TI Designs: TIDEP-01004
Deep Learning Inference For Embedded Applications Reference Design

Description

This reference design demonstrates how to use TI Deep Learning (TIDL) on a Sitara AM57x System-on-Chip (SoC) to bring deep learning inference to an embedded application. This design shows how to run deep learning inference on either C66x DSP cores (available in all AM57x SoCs) and Embedded Vision Engine (EVE) subsystems, which are treated as black boxed deep learning accelerators on the AM5749 SoC.

This reference design is applicable to any application that is looking to bring deep learning inference into an embedded application.

Customers looking to quickly get started with a deep learning network or to evaluate their own networks performance on an AM57x device will find a step-by-step guide on how to use TIDL available as part of TI's free AM57x Processor SDK.

Resources

TIDEP-01004	Design Folder
AM5749	Product Folder
AM57x	AM57x SoC Family Folder
AM5749 IDK EVM	AM5749 IDK EVM Folder
TIDA-010013	Design Folder
AM57x Software Development Kit (SDK)	AM57x SDK Download Folder

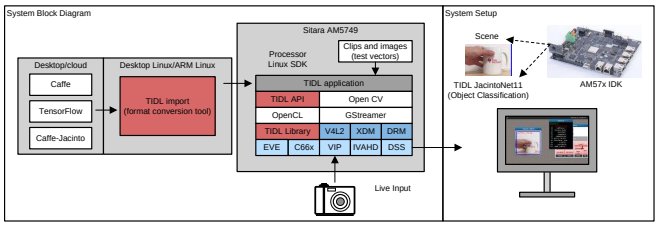
TI E2E™ Community — ASK Our E2E™ Experts

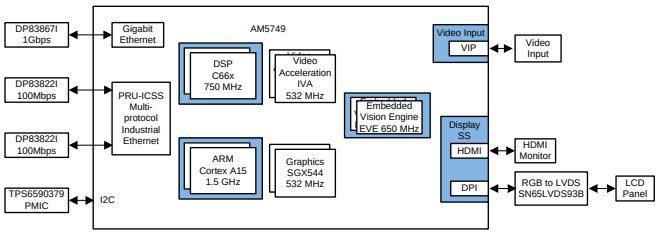
Features

- Embedded deep learning inference on AM57x SoC
- Performance scalable TI deep learning library (TIDL library) on AM57x using C66x cores only, EVE subsystems only, or C66x + EVE combination.
- Performance optimized reference CNN models for object classification, detection and pixel-level semantic segmentation.
- Full walk-through of TIDL development flow: training, import and deployment
- Benchmarks of several popular deep learning networks on AM5749
- This reference design is tested on AM5749 IDK EVM and includes TIDL library on C66x core and EVE subsystem, reference CNN models and Getting Started guide.

Applications

- Automated sorting equipment
- Optical inspection
- Vision computer
- Code readers
- Industrial robots
- Logistics robots
- Currency counters
- ATMs
- Patient monitors
- Building automation
- Industrial transport
- Space, avionics & defense





An IMPORTANT NOTICE at the end of this TI reference design addresses authorized use, intellectual property matters and other important disclaimers and information.

1 System Description

Deep learning is a type of machine learning that trains a computer to perform human-like tasks, such as identifying images, recognizing speech, or making predictions in time series. Instead of organizing data to run through predefined equations, deep learning sets up basic parameters about the data and trains the computer to learn on its own by recognizing patterns using many layers of processing. Deep learning is one of the foundations of artificial intelligence (AI) and is influencing industry after industry by enabling products to behave intelligently like humans.

NOTE: If you are new to deep learning and want a quick overview, it is recommended to watch this video before continuing.

Deep learning models are based on deep artificial neural networks. They are also known as Deep Neural Networks (DNNs). Within DNNs, there are many different neural network architectures like Convolution Neural Networks (CNNs) and Recurrent Neural Networks (RNNs). These different architectures lend themselves to solving different types of problems. CNN models are popular in solving computer vision problems and are ideal for image classification, object detection and semantic segmentation.

At a high level, deep learning, similar to any automated system based on statistical machine learning techniques, works as a two-stage process: Training and Inference. Training is the process of developing a deep learning algorithm. After training is completed, the networks are deployed into the field for "inference" — classifying data to "infer" a result. Training a deep learning model usually occurs offline using a large data set on servers or PCs with external accelerators such as graphics processing units (GPUs). Real-time performance or the thermal solution is not an issue during this phase. However, during inference in an embedded system, real-time performance and device power consumption can be a key care about for many end products.

TI addresses the need for bringing deep learning inference at the edge for embedded applications with the highly integrated AM57x family of Sitara™ processors. The Sitara processors provide industrial grade solutions with single to multicore Arm® processors, with the AM57x family equipped with high performance Arm Cortex®-A15 cores running at up to 1.5 GHz. The scalable family of AM57x provides dedicated hardware for accelerated multimedia and industrial communication, multiple capture and display interfaces and a rich set of connectivity peripherals.

The AM57x family also features single and dual core C66x processors that are capable of running deep learning inference as well as traditional machine vision algorithms. For additional inference performance, the AM5749 processor includes the addition of two EVE subsystems.

TIDL enables running the real-time inference part of deep learning at low power on both the C66x cores and the EVE subsystems. It is a set of open-source Linux software packages and tools that enables offloading of Deep Learning inference compute workloads from Arm cores to EVE subsystems and C66x cores. Developing and deploying CNN for image classification, object detection and pixel-level semantic segmentation use cases on the AM5749 SoC is described in this document.

NOTE: TIDL does not address the training phase of deep learning models. It enables running deep learning inference at the edge.

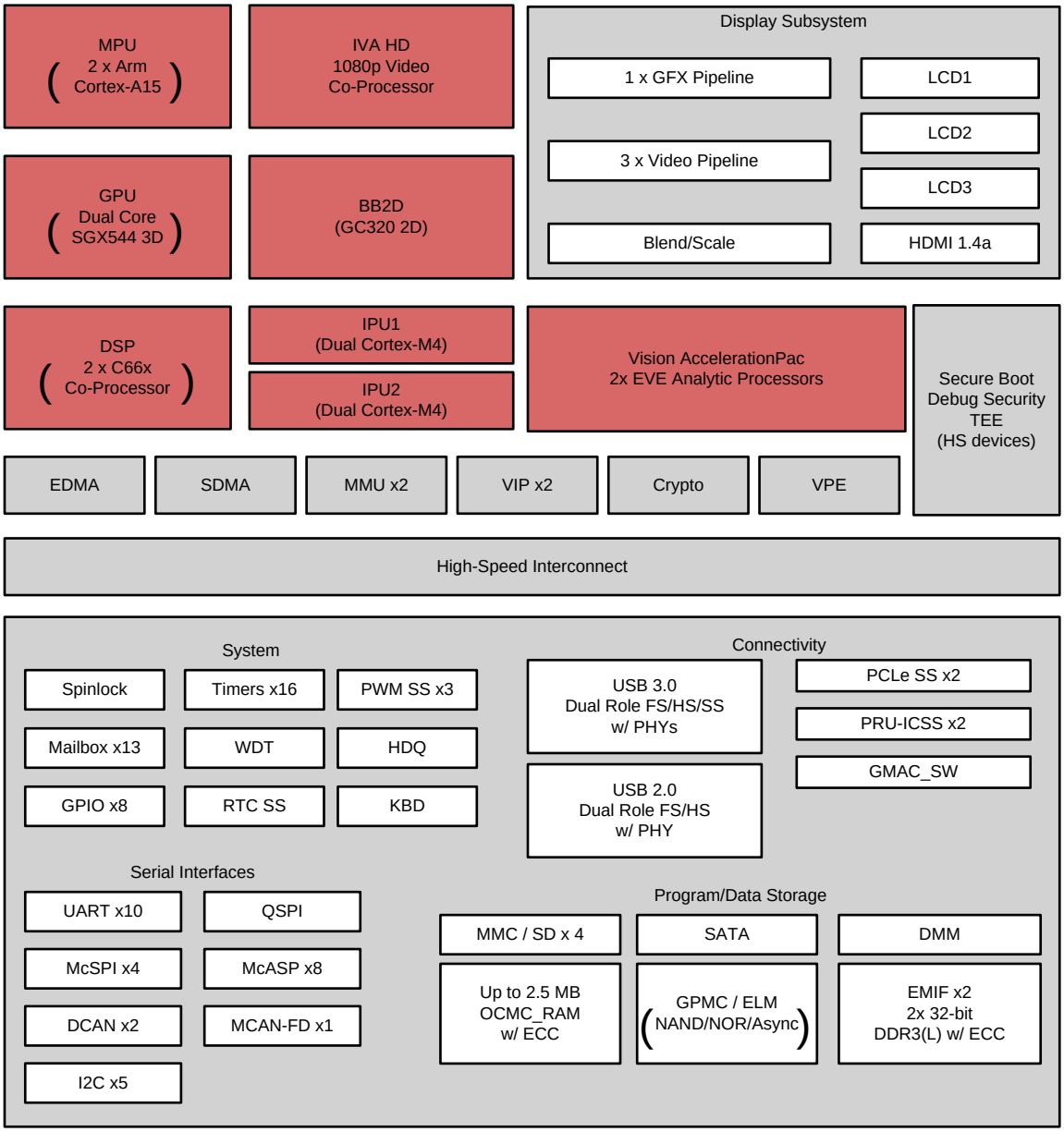
1.1 Key System Specifications

1.1.1 AM57x SoC

The AM57x is a highly integrated, pin compatible, scalable, Sitara class processor. It has single or dual core Arm Cortex-A15s. The AM57x is designed for embedded applications including PLCs, industrial network switches, industrial gateways for protocol translation, human machine interface (HMI), grid infrastructure protection and communications, and other industrial use applications. The device includes the following subsystems:

