BONSEYES

ARTIFICIAL INTELLIGENCE MARKETPLACE

Deliverable D3.5

Initial Platform Deployment Methods and Tools

Point of Contact	Athena Elafrou
Institution	ICCS
E-mail	athena@cslab.ece.ntua.gr
Phone	+306947575203

Project Acronym	**BONSEYES**
Project Title	**Platform for Open Development of Systems of Artificial Intelligence**
Grant Agreement No.	**732204**
Topic	**H2020-ICT-2016 Smart Cyber-Physical Systems**
Project start date	**1 December 2016**
Nature	**Other**
Dissemination level	**Public**
Due date	**M18**
Date of delivery	**M18**
Lead partner	**ICCS**
Contributing partners	**UCLM, TCD, UEDIN, HES-SO, ARM, RTRK**
Authors	**Athena Elafrou (ICCS), Georgios Goumas (ICCS)**
Reviewers	**Itsik Arbel (NVISO), Daniel Ostler (TUM)**

Schweizerische Eidgenossenschaft
Confédération suisse
Confederazione Svizzera
Confederaziun svizra

Swiss Confederation

Federal Department of Economic Affairs,
Education and Research EAER
State Secretariat for Education,
Research and Innovation SERI

This project has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No 732204 (Bonseyes).

This work is supported by the Swiss State Secretariat for Education, Research and Innovation (SERI) under contract number 16.0159.

The opinions expressed and arguments employed herein do not necessarily reflect the official views of these funding bodies.

Revision History

Version	Date	Author	Comment
0.1	15 Feb 2018	Athena Elafrou	Table of Contents
0.7	9 May 2018	Athena Elafrou	First draft, submission for consortium review
0.8	24 May 2018	Athena Elafrou	Second draft with added content
0.9	30 May 2018	Itsik Arbel	Quality assurance
1.0	31 May 2018		Final draft

Contents

Abbreviations, Participant short names and Glossary

Abbreviations

AI	**Artificial Intelligence**
CPS	**Cyber-Physical Systems**
IoT	**Internet of Things**
DNN	**Deep Neural Network**
CNN	**Convolutional Neural Network**
LPDNN	**Low Power Deep Neural Network**
SDK	**Software Development Kit**
SSD	**Single Shot Multibox Detector**
GEMM	**General Matrix Matrix Multiply**
GEMV	**General Matrix-Vector Multiply**
SpGEMM	**Sparse Matrix - Dense Matrix Multiply**
SpMV	**Sparse Matrix - Dense Vector Multiply**

Participant short names

NVISO	**nViso SA**
UCLM	**Universidad de Castilla – La Mancha**
TCD	**Trinity College Dublin**
UEDIN	**University of Edinburgh**
FHNW	**Fachhochschule Nordwestschweiz**
TUM-Med	**Klinikum rechts der Isar der Technischen Universität München**
ICCS	**Institute of Communication and Computer Systems**
SYNYO	**Synyo GmbH**
HES-SO	**Haute Ecole Spécialisée de Suisse Occidentale**
ARM	**ARM Limited**
ZFFNAG	**ZF Friedrichshafen AG**
RTRK	**Institut RT-RK**
SCIPROM	**SCIPROM Sàrl**
BTH	**Blekinge Tekniska Högskola**

Glossary

The Bonseyes glossary is available on https://www.bonseyes.com/glossary/.

List of Tables

List of Figures

Summary

This document provides an overview and evaluation of the current version of the *Low-Power Deep Neural Network* (LPDNN) inference framework, which is a software package that incorporates platform deployment methods and tools that are being developed as part of WP3 of the Bonseyes project, LPDNN aims to enable the deployment of deep learning models on resource-constrained, embedded platforms. LPDNN will be used in the realisation of the project's demonstrators in the frame of WP5.

