 4.3 shows the dataset augmentation's impact on model accuracy.

Table 4.3. Model Performance Data with Dataset Augmentation

Dataset Augmentation	Accuracy on test set
D1: Canvas Shift	55%
D2: D1 + Brightness augmentation	58%
D3: D2 + Contrast augmentation	60%

Note: The Fps data contains information of model with only one pooling in one fire layer. The released model contains two pooling in fire1 and fire2 to increase FPS.

4.5. Training from Scratch and/or Transfer Learning

To train the machine:

1. Modify the training script. The training script *@train.sh* is used to trigger training. Figure 4.25 shows the input parameters to be configured.

```
python train_and_prune.py --dataset_path='/home/user/datasets/vga-dataset' \
                --logdir='logs' \
                --val_set_size=50 \
                --validation_freq=5 \
                --gray \
                --early_pooling \
                --filterdepths='32,32,64,64,88,128' \
                --sparsity='0.25,0.25,0.35,0.35,0.45' \
                --epochs='80,5,50' \
                --gpuid=0 \
                --configfile='squeezedet.config' \
                # --runpruning \
                # --usecov3 \
                # --usedefaultvalset \
                # --freeze_landmark='fire1' \
                # --init=<check point path>
```

Figure 4.25. Training Input Parameter

Some of the options you can provide during training are the following:

- --dataset_path – Dataset directory path. /home/dataset/vga_dataset is example.
- --logdir – log directory where checkpoint files are generated while model is training.
- --val_set_size – Validation split percentage.
- --validation_freq – Validation frequency in terms of number of epochs.
- --gray – Add flag to train model with grayscale images.
- --early_pooling – Add flag to use early-pooling.
- --filterdepths – comma separated list of number of features for each layer. You can use depth length of 5 to 10 (default value is 7).
- --sparsity – List of fraction to prune layer channels.
  Note: The first fire layer is not pruned, so length of sparsity list should be one less than length of filter depths.
- --epochs – Comma separated epoch list for training, pruning and fine-tuning.
- –gpuid – If the system has more than one gpu, it indicates the one to use.
- --configfile – Config file name. If file exist in logdir, the code reuses it. Otherwise, it creates a new file.
- --runpruning – Add flag to run pruning after training is completed.
- --usecov3 – Use Normal convolution as first layer instead of depthwise 1x1 conv layer.
- --usedefaultvalset – Add this flag if you want to reuse validation set images from val.txt. If flag is not present, the code creates a new validation set.
- --freeze_landmark – If you want to freeze some layers in the network, input the subpart or layer name as an argument and the code freezes weights up to the layer name mentioned in this argument. For example, *Fire1*, *Fire6,* and so on.
- --init – If you want to specify pretrained model to load weights.

2. Execute the train.sh script which starts training.

```
./train.sh
Creating Direcory: /home/user/vga/training/logs
Creating Direcory: /home/user/vga/training/logs/data
Creating Direcory: /home/user/vga/training/logs/config
Number of images: 48295
Number of epochs: 1
Number of batches: 2414
Batch size: 20
2414/2414 [==============] - 2586s 1s/step - loss: 1.7831 - loss_without_regularization: 1.7831
- bbox_loss: 1.1773 - class_loss: -1.1921e-07 - conf_loss: 0.6058
```

Figure 4.26. Execute training Script

```
./train.sh
Creating Direcory: /home/user/vga/training/logs
Creating Direcory: /home/user/vga/training/logs/data
Creating Direcory: /home/user/vga/training/logs/config
Number of images: 48295
Number of epochs: 1
Number of batches: 2414
Batch size: 20
Weights initialized by name from /home/user/vga/training/logs/train/checkpoins/model.10-0.80.hdf5
```

Figure 4.27. Execute training with transfer learning

```
./train.sh
Creating Direcory: /home/user/vga/training/logs
Creating Direcory: /home/user/vga/training/logs/data
Creating Direcory: /home/user/vga/training/logs/config
Number of images: 48295
Number of epochs: 1
Number of batches: 2414
Batch size: 20
Weights initialized by name from /home/user/vga/training/logs/train/checkpoins/model.10-0.80.hdf5
layer frozen till fire5_PW
```

Figure 4.28. Execute training with transfer learning + frozen layers

Note: If model is not converging in pruning user can reduce sparsity and try again.

3. Start TensorBoard.

```
$ tensorboard –logdir=<log directory of training>
```

For example: tensorboard –logdir=’./logs/train/tensorboard’

4. Open the local host port on your web browser.

```
$ tensorboard --logdir log/train/tensorboard/
TensorBoard 1.15.0 at http://gpu-server:6006/ (Press CTRL+C to quit)
```

Figure 4.29. TensorBoard – Generated Link

5. Check the training status on TensorBoard

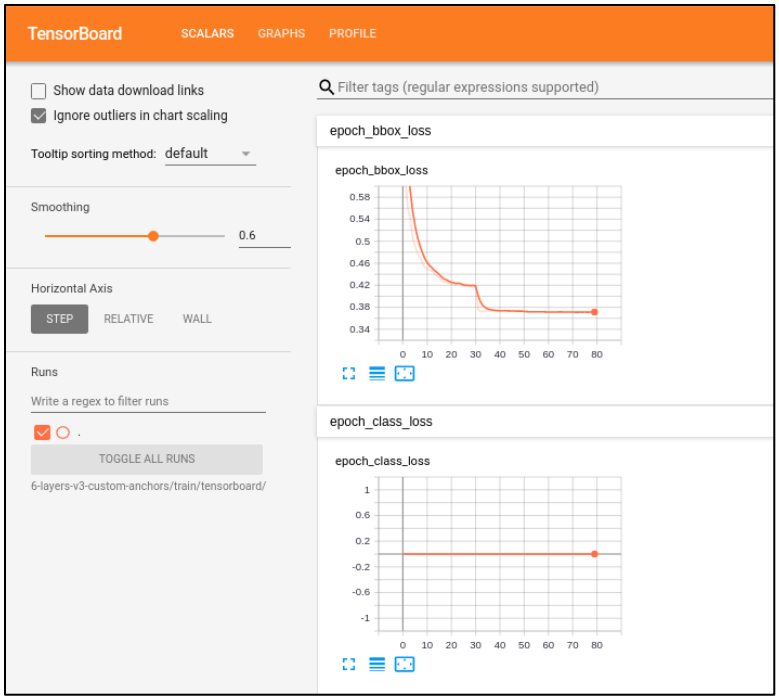

Figure 4.30. TensorBoard

Figure 4.30 shows the backbone graph.

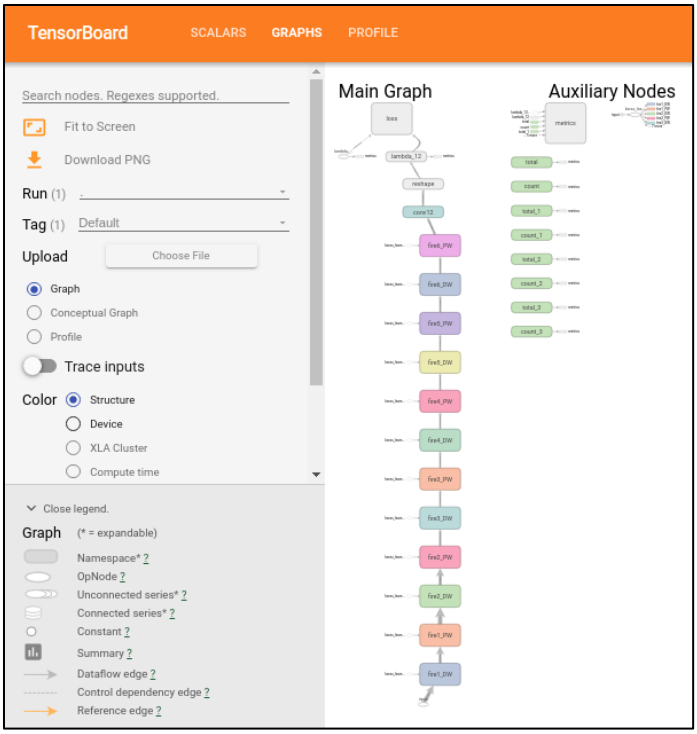

Figure 4.31. Backbone Graph

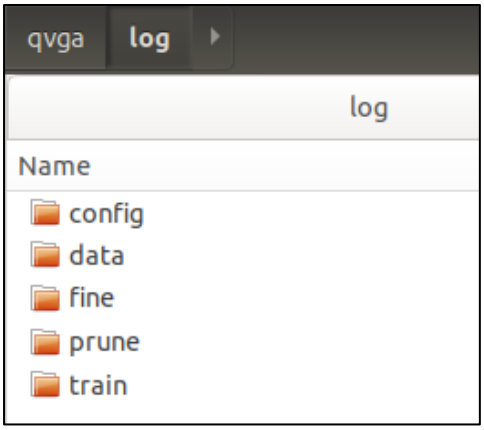

Figure 4.32. Example of Files at Log Folder

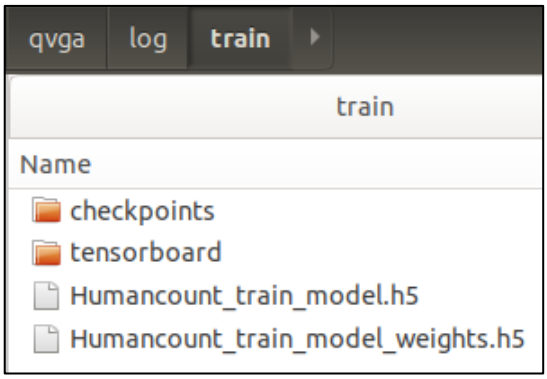

Figure 4.33. Example of Checkpoint and Trained Model

5. Evaluating the Model

This section describes the procedure to calculate model performance in terms of mAP.

5.1. Converting Keras Model to TensorFlow File

The VGA code contains the *keras2tf.py* file under the *keras-to-tf-converter* directory as shown in Figure 5.1.

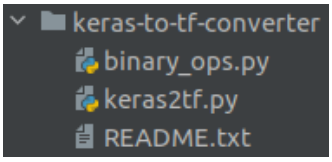

Figure 5.1. Keras to tf converter Directory

Note: If you perform any quantization change in training code *binary_ops.py*, you need to replicate those changes in *binary_ops.py*.

Run the command below to generate .pb file in the same path of the h5 file.

```
$ python keras2tf.py –kerasmodel <h5 model path>
```

The script saves the .pb file in the directory assigned to kerasmodel argument.

5.2. Running Inference on Test Set

The VGA code contains *vga-inference.py* under inference directory as shown in Figure 5.2.

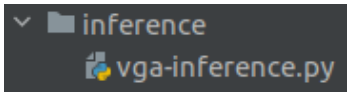

Figure 5.2. Inference Directory

Note: If you performed any changes in the training code regarding image size, number of anchors, or grid size, you need to replicate those changes in the inference script.

Run the command below to run inference on the test set.

```
$ python vga-inference.py –pb <converted pb path> --input_image <test set images path>
```

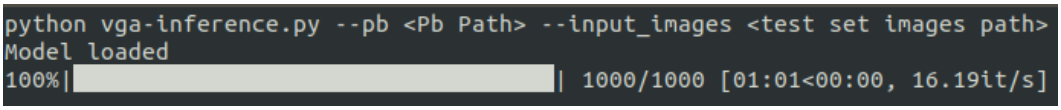

Figure 5.3. Run Inference

The command above saves the images with bbox drawn in *inference_output/image_output* and resultant kitti output in inference_output/predictions as shown in Figure 5.4.

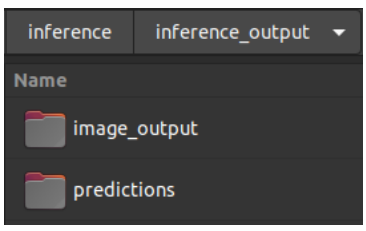

Figure 5.4. Inference Output

5.3. Calculating mAP

The *VGA* code contains the *main.py* file under the mAP directory as shown in Figure 5.5.

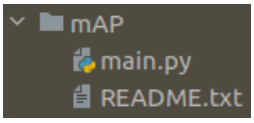

Figure 5.5. mAP Directory Structure

Run the command below to calculate mAP using the predictions generated from inference and groundtruth from the test set.

```
$ python main.py –input_images <input test set images path> --ground_truth
<input test set labels path> --predictions <path to prediction generated from
inference> --no-animation –no-plot
```

```
mAP$ python main.py --input_images <input test images path> \
                    --ground_truth <input testset groundtruth path> \
                    --predictions <pridictions path> \
                    --no-animation  --no-plot
XX.XX% = person AP
mAP = XX.XX%
```

Figure 5.6. mAP Calculation

After successfully running the script, it shows the mAP for each class and the total mAP.

6. Creating Binary File with Lattice sensAI

This chapter describes how to generate binary file using the Lattice sensAI version 4.0 program.

Figure 6.1. sensAI Home Screen

To create the project in sensAI tool:

1. Click File > New.
2. Enter the following settings:
   - Project name
   - Framework– Keras
   - Class–CNN
   - Device– CrossLink-NX Voice and Vision Machine Learning (VVML) Board, Rev B
   - 'Compact Mode' should be unchecked.
3. Click **Network File** and select the network (h5) file.