- Cortex-A15 microprocessor unit (MPU) subsystem, including two Arm Cortex-A15 cores
- Two digital signal processor (DSP) C66x subsystems
- Two Embedded Vision Engine (EVE) susbsytems
- Two Cortex-M4 subsystems, each including two Arm Cortex-M4 cores
- Two dual-core Programmable Real-Time Unit for Industrial Communications (PRU-ICSS)
- Display subsystem (DSS)
- Video Processing subsystem (VPE)
- Video Input Capture (VIP)
- 3D-graphics processing unit (GPU) subsystem, including POWERVR™ SGX544 dual-core
- 2D-graphics accelerator (BB2D) subsystem, including Vivante™ GC320 core
- Real-time clock (RTC) and Debug subsystems
- The device provides a rich set of connectivity peripherals, including among others: USB 3.0 and 2.0, SATA 2, PCIe Gen2 Gigabit Ethernet Switch subsystems
- The device includes support for functional safety system requirements
  - Error Detection and Correction:
    - Parity bit per byte on C66x DSP
    - L1 program cache and Single-Error Correction Dual-Error Detection (SECDED) on L2 memories
    - SECDED on Large L3 memory
    - SECDED on external DDR memory interface (EMIF1 only)
  - MMU/MPU
    - MMU used for key masters (Cortex-A15 MPU, Cortex-M4 IPU, C66x DSP, EDMA)
    - Memory protection of C66x cores
    - MMU inside the Dynamic Memory Manage
- The device also integrates on-chip memory, external memory interfaces and memory management.

Figure 1. AM574x Block Diagram



Executing deep learning models for inference are computationally intensive. They also have high data input/output (I/O) bandwidth requirement. A good choice of embedded processor for Deep Learning is the one that not only has the data processing power but is also able to fetch the needed data for processing in real time, else the actual performance throughput can be far less than the theoretical claimed one. The processor should also have enough memory to fit in the model and its parameters.

The EVE subsystems and the C66x cores inside AM57x processor are a good fit for accelerating processing of Deep Learning layers. The EVE subsystem has Single Instruction Multiple Data (SIMD) architecture. The C66x core is Very Long Instruction Word (VLIW), SIMD + MIMD architecture. Both the EVE subsystem and C66x architectures have cache, EDMA and internal memories. This allows programming of the engine in such a way that the processing engine does not have to pay much penalty for the data I/O resulting in a good actual throughput (upto 80%) compared to the theoretical one. The AM57x device can address 4 GiB of physical SDRAM space. With such an architecture, there are limited constraints on the network model or parameter size. While both C66x cores and EVE subsystems are a good fit for accelerating the deep learning layers, the EVE subsystem can process certain layers up to 4x times faster compared to the C66x core due to its architecture, like the presence of the Vector Coprocessor (VCOP) SIMD engine and higher data I/O bandwidth.

1.1.2 Processor SDK With TIDL

The Processor Software Development Kit (PSDK) for Linux is a unified software platform for TI embedded processors providing easy setup and fast out of box access to benchmarks and demos. All releases of Processor SDK are consistent across TI's broad portfolio, allowing developers to seamlessly reuse and migrate software across devices.

Linux Highlights:

- Long-term stable (LTS) mainline Linux kernel support
- U-Boot bootloader support
- Linaro GNU compiler collection (GCC) tool chains
- Yocto Project™ OE Core compatible file systems

TIDL is part of PSDK for Linux, which enables accelerating the deep neural network layers on EVE subsystems and C66x cores on AM57x SoCs using C++ based TIDL APIs. TIDL provides the right balance of accuracy, speed and memory usage trade-offs. It also provides an easy way to use a model from one of the popular deep learning training frameworks and runs it on an AM57x-based embedded platform. Ease of use, high performance and low power integrated solution are differentiations offered by TIDL.

Initial release of TIDL supports only CNN layer types. It supports most of the popular CNN layers present in frameworks such as Caffe and TensorFlow. The layers come with parametric restrictions. Future releases of TIDL will also support RNN layers. To learn about the latest list of TIDL supported layers and restrictions on the parameters, see the Processor SDK Linux document.

CNN layers on TIDL can process signals from vision camera, Time of Flight (ToF), mmWave/Radar or other sources.

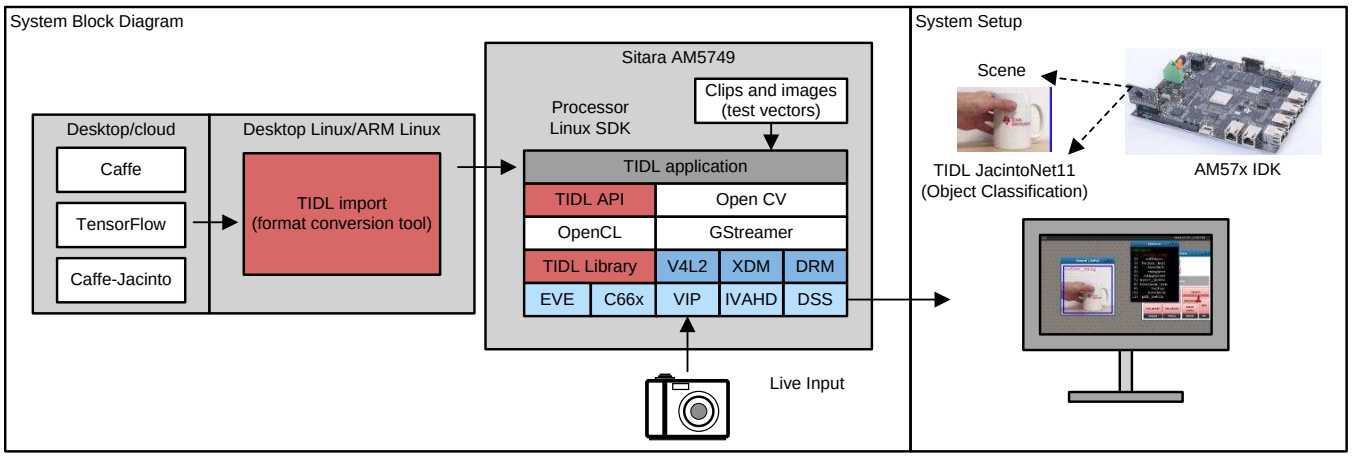
The PSDK Linux comes with TI developed performance optimized example CNN models that demonstrates real-time processing of applications involving image classification, object detection and pixel-level semantic segmentation.

NOTE: Support for performance optimized RNN layers and example network models are planned for future release of TIDL.

2 System Overview

2.1 Block Diagram

Figure 2. TIDEP-01004 Block Diagram



2.2 Highlighted Products

2.2.1 AM5749 SoC

AM5749 SoC brings high performance Deep Learning inference through the maximum flexibility of a fully integrated mixed processor solution utilizing two C66x cores and two EVE subsystems. The devices also combine programmable video processing with a highly integrated peripheral set. Cryptographic acceleration is available in every AM574x device.

Arm allows developers to keep control functions separate from other algorithms programmed on the C66x cores and coprocessors, thus reducing the complexity of the system software.

Cryptographic acceleration is available in all devices. All other supported security features, including support for secure boot, debug security and support for trusted execution environment are available on High-Security (HS) devices. For more information about HS devices, refer to this document.

2.2.2 AM5749 IDK

The AM574x Industrial Development Kit (IDK) is a development platform for evaluating the industrial communication and control capabilities of Sitara AM574x processors for applications in factory automation, drives, robotics, grid infrastructure, and more.

AM574x processors include dual Programmable Real-Time Unit for Industrial Communications (PRU-ICSS) subsystems that can be used for industrial Ethernet protocols such as Profinet, EtherCAT, Ethernet/IP, and others.

The TMDXIDK5749 breaks out six ports of Ethernet, four of which can be used concurrently: 2x Gb Ethernet ports and 2x 10/100 Ethernet ports from the PRU-ICSS subsystems. It includes four Ethernet ports with concurrent operation including two from PRU-ICSS, 2GB DDR3, Profibus connection, EtherCAT and RS485 Headers, On-board eMMC, Mini PCIe, USB3, and HDMI connectors.