1. Introduction

The main goal of the Bonseyes project is to develop a platform, the *Bonseyes platform*, consisting of an *Artificial Intelligence (AI) Marketplace*, a *Deep Learning Toolbox* and *Developer Reference Platforms* for organizations aiming to adopt AI in low-power Internet-of-Things (IoT) devices, embedded computing systems, or datacentre servers. The Deep Learning Toolbox will incorporate methods and tools developed in the context of WP3 of the Bonseyes project that enable the creation of Deep Neural Network (DNN) models that are tailored for the target platform and the optimal execution of these models on selected Developer Reference Platforms (Bonseyes WP4). This document introduces a component of the Deep Learning Toolbox, namely the *Low-Power Deep Neural Network* (LPDNN) inference framework, which is a software package that includes platform deployment methods and tools that enable high-performance execution of DNN models on resource-constrained devices. LPDNN plays a significant role in fulfilling the goals of the Bonseyes platform.

LPDNN is responsible for (a) generating code for AI DNN-based inference that is optimized for the selected developer platform, (b) easing the integration of the model into an AI application and (c) automating the deployment of the model on the target platform. Furthermore, it provides benchmarking tools that enable the analysis of model accuracy and inference performance in terms of execution time, memory requirements, etc. These tools serve multiple purposes, including provisioning of data for performance auto-tuners that will automate the end-to-end model optimization workflow in LPDNN, quality assessment of DNN models that are published on the AI Marketplace, and the realisation of a feedback loop between low-level software engineers and data scientists that allows the iterative refinement of models published on the AI Marketplace.

From a technical perspective, the LPDNN inference framework has been designed to:

- Run even on a bare-metal platform with no file system support.
- Support multiple accelerators such as DSPs, SIMD engines, embedded GPUs that are generally present in today's embedded SoCs.
- Integrate third-party deep learning acceleration libraries as plugins.
- Allow the automated performance tuning of DNN inference through optimal selection of DNN primitives.
- Efficiently support quantized and/or sparse DNNs that are tailored for resource-constrained devices.

The combination of these features differentiates LPDNN from other deep learning frameworks, such as TensorFlow Lite [1], TensorRT [2] or Caffe2 [3], which typically support a subset of these features.

This document assumes basic familiarity with deep learning concepts and is structured as follows:

- Section 2 describes the design and features available in the current version of the LPDNN framework.
- Section 3 demonstrates how LPDNN can be used to integrate DNNs in an AI application.
- Section 4 presents an evaluation of the LPDNN framework on selected developer platforms.

2. LPDNN Inference Framework

This section provides an overview of the LPDNN framework and a description of the deployment methods and tools that are currently implemented. LPDNN is organized into (see Figure 1):

- A *core library* that implements all the required functionality for DNN inference.
- *Tools* that enable model importing and optimization.
- A *software development kit* (SDK) that handles all the sample applications, testing and benchmarking.

LPDNN core is dependency-free and uses a plugin architecture to support heterogeneous hardware, vendor optimized libraries, and dependency code paths. This allows LPDNN to run on a diverse set of devices ranging from small ultra-low power bare-metal systems up to high-end datacentre servers. Complementary tools enable importing pre-trained DNN models, analysing their memory and computational requirements, quantizing them, optimizing inference code for the target platform, etc. The SDK supports model testing and optimization through debugging and benchmarking tools. Benchmarking results produced by the SDK will be published in the AI Marketplace of Bonseyes (Bonseyes WP2) through feedback loop mechanisms to enable the iterative improvement of the quality and performance of AI systems.

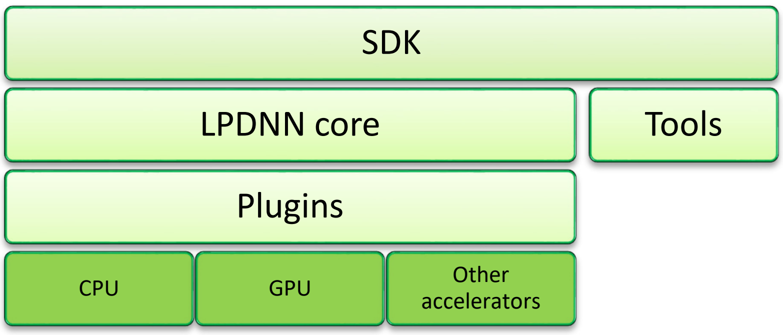


Figure 1. LPDNN software stack

In a typical use-case of LPDNN, one would first import a DNN model that has been trained using the Deep Learning Toolbox of the Bonseyes platform (or another deep learning framework) into LPDNN core to generate a compiled version of the model along with optimized inference code, and then use the SDK to develop, test and benchmark an AI application that integrates the executable model. A simple example of this workflow is given in Section 3.