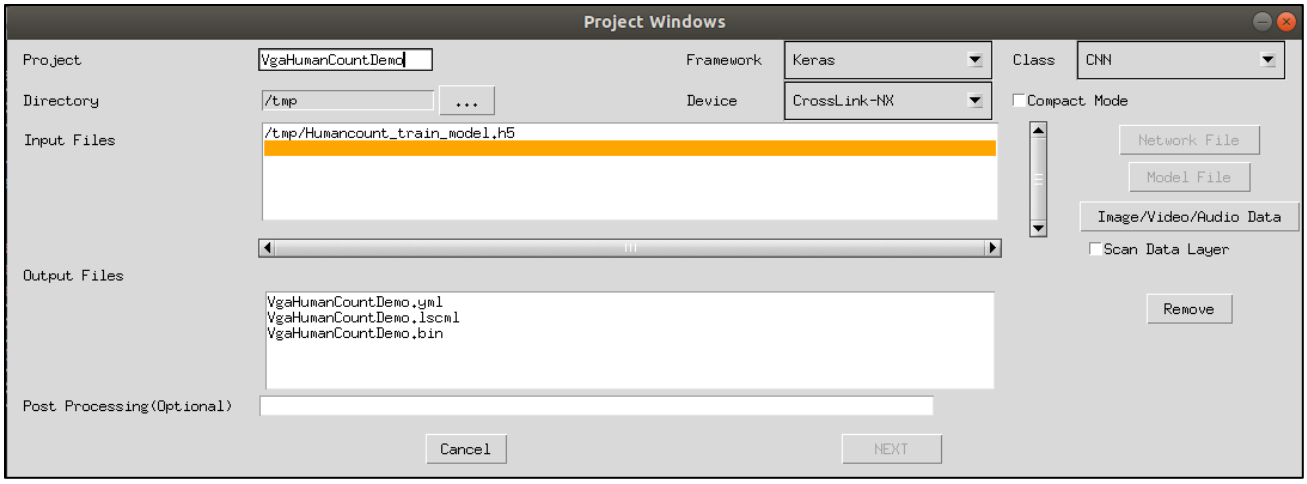

Figure 6.2. sensAI –Network File Selection

4. Click **Image/Video/Audio Data** and select the image input file.

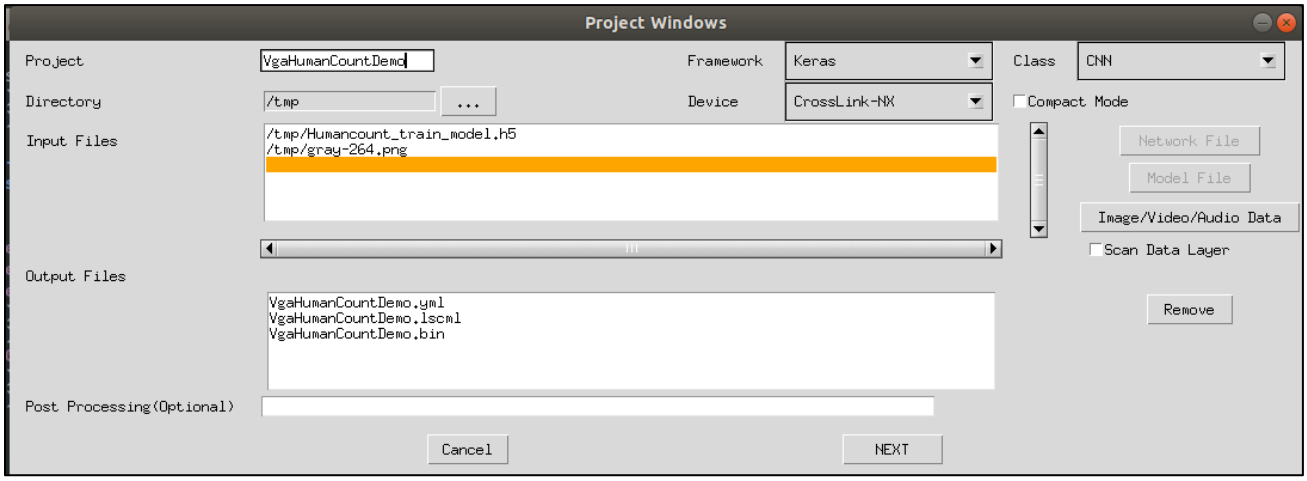

Figure 6.3.sensAI –Image Data File Selection

5. Click **NEXT**.
6. Configure your project settings.

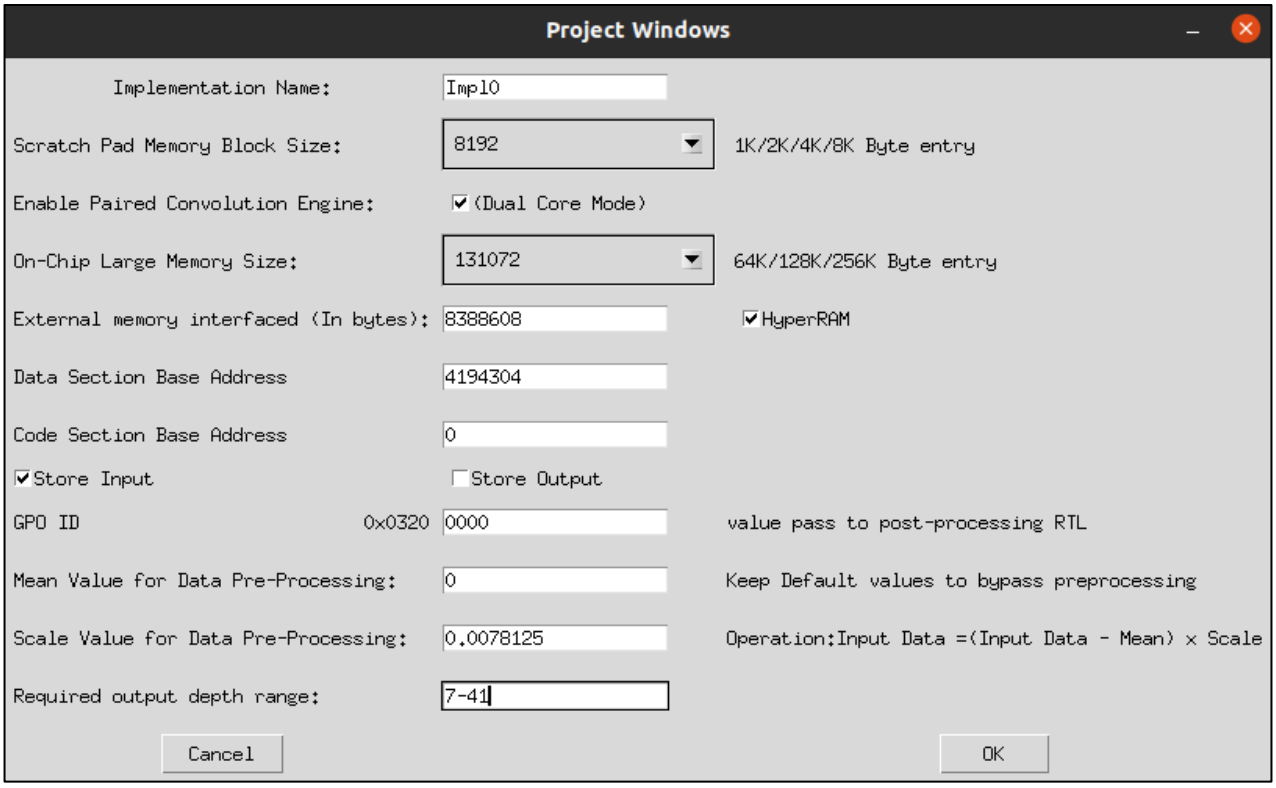

Figure 6.4. sensAI – Project Settings

7. Scratch pad memory block size and data section base address should match the FPGA RTL code.
8. Set depth range. With this setting, the class probability values are not part of the output values.
9. Click **OK** to create the project.
10. Double-click **Analyze**.

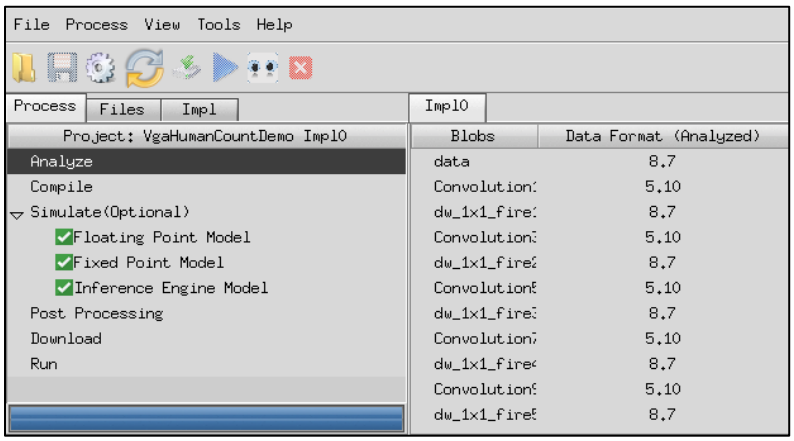

Figure 6.5. sensAI – Analyze Project

The tool generates the Q format for each layer based on range analysis, which is based on input image and other parameters.

11. Confirm the Q format of each layer as shown in Figure 6.6 and update the fractional bit for each layer by double clicking on the values against each layer one by one.

Blobs	Data Format (Analyzed)	Stored Data Format	Internal(External) Mem Bytes
data	8.7	1.7	8960 (307200)
Convolution1	5.10	5.10	None
dw_1x1_fire1_fire1	8.7	1.7	61440 (614400)
Convolution3	5.10	5.10	None
dw_1x1_fire2_fire2	8.7	1.7	20480 (38400)
Convolution5	5.10	5.10	None
dw_1x1_fire3_fire3	8.7	1.7	9600 (N/A)
Convolution7	5.10	5.10	None
dw_1x1_fire4_fire4	8.7	1.7	9600 (N/A)
Convolution9	5.10	5.10	None
dw_1x1_fire5_fire5	8.7	1.7	13200 (N/A)
Convolution11	5.10	5.10	None
dw_1x1_fire6_fire6	8.7	1.7	19200 (N/A)
conv12/Conv2D	5.10	5.10	None
CBSR_conv12/Conv2D	8.7	5.10	25200 (N/A)

Figure 6.6. Q Format Settings for Each Layer

12. After changing the fractional bit, double click on **Analyze** again.
13. Double-click **Compile** to generate the Firmware file.

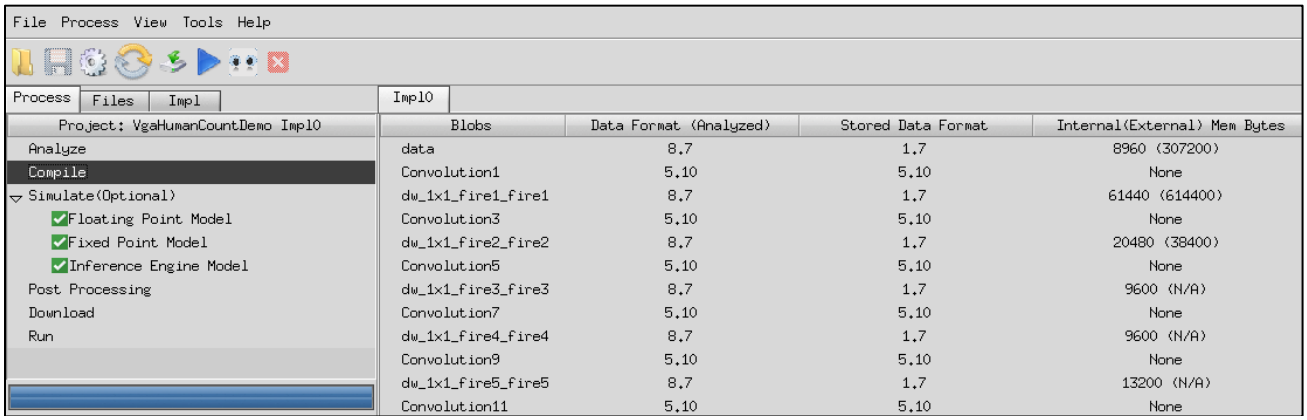

Figure 6.7. Compile Project

7. Hardware Implementation

7.1. Top Level Information

7.1.1. Block Diagram

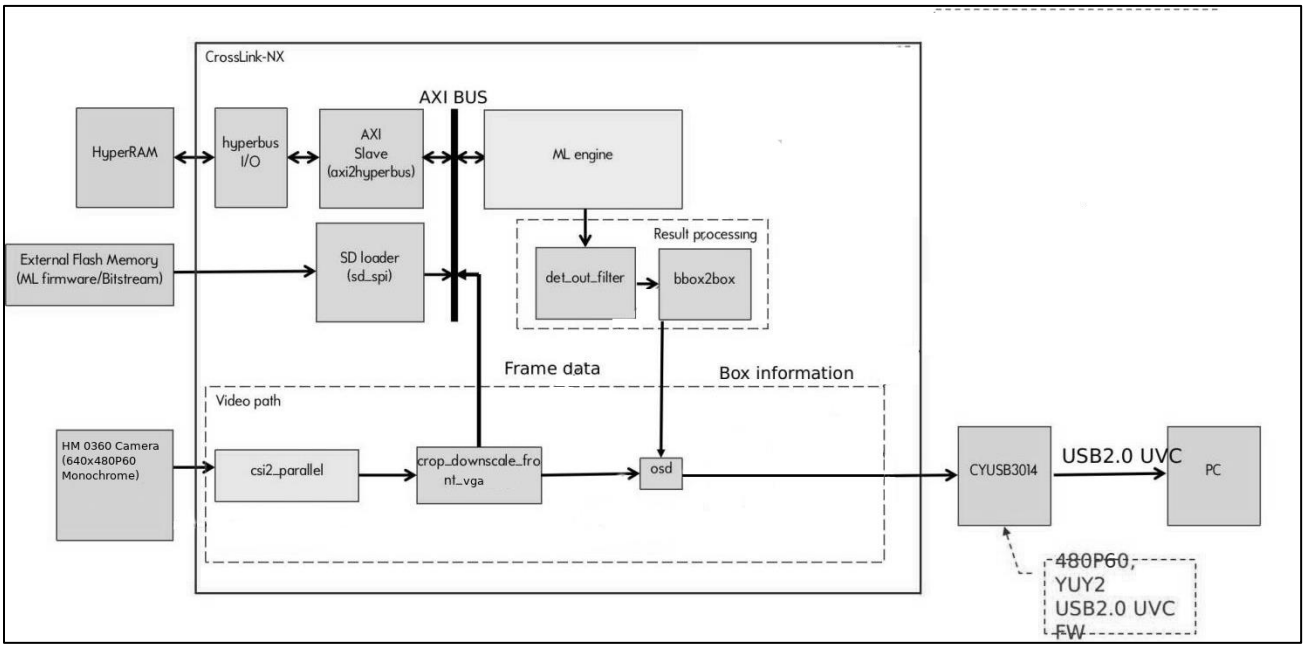

Figure 7.1. RTL Top Level Block Diagram

7.1.2. Operational Flow

This section provides an overview of the data flow across the CrossLink-NX VVML Board, Rev B board.

- The CNN module is configured with the help of a binary (BIN) file stored in a SD card. The .bin file is a command sequence code which is generated by the Lattice Machine Learning software tool.
- The command code is written in hyperRAM through AXI before the execution of CNN Accelerator IP Core starts. CNN reads command code from hyperRAM during its execution and does calculation with it per command code. Intermediate data may be transferred from/to hyperRAM per command code.
- The RAW8 data from *csi2_to_parallel* module is stored in 640 × 480 image resolution in *crop_downscale_front_vga* module FIFO. This data is written into hyperRAM memory through *rpc2_ctrl_controller* through the *axi_ws2m* AXI interface module.
- After command code and input data are available, CNN Accelerator IP Core starts calculation at the rising edge of start signal.
- Output data of CNN is passed to *det_out_filter* for post processing. *det_out_filter* generates bounding box co-ordinates X, Y, W, H associated with top 5 confidence value indexes for 640x480 image resolution.
- These co-ordinates are passed to *osd_back_vga_human_count* for resizing them to fit the actual image resolution on PC.