To learn more about the board and place order for the kit, click here.

2.3 Design Considerations

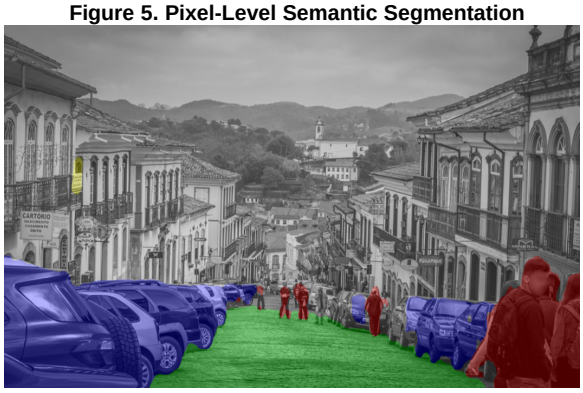
For applications targeting CNN models, TI has developed performance optimized CNN reference models for three different application areas:

- **Object classification**: Prediction about what class of object is present in the scene.
- **Object detection**: Making predictions over potentially multiple objects in the scene, and getting a rough idea of where they are located.
- **Pixel level semantic segmentation**: Locating the regions taken up by the different objects with much more detail, to create precise outlines of the objects in the scene. In this, each pixel is assigned a class.

Figure 3. Object Classification

Figure 4. Object Detection

Figure 5. Pixel-Level Semantic Segmentation



TI developed an example network model named as JacintoNet11 to demonstrate object classification, JDetNet model to demonstrate object detection and JSegNet21 to demonstrate pixel-level semantic segmentation.

Subsequent subsections describe the design approaches taken to develop the performance optimized CNN models that can run real time on AM57x SoC using TIDL.

2.3.1 Deep Learning Solutions Development flow

Developing solutions using deep learning technology is a multi-step process. It involves:

- Selecting and preparing a data set for training the network
- Selecting a framework to develop a network model
- Designing initial network model

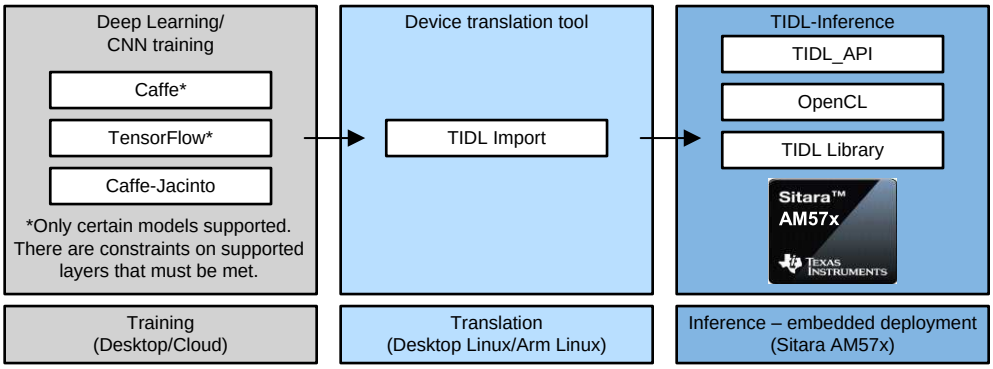
NOTE: You can start from scratch or use one of the available open source network models or the TI provided reference model as a starting point.

Steps 1 through 6 are grouped together to be called as training phase.

1. Iterating on the network design with translate (adjust breadth, depth, and so forth) until performance (frames) is appropriate for the application.
2. Training offline on desktop or cloud and determine accuracy, refining the network design as appropriate.
3. Repeat up to step 4 until both performance and accuracy goals are met.
4. Once satisfied with network performance and accuracy, translating the model to a format understood by deployment setup.
5. Deploying the trained model on production-setup for inference.
6. Optional - Collect more training data set from the inference run and re-train the network model for improved accuracy by going back to step 5.

Table 1 illustrates the development flow for deploying deep learning solutions using TIDL on AM57x SoC.

Figure 6. TIDL Development Flow



The first part of the development flow is for training a network model and is best accomplished within popular training frameworks. Several publicly available deep learning frameworks enable the training of CNN or other deep learning models. Two of the most popular frameworks are Caffe and TensorFlow. TIDL supports Caffe and TensorFlow framework with some restrictions and has also developed its own framework Caffe-Jacinto to enable network model for embedded processors.

The next step is using the TIDL device translator tool to convert network models into an internal format best suited for use inside the TIDL library.

The final step is to run the converted network model on the AM57x SoC device using TIDL APIs.

2.3.1.1 Training Framework

TI developed its own framework forked from NVIDIA/Caffe, which in-turn is derived from BVLC/Caffe. The modifications in this fork enable training of complex models that can be used in embedded platforms. The framework is named as Caffe-Jacinto and is community-driven, open source and well documented. It is hosted on github. There are example network models provided for image classification, object detection and pixel-level semantic segmentation. Example scripts are also provided for training the models using Caffe-Jacinto.

Along with Caffe-Jacinto, TIDL supports importing models from Caffe and TensorFlow framework using the device translator tool. It also supports various Caffe flavors including BVLC/Caffe and NVIDIA/Caffe.

NOTE: Only certain models are supported with BVLC-Caffe and TensorFlow frameworks. There are constraints on supported layers that must be met for TI device translator tool to work on it.

2.3.1.2 Designing Network Model

When developing a deep learning solution for embedded processors, complexity of the overall network should be restricted such that it fits well within the computing capability of the targeted device. The computational complexity of full frame CNN applications is extremely high. This kind of complexity is out of reach for typical low-power embedded devices, which are typically constrained to power consumption in single digit Watts. For CNN inference to be feasible on AM57x SoC, the compute requirement has to come around or below 40 GMAC/sec or 80 GOPs/sec (1 MAC is equivalent to 2 OPs).

With TIDL, TI adopted embedded CNN approaches like efficient CNN configuration, sparsity and fixed-point quantization to develop example network models that can run the inference on C66x cores and EVE subsystems on AM57x SoC.

2.3.1.2.1 Efficient CNN Configuration

When designing network models for embedded processors, consideration needs to be made for inference complexity including computation load, memory usage and data bandwidth (for transfer of weights and activation), such that it fits the need of SoCs on which the model will be deployed to run inference.

While popular networks can run on TIDL, to achieve higher performance (run more frames per sec) out of this low-power embedded device, the network deployed must fit within the capability of the AM57x SoC.

In CNN models, the convolution layers are the most computationally intense and determines how fast the inference runs, so it is important to reduce the complexity of convolution layers. Grouped convolutions and depth-wise separable convolutions are some of the popular techniques that helps reduce the computation cost from convolution layers.

Grouped convolution technique is adopted in the design of example network models JacintoNet11, JSegNet21 and JDetNet. The base classification network in these examples is inspired by ResNet10. Residual connections were removed and groups of four added to every alternate layers to reduce complexity. Grouped convolutions also helps in data bandwidth reduction.

2.3.1.2.2 Sparsity

As mentioned in Section 2.3.1.2.1, in CNN models, the convolution layers are the most computationally intense. When grouped or depth-wise separable convolution by itself does not meet the real-time performance need of end products on a given embedded processor, then sparsity is another tool that can help reduce the computation cost.

Sparsity is a technique used during training to decrease the number of non-zero weights by zeroing out small coefficients. Using sparse convolution algorithms eliminates the need for multiplications whenever the weights are zeros. Sparse training methods can induce 80% or more sparsity in most convolution layers, i.e. making 80% of the convolution weights zero. In this technique, the structure of the network does not change when zeroing out small coefficients.

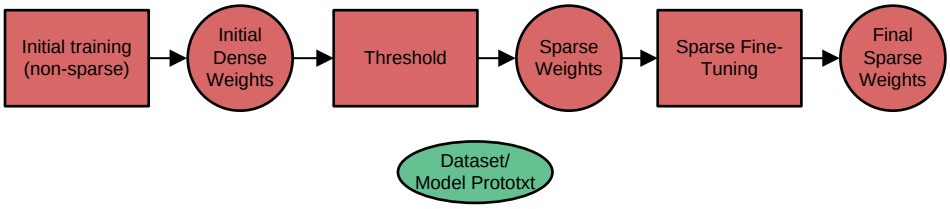
Sparsification at training time is useful only if the inference framework is capable of performing sparse convolutions efficiently. TIDL supports sparse convolution, which can execute the inference much faster when there are a lot of zero coefficients.

Forcing convolution weights to zero can reduce the accuracy of the deployed algorithm. Accuracy lost due to sparsity can be gained back by fine tuning the non-zero coefficients. During the fine tuning of the training stage, coefficients that are already zero are not updated during back propagation update. Training models with sparsity (sparsification) without losing significant accuracy are an important aspect of the training phase.