2.1 DNN Architecture Support

DNN architectures are typically comprised of several distinct layers. In the following, we distinguish between layers that are *parameter-free,* i.e. they do not have any trainable parameters, and layers that require trainable parameters. LPDNN implements layers commonly found in state-of-the-art DNNs.

2.1.1 Layers with trainable parameters

- *Convolutional*: This layer is the basic building block of Convolutional Neural Networks (CNNs), a category of neural networks that has been designed to exploit spatially local correlation in images in order to reduce the number of learnable parameters. This is achieved by enforcing a local connectivity pattern between neurons of adjacent layers: each neuron is connected only to a small subset of the neurons in the previous layer, called its receptive field. The layer's parameters form a set of filters that is convolved over the input in a sliding window fashion to perform feature extraction.
- *Fully-Connected (or Inner Product)*: This layer is typically used at the end of a neural network to perform the high-level reasoning for the AI task. For instance, in image classification using CNNs, a fully-connected layer at the end of the network determines which high-level features most strongly correlate to a particular class. During training, its parameters learn to produce the correct probabilities for the different

classes. Neurons in a fully connected layer have connections to all neurons in the previous layer, leading to a large number of parameters.
- *Batch Normalization* [4]: This layer is used to normalize the activations in hidden layers of the network to reduce internal covariance shift. For DNN deployment, the parameters of this layer are typically merged with the parameters of the previous layer to improve inference performance.
- *Scale*: This layer scales and shifts the elements of the input tensor and may be used to implement Batch Normalization.

2.1.2 Parameter-free layers

- *Pooling*: The role of this layer is to progressively reduce the spatial size of the representation to reduce the number of parameters in the network, and hence to also control overfitting. LPDNN supports *max* and *average* pooling.
- *Activation*: This layer determines which neurons should be activated or not. Whether the information that a neuron is receiving is relevant for the given information or should it be ignored. This is done by applying a non-linear transformation, called an *activation function*, over the input signal. LPDNN implements the ReLU, Leaky-ReLU, Sigmoid and TanH activation functions.
- *Local Response Normalization*: This layer normalizes the activations of hidden layers in the network to improve generalization of the model by implementing a form of lateral inhibition inspired by the type found in real neurons, creating competition for big activities amongst neuron outputs computed using different kernels [5].
- *Element-wise Sum*: This layer computes the element-wise weighted sum of the input tensors with user-specified coefficients.
- *Concatenation*: This layer concatenates tensors along one dimension.
- *Loss*: The loss layer specifies how training penalizes the deviation between the predicted and true labels and is normally the final layer. The loss function may differ depending on the task. LPDNN implements the SoftMax loss, which is typically used for predicting a single class of N mutually exclusive classes.
- *Layers of Single-Shot-Multibox-Detector (SSD)*: SSD is a popular framework for object detection that uses a single CNN to provide both bounding boxes and classifications [6].

Any DNN model using a combination of the above parametrical and parameter-free layers can be deployed using LPDNN. New layers can be added by extending the core library and implementing the layer in a new or existing plugin. The following popular DNN architectures are supported out-of-the box:

- AlexNet [5]
- VGG [8]
- GoogLeNet [9]
- MobileNet [7]
- SqueezeNet [10]
- ResNet [11]