7.1.3. Core Customization

Table 7.1. Core Parameter

Constant	Default (Decimal)	Description
OVLP_TH_2X	5	Intersection Over Union Threshold (NMS)
NUM_FRAC	10	Fraction Part Width in Q-Format representation.
EN_INF_TIME	0	Enable Timing measurement logic By default, it is zero and the memory file used is *human_count.memI.* If assigned 1, timing measurement is enabled and the memory file used is *human_count_INF.mem*. To configure the respective memory file, follow the steps below: 1. Open *dpram8192x8_human_count.ipx* from File List in Lattice Radiant. 2. Click on *Browse Memory File* from Initialization section. 3. Update mem file path: • For 0 – /src/jedi_common/human_count.mem • For 1 – /src/jedi_common/human_count_INF.mem
INF_MULT_FAC	15907	Inference time multiplying factor calculated as per CNN clock frequency and using Q-Format (Q1.31). CNN clock frequency = 135 MHz Hence, CNN clock period = 1/(135 × 10-6) µs = 0.000007407 ms Now, Q1.31 = 0.000007407 × 231 = ~15907
FLASH_START_ADDR	24'h300000	SPI Flash Read Start address (keep same address in programmer while loading Firmware file) For example, for current start address, programmer address should be 0x00300000.
FLASH_END_ADDR	24'h400000	SPI Flash Read End address (keep same address in programmer while loading firmware file). The address must be in multiple of 512 Bytes. For example, for current end address, programmer address should be: 0x00400000.
Constant Parameters (Not to be modified)		
NUM_ANCHOR	2100	Number of reference bounding boxes for all grids
NUM_GRID	300	Total number of Grids (X × Y)
NUM_X_GRID	20	Number of X Grids
NUM_Y_GRID	15	Number of Y Grids
PIC_WIDTH	640	Picture Pixel Width (CNN Input)
PIC_HEIGHT	480	Picture Pixel Height (CNN Input)
TOP_N_DET	10	Number of top confidence bounding boxes detection
HYPERRAM_BASEADDR	4194304	Indicates hyperRAM starting base address location value. This should match in sensAI compiler while generating firmware.
RAW8_OFFSET	0	Indicates hyperRAM starting address location value to store RAW8

7.1.4. Architecture Details

7.1.4.1. SPI Flash Operation

RTL module spi_loader_spram provides SPI Flash read operation and writes that data into HyperRAM through the AXI interface. It reads from SPI Flash as soon as board gets powered up and .bit and .bin files are loaded in expected addresses.

- Expected Address for BIT File (Programmer) – 0x0000000 - 0x00100000
- Expected Address for Firmware File (Programmer) – FLASH_START_ADDR - FLASH_END_ADDR

Typical sequence of SPI Read commands for SPI Flash MX25L12833F is implemented using FSM in RTL as per the flow of operation below.

- After FPGA Reset, RELEASE FROM DEEP POWER DOWN command (0xAB) is passed to SPI Flash memory. Then RTL waits for 500 clock cycle for SPI flash to come into Standby mode if it is in Deep Power Down mode.
- RTL sends FAST READ command code (0x0B) on SPI MOSI signal for indication of Read Operation to SPI Flash.
- RTL sends three bytes of address on SPI MOSI channel, which determines the location in SPI flash from where the data needs to be read.
- This SPI Flash has eight dummy cycles as wait duration before read data appears on MISO channel. After waiting for eight dummy cycles, the RTL code starts reading data.
- This read sequence is shown in Figure 7.2. The SPI Interface Signal Mapping with RTL signals are as follows:
  - CS (Chip Select) => SPI_CSS
  - SCLK (Clock) => SPI_CLK
  - SI (Slave In) => SPI_MOSI
  - SI (Slave Out) => SPI_MISO
- The Read Data on MISO signal is stored in a FIFO in RTL, which then reads the data in multiples of 512 bytes. After 512 Bytes Chip Select is deasserted, the AXI FSM state is activated.

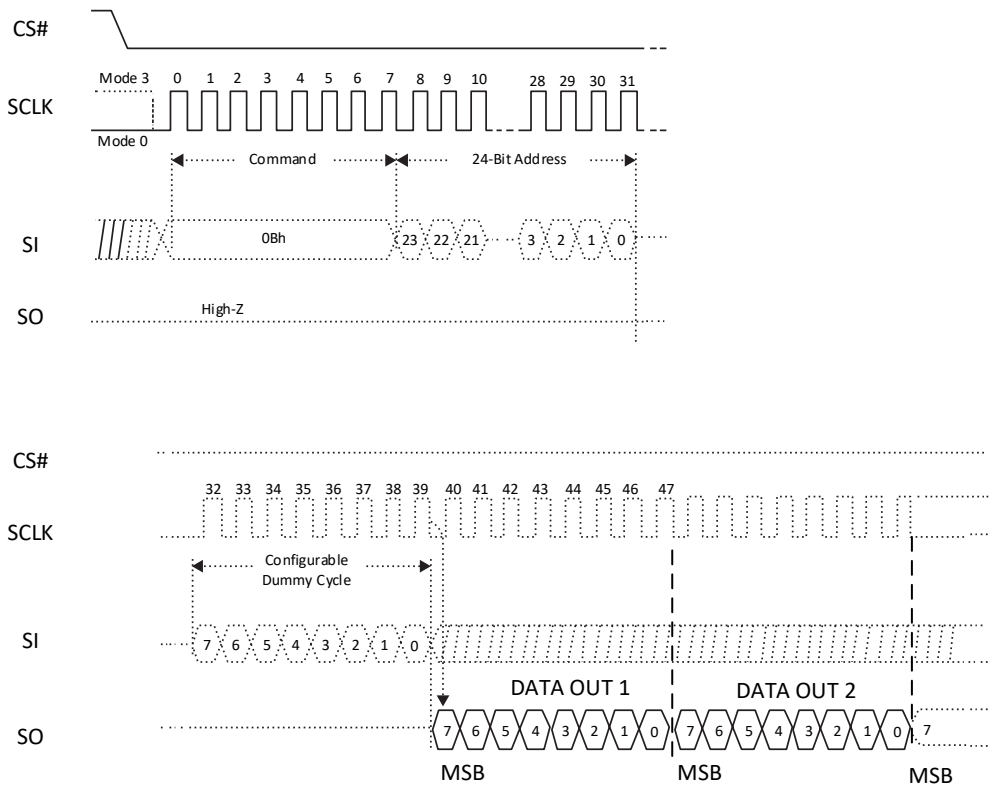

Figure 7.2. SPI Read Command Sequence

- AXI logic reads the data from FIFO in burst of four on AXI write channel, with each burst having 128 bytes.

- In accessing HyperRAM, the *axi_ws2m* module is used as a Muxing module among multiple input slave AXI interfaces as shown in Figure 7.4. The *spi_loader_spram* module is considered as SLAVE 0 and given priority to write into HyperRAM. The Master interface connects to the *axi2_hyperbus* module, which provides output interface for accessing HyperRAM.
- After writing into HyperRAM, the 512 bytes are fetched from the SPI Flash using same command sequence as explained above until the FLASH_END_ADDR is reached.

7.1.5. Pre-processing CNN

The output from *csi2_to_parallel* module is a stream of RAW8 data that reflects the camera image, which is given to *crop_downscale_front_vga* module.

The *crop_downscale_front_vga* module processes that image data and generates input of 640 × 480 image data interface for CNN IP.

7.1.5.1. Pre-processing Flow:

RAW8 data values for each pixel are fed serially line by line for an image frame. This accumulated value is written into Line Buffer. Line Buffer is a FIFO. Data from FIFO to be read depends on Pixel Count, which is calculated below.

In below example, FIFO Read cycles are calculated by two factors such as the Incoming Horizontal Pixels and the Number of bytes read in one cycle. FIFO Read cycle can be written in the equation as (Incoming Horizontal Pixels/Number of bytes read in one cycle)

FIFO Read Cycles = 640/80 = 80

Pixel Count >= Incoming Horizontal Pixels – ((FIFO Read Cycles × Pixel Clock Frequency)/Read Clock Frequency from True Dual Port RAM)

For example: Pixel Count >= 640 – ((80 × 24)/81) ~= 616

Data from Memory is read and stored in HyperRAM for CNN input through rpc2_ctrl_controller, through the axi_w2sm module which acts as an AXI interface to write data from slave (crop_downscale_front_vga) to master (axi2_hyperbus). This process is described in the next section.

7.1.6. HyperRAM Operations

The CrossLink-NX VVML Board, Rev B board uses external HyperRAM for faster data transfer mechanism among the internal blocks and enhances the system performance. The *crop_downscale_front_vga* module uses HyperRAM to store the image data.

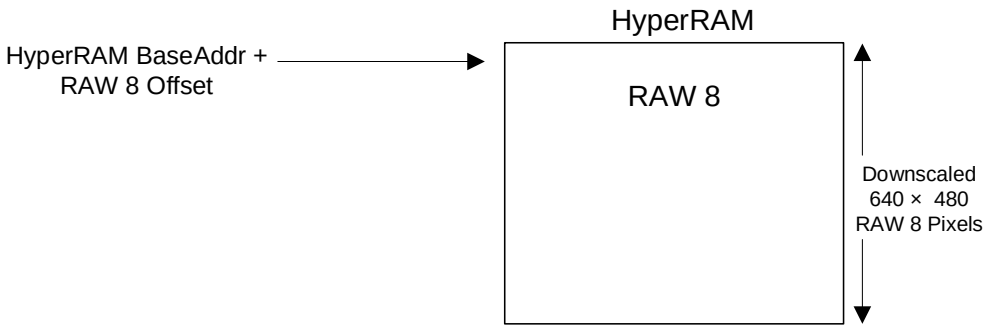

Figure 7.3. HyperRAM Memory Addressing

Primarily, crop_downscale_front_vga module stores 640 values of RAW8 into a local FIFO for all 480 horizontal blocks. Later this stored data is written to HyperRAM through AXI write data channel.

Figure 7.3 shows, when final data is written out, 640x480 RAW8 pixels are stored into HyperRAM starting from HyperRAM Base address location.

The 640 × 480 pixel values stored in HyperRAM are then obtained by CNN engine after getting command sequence through AXI interface.

For the crop_downscale_front_vga module to access HyperRAM for above explained operations, axi_ws2m module functions as a Muxing module for multiple input slave AXI interfaces as shown in block diagram Figure 7.4.

For the internal blocks to access HyperRAM, axi_ws2m considers spi_loader module as SLAVE 0, cnn_opt module as SLAVE 1, crop_downscale_front_vga module as SLAVE2 and the MASTER connects these slaves to axi2_hyperbus module.

The priority to select write channel is given to respectively spi_loader slave, cnn_opt slave and then crop-downscale slave. Whenever valid address is available from the respective Slave on its write address channel, that slave is given access of master channel if other priority slaves are not accessing it.

Thus, when valid write address is obtained from crop_downscale_front_vga module, access is given to Slave 2 to use HyperRAM.

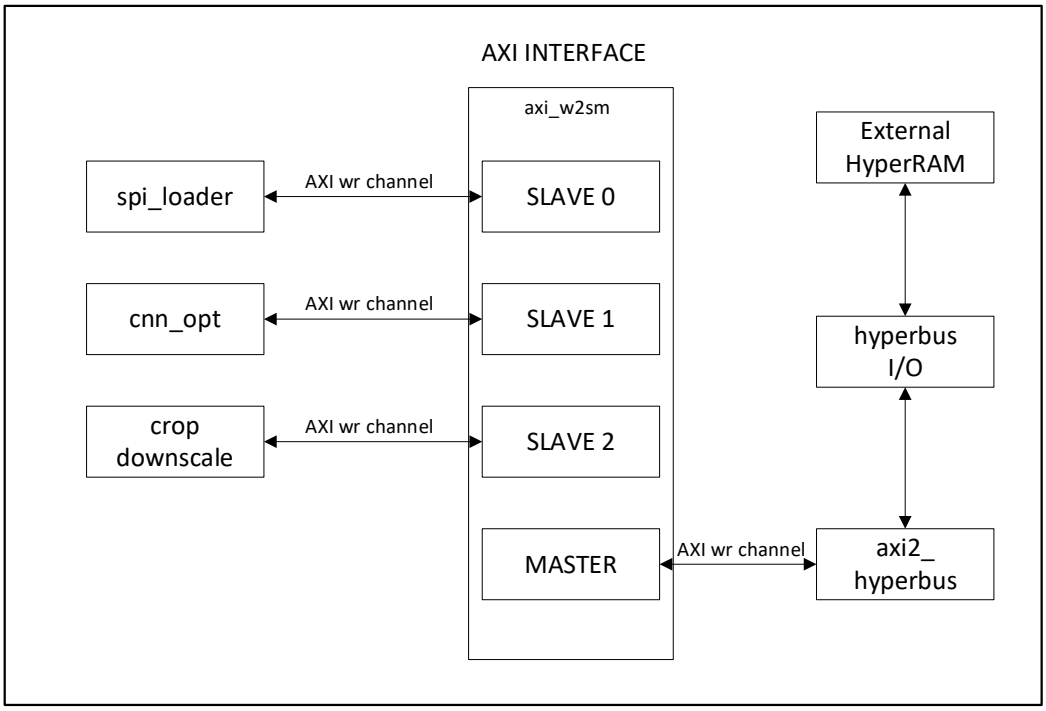

Figure 7.4. HyperRAM Access Block Diagram

7.1.7. Post Processing CNN

CNN provides total of 10500 [2100 × 5 (C, X, Y, W, H)] values which are given to the *det_out_filter* module. The CNN output data consists of the following parameters.

Table 7.2. Data Parameters of CNN Output

Parameter	Description
C	This parameter indicates the confidence of detected object class. For each grid cell (20 × 15), one confidence value (16 bit) for each anchor box (7) is provided making total values of confidence 20 × 15 × 7 = 2100 from CNN output.
X	This parameter indicates the relative X coordinate to transform the anchor box into a predicted bounding box for detected object. For each grid cell, one relative X value (16 bit) for each anchor box is provided making total values of 20 × 15 × 7 = 2100 for X from CNN output.
Y	This parameter indicates the relative Y coordinate to transform the anchor box into a predicted bounding box for detected object. For each grid cell, one relative Y value (16 bit) for each anchor box is provided making total values of 20 × 15 × 7 = 2100 for Y from CNN output.
W	This parameter indicates the relative W (Width) coordinate to transform the anchor box into a predicted bounding box for detected object. For each grid cell, one relative W value (16 bit) for each anchor box is provided making total values of 20 × 15 × 7 = 2100 for W from CNN output.
H	This parameter indicates the relative H (Height) coordinate to transform the anchor box into a predicted bounding box for detected object. For each grid cell, one relative H value (16-bit) for each anchor box is provided making total values of 20 × 15 × 7 = 2100 for H from CNN output.

Figure 7.5 shows the format of CNN output.

OutputData	C				X	Y	W	H	X	Y	W	H
Index No	0-299	300-599	...	1800-2099	2100-2399	2400-2699	2700-2999	3000-3299	3300-3599	3600-3899	3900-4199	4200-4499
Grid No	0-299	0-299		0-299	0-299	0-299	0-299	0-299	0-299	0-299	0-299	0-299
Anchor No	1	2	...	7	1				2			

Figure 7.5. CNN Output Data Format

The primary functionality of the *det_out_filter* module is to capture the CNN valid output and modifying it to make it work with the osd_back_vga_human_count module.

The *det_out_filter* module contains two sub-modules: *det_sort_conf* and *det_st_bbox*.

- 2100 values of confidence are passed to det_sort_conf module. It sorts out top 10 highest confidence values and stores their indexes. Index values are passed to *det_st_bbox* modules.
- 2100 × 4 values of coordinates are passed to *det_st_bbox* module. It calculates the bounding box coordinates, performs NMS and provides valid box bitmap.

The *osd_back_vga_human_count* module contains logic for post processing

- The *draw_box_simple* module calculates the box coordinates for 640 × 480 coordinates.
- The *lsc_osd_text* module generates character bitmap for showing text on the display.