It is seen from experiments that the classification accuracy drop for a typical CNN network is around 1%, while inducing 80% sparsity. 2× to 4× execution speed increase has been observed when nearly 80% of the weights in the convolution layers are zeros.

Caffe-Jacinto is a good training framework for generating sparse models. Figure 7 shows sparse training using Caffe-Jacinto. First a network is trained following a regular training procedure to get dense weights. Once dense weights are achieved, sparsity is introduced using the thresholding mechanism. After achieving sparse weights, fine tuning of the network is done to regain some of the lost quality due to sparsity.

Figure 7. Caffe Jacinto-Based Sparse Training



The overall training procedure is as follows:

Table 1. Training Procedure

Stage	Steps	Description/comments
1	Pre-training	Pre-training on a large dataset before doing the training on target dataset. This is widely used by semantic segmentation and object detection models. Optional, highly recommended
2	L2-Regularized training	First round of training with L2-regularization on the target dataset before attempting to induce sparsity by L1-regularization. Optional, highly recommended.
3	L1-Regularized training	Technique that promotes sparsity by inducing several small coefficients (weights) and this makes it easy for the thresholding stage late
4	Sparsification using thresholding	Threshold step looks at the Caffe-Jacinto model in a layer-by-layer fashion and try to zero out as many coefficients as specified in the Caffe-Jacinto executable "threshold" options
5	Sparse fine tuning	Recovery of quality lost during thresholding stage by further fine-tuning the non-zero coefficients.
6	Batch Norm Optimization	Merges the batch-norm coefficients into the convolution layers. Subsequent training or testing stages need not use batch-norm layers; is not needed on the inference side

NOTE: Sparsity is optional. TIDL can work with conventional non-sparse models as well.

NOTE: It is recommended to use dense convolution for Input/Feature maps that are smaller than 32x32. Sparse convolution has high overhead for smaller than 32 x 32 convolution.

NOTE: Initial training (non-sparse) block in Figure 7 corresponds to training procedure optional step 1, optional step 2 and step 3 mentioned in Table 1.

2.3.1.3 Quantization

The trained model is a floating-point model. However, floating point is not the best for execution speed on low-power embedded devices. Thus, it is important to convert the floating-point model such that inference execution can use fixed-point operations (with example convolutions done using 8-bit or 16-bit integer multiplications).

TIDL and its device translator tool automatically converts floating point to fixed point so that the training algorithm or framework does not need to do anything special for fixed-point inference in TIDL. This is called on-the-fly quantization, a sophisticated feature that significantly increases execution speed and takes care of varying input-signal characteristics and intermediate layer outputs. TIDL supports both 8-bit and 16-bit quantization. The drop in accuracy due to quantization is small for several popular networks.

2.3.1.4 Translation

The TI device translator tool named as import tool enables development on open frameworks and provides push-button PC-to-embedded porting. The import tool runs on Linux x86 or the Arm Linux port and converts network models developed on TIDL supported frameworks into an internal format suited for use inside the TIDL library. It is built using the protobuf library version 3.2.0rc2.

Following bullets list the features, restrictions and the usage of the import tool:

- The import tool accepts various parameters through import configuration file.
- There are parametric restrictions on certain TIDL supported layer types. These restrictions are documented in the PSDK Linux document under the TIDL Import Process section.
- Import tool can be run on Linux host machine using the following instruction:
  - #tidl_model_import.out.exe <Import configuration File Name>
- On successful execution, it creates two translated binary output files: one for network and other for parameters.

Along with the "import tool", TIDL is packaged with the "viewer" and "simulation" tools.

The viewer tool does visualization of the imported network model. More details can be found here.

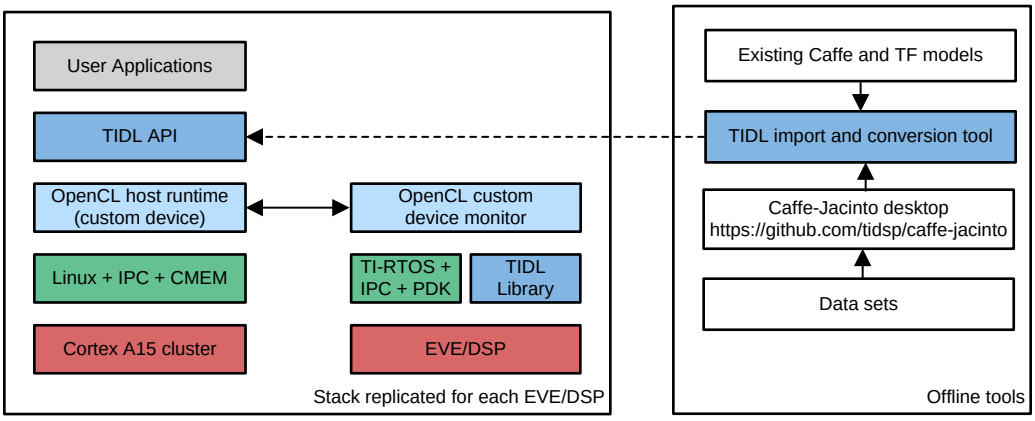
The "simulation tool" is a convenience tool for testing the model on Linux x86 setup. This is bit-exact simulation, so the output of the simulation tool is expected to be identical to the output of the A57x target. It can be used to verify converted model accuracy (FP32 vs 8-bit implementation). Performance of the simulation tool cannot be used to predict the execution time on target EVM.

2.3.1.5 Deployment

Translated network models can be deployed on AM57x SoC using the TIDL APIs. TIDL API is a C++ API on Arm/Linux. The API provides a simple interface to the user and abstracts the mechanics of offloading networks to one or more C66x cores or EVE subsystems. It provides a common host abstraction for user applications across EVEs and C66xs on AM57x.

Figure 8 shows the software stack for deep learning applications on AM57x SoC. TIDL API leverages TI's OpenCL implementation to offload networks to C66xs and EVEs.

Figure 8. TIDL Software Stack



In this example, a configuration object is created from reading a TIDL network configuration file. An executor object is created with two EVE subsystems. It uses the configuration object to setup and initialize TIDL network on EVEs. Each of the two execution objects dispatches TIDL processing to a different EVE subsystem. Because the OpenCL kernel execution is asynchronous, the frames can be pipelined across two EVEs. When one frame is being processed by an EVE, the next frame can be processed by another EVE.

These steps illustrate how easy it is to use TIDL APIs to leverage deep learning in user applications.

1. Determine if there are any TIDL capableOpenCL devices (C66x or EVE) on the AM57x SoC.
2. Create a Configuration object by reading it from a file or by initializing it directly. The listing below parses a configuration file and initializes the Configuration object.
3. Create an Executor with the appropriate device type, set of devices and a configuration.
4. Get the set of available ExecutionObjects and allocate input and output buffers for each ExecutionObject.
5. Run the network on each input frame. The frames are processed with available execution objects in a pipelined manner with additional num_eos iterations to flush the pipeline (epilogue).

```
// Step #2 - Read a TIDL network configuration file
Configuration configuration;
bool status = configuration.ReadFromFile(“./tidl_j11v2_net");

// Step #3 - Create an executor with 2 EVEs and configuration
DeviceIds ids = {DeviceId::ID0, DeviceId::ID1};
Executor executor(DeviceType::EVE, ids, configuration);

// Step #4 - Query Executor for set of ExecutionObjects created
const ExecutionObjects& eos = executor.GetExecutionObjects();
int num_eos = eos.size();  // 2 EVEs

// Allocate input and output buffers for each execution object
for (auto &eo : eos)
{
     ArgInfo in(eo->GetInputBufferSizeInBytes());
     ArgInfo out(eo->GetOutputBufferSizeInBytes());
     eo->SetInputOutputBuffer(in, out);
}

// Step #5 - Pipelined processing with 2 EVEs
for (int idx = 0; idx < configuration.numFrames + num_eos; idx++)
{
    ExecutionObject* eo = eos[idx % num_eos].get();
    // Wait for previous frame on the same eo to finish processing
    if (eo->ProcessFrameWait())  WriteFrameOutput(*eo);

    // Read a frame and start processing it with current eo
    if (ReadFrameInput(*eo, idx))  eo->ProcessFrameStartAsync();
}
```

ReadFrameInput and WriteFrameOutput functions are used to read an input frame and write the result of processing. For example, with OpenCV, ReadFrameInput is implemented using OpenCV APIs to capture a frame. To execute the same network on DSPs, the only change is to replace DeviceType::EVE with DeviceType::DSP.

The TIDL API provides the following important features:

- Multiple network models running concurrently on different EVE subsystems and C66x cores: TIDL API supports running as many different network models concurrently as there are TIDL accelerators available on the AM57x device. In an AM5749 device, the maximum number of concurrent network models would be 4, to match the 4 (2x EVE + 2x C66x DSPs) TIDL accelerators available in the AM5749.
- Split of network model between EVE subsystem and C66x core: Softmax and InnerProduct layers run faster on C66x core compared to EVE subsystem. Network models can be split between C66x core and EVE subsystem to improve upon the performance by following the concept of layer groups.