2.2 Core Library

LPDNN contains a core module comprising a set of dependency-free functions implemented in ANSI C that supports inference of neural networks on any embedded device. The core module is complemented by a set of plugins that can be built together with the core to produce optimized code for a specific computing system. Each plugin can make use of other available computing libraries. The plugin-based architecture allows the main core to remain small and dependency-free, while additional libraries are only included when needed and for specific platforms, notably increasing the portability across systems. Furthermore, cross-compilation and platform-specific tools are added to support a wide range of heterogeneous computing platforms such as CPUs, GPUs, DSPs, VPUs. The core is able to support multiple plugins at the same time; this flexibility makes it easy to write and experiment with optimized algorithms for some specific layer types and to dispatch the execution of each layer to the most suitable implementation according to the network architecture, target platform and desired accuracy and performance specification. The previous goals can be achieved by providing input directives to the code generation tool, to specify the plugin, computing library and algorithm to be used for each layer. For instance, one may choose to deploy all Convolutional layers on the GPU and the rest on the CPU. In this way, efficient end-to-end inference code is generated which is then natively compiled or cross-compiled and linked to the specific embedded platform libraries.

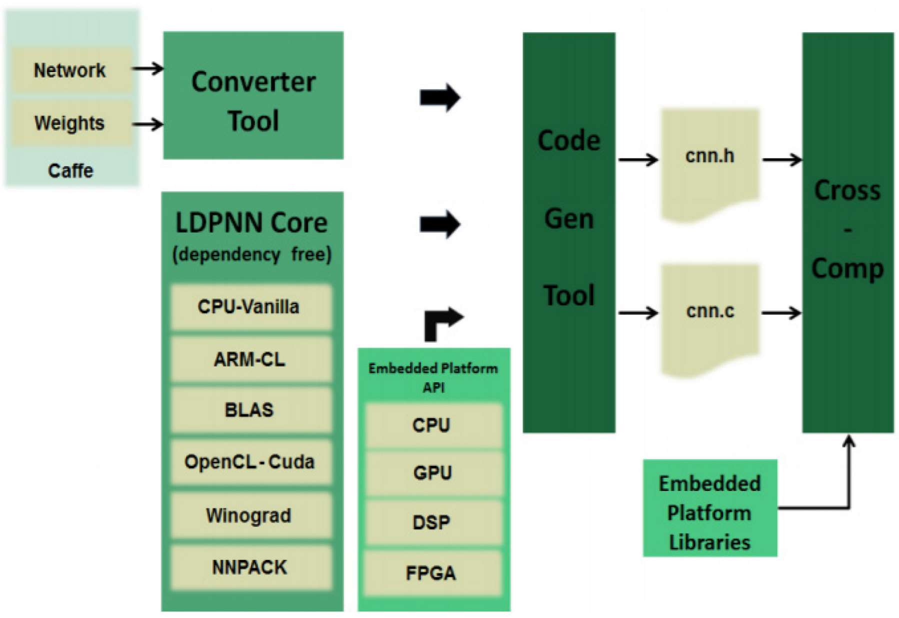


Figure 2. LPDNN core library overview

2.3 Plugins

Deep Learning is an area of intensive research that produces a rapid succession of innovations and technologies. Researchers are continuously developing new algorithms, data encodings, and libraries. Device manufacturers are building new platforms and adapt hardware and protocols to the latest insights about how to solve deep learning tasks. To embrace the rapidly evolving variety of technologies, Bonseyes takes the approach of designing the Deep Learning Toolbox (see Section 1) with a plugin architecture so that libraries and tools can be easily integrated and deployed.

LPDNN has been designed to facilitate the integration of optimized implementations of DNN layers, which we will henceforth refer to as *primitives*, through the plugin architecture illustrated in Figure 2. Primitives may be implemented directly in LPDNN or indirectly through third-party libraries or libraries developed in the context of Bonseyes WP3. In any case, the primitives are provided through a plugin that complies to a specific API, defined by the LPDNN core. A plugin may implement any number of layers currently supported in LPDNN (see Section 2.1). Support for heterogeneous hardware comes through hardware-specific plugins. LPDNN currently includes plugins for CPUs from multiple vendors and NVIDIA GPUs. DNN inference code can use layer implementations from a single or multiple plugin. This enables heterogeneous computing as well as optimal primitive selection.

2.3.1 Plugins for Dense DNNs

The following plugins use dense computations to perform the inference and should typically be used for dense DNNs, i.e., DNNs with no or low levels of sparsity in the model parameters.