7.1.7.1. Confidence Sorting

- All input confidence values (2100) are compared with threshold parameter CONF_THRESH value. Confidence values which are greater than threshold are considered as valid for sorting.
- The *det_sort_conf* module implements an anchor counter (0–2099), which increments on each confidence value. It provides the index of confidence value given by the CNN output.
- Two memory arrays are generated in this module: (1) Sorted top 10 (TOP_N_DET) confidence value array, and (2) sorted top 10 confidence index array.
- For sorting, a standard sorting algorithm is followed. As input confidence values start arriving, each value is compared with stored/initial value at each location of the confidence value array.
- If the input value is greater than stored/initial value on any array location and lesser than stored/initial value of previous array location, the input value is updated on current array location. The previously stored value of current location is shifted into the next array location.
- Refer to Figure 7.6 for sorting of new value of confidence into existing confidence value array. The calculated confidence index (anchor count value) is also updated in the confidence index array along with confidence value array.

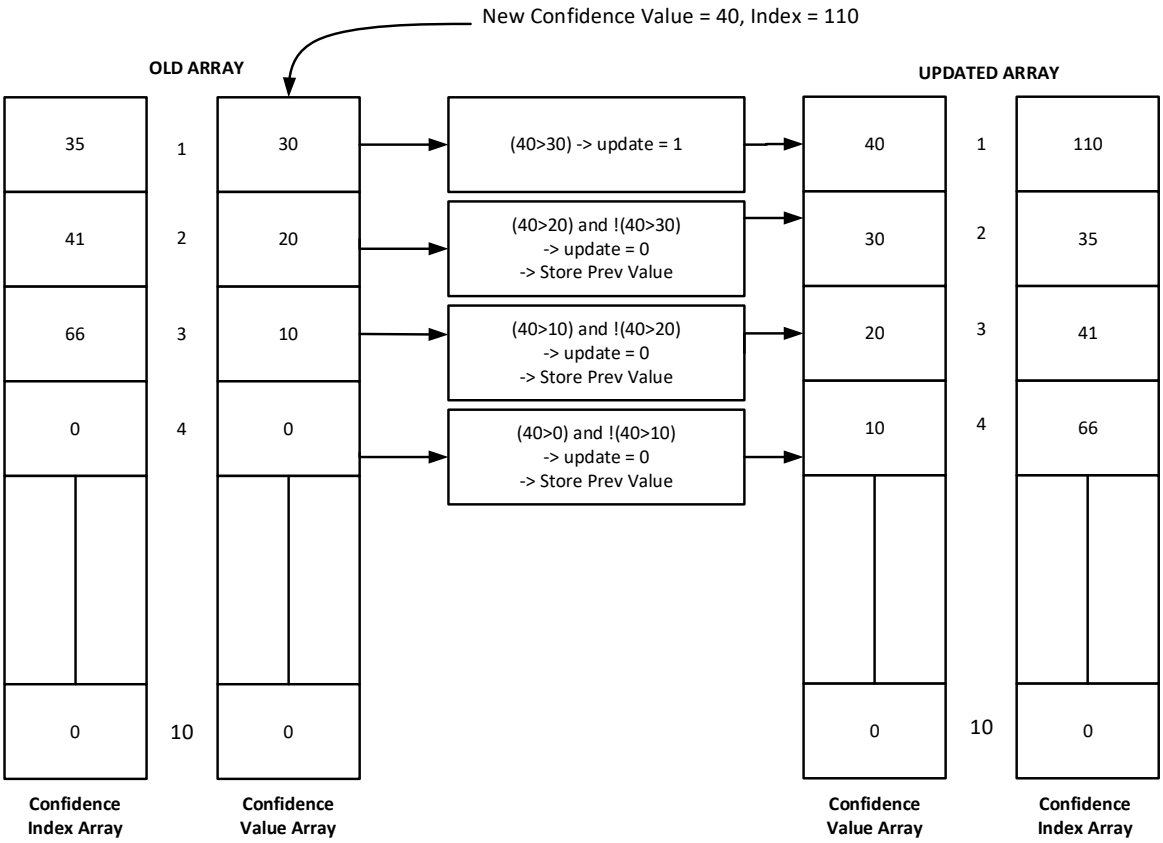

Figure 7.6. Confidence Sorting

- This process is followed for all 2100 confidence values. This module provides 10 indexes (*o_idx_00 to o_idx_09*) as output along with the count of valid indexes (*o_num_conf*). *o_idx_00* contains highest confidence value index and *o_idx_09* contains lowest confidence value index.

7.1.7.2. Bounding Box Calculation

The Neural Network for Object Detection is trained with seven reference boxes of pre-selected shapes having constant W (Width) and H (Height). These reference boxes are typically referred as anchors.

Table 7.3. Pre-Selected Width and Height of Anchor Boxes

Anchor No.	1	2	3	4	5	6	7
W × H (pixel)	281x278	194x219	144x178	108x143	160x90	82x107	62x74

Anchors are centered around 20 × 15 grid cells of image. Therefore, each grid center has above seven anchors with pre-selected shape. 20 × 15 are the number of grid centers along horizontal and vertical directions. The grid center (X, Y) pixel values are shown in Table 7.3.

Table 7.4. Grid Center Values (X, Y) for Anchor Boxes

Grid No.	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
X (pixel)	30	61	91	122	152	183	213	244	274	305	335	366	396	427	457	488	518	549	579	610
Y (pixel)	30	60	90	120	150	180	210	240	270	300	330	360	390	420	450					

CNN provides total 2100 (20 × 15 × 7) values of each relative coordinates X, Y, W, and H to transform the fixed size anchor into a predicted bounding box. Input X, Y, W, and H values associated with top 10 sorted confidence indexes are used for box calculation in the *det_st_bbox* module.

Each anchor is transformed to its new position and shape using the relative coordinates as shown in Logic 1.

```
LOGIC 1
X′ = X coordinate of Predicted Box
X = Grid Center X according to Grid number
W = Width of Anchor according to Anchor number
DeltaX = Relative coordinate for X (CNN output)
X′ = X + W × DeltaX
Y′ = Y + H × DeltaY
W′= W × DeltaW
H′ = H × DeltaH
```

The box coordinates are passed to the *bbox2box* module in *jedi_human_count_top.v* after NMS process.

NMS is implemented to make sure that in object detection, a particular object is identified only once. It filters out the overlapping boxes using OVLP_TH_2X value.

NMS process is started when the CNN output data is completely received.

- The process starts from the box having the highest confidence coordinates: 0^{th} location in X, Y, W, H array.
- These coordinates are compared against the second highest confidence coordinates: First location in X, Y, W, H array. From this comparison, Intersection and Union coordinates are found.
- From these coordinates, Intersection and Union area are calculated between highest confidence box and the second highest confidence box as shown in Figure 7.7.

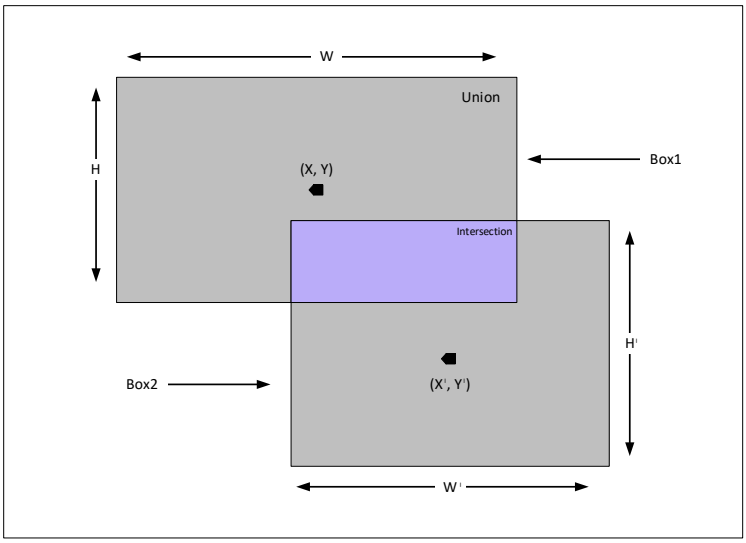

Figure 7.7. Intersection-Union Area NMS

- If *Intersection Area × (OVLP_TH_2X/2) > Union Area*, the box with lower confidence value is blocked in the final output.
- This NMS calculation is performed between all the combinations of two boxes.
- After all combinations are checked, the output array *o_bbox_bmap* contains boxes, which are correctly overlapped or non-overlapped. *o_out_en* provides valid pulse for *bbox2box* for further processing on these box coordinates.

7.1.7.3. Bounding Box Upscaling

The process of upscaling bounding boxes for 640x480 resolution is accomplished by two different modules bbox2box and draw_box_simple.

Initially bbox2box module in jedi_human_count_top.v obtains box coordinate outputs from det_out_filter.

Considering (X, Y) as center of the Box of Width W and Height H it calculates extreme ends of the Box (X1, X2 & Y1, Y2) for 640x480 resolution. It also clamps the coordinate values so that the box remains out of masking area. This is shown in logic 2.

```
LOGIC 2
X1 = If ((X'- W'/2) < 0)       => 0       else (X'- W'/2)
Y1 = If ((Y'- H'/2) < 0)        => 0       else (Y'- H'/2)
X2 = If ((X'+ W'/2) > 640) => 640 else (X'+ W'/2)
Y2 = If ((Y'+ H'/2) > 480) =>  480 else (Y'+ H'/2)
```

The final calculated X1, X2, Y1 & Y2 values for all the boxes in bbox2box are then sent to draw_box_simple module through *osd_back_vga_human_count* module

Pixel counter and Line counter keeps track of pixels of each line and Lines of each frame. Outer boundary of the box and Inner boundary of the box are calculated when Pixel and Line counter reaches to co-ordinates (X1, X2) and (Y1, Y2) respectively. Calculations are done as per logic 3.

```
LOGIC 3
Outer Box = (Pixel Count >= (X1 - 1)) & (Pixel Count <= (X2 + 1)) &
                        (Line Count >= (Y1 - 1)) & (Line Count <= (Y2 + 1))
Inner Box = (Pixel Count > (X1 + 1)) & (Pixel Count < (X2 - 1)) &
                        (Line Count > (Y1 + 1)) & (Line Count < (Y2 - 1))
```

Each Bounding Box is calculated by removing the intersecting area of the outer and the inner box. Box is only displayed if Box-Bitmap for that box is set to 1 (From det_st_bbox through bbox2box module). Box on calculations are as done as logic 4.

```
LOGIC 4
Box_on[1] = Outer Box[1] & ~Inner Box[1] & Box-Bitmap[1]
Box_on[2] = Outer Box[2] & ~Inner Box[2] & Box-Bitmap[2]
.
.
Box_on[20] = Outer Box[20] & ~Inner Box[20] & Box-Bitmap[20]
```

o_box_obj signal is asserted when any of the above Box_on signal is set which is then connected to green_on signal and processed for Bounding Box display in the output.

7.1.7.4. OSD Text Display

lsc_osd_text module provides bitmap of each ASCII character to be displayed with specified position on screen. It takes count of detected Humans.

It sets an output signal (text_on) when Text is to be displayed on the output screen through USB. When text_on is set, YCbCr value for that pixel location is assigned FF, 7F, 7F respectively values (White color) and sent to USB output instead of original pixel value.

7.1.7.5. USB Wrapper

The module Wrapper_USB3 is used to transmit 16 Bit data to the output 16 Bit interface every clock cycle.

This module takes input data in YCbCr 24 Bit format and gives output as 16 Bit YCb and YCr format. This module does not change or regenerate input timing parameters.

7.1.7.6. Inference Time Calculation

The time taken by a trained neural network model to infer/predict outputs after obtaining input data is called inference time. The process of this calculation is explained as follows.

Following logic is added in crop_downscale_vga_front.v

The inference time is calculated by implementing a counter to store the count of CNN engine cycles per frame.

When signal i_rd_rdy (for example, o_rd_rdy coming from CNN engine) is high, the CNN engine indicates that it is ready to get input and when it is low, the engine indicates that it is busy.

When i_rd_rdy signal is low, the CNN counter begins and stops when the i_rd_rdy signal goes high again indicating that previous execution is over and the CNN is ready for new input.

As shown below in Figure 7.8 when rdy_h2l (ready high-to-low) pulse is asserted, the CNN Up-counter starts from 1 and the count value increases till i_rd_rdy is not high again. The count value is stored in (count).

Similarly when rdy_l2h (ready low-to-high) pulse is asserted, the Up-counter stops and the final CNN count value is obtained (cnn_count).

```
assign rdy_l2h          =  i_rd_rdy & ~rdy;
assign rdy_h2l          = ~i_rd_rdy & rdy;

// Counter to calculate individual frame inference time
assign count_c          = (o_rd_done & rdy_h2l) ? 27'd1 : (i_rd_rdy) ? count : (count + 27'd1);

// Store individual cnn frame counter
assign cnn_count_c      =  rdy_l2h ? count : cnn_count;
```

Figure 7.8. CNN Counter Design

The methodology used to obtain stable inference time is to calculate inference time per frame and obtain the average inference time value after 16 CNN frames are over as discussed below.

After completion of every frame, the new count value (cnn_count) obtained as explained above is added to the previous value and stored in (cnn_adder).

A frame counter keeps monitoring the frame count and after 16 frames when the frame count is done, this cnn_adder value is reset as shown in Figure 7.9.

```
// Frame counter to calculate CNN frames upto 16 (0 - 15)
assign frame_counter_c   = (rdy_l2h_rr)? (frame_counter + 4'd1) : frame_counter;

// keep adding indiviual cnn frame counter for 16 frames. Then clear to 0
assign cnn_adder_c       = (count_done)? 31'd0 : (rdy_l2h_r) ? (cnn_adder + {4'd0,cnn_count}) : cnn_adder;

// Counter addition done when 1ll 16 cnn frame counter values are added and averaged
assign count_done        = (frame_counter == 4'd15) & (rdy_l2h_rr);
```

Figure 7.9. Frame Counter Design for 16 CNN Frames Average

To get the average inference time value (avg_inf_time_hex) after frame count is done, the final cnn_adder value is divided by 16 as shown in Figure 7.10.

```
// Average Inference Time calculated by dividing by 16
assign inf_time_c        = (count_done)? cnn_adder[30:4] : inf_time;

// 32 Bit average Inference time in Hex
assign avg_inf_time_hex  = {5'd0,inf_time};
```

Figure 7.10. Average Inference Time Calculation

Using Lattice Multiplier library module this average inference time value is multiplied by INF_MULT_FAC, a parameter indicating inference multiplying factor explained in Table 7.1.

The inference time in millisecond (inf_time_ms) is obtained by dividing the output obtained from this multiplier by 2^31 as per the Q-Format, shown in below images.

All the above obtained values, namely, the CNN count, the average inference time, and the inference time in millisecond are passed on to lsc_osd_text_human_count module for getting bitmap to display characters.

```
assign inf_time_ms          = inf_time_mult[46:31];
```

Figure 7.11. Inference Time in Millisecond

7.1.7.7. Inference Time Display Management

This module *lsc_osd_text_human_count.v* mainly consists of a DPRAM, which holds the characters at pre-defined address positions indicated by text_addr and an 8x8 font ROM which provides the bitmap of these characters for PC display.