For details, see the TIDL API User's Guide and out of box TIDL example code included on the filesystem.

2.3.2 TIDL Performance, Accuracy and Power Benchmarking

2.3.2.1 Performance Benchmarking

Mobilenet, SqueezeNet, JacintoNet11, JSegNet21 and JDetNet network models are validated and benchmarked on AM5749 IDK EVM. In this benchmarking setup, EVE is clocked at 650 MHz and C66x is clocked at 750 MHz. The performance number reported in Table 2 is for one EVE Subsystem and one C66x core. It is benchmarked using Processor SDK 5.0. For latest performance number, see the TIDL performance numbers reported in Processor SDK document.

Table 2. TIDL Performance Benchmarking

Network Topology	ROI Size	MMAC (million MAC)	Sparsity (%)	EVE Subsystem Performance (in ms)		C66x core Performance (in ms)		EVE + C66x Performance (in ms)
				Using Sparse Model	Using Dense Model	Using Sparse Model	Using Dense Model	Sparse Model
MobileNet	224x224	567.7	1.42	-	570.78	-	717.11	-
SqueezeNet	227x227	390.8	1.46	-	243.12	-	1008.92	-
InceptionNet V1	224x224	1497.37	2.48	-	655.05	-	2235.99	-
JacintoNet11	224x224	405.81	73.15	105.55	195.69	115.91	370.64	62.06 (~56 ms on EVE + ~6 ms on C66x)
JSegNet21	1024x512	8506.5	76.47	313.15	1014.45	1101.12	3825.95	-
JDetNet	768x320	2191.44	61.84	-	-	-	-	168.05

NOTE: The Dense model refers to model output after L2 regularized training.

MMAC reported in Table 2 is for the dense model.

The Sparsity column helps to estimate the MMAC for sparse model.

C66x core is efficient in processing the Pooling and Softmax layer compared to EVE subsystem.

For the JacintoNet11 EVE + C66x solution, Convolution + Relu layers + Inner product layers are processed on EVE subsystem. Pooling and SoftMax layers are processed on C66x core.

The JDetNet model does not have EVE subsystem or C66x core only solution. Layers are grouped and processing is split between C66x core and EVE subsystem.

TIDL API adds roughly 2 ms of overhead per process call (ProcessFrameStartAsync). For example, JacintoNet11 sparse model will take approximately 107.5 (105.55 + 2 ms) to process one frame.

The numbers reported above include memory access overhead (data input/output) by these layers.

Overall performance of a network model on a particular AM57x SoC depends upon the number of C66x cores and EVE subsystems on that SoC and how they are utilized. For example, on AM5749 IDK EVM, when the network model processing is distributed such that the layers are grouped and split between C66x core and EVE subsystem, JacintoNet11 sparse model can achieve around 32 fps utilizing two EVE subsystem and one C66x core. To understand how to estimate fps, refer to Table 2. We see that JacintoNet11 takes ~62 ms to process one frame on one EVE subsystem + one C66x core. This is equivalent to 16 fps (1000/62). Since C66x core takes only 6 ms to process the layer group, it can handle layer group processing from another frame. With this calculation, when utilizing both the EVE susbsystem and only one C66x core, JacintoNet11 can process 32 fps on AM5749 SoC.

Considering the same JacintoNet11 sparse model as an example, when the processing is configured such that each C66x core and EVE subsystem is handling the processing of the entire frame, JacintoNet11 can process around 4 frames in 116 ms (slower core governs the speed of concurrent processing and C66x core is slower core in this example). This is equivalent to around 35 fps. In this scenario, there will be a latency of 4 captured frames. In case of live image capture, to be able to feed frames to these 4 accelerators simultaneously, the camera capture rate is expected to be 4x of the processing rate.

2.3.2.2 Power Consumption

Table 3 shows power consumption of different network models running on EVE subsystem and C66x cores of AM5749 SoC. The power numbers are benchmarked using out of box Processor SDK for Linux software, which is not optimized for deep learning specific application use cases. All of the cores, IPs and peripherals are getting clocked at its default clock speed.

A15 clocked @1 GHz, DDR clock frequency @666 MHz, C66x@750 MHz, EVE@650 MHz, all IPs including, GPU, IVA, IPUs and other peripherals are clocked.

This measurement is at room temperature and on nominal silicon. SoC power consumption when the AM549 IDK EVM has boot up but no deep learning application is running is 2602.5 mw.

Table 3. Power Consumption of Different Network Models

Network Model	Total SoC Power Consumption (in mw)	C66x/EVE Only Power Consumption (in mw)
JacintoNet11 on 2xEVE cores only	3918.1	216.4
JacintoNet11 on 2xC66x cores only	4186.5	516.0
JSegNet21 on 2xEVE cores only	4122.1	217.3
JDetNet on 2xEVE + 2xC66x	4776.7	339.7

NOTE: Test application used for power benchmarking exercise receives the input images from file and wtites the output to a file (exception JacintoNet11 for which the output is displayed on HDMI monitor). Network File System (NFS) is used and hence Ethernet is running during file read/writes.

Total SoC power consumption in Table 3 includes test overhead related to file read/write, the size of which varies per network model.

Total SoC power reported inTable 3 includes power consumed by C66x cores and EVE subsystems.

For JDetNet model, EVE subsystem and C66x core are running in sequential.

The power numbers reported in Table 3 can be further optimized for specific use cases by adjusting OPPs of cores for required performance, disabling unused IPs, peripherals and cores and adopting other power optimizing techniques.

2.3.2.3 Accuracy

Accuracy of the JacintoNet11, JSegNet21 and JDetNet model is benchmarked for dense (non-sparse) and sparse models. As can be seen from Table 4, there is minimal drop in accuracy of sparse models compared to dense model. The sparse models can perform 2x to 4x faster compared to dense model when the sparsity percentage is around 80%.

- **Image classification**: Table 4 shows Top-1 classification accuracy of JacintoNet11 model on 1000 classes of ImageNet data base. More details on JacintoNet11 accuracy benchmarking setup can be found here.

Table 4. JacintoNet11 Accuracy Results

Configuration-Dataset Imagenet (1000 classes)	Top-1 Accuracy (%) (1)
JacintoNet11 non-sparse	60.9%
JacintoNet11 layer wise threshold sparse (80%)	57.3%
JacintoNet11 channel wise threshold sparse (80%)	59.7%

(1) Top-1 classification accuracy indicates probability that ground truth is ranked highest.

For reference, the Top-1% and Top-5% accuracy of known popular network models are listed in Table 5. This table is shown for the comparison of the topology, irrespective of the implementation.

Table 5. JacintoNet11 Accuracy Result Comparison with Known Network Models

Configuration	ROI Size	Top-1 Accuracy(%)	Top-5 Accuracy(%)	Complexity for 1000 Classes (GigaMACS)
ResNet10 [1]	224x224	63.9	85.2	0.910
BVLC AlexNet [2]	227x227	57.1	80.2	0.687
JacintoNet11	224x224	60.9	83.05	0.410

NOTE:

- ImageNet pre-trained models with batch normalization:
  - https://arxiv.org/pdf/1612.01452.pdf
- BVLC/caffe/models/bvlc_alexnet:
  - https://github.com/BVLC/caffe/tree/master/models/bvlc_alexnet

- **Image segmentation**: Table 6 captures the JSegNet21 model trained on Cityscapes database accuracy for 5-class (background, road, person, road, signs, vehicle). Mean Intersection Over Union (IOU) is the ratio between True Positives and sum of True Positives, False Negatives and False Positives. More details on JSegNet21 benchmarking setup can be found here.

Table 6. JSegNet21 Accuracy Results

Configuration-Dataset Cityscapes (5-classes)	Pixel Accuracy	Mean IOU (%)
Initial L2 regularized training	96.20%	83.23%
L1 regularized training	96.32%	83.94%
Sparse fine tuned (~80% zero coefficients)	96.11%	82.85%
Sparse (80%), Quantized (8-bit dynamic fixed point)	95.91%	82.15%

NOTE: Regularizer are kind of a constraining function.

L2 regularizer tries to minimize the sum of the square of the weights.

L1 regularizer tries to minimize the sum of the absolute value of the weights.

L1 regularizer has a natural tendency to have only few large coefficients, leaving most of them as very small. This is useful step for training network with sparsity.

- **Object Detection** : Table 7 captures accuracy results for the JDetNet model trained on the PASCAL VCOCO712 database. Validation accuracy is reported in mean Average Precision (mAP). More details on the JDetNet model accuracy benchmarking setup can be found here.

Table 7. JDetNet Accuracy Results

Configuration-Dataset VOC0712	mAP (%)
Initial L2 regularized training	68.66%
L1 regularized fine tuning	68.07%
Sparse fine tuned (~61% zero coefficients)	65.77%

3 Hardware, Software, Testing Requirements, and Test Results

3.1 Required Hardware and Software

3.1.1 Hardware

TIDL can run on any TI AM57xx board supported by Processor SDK Linux as it can run on DSPs or EVEs. In this design document, the AM5749 IDK EVM is used to demonstrate the required hardware and software steps.