2.3.1.1 CPU

2.3.1.1.1 cpu_vanilla

This plugin provides naive, dependency-free implementations of all layers currently supported in LPDNN (see Section 2.1). These are the default implementations and can be overridden by other plugins. This plugin is typically used for layers that are not critical to the inference performance. It is also used for testing purposes.

2.3.1.1.2 cpu_blas

This plugin includes optimized implementations of the Convolutional and Fully-Connected layers using high-performance matrix-matrix and matrix-vector multiplication operations, which we will henceforth refer to as GEMM and GEMV respectively, found in BLAS libraries.

The implementations of the Convolutional layer follow the widely-used approach of lowering the convolutions into a GEMM operation by appropriately reshaping the tensors involved in the operation. This approach is most effective when it creates large matrices for multiplication, since GEMM is characterized by a computational intensity, i.e., ratio of floating-point operations per byte of data, that increases as the matrices get larger. This plugin implements multiple reshaping methods, with different strengths and weaknesses, including the popular *im2col* transformation [13], as well as the *im2row*, *kn2col*, and *kn2row* variations [14]. Any implementation following this approach typically involves a lowering step, which transforms the input, weight and/or output tensors appropriately using auxiliary memory, a GEMM operation that performs the bulk of the work, and a lifting step that converts the output of the previous step into the layout required by the next layer. In the following, *K* will refer to the rows/columns of the convolution filter.

- im2X methods:
  This family of lowering methods transform the input tensor into a matrix by copying patches of the input and unrolling them into columns (im2col) or rows (im2row) of this intermediate matrix. This implies that input data is replicated, increasing the memory requirements over a direct implementation by a factor of K^2. Therefore, this family of methods may not be appropriate for memory-constrained embedded systems. A more detailed description can be found in [13] [14].
- kn2X methods:
  This family of lowering methods focuses on eliminating the data replication issue of the im2X methods by expressing a KxK convolution as the sum of K^2 1x1 convolutions, driven by the fact that a 1x1 convolution can be implemented using a GEMM operation without replicating any input data. The convolutional layer consists of a simple lowering of tensors to matrices, followed by a GEMM operation and, finally, a lifting step that properly sums each of the K^2 submatrices to compute the result. These methods still incur a memory overhead of K^2 compared to direct methods and may also reduce the computational intensity of the GEMM operation. Furthermore, by decoupling some of the additions from multiplications by shifting them to the lifting phase, devices with hardware support for fused multiply-add may be underutilized. A more detailed description of the kn2X family of methods can be found in [14].

The Fully-Connected layer is trivially reduced to a matrix-vector multiplication if the batch size is set to 1 or a matrix-matrix multiplication for larger batch sizes.

LPDNN currently supports the following third-party BLAS libraries:

- Atlas [15], with multi-vendor CPU support
- OpenBLAS [16], with multi-vendor CPU support
- BLIS [17], with multi-vendor CPU support
- Intel MKL [18], with x86 and x86_64 CPU support

Optimized GEMM implementations have also been provided by ARM Ltd.

2.3.1.1.3 cpu_nnpack

This plugin integrates the open-source NNPACK deep learning acceleration package which is publicly available on https://github.com/Maratyszcza/NNPACK. NNPACK provides many optimized implementations of the Convolutional layer, but it also supports Fully-Connected, Max Pooling, ReLU Activation and SoftMax layers.

NNPACK implements multiple algorithms for Convolutional layers including Direct, GEMM-based, FFT-based and Winograd-based convolution. FFT-based methods perform FFT convolution via the convolution theorem, by first computing the Fourier transform of the input image and the filter, applying a pointwise multiplication, and then computing the inverse Fourier transform of the resulting matrix to produce the output. FFT-based implementations have been shown to work well for large filters [19], which, however, are not that common in state-of-the-art DNNs. NNPACK provides FFT-based implementations for non-strided Convolutional layers with filters up to 16x16. Winograd-based methods use the Coppersmith-Winograd algorithm [20], which shows how to reduce the number of multiplications at the cost of more additions and a large number of intermediate products. It has been shown in [21] that Winograd-based convolution can be efficient for small filters. NNPACK provides implementations for non-strided Convolutional layers with 3x3 filters.