This module basically functions by using two entities. One is the position of the character where it has to be displayed, and other is by reading the ASCII value of the character to be displayed.

For this purpose, once the CNN count, individual frame inference time and the inference time in millisecond values are obtained, they are converted from hex into ASCII values as shown below.

The average inference time input values (i_avg_inf_time_hex) are converted from hex to ASCII values as shown below. To display 8 characters of this value on PC, this input is stored in respective r_avginfhex_ch. The characters obtained by adding 7'h30 and 7'h37 are shown in Figure 7.12**.**

```
r_avginfhex_ch0 <= (i_avg_inf_time_hex[31:28] > 4'd9)? (i_avg_inf_time_hex[31:28] + 7'h37) : (i_avg_inf_time_hex[31:28] + 7'h30);
r_avginfhex_ch1 <= (i_avg_inf_time_hex[27:24] > 4'd9)? (i_avg_inf_time_hex[27:24] + 7'h37) : (i_avg_inf_time_hex[27:24] + 7'h30);
r_avginfhex_ch2 <= (i_avg_inf_time_hex[23:20] > 4'd9)? (i_avg_inf_time_hex[23:20] + 7'h37) : (i_avg_inf_time_hex[23:20] + 7'h30);
r_avginfhex_ch3 <= (i_avg_inf_time_hex[19:16] > 4'd9)? (i_avg_inf_time_hex[19:16] + 7'h37) : (i_avg_inf_time_hex[19:16] + 7'h30);
r_avginfhex_ch4 <= (i_avg_inf_time_hex[15:12] > 4'd9)? (i_avg_inf_time_hex[15:12] + 7'h37) : (i_avg_inf_time_hex[15:12] + 7'h30);
r_avginfhex_ch5 <= (i_avg_inf_time_hex[11:8]  > 4'd9)? (i_avg_inf_time_hex[11:8]  + 7'h37) : (i_avg_inf_time_hex[11:8]  + 7'h30);
r_avginfhex_ch6 <= (i_avg_inf_time_hex[7:4]   > 4'd9)? (i_avg_inf_time_hex[7:4]   + 7'h37) : (i_avg_inf_time_hex[7:4]   + 7'h30);
r_avginfhex_ch7 <= (i_avg_inf_time_hex[3:0]   > 4'd9)? (i_avg_inf_time_hex[3:0]   + 7'h37) : (i_avg_inf_time_hex[3:0]   + 7'h30);
```

Figure 7.12. Average Inference Time Value to ASCII Conversion

Table 7.5. Signal Values to ASCII Conversion

Characters for Display	Value to be Added to Signal	ASCII HEX Value	ASCII Decimal Value
1	7'h30	31	49
2	7'h30	32	50
3	7'h30	33	51
4	7'h30	34	52
5	7'h30	35	53
6	7'h30	36	54
7	7'h30	37	55
A	7'h37	41	65
B	7'h37	42	66
C	7'h37	43	67
D	7'h37	44	68
E	7'h37	45	69
F	7'h37	46	70

- Similarly, to display eight characters of individual frame inference time, the input signal *i_inf_time_hex* is converted from hex to ASCII and stored in respective r_infhex_ch signal as shown in Figure 7.13.
- In the same way, to display four characters of inference time in ms, the input signal *i_inf_ms* is converted from hex to ASCII and stored in respective r_inf_ms signal as shown below.

```
r_infhex_ch0    <= (i_inf_time_hex[31:28] > 4'd9)? (i_inf_time_hex[31:28] + 7'h37) : (i_inf_time_hex[31:28] + 7'h30);
r_infhex_ch1    <= (i_inf_time_hex[27:24] > 4'd9)? (i_inf_time_hex[27:24] + 7'h37) : (i_inf_time_hex[27:24] + 7'h30);
r_infhex_ch2    <= (i_inf_time_hex[23:20] > 4'd9)? (i_inf_time_hex[23:20] + 7'h37) : (i_inf_time_hex[23:20] + 7'h30);
r_infhex_ch3    <= (i_inf_time_hex[19:16] > 4'd9)? (i_inf_time_hex[19:16] + 7'h37) : (i_inf_time_hex[19:16] + 7'h30);
r_infhex_ch4    <= (i_inf_time_hex[15:12] > 4'd9)? (i_inf_time_hex[15:12] + 7'h37) : (i_inf_time_hex[15:12] + 7'h30);
r_infhex_ch5    <= (i_inf_time_hex[11:8]  > 4'd9)? (i_inf_time_hex[11:8]  + 7'h37) : (i_inf_time_hex[11:8]  + 7'h30);
r_infhex_ch6    <= (i_inf_time_hex[7:4]   > 4'd9)? (i_inf_time_hex[7:4]   + 7'h37) : (i_inf_time_hex[7:4]   + 7'h30);
r_infhex_ch7    <= (i_inf_time_hex[3:0]   > 4'd9)? (i_inf_time_hex[3:0]   + 7'h37) : (i_inf_time_hex[3:0]   + 7'h30);
```

Figure 7.13. CNN Count Values to ASCII Conversion

```
r_infms_ch0     <= (i_inf_time_ms[15:12] > 4'd9)? (i_inf_time_ms[15:12] + 7'h37) : (i_inf_time_ms[15:12] + 7'h30);
r_infms_ch1     <= (i_inf_time_ms[11:8] > 4'd9)? (i_inf_time_ms[11:8] + 7'h37) : (i_inf_time_ms[11:8]  + 7'h30);
r_infms_ch2     <= (i_inf_time_ms[7:4] > 4'd9)? (i_inf_time_ms[7:4] + 7'h37) : (i_inf_time_ms[7:4] + 7'h30);
r_infms_ch3     <= (i_inf_time_ms[3:0] > 4'd9)? (i_inf_time_ms[3:0] + 7'h37) : (i_inf_time_ms[3:0] + 7'h30);
```

Figure 7.14. Inference time in millisecond values to ASCII conversion

The positions where these values have to be displayed are given using text_addr signal as shown in Figure 7.15. The use of these locations is shown in and 7.17.A memory initialization file *human_count_INF.mem* is used by Lattice Radiant tool to store characters at address locations for display.

```
assign w_avginfhex_ch0_pos  = (text_addr == 13'd1136);
assign w_avginfhex_ch1_pos  = (text_addr == 13'd1137);
assign w_avginfhex_ch2_pos  = (text_addr == 13'd1138);
assign w_avginfhex_ch3_pos  = (text_addr == 13'd1139);
assign w_avginfhex_ch4_pos  = (text_addr == 13'd1140);
assign w_avginfhex_ch5_pos  = (text_addr == 13'd1141);
assign w_avginfhex_ch6_pos  = (text_addr == 13'd1142);
assign w_avginfhex_ch7_pos  = (text_addr == 13'd1143);

assign w_infhex_ch0_pos     = (text_addr == 13'd1172);
assign w_infhex_ch1_pos     = (text_addr == 13'd1173);
assign w_infhex_ch2_pos     = (text_addr == 13'd1174);
assign w_infhex_ch3_pos     = (text_addr == 13'd1175);
assign w_infhex_ch4_pos     = (text_addr == 13'd1176);
assign w_infhex_ch5_pos     = (text_addr == 13'd1177);
assign w_infhex_ch6_pos     = (text_addr == 13'd1178);
assign w_infhex_ch7_pos     = (text_addr == 13'd1179);
// Milisecond display
assign w_infms_ch0_pos      = (text_addr == 13'd1146);
assign w_infms_ch1_pos      = (text_addr == 13'd1147);
assign w_infms_ch2_pos      = (text_addr == 13'd1148);
assign w_infms_ch3_pos      = (text_addr == 13'd1149);
```

Figure 7.15.Text Address Positions to Display Input Values

- The address location structure for displaying average inference time (of 16 CNN frames) and inference time in millisecond values along with their strings are stored in *human_count_INF.mem* is shown in Figure 7.16.

	Characters stored as ASCII Values in human_count.mem file for string display											Avg Inf Time	Characters in human_count.mem file except Inf time value				
Character	A	v	g	l	n	f	T	i	m	e	:		(	Inf Time	m	s	)
Address in decimal	1122	1123	1124	1126	1127	1128	1130	1131	1132	1133	1134	1136 to 1143	1145	1146 to 1149	1150	1151	1152

Figure 7.16. Address Locations to Display Individual Frame Time and Inference Time with String in Display

- The address location structure for displaying individual frame inference time values along with the string are stored in *human_count_INF.mem* is shown in Figure 7.17.

Characters stored as ASCII Values in human_count.mem file for string display *									Individual Frame Inf Time
Character	I	n	f	T	i	m	e	:	
Address in decimal	1162	1163	1164	1166	1167	1168	1169	1170	1172 to 1179

Figure 7.17. Address Locations to Display CNN Count Value and its String in Display Output

- To display the input values in the address locations shown in Figure 7.16 and Figure 7.17, the ASCII values obtained as shown in Figure 7.13, Figure 7.14, and Figure 7.15 are sent to the 8 × 8 font ROM with the help of *font_char* signal to obtain the bitmap for display as shown in Figure 7.18.

```
assign font_char    = (r_face_ch0_pos )? r_face_ch0 :
                      (r_face_ch1_pos )? r_face_ch1 :
                      (r_th_sign_pos  )? r_th_sign :
                      (r_th_ch0_pos   )? r_th_ch0 :
                      (r_th_ch1_pos   )? r_th_ch1 :
                      (r_th_ch2_pos   )? r_th_ch2 :
                      (r_th_ch3_pos   )? r_th_ch3 :
                      //Inference Time Logic
                      (r_avginfhex_ch0_pos )? r_avginfhex_ch0 :
                      (r_avginfhex_ch1_pos )? r_avginfhex_ch1 :
                      (r_avginfhex_ch2_pos )? r_avginfhex_ch2 :
                      (r_avginfhex_ch3_pos )? r_avginfhex_ch3 :
                      (r_avginfhex_ch4_pos )? r_avginfhex_ch4 :
                      (r_avginfhex_ch5_pos )? r_avginfhex_ch5 :
                      (r_avginfhex_ch6_pos )? r_avginfhex_ch6 :
                      (r_avginfhex_ch7_pos )? r_avginfhex_ch7 :
                      (r_infhex_ch0_pos    )? r_infhex_ch0 :
                      (r_infhex_ch1_pos    )? r_infhex_ch1 :
                      (r_infhex_ch2_pos    )? r_infhex_ch2 :
                      (r_infhex_ch3_pos    )? r_infhex_ch3 :
                      (r_infhex_ch4_pos    )? r_infhex_ch4 :
                      (r_infhex_ch5_pos    )? r_infhex_ch5 :
                      (r_infhex_ch6_pos    )? r_infhex_ch6 :
                      (r_infhex_ch7_pos    )? r_infhex_ch7 :
                      (r_infms_ch0_pos    )? r_infms_ch0 :
                      (r_infms_ch1_pos    )? r_infms_ch1 :
                      (r_infms_ch2_pos    )? r_infms_ch2 :
                      (r_infms_ch3_pos    )? r_infms_ch3 :
                                            text_data[6:0];
```

Figure 7.18. Bitmap Extraction from Font ROM

8. Creating FPGA Bitstream File

8.1. Generating Bitstream using Lattice Radiant Software

To create the FPGA bitstream file:

1. Open the Lattice Radiant software. Default screen in shown in Figure 8.1.

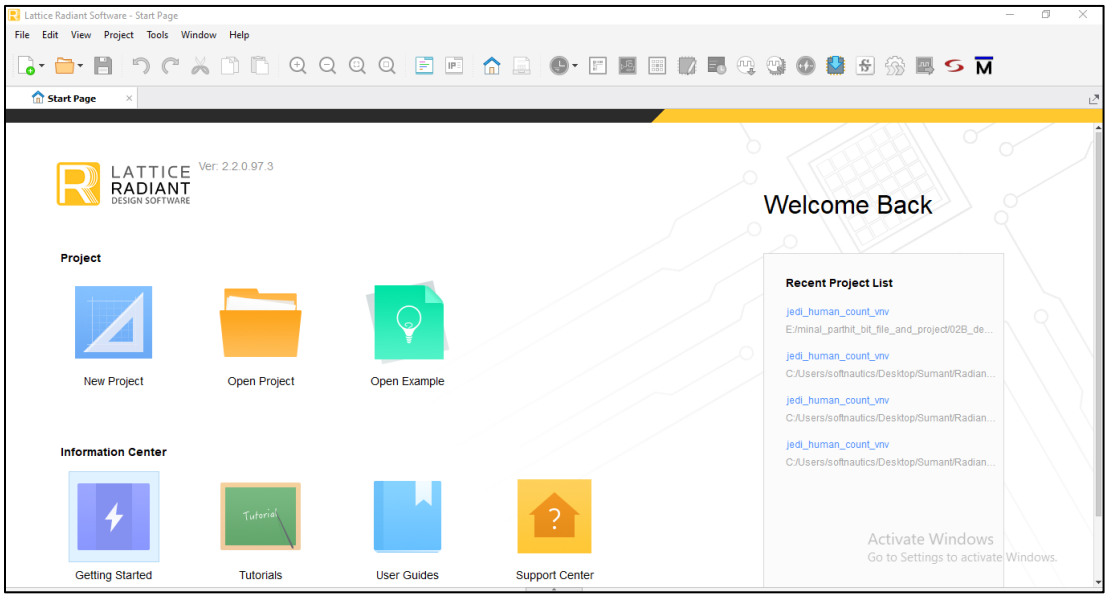

Figure 8.1. Lattice Radiant – Default Screen

2. Go to **File > Open > Project**.
3. Open the Lattice Radiant project file (.rdf) for the CrossLink-NX Voice and Vision Human Count Demo RTL. As shown in Figure 8.2, you can also open project by triggering the yellow folder shown in the user interface.