The following hardware setup is needed to run the TIDL applications:

- AM5749 IDK EVM
- Power supply
- LCD panel or HDMI monitor along with HDMI cable
- micro SD card

3.1.2 Software

TI Deep Learning is included in the AM57x Processor SDK for Linux. The Processor SDK Linux provides a fundamental software platform for development, deployment, and execution of Linux-based applications. The package contains a software user's guide and additional documentation for setting up and running the demonstration applications.

3.2 Testing and Results

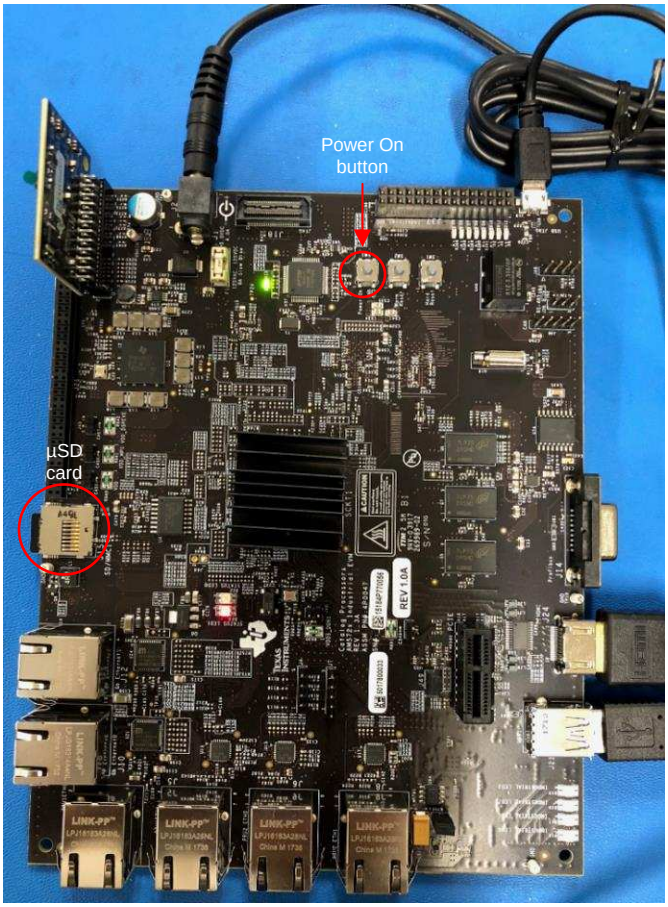
3.2.1 Test Setup

3.2.1.1 Hardware Test Setup

1. Flash the micro SD card following the directions in Software Test Setup Section 3.2.1.2.
2. Insert the flashed micro SD card in the board, connect the USB mouse, HDMI cable and power cable. Connect other end of the HDMI cable to the HDMI monitor. Press the "power on" button on the board.

Figure 9 shows the AM5749 hardware setup connected to HDMI monitor.

Figure 9. AM574x IDK EVM



NOTE: If using the LCD panel as a display, connect the panel to the IDK EVM using the steps mentioned in this video.

A USB mouse is needed when the HDMI monitor is used for display. The Weston manager will not run if there is no input device detected, hence the need for mouse.

3.2.1.2 Software Test Setup

1. Download the package from http://www.ti.com/tool/PROCESSOR-SDK-AM57X.
   a. Download the SDK installer from here. Use http://www.ti.com/tool/PROCESSOR-SDK-AM57X, with description field: “AM57xx EVM Linux SDK (64-bit Binary)“.
   b. Create the SD card with default images using the SDK Create SD Card Script
2. Getting started guide can be found here.

3.2.2 Test Results

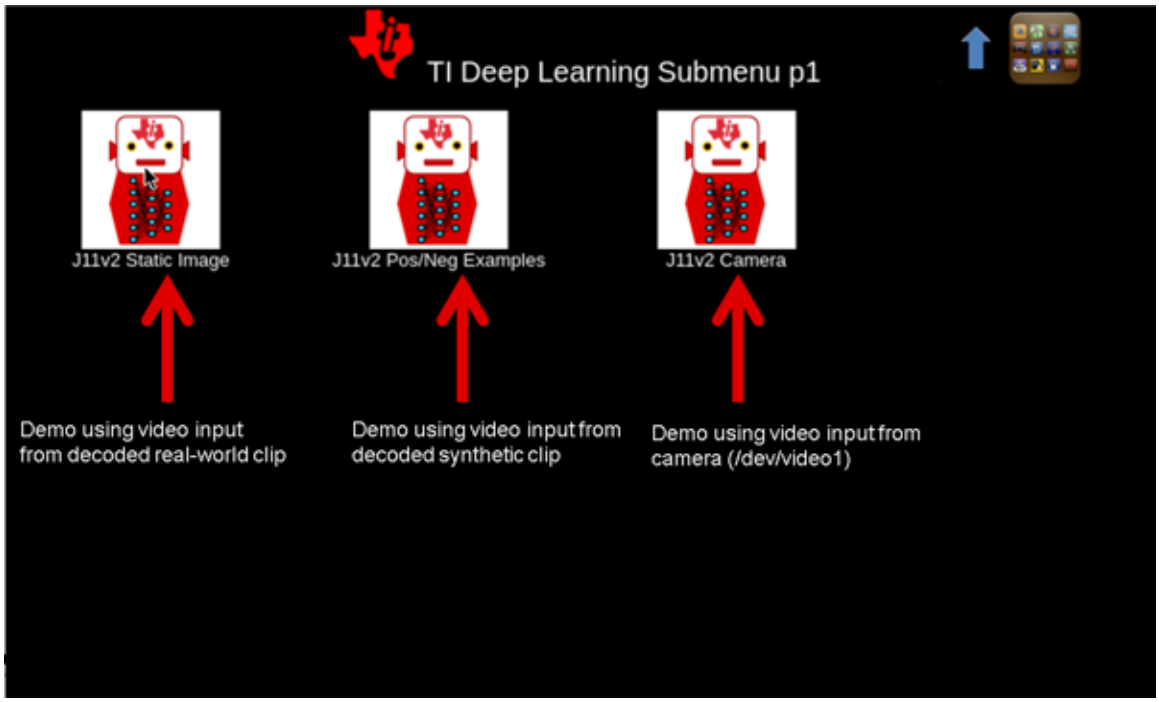
Once the board has boot up, the Matrix-GUI application will be launched and the screen shown in Figure 10 will be seen on the display panel or monitor.

1. Select the TI Deep Learning demo from Matrix-GUI.

Figure 10. AM57xx Matrix-GUI Launcher With TIDL Submenu Button

2. Once that is selected, the following screen will be seen.

Figure 11. TIDL Submenu



This submenu is from the processor SDK 5.0 release. In this release, all three buttons are using the image classification model JacintoNet11 inference run, but with different input source. In future releases, examples with image segmentation and object detection will be added in the Matrix under "TI Deep Learning" icon.

In this submenu, video inputs for the JacintoNet11 application can come from either:

- Camera /dev/video1 (demo started with: /usr/bin/runTidlLiveCam.sh), default resolution 640 x 480
- Decoded real-world video clip, /usr/share/ti/tidl/examples/classification/clips/test1.mp4 (demo started with: /usr/bin/runTidlStaticImg.sh), 320 x 320 resolution, H264 encoded
- Decoded synthetic video clip, /usr/share/ti/tidl/examples/classification/clips/test2.mp4 (demo started with: /usr/bin/runTidlPnExamples.sh), 320 x 320 resolution, H264 encoded

Since JacintoNet11 model expects input images at 224 x 224 size, original input is resized and central cropped to 224 x 224 (to preserve the aspect ratio).

3. Select the J11v2 static image icon. The image shown in Figure 12 will be seen on the screen.

Figure 12. J11v2 Static Image Application Description

4. Click the run button. The run button invokes the /run/bin/runTidlStaticImg.sh script. Content of the script looks like that shown below. Argument settings for the tidl_classification application are explained later in this section.

```
root@am57xx-evm: cat /run/bin/runTidlStaticImg.sh script
./tidl_classification -n 2 -t d -l ./imagenet.txt -s ./classlist.txt -i ./clips/test1.mp4 -
c ./stream_config_j11_v2.txt & pid=$!
```

NOTE: The out of box command line argument settings for all three tidl_classification application binary launched through Matrix-GUI sets the core type as C66x DSP core (-t is set as d). No EVE subsystem is utilized to make the example generic that can be run on any AM57x SoC.

Better performance of the model can be seen when both C66x cores and EVE subsystems are utilized. The TIDL application for object classification utilizing both C66x DSP core and EVE subsystem will be part of Processor SDK 5.1 release.