2.3.1.1.4 cpu_arm_conv

This plugin integrates highly-optimized Convolutional layers for ARM CPUs, developed under WP3 of the Bonseyes project. Specifically, multiple Winograd-based implementations for 3x3 and 5x5 filters are provided, as well as 3x3 Depthwise Convolutional layers, which are typically used in combination with 1x1 Convolutional layers to implement so-called Depthwise Seperable convolutions. A depthwise convolution is a spatial convolution performed independently over each - instead of every - channel of the input. This type of convolution is used in Xception [12] and MobileNets [7], both popular networks for resource-constrained devices.

2.3.1.2 GPU

2.3.1.2.1 gpu_cudnn

This plugin integrates the NVIDIA CUDA Deep Neural Network library (cuDNN) [22] into LPDNN. cuDNN is used in most DNN frameworks such as Caffe [26], TensorFlow [1], etc. to accelerate both training and inference of neural networks on NVIDIA GPUs. This plugin implements the following layers: Convolutional, Activation (ReLU), Batch Normalization, Local Response Normalization, Pooling and SoftMax Loss.

2.3.1.2.2 gpu_blas

This plugin is complementary to the "gpu_cudnn" plugin, in the sense that it implements layers that are not available in "gpu_cudnn". Specifically, this plugin currently implements the Fully-Connected layer using the NVIDIA cuBLAS library [23].

2.3.2 Plugins for Sparse DNNs

The following plugins use sparse computations to perform the inference and should typically be used for sparse DNNs, i.e., DNNs with high levels of sparsity in the model parameters introduced through fine-grained pruning techniques [24].

2.3.2.1 CPU

2.3.2.1.1 cpu_sparse

This plugin includes multiple implementations of sparse Convolutional and Fully-Connected layers, which can be used to compress the model representation in memory, i.e., the weights tensor(s), when the model has been pruned with techniques that introduce high levels of sparsity. The plugin includes both direct and BLASbased implementations using the im2X family of lowering methods (see Section 2.3.1.1.2) that rely on sparse matrix - dense matrix multiplication (SpGEMM) and sparse matrix - dense vector multiplication (SpMV). The sparse weight tensor is represented using one of the de facto standard sparse matrix storage formats, including the Coordinate (COO), Compressed Sparse Row (CSR) and Compressed Sparse Column (CSC) formats [25].

LPDNN currently supports the following third-party sparse BLAS libraries:

- Intel MKL [18], with x86 and x86_64 CPU support

SpGEMM and SpMV kernels have also been developed directly in the plugin to enable execution on ARM CPUs.

2.3.3 Supported Data Layouts

The inputs, weights and outputs of DNN layers are multidimensional tensors, so there are many possible orderings of the dimensions in memory. Many deep learning frameworks use a single canonical tensor layout. For example, Caffe [26] uses the N × C × H × W layout for the input tensor, where N is the batch size, C is the number of input channels, H is the height of a channel and W is the width of a channel, and the M × C × K × K layout for the weight tensor, where M is the number of filters, C is the number of input channels and K is the filter height/width. Different layouts may be beneficial for different layers, algorithms and hardware platforms, making the selection non-trivial. Furthermore, if consecutive layers use different layouts then a conversion needs to be performed on-the-fly, which may outweigh the isolated performance gains of the individual layers. A solution to this problem has been developed under WP3 of Bonseyes via a Partitioned Boolean Quadratic Assignment Problem solver. More details on this work, which will be integrated in the LPDNN framework, can be found in [27]. LPDNN currently provides a data layout conversion layer, which is implicitly inserted between consecutive layers when their tensor layouts are incompatible.