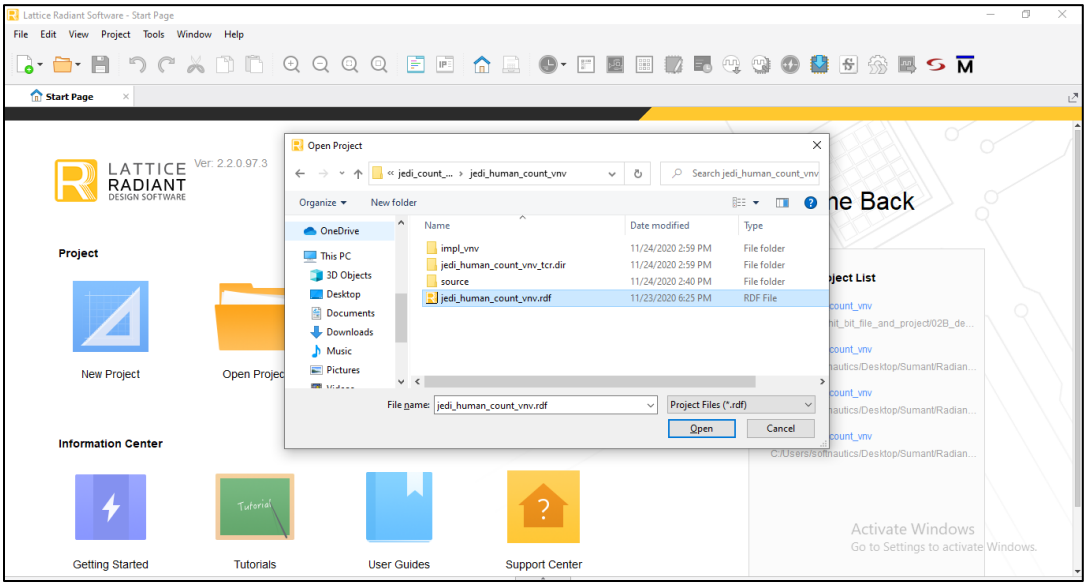

Figure 8.2. Lattice Radiant – Open CrossLink-NX VVML Board, Rev B Project File (.rdf)

4. After opening the project file, check the following points shown in Figure 8.3.
   - Design loaded with zero errors message shown in the Output window.
   - Check for this information in Project Summary window.
   - Part Number – LIFCL-40-9MG289I
   - Family – LIFCL
   - Device – LIFCL-40
   - Package – CSBGA289

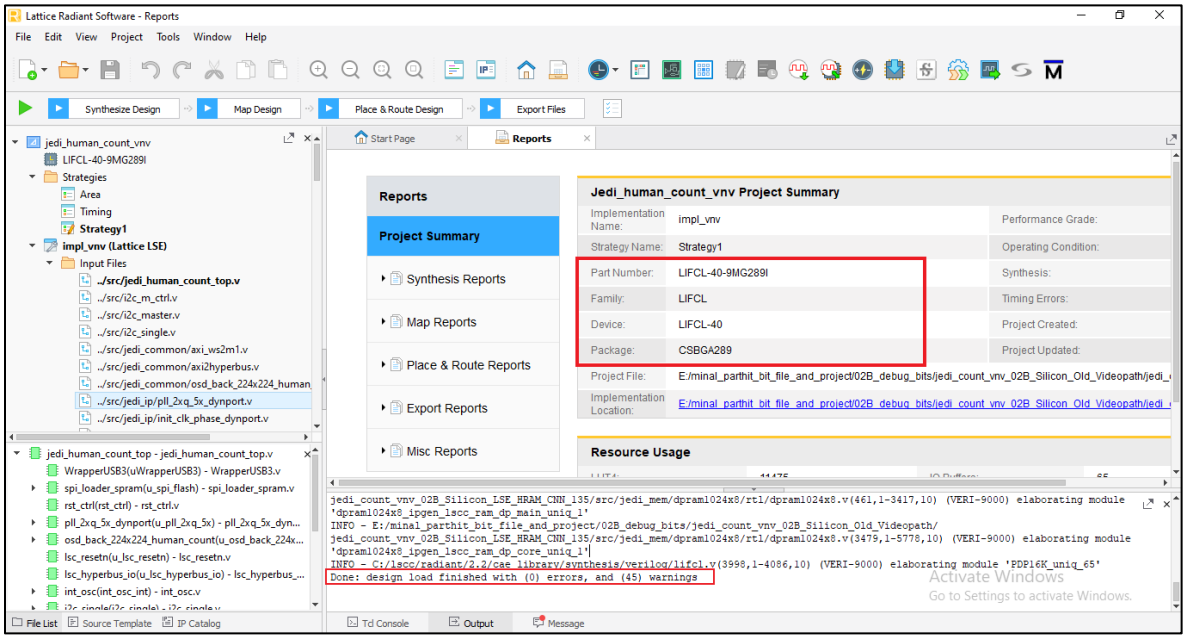

Figure 8.3. Lattice Radiant – Design load check after opening Project File

5. If the design is loaded without errors, click the **Run** button to trigger bitstream generation as shown in Figure 8.4.

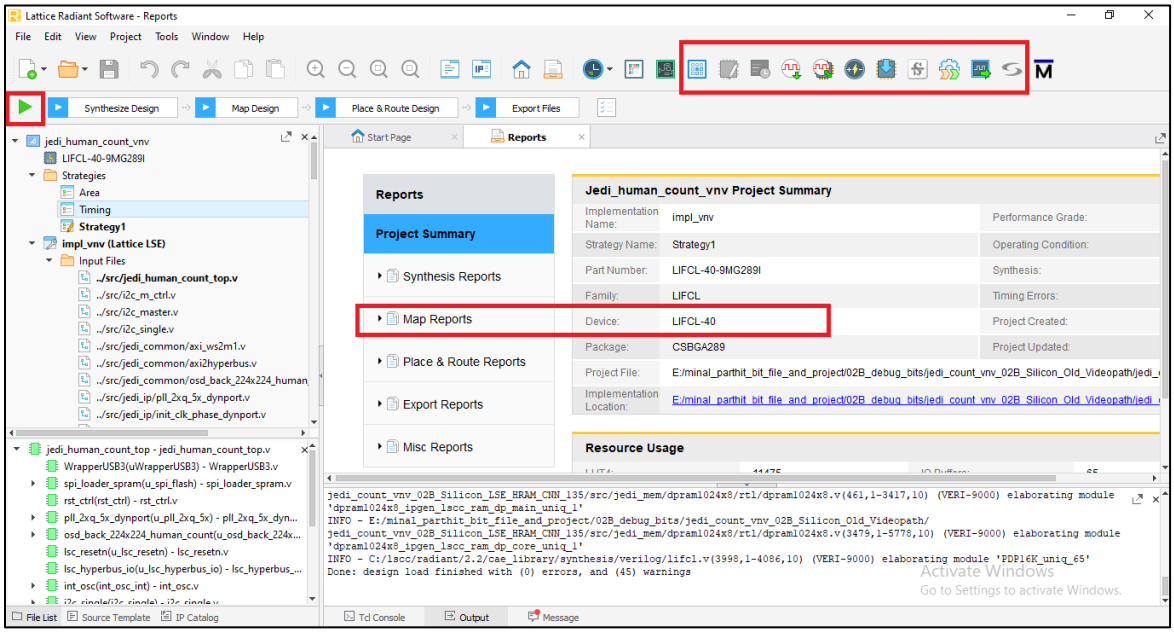

Figure 8.4. Lattice Radiant – Trigger Bitstream Generation

6. The Lattice Radiant tool displays *Saving bitstream in …* message in the **Reports** window. Bitstream is generated at *Implementation Location* shown in Figure 8.5.

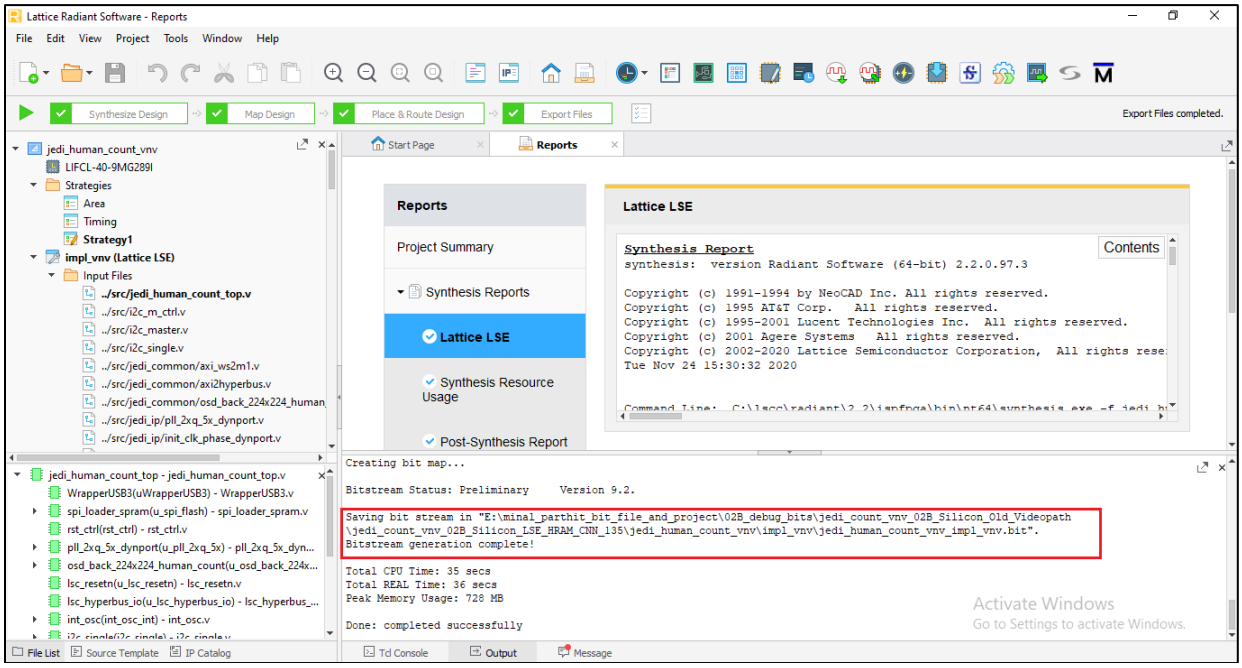

Figure 8.5. Lattice Radiant – Bit file Generation Report Window

8.2. Configuring IP in Lattice Radiant Software

After loading the design without any errors, perform the steps below to uninstall the old version of an existing IP or to install the latest version of an IP.

To uninstall an existing IP:

1. Click **IP Catalog** and go to **IP > DSP** in the IP tree.
2. Select the IP to uninstall and click the delete option.
3. Click **Yes** as shown in Figure 8.6 to remove the IP from the tree.

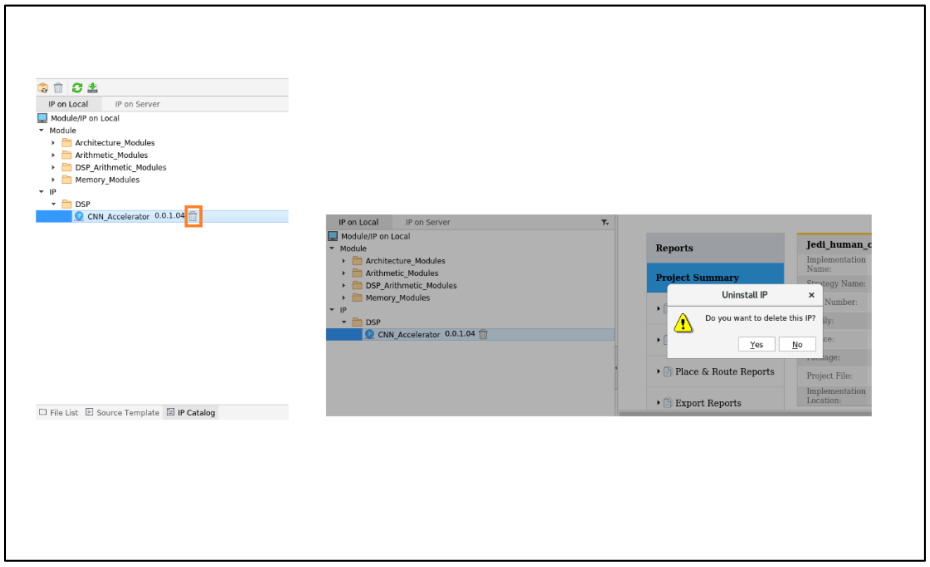

Figure 8.6. Lattice Radiant – Uninstall Old IP

To install a new IP:

1. Click the **Install a User IP** option as shown in Figure 8.7.

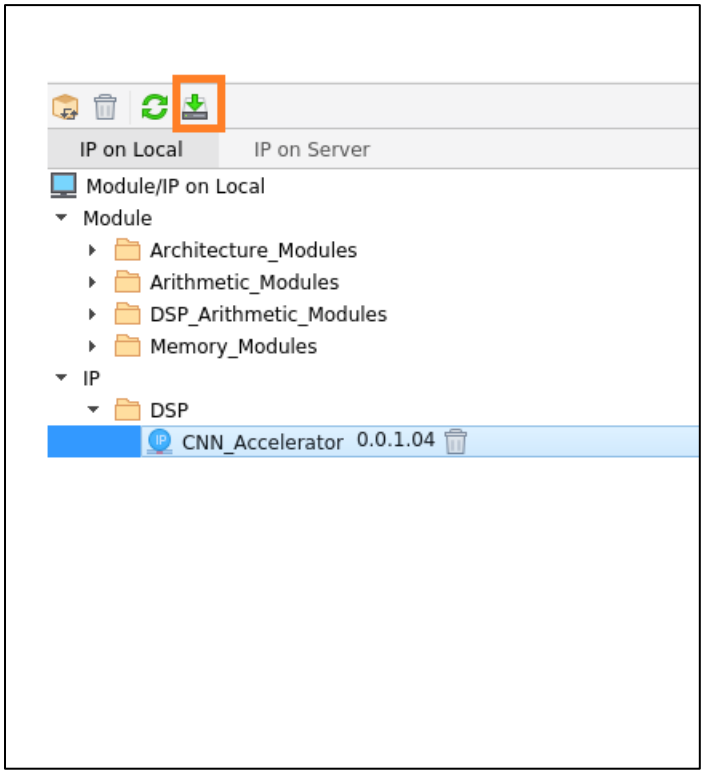

Figure 8.7. Lattice Radiant – Install New IP

2. Select and open the **IP package** (.ipk).

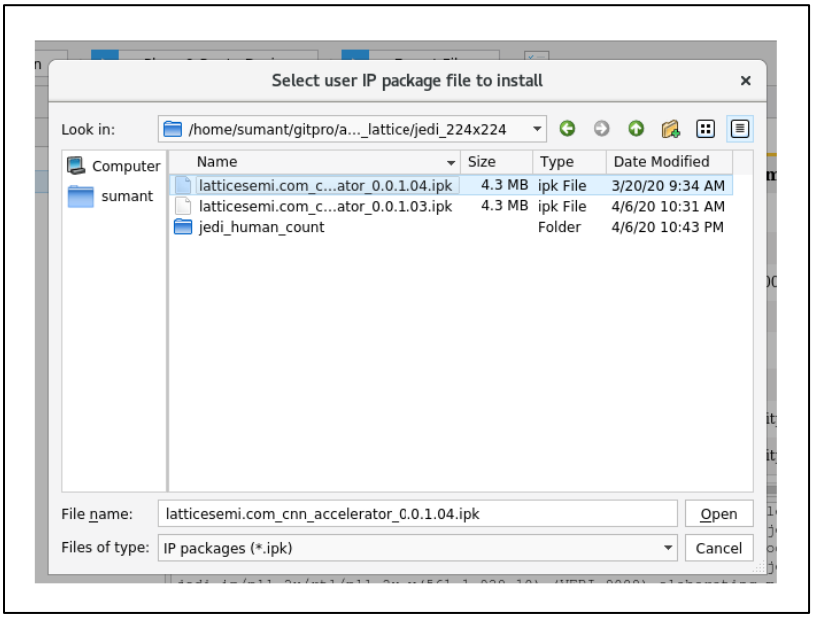

Figure 8.8. Lattice Radiant – Select User IP Package

3. The *IP License Agreement* window appears. Select **Accept**, and the IP is installed in the IP tree.

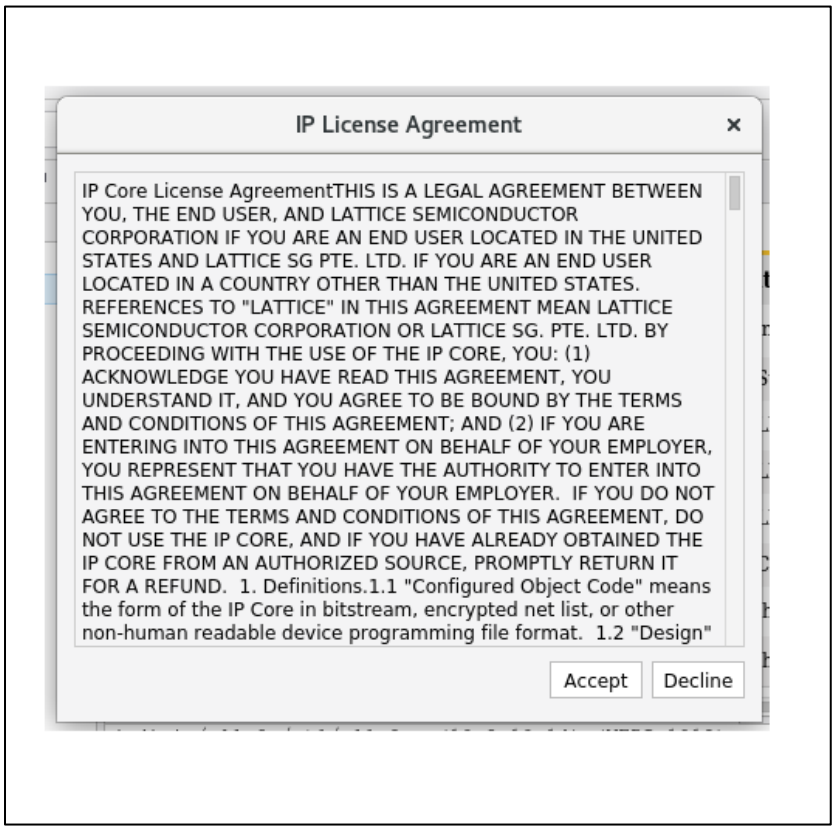

Figure 8.9. Lattice Radiant – IP License Agreement

Once the IP is installed, trigger the **Run** button to generate the bitstream.