5. The "tidl_classification" application in the example "runTidlStaticImg.sh" script reads pre-recorded 320 x 320 resolution H.264 encoded test1.mp4 clip, decodes it on IVA-HD hardware accelerator and sends the decoded image to C66x cores for image classification. The image is detected, classified and labeled as shown in Figure 13. Along with the window displaying the classified object, two other windows will be seen: one of the window shows the TIDL Software Stack and another one shows the class list. The "ClassList" window also displays network model performance.

Figure 13. Classification in Progress ("J11v2 StaticImage" button)

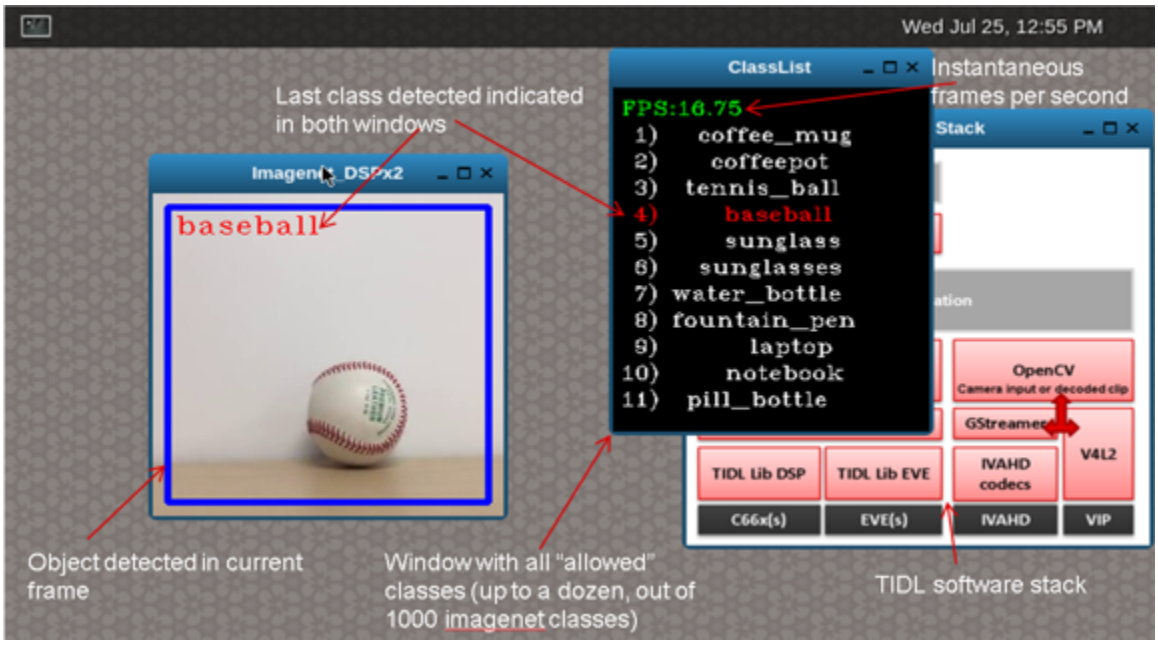
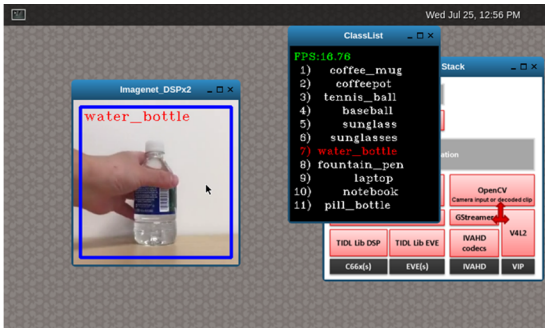
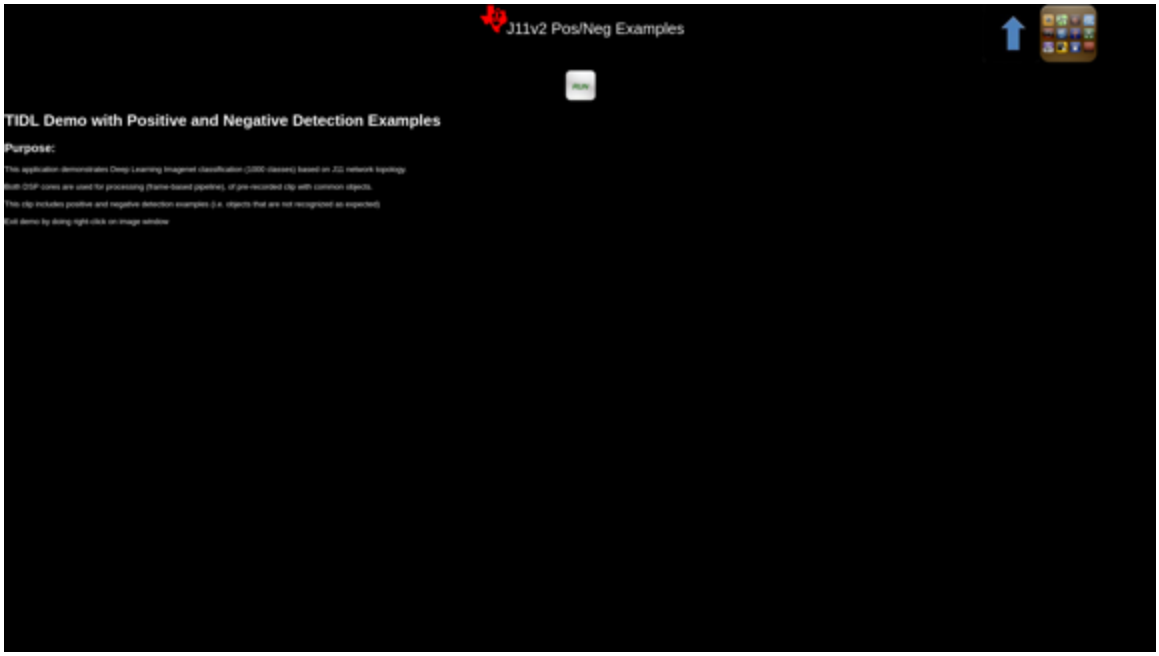


Figure 14. Additional Objects Detected



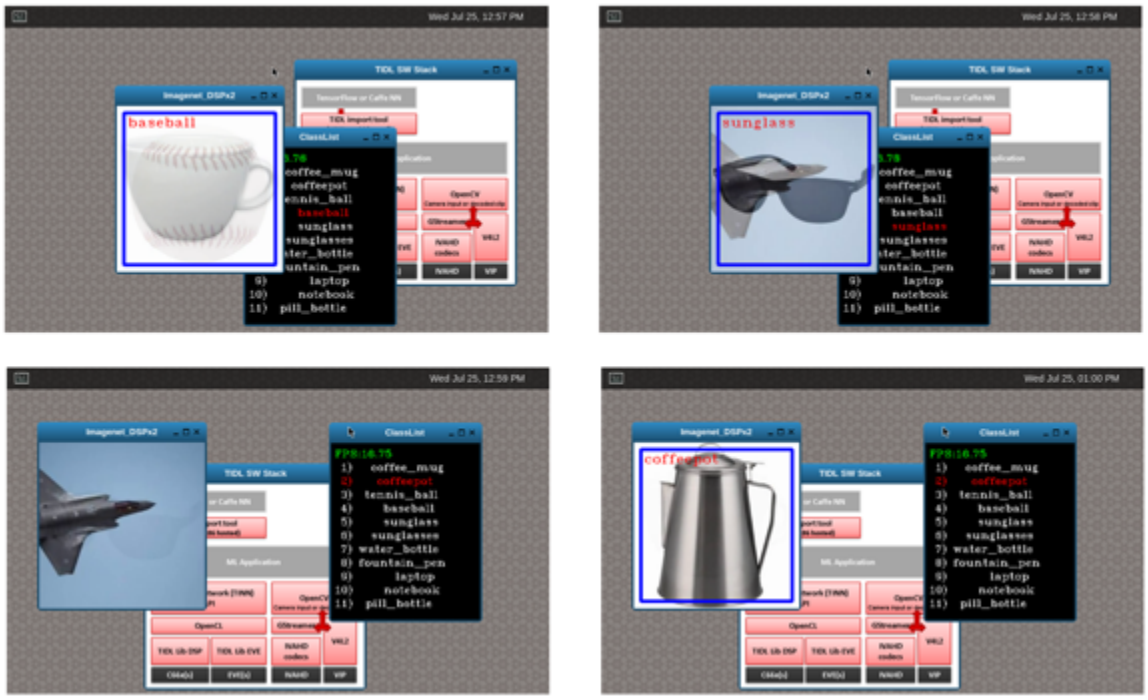
6. At the end of demo, after going back to Matrix-GUI display, select the TIDL submenu and then the second button: "J11v2 Pos/Neg Examples". The screen shown in Figure 15 will be seen. Click the Run button.

Figure 15. j11v2 Pos/neg Examples



It will start playing and classifying the pre-recorded /usr/share/ti/tidl/examples/classification/clips/test2.mp4 clip.