2.3.4 Supported Data Types

DNN training is typically performed using high-precision floating-point arithmetic. However, it has been shown that DNN inference can be performed using lower-precision fixed-point arithmetic with a minor loss in accuracy [28]. This is especially important for mobile and embedded devices that do not efficiently support floating-point computations, e.g., Internet of Things (IoT) and robotics devices. Since fixed-point arithmetic can be trivially performed with integer arithmetic, which is natively supported on any device, it is important that a DNN inference framework provides fixed-point layer implementations using integer arithmetic. Since, ANSI C does not provide native fixed-point datatypes, LPDNN creates the definitions that are required to implement the variables. While the width and binary point position of the fixed-point representation are fully customisable in the LPDNN core, layer implementations are currently provided only for 16-bit signed integers for all tensors using the “cpu_vanilla” plugin (see Section 2.3.1.1.1). This implies that LPDNN can import low-precision models of any fixed-point representation, however, inference will be performed using a 16-bit representation and, therefore, requires a data type conversion. Operations using mixed fixed-point arithmetic are non-trivial to implement and will be added if required. Table 1 shows the support for different data representations in the plugins of LPDNN. A question mark (?) indicates planned support.

Table 1. Datatype support in LPDNN plugins

LPDNN plugin	Floating-point	Fixed-point [IL.FL]
cpu_vanilla	float	16 bits, using int16
cpu_blas	float	✗
cpu_nnpack	float	✗
cpu_armconv	float	**?**

cpu_sparse	float	**?**
gpu_cudnn	float/half	✗
gpu_blas	float	✗

2.4 Tools

In this section we introduce the most important tools available in LPDNN that enable and facilitate the task of deploying and optimizing DNN models on a target platform in support of an AI challenge.

2.4.1 DNN model importing

LPDNN currently supports importing models from the Caffe format [26]. This implies that any model trained with a different deep learning framework, e.g. PyTorch [29], TensorFlow [1], etc., will need to be converted first to the Caffe format before it can be imported to LPDNN. This tool converts the Caffe model into LPDNN's internal representation. Specifically:

1. It converts the Caffe network specification present in the form of a .prototxt file into a simple array of structured format that is compatible with LPDNN.
2. It converts the trained model file in the .caffemodel format into a LPDNN compatible format, using either a fixed or floating-point data representation.

Steps 1 generates an ANSI C network specification in "lpdnn_netspec.h/.c" files and Step 2 generates the model weights either in "lpdnn_model.h/.c" files or in the HDF5 format, a popular format for storing and managing data. These files can be compiled and linked to an AI task that uses LPDNN for inference (see Section 3).

2.4.2 DNN model analyzer

LPDNN provides a tool for visualizing and analysing DNN architectures in terms of their computational and memory requirements. Specifically, the tool parses a model description given in the Caffe format [26] and (a) produces plots that demonstrate per layer and total memory requirements of weights and activations given a user-defined bit-length for the data representations, and (b) per layer and total statistics on the number of operations, e.g., additions, multiplications, divisions, etc., required for the inference of the model. This is useful for estimating the suitability of a model for deployment on a target platform. Figure 3 shows the per layer memory requirements of AlexNet's [5] weights assuming a 1-bit data representation.

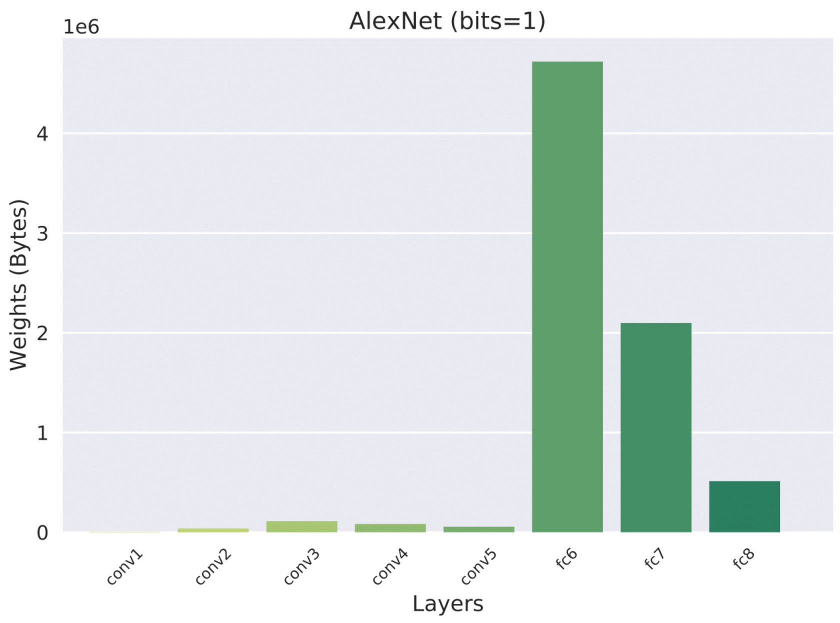


Figure 3. Memory requirements of AlexNet [5] model weights using a 1-bit data representation.