9. Programming the Demo

9.1. Load Firmware in FX3 I²C EEPROM

To load the firmware:

1. Connect the USB3 port of the CrossLink-NX VVML Board (Rev B) to the PC using the USB3 cable.
2. Open the USB Control Centre application. Cypress FX3 SDK should also be installed.
3. Use the CrossLink-NX VVML (Rev B) board and put the jumper on **J13** to make the FX3 firmware programmable.
4. Connect the **FX3** cable to PC.
5. Press the **Push** button **SW2** to reset the FX3 chip. Figure 9.1 shows the boot loader device screen.

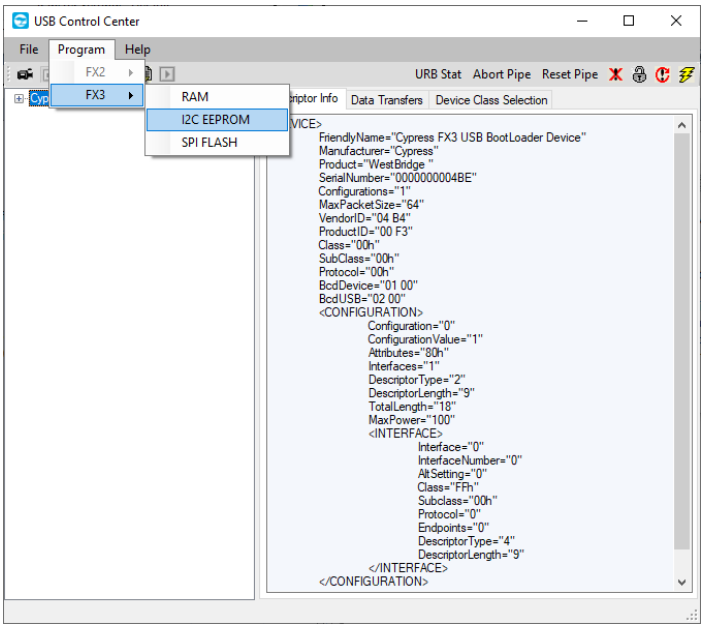

Figure 9.1. Selecting FX3 I²C EEPROM in USB Control Center

6. Select Cypress USB Bootloader.
7. Go to **Program > FX3 > I2C E2PROM**.
8. Open and select the FX3 image file for the 640×480p60 16-bit configuration, and the firmware is programmed in the I²C E2PROM. Wait for the *Programming Successful* message to appear in the bottom taskbar.
9. After successfully programming the files, remove the J13 jumper.
10. Power off and power on the board to boot the FX3 from I²C E2PROM.

9.2. Programming the CrossLink-NX VVML Board, Rev B SPI Flash

9.2.1. Erasing the CrossLink-NX VVML Board, Rev B SRAM Prior to Reprogramming

If the CrossLink-NX VVML Board, Rev B device is already programmed (either directly, or loaded from SPI Flash), follow this procedure to first erase the CrossLink-NX VVML Board, Rev B SRAM memory before re-programming the SPI Flash. If you are doing this, keep the board powered when re-programming the SPI Flash (so it does not reload on reboot).

To erase CrossLink-NX VVML Board, Rev B:

1. Start Lattice Radiant Programmer. In the **Getting Started** dialog box, select **Create a new blank project**.

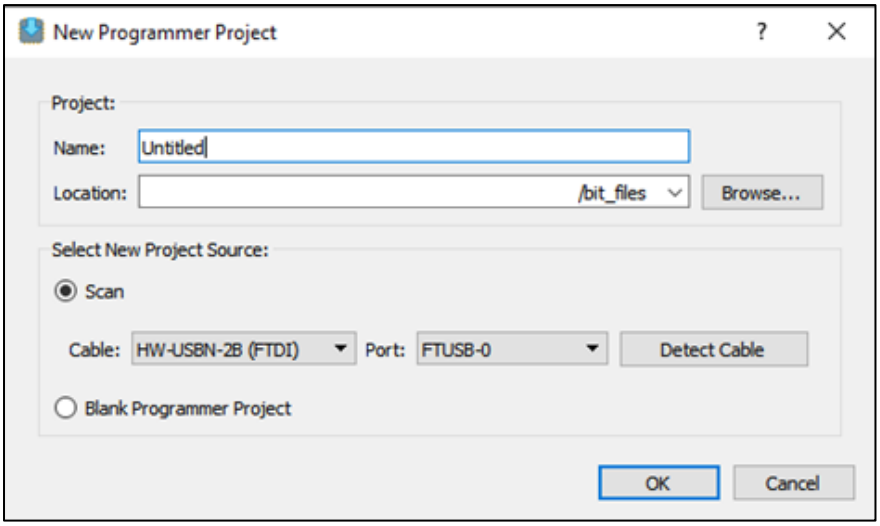

Figure 9.2. Lattice Radiant Programmer – Default Screen

2. Click **OK**.
3. Select **LIFCL** for **Device Family** and **LIFCL-40** for **Device** as shown Figure 9.3.

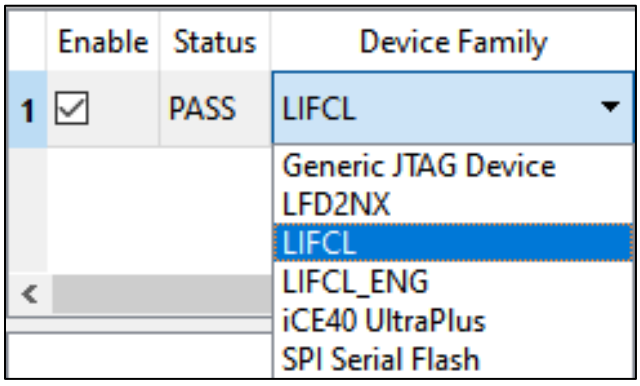

Figure 9.3. Lattice Radiant Programmer – Device Selection

4. Right-click and select **Device Properties**.
5. Select **JTAG** for Port Interface, **Direct Programming** for Access Mode, and **Erase Only** for Operation as shown in Figure 9.4.

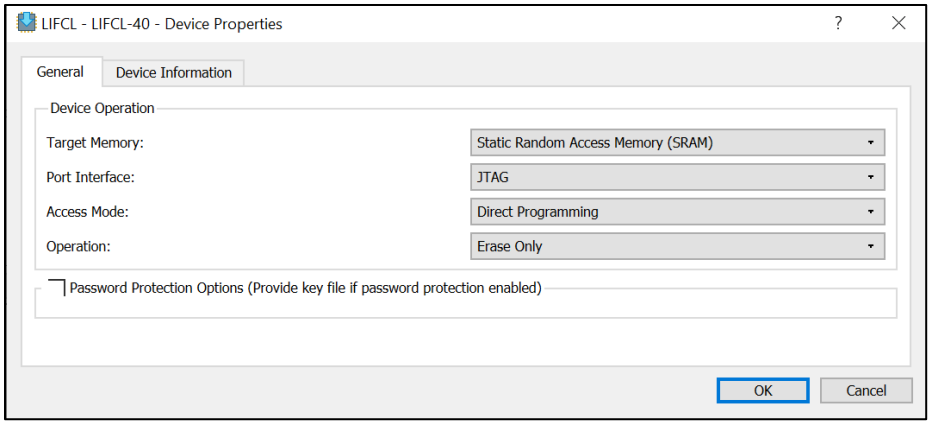

Figure 9.4. Lattice Radiant Programmer – Device Operation

6. Click **OK** to close the **Device Properties** dialog box.
7. Press the **SW5** push button switch. Click the **Program** button. Hold it until you see the *Successful message* in the Lattice Radiant log window.
8. In the Lattice Radiant Programmer main interface, click the **Program** button to start the erase operation.

9.2.2. Programming the CrossLink-NX VVML Board, Rev B Board

To program the CrossLink-NX VVML Board, Rev B SPI flash:

1. Ensure that the CrossLink-NX VVML Board, Rev B device SRAM is erased by performing the steps in Programming the CrossLink-NX VVML Board, Rev B SPI Flash

Erasing the CrossLink-NX VVML Board, Rev B SRAM Prior to Reprogramming.

3. In the Radiant Programmer main interface, right click the CrossLink-NX Voice and Vision Machine Learning (VVML) Board, Rev B row and select **Device Properties** to open the **Device Properties** dialog boxes as shown in Figure 9.5.
4. Select SPI FLASH for Access mode, JTAG2SPI for Port Interface, and Direct Programming for Access Mode.
5. For Programming File, browse and select the CrossLink-NX VVML Board, Rev B bitfile (*.bit).
6. For SPI Flash Options, make the selections in Figure 9.5 to select the Macronix 25L12833F device.

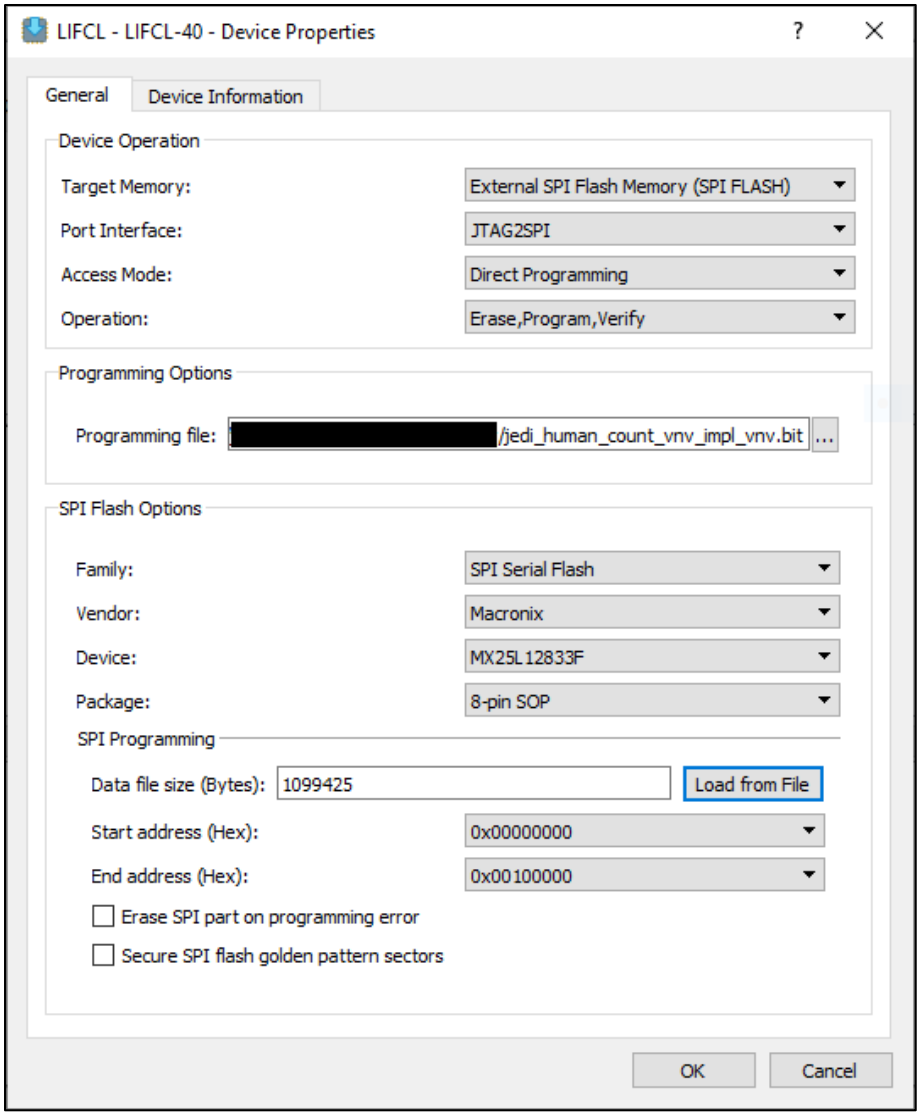

Figure 9.5. Lattice Radiant Programmer – Selecting Device Properties Options for CrossLink-NX Flashing

a. Click **Load from File** to update the Data file size (Bytes) value.

b. Ensure that the following addresses are correct:
   - Start Address (Hex) – 0x00000000
   - End Address (Hex) – 0x00100000

7. Click **OK**.

8. Press the **SW5** push button switch before clicking **Program** button as shown in Figure 9.6. Hold it until you see the *Successful message* in the Lattice Radiant log window.