Figure 16. Classification Using Synthetic Video Clip ("J11v2 PnExamples" button)



7. Positive detection are on three windows as shown in the example above, where as lower left window ("plane") is without detection ("blue" rectangle is missing) - since airplane class is not included in classlist.txt.
8. Once started, each demo runs for approximately 90 seconds and stops, returning to the main Matrix-GUI display. It is possible to stop the demo sooner than that, by doing right-click on the main image window (Window title "Imagenet_XXX").
9. In order to experiment with this program, you can login to the EVM terminal and go to the /usr/share/ti/tidl/examples/classification folder. First you need to stop matrix-gui: /etc/init.d/matrix-gui stop.
10. The "tidl_classification" application demos in the matrix-gui TIDL are based on the TIDL API. It has the command line arguments as shown in code section below.

```
root@am57xx-evm:/usr/share/ti/tidl/examples/classification#./tidl_classification -h
Usage: tidl
  Will run all available networks if tidl is invoked without any arguments.
  Use -c to run a single network.
Optional arguments:
 -c                   Path to the configuration file
 -n <number of cores> Number of cores to use (1 - 4)
 -t <d|e>             Type of core. d -> DSP, e -> EVE
 -l                   List of label strings (of all classes in model)
 -s                   List of strings with selected classes
 -i                   Video input (for camera:0,1 or video clip)
 -v                   Verbose output during execution
 -h                   Help
```

11. The demo can be started with the command:

    ./tidl_classification -n 2 -t d -l ./imagenet.txt -s ./classlist.txt -i ./clips/test1.mp4 -c ./stream_config_j11_v2.txt

12. To run the application on a different video clip, provide the path to the video clip after "-i" argument. For live camera input, provide "1" as input after "-i" (device node index from /dev/video1).
13. A ist of “allowed” classes can be edited in classlist.txt file by adding or deleting classes listed in imagenet.txt. Both these files "classlist.txt" and "imagenet.txt" are located in the "/usr/share/ti/tidl/examples/classification" folder on the target file system.
14. Note that imagenet.txt should not be changed unless you are using a different model, which is defined in stream_config_j11_v2.txt file:

```
root@am57xx-evm:/usr/share/ti/tidl/examples/classification#cat stream_config_j11_v2.txt
numFrames   = 9000
inData   = /usr/share/ti/tidl/examples/test/testvecs/input/preproc_0_224x224.y
outData   = "/usr/share/ti/tidl/examples/classification/stats_tool_out.bin"
netBinFile      =
"/usr/share/ti/tidl/examples/test/testvecs/config/tidl_models/tidl_net_imagenet_jacintonet11v2.
bin"
paramsBinFile   =
"/usr/share/ti/tidl/examples//test/testvecs/config/tidl_models/tidl_param_imagenet_jacintonet11
v2.bin"
inWidth = 224
inHeight = 224
inNumChannels = 3
```

3.2.2.1 Additional TIDL Examples

There are more file read write based examples that cannot be started directly from Matrix-GUI, but can be easily accessed from the command line.

- Image segmentation: /usr/share/ti/tidl/examples/segmentation
  - Pixel level segmentation creating output images with class indices (0-4) for each pixel. The model used is version of JSegNet21 model.

  Below is a snippet of the logs from the segmentation application run:

```
root@am57xx-evm:/usr/share/ti/tidl/examples/segmentation# ./segmentation -
i ../test/testvecs/input/roads/pexels-photo-972355.jpeg
Input: ../test/testvecs/input/roads/pexels-photo-972355.jpeg
frame[0]: Time on device:    252ms, host:  259.3ms API overhead:   2.82 % Saving frame 0
overlayed with segmentation to: overlay_0.png
segmentation PASSED
```

- Object detection: /usr/share/ti/tidl/examples/ssd_multibox
  - Object detection creating a list of bounding boxes: xmin, ymin, xmax, ymax,class index (0-3) and probability. The mode used is version of JDetNet model.

  Below is a snippet of the logs from the ssd_multibox application application run:

```
root@am57xx-evm:/usr/share/ti/tidl/examples/ssd_multibox#./ssd_multibox -
i ../test/testvecs/input/roads/pexels-photo-378570.jpeg
Input: ../test/testvecs/input/roads/pexels-photo-378570.jpeg
frame[0]: Time on EVE:  147.3ms, host:  151.4ms API overhead:   2.66 %
frame[0]: Time on DSP:  22.38ms, host:  23.73ms API overhead:   5.71 %
Saving frame 0 with SSD multiboxes to: multibox_0.png
Loop total time (including read/write/print/etc):  485.5ms
ssd_multibox PASSED
```

Input and output images of object detection and image segmentation:

Figure 17. "ssd_multibox" Input Image

Figure 18. "ssd_multibox" Output With Detected Objects of Various Classes

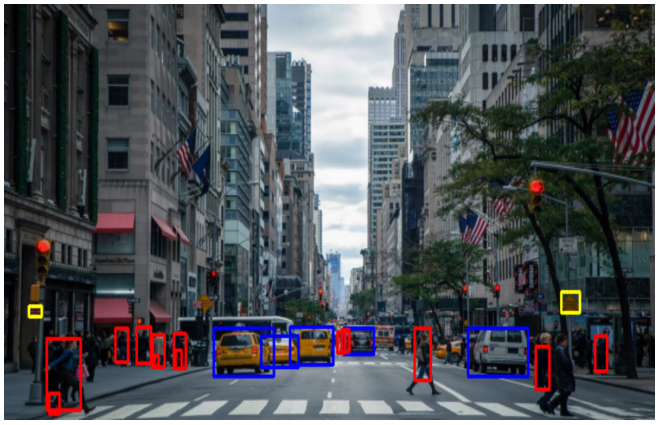


Figure 19. "segmentation" Input Image

Figure 20. "segmentation" Output With Pixel Level Classification

4 Design Files

4.1 Schematics

To download the hardware design files for the AM5749 IDK EVM, see the design files at TIDEP-01004.

5 Software Files

Download the Processor SDK Linux for AM57x from the AM57x software product page.

6 Related Documentation

1. TI Deep Learning on Processor SDK Linux
2. TIDL API
3. Caffe-Jacinto framework
4. Caffe-Jacinto models
5. CVPR Paper for Sparse Convolution

6.1 Trademarks

7 Terminology

AI - Artificial Intelligence

CNN - Convolution Neural Network

DNN - Deep Neural Network

DSP - Digital Signal Processor

EVE- Embedded Vision Engine

IOU - Intersection Over Union

mAP - mean Average Precision

RNN - Recurrent Neural Network

SoC - System on Chip

SIMD - Single Instruction Multiple Data

TIDL - TI Deep Learning

VLIW - Very Long Instruction Word

8 About the Author

MANISHA AGRAWAL is Lead Applications Engineer with the Catalog Processor Group. She has worked at TI since 2006. Her recent roles focus on growing and supporting Sitara line processors for multimedia, machine vision, and industrial applications.

DJORDJE SENICIC is a Senior Software Engineer with the Catalog Processor Group. His recent roles focus on Processor Linux software packages for Sitara an K2 devices, including machine vision applications.

MARK NADESKI is a Business Development Manager in TI's Processors Business Unit. He has worked at TI since 1992. His current focus is on machine vision.

IMPORTANT NOTICE AND DISCLAIMER

TI PROVIDES TECHNICAL AND RELIABILITY DATA (INCLUDING DATASHEETS), DESIGN RESOURCES (INCLUDING REFERENCE DESIGNS), APPLICATION OR OTHER DESIGN ADVICE, WEB TOOLS, SAFETY INFORMATION, AND OTHER RESOURCES “AS IS” AND WITH ALL FAULTS, AND DISCLAIMS ALL WARRANTIES, EXPRESS AND IMPLIED, INCLUDING WITHOUT LIMITATION ANY IMPLIED WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE OR NON-INFRINGEMENT OF THIRD PARTY INTELLECTUAL PROPERTY RIGHTS.

These resources are intended for skilled developers designing with TI products. You are solely responsible for (1) selecting the appropriate TI products for your application, (2) designing, validating and testing your application, and (3) ensuring your application meets applicable standards, and any other safety, security, or other requirements. These resources are subject to change without notice. TI grants you permission to use these resources only for development of an application that uses the TI products described in the resource. Other reproduction and display of these resources is prohibited. No license is granted to any other TI intellectual property right or to any third party intellectual property right. TI disclaims responsibility for, and you will fully indemnify TI and its representatives against, any claims, damages, costs, losses, and liabilities arising out of your use of these resources.

TI’s products are provided subject to TI’s Terms of Sale (www.ti.com/legal/termsofsale.html) or other applicable terms available either on ti.com or provided in conjunction with such TI products. TI’s provision of these resources does not expand or otherwise alter TI’s applicable warranties or warranty disclaimers for TI products.

Mailing Address: Texas Instruments, Post Office Box 655303, Dallas, Texas 75265