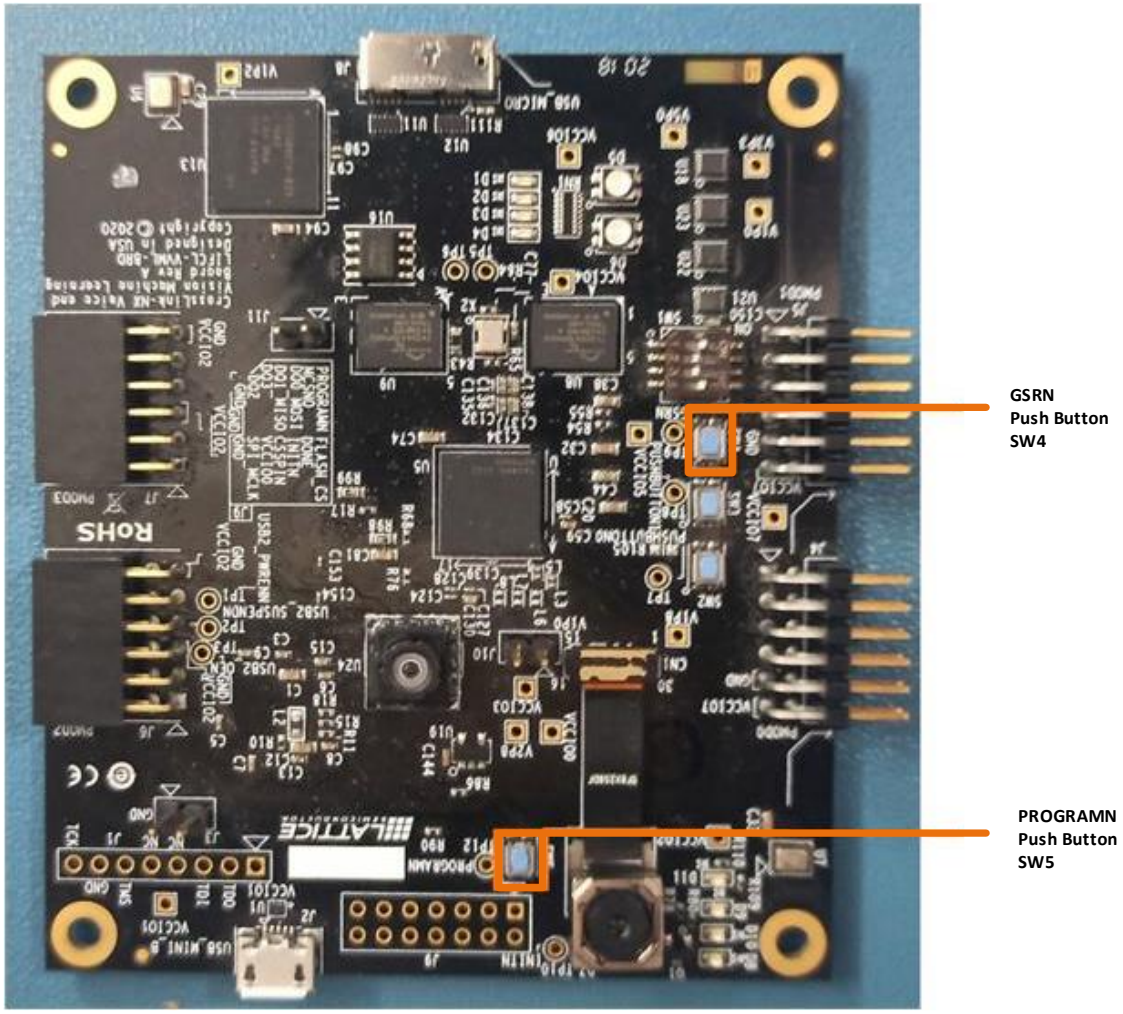

Figure 9.6. CrossLink-NX VVML Board Flashing Switch – SW5 Push Button

9. After successful programming, the **Output** console displays the result as shown in Figure 9.7.

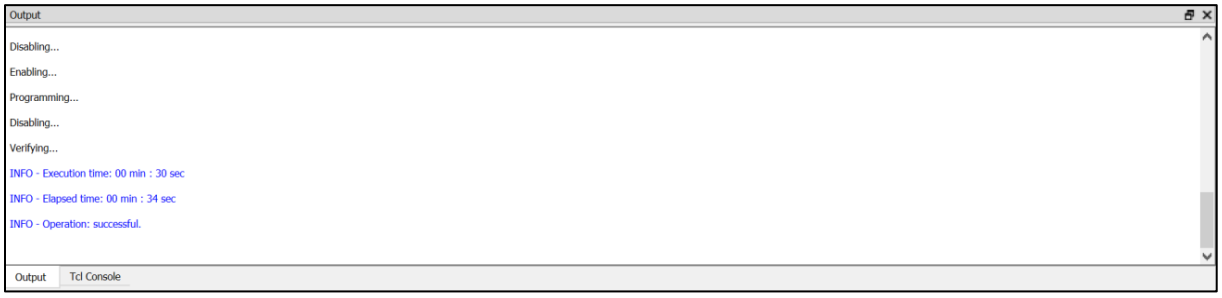

Figure 9.7. Lattice Radiant Programmer – Output Console

9.2.3. Programming sensAI Firmware Binary to the CrossLink-NX VVML SPI Flash

9.2.3.1. Flash sensAI Firmware Hex to CrossLink-NX VVML SPI Flash

To program the CrossLink-NX SPI flash:

1. Ensure that the CrossLink-NX device SRAM is erased by performing the steps in Programming the CrossLink-NX VVML Board, Rev B SPI Flash

Erasing the CrossLink-NX VVML Board, Rev B SRAM Prior to Reprogramming before flashing the bitstream and sensAI firmware binary.

3. In the Lattice Radiant Programmer main interface, right-click the CrossLink-NX row and select **Device Properties**.
4. Apply the settings below:
   a. Under Device Operation, select the options below:
      - Port Interface – JTAG2SPI
      - Target Memory – SPI FLASH
      - Access Mode – Direct Programming
   b. Under Programming Options, select the CrossLink-NX sensAI firmware binary file after converting it to hex (*.mcs) for the **Programming File**.
   c. For **SPI Flash Options**, make the selections as shown in Figure 9.8.

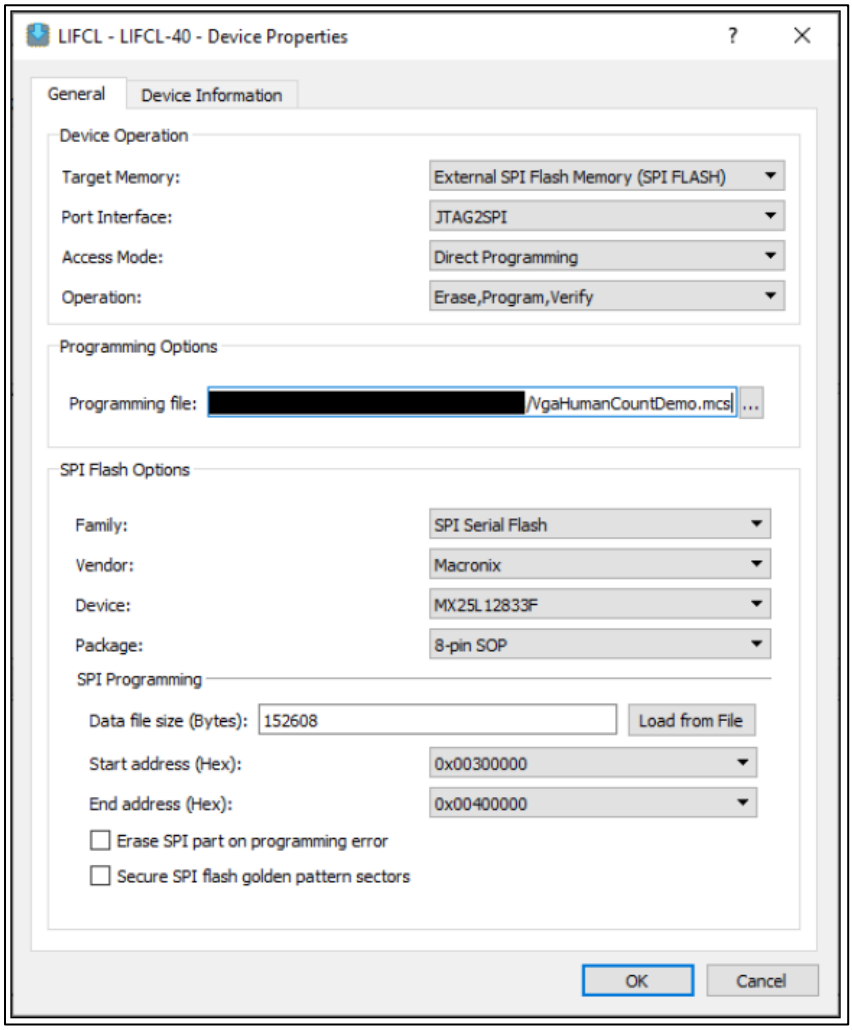

Figure 9.8. Lattice Radiant Programmer – Selecting Device Properties Options for CrossLink-NX Flashing

d. Click **Load from File** to update the data file size (bytes) value.

e. Ensure that the following addresses are correct:
   - Start Address (Hex) – 0x00300000
   - End Address (Hex) – 0x00400000

5. Click OK.
6. Press the **SW5** push button switch. Click the **Program** button and hold it until you see the *Successful* message in the Lattice Radiant log window.
7. After successful programming, the **Output** console displays the result as shown in Figure 9.9.

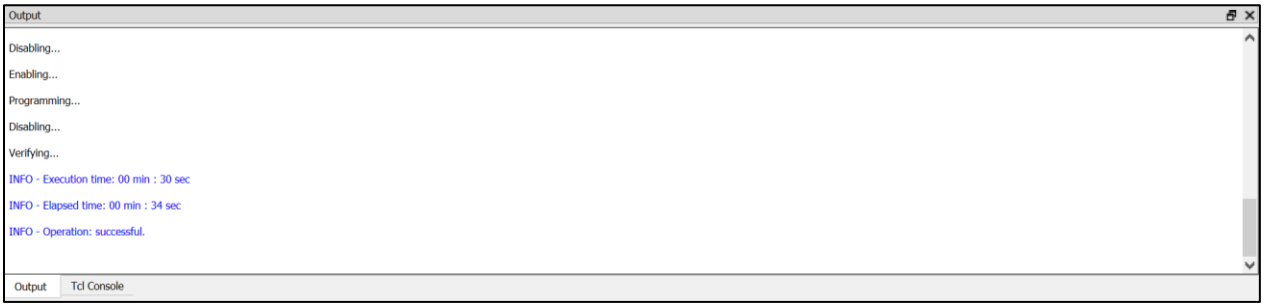

Figure 9.9. Lattice Radiant Programmer – Output Console

10. Running the Demo

To run the demo:

1. Power on the CrossLink-NX VVML board. Make sure the position of SWITCH0 is ON to boot the device from I2C EEPROM.
2. Connect the CrossLink-NX VVML board to the display monitor through the board's USB3 port.
3. Open the AMCap or VLC application and select the FX3 device as source.
4. The camera image should be displayed on monitor as shown in Figure 10.1.

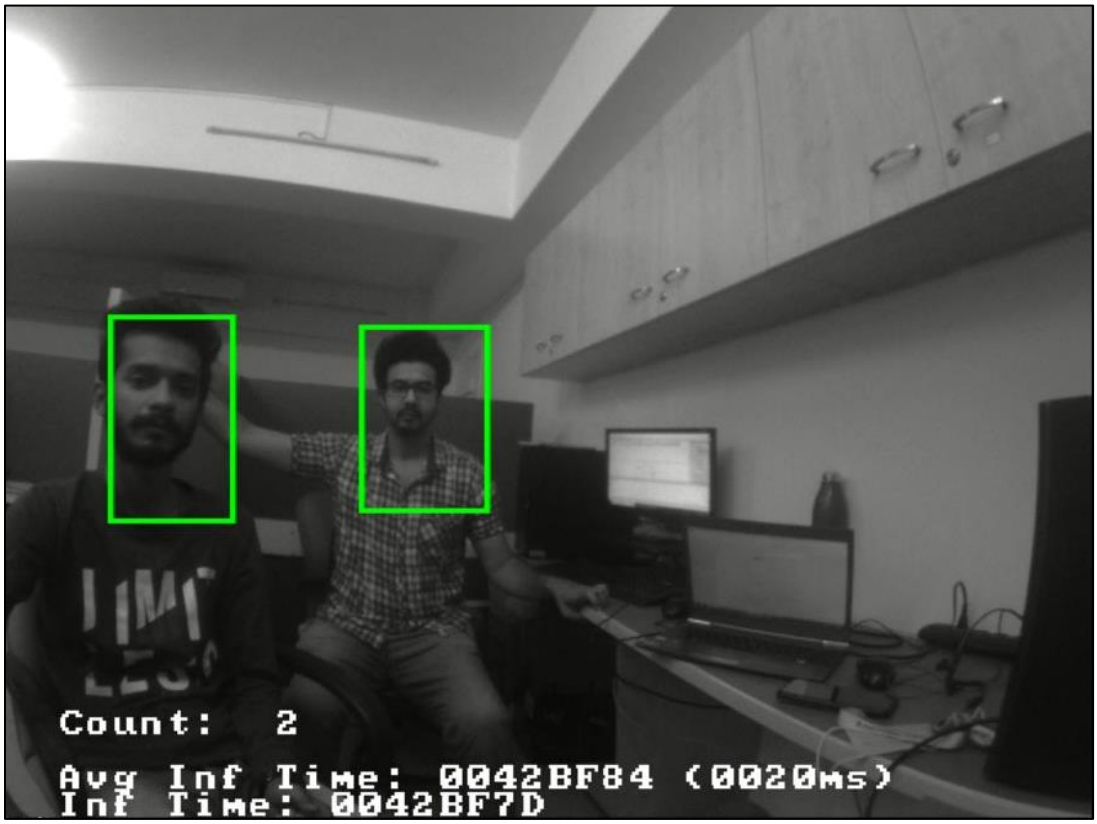

Figure 10.1. Running the Demo

5. The demo output contains the bounding boxes for detected humans in a given frame and it displays the total number of detected humans in a given frame on the display.

Appendix A. Other Labeling Tools

Table A.1 provides information on other labeling tools.

Table A.1. Other Labeling Tools

Software	Platform	License	Reference	Converts To	Notes
annotate-to-KITTI	Ubuntu/Windows (Python based utility)	No License (Open source GitHub project)	https://github.com/SaiPrajwal95/annotate-to-KITTI	KITTI	Python based CLI utility. Just clone it and launch. Simple and Powerful.
LabelBox	JavaScript, HTML, CSS, Python	Cloud or On-premise, some interfaces are Apache-2.0	https://www.labelbox.com/	json, csv, coco, voc	Web application
LabelMe	Perl, JavaScript, HTML, CSS, On Web	MIT License	http://labelme.csail.mit.edu/Release3.0/	xml	Converts only jpeg images
Dataturks	On web	Apache License 2.0	https://dataturks.com/	json	Converts to json format but creates single json file for all annotated images
LabelImg	ubuntu	OSI Approved :: MIT License	https://mlnotesblog.wordpress.com/2017/12/16/how-to-install-labelimg-in-ubuntu-16-04/	xml	Need to install dependencies given in reference
Dataset_annotator	Ubuntu	2018 George Mason University Permission is hereby granted, Free of charge	https://github.com/omenyayl/dataset-annotator	json	Need to install app_image and run it by changing permissions

References

- Google TensorFlow Object Detection GitHub
- Pretrained TensorFlow Model for Object Detection
- Python Sample Code for Custom Object Detection
- Train Model Using TensorFlow

Technical Support Assistance

Submit a technical support case through www.latticesemi.com/techsupport.

Revision History

Revision 1.0, June 2021

Section	Change Summary
All	Initial release.

www.latticesemi.com