2.4.3 DNN model quantization

Quantization techniques store and calculate numbers in more compact formats. In the context of deep learning, quantization has been mainly used to compress the models, but also replace power-hungry high-precision floating-point computations with low-power reduced-precision fixed-point computations. The intuition behind these techniques is that fixed-point representations of real numbers can be represented by integers and, therefore, computations on such representations can be implemented using integer arithmetic. Therefore, the advantages of quantization are many-fold: apart from explicitly reducing the size of a DNN model, it can also enable the deployment of DNNs on devices with no FPU units, as well as provide performance and energy-efficiency benefits. Nonetheless, quantization introduces a trade-off between memory savings and model accuracy. Multiple quantization schemes for DNNs have been proposed in the literature with different strengths and weaknesses [30] [28] [31] [32]. We draw a basic distinction between methods that directly train a reduced-precision model and methods that convert a model pre-trained in high precision into low-precision as a post-processing step. Here, we focus on post-training quantization schemes that do not require a retraining phase to restore model accuracy and, therefore, can be directly implemented in the LPDNN inference framework. We distinguish the following approaches:

- *Quantization of weights and/or activations by directly reducing the numerical precision of the values:* This family of techniques directly converts the floating-point representation of pre-trained weights and/or activations to a fixed-point representation in the form [IL.FL], where IL is the bit-length of the integer part and FL is the bit-length of the fractional part. Figure 4 shows a signed fixed-point representation using 8 bits. The conversion from floating to fixed-point is typically done using deterministic round-to-nearest or stochastic rounding [30]. The fixed-point representation may be the same for the entire network or may be adjusted per layer, i.e. *dynamic* fixed-point.

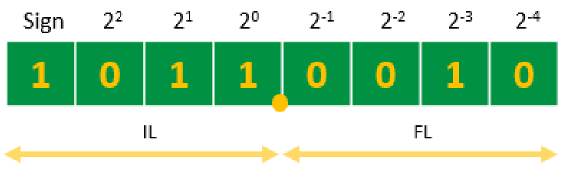
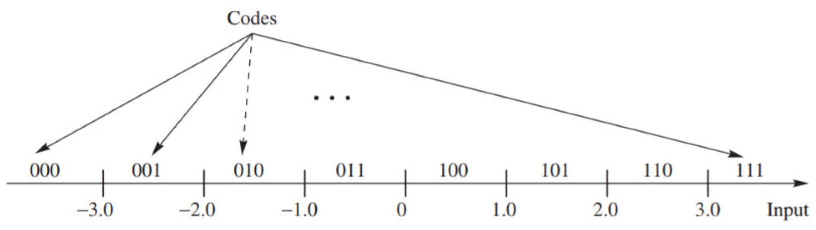


Figure 4. Fixed-point representation of real numbers using 8 bits.

- Quantization *of weights and/or activations through classical scalar or vector quantization schemes*: This family of techniques employs a scalar or vector quantizer to map the original set of floating-point values to a much smaller set of quantized values. First, the quantized values are determined and typically range between the minimum and maximum value of the original set of values. Each quantized value is mapped to a binary codeword. The mapping of binary codewords to quantized values forms the codebook of the quantizer. The original values are then assigned the binary codeword of the "closest" quantized value. An example of a uniform scalar quantizer is given in Figure 5. Therefore, in this case the weights and/or activations of the DNN no longer contain a particular value, but instead store an index to a lookup table (codebook).



Input Codes	Output
000	−3.5
001	−2.5
010	−1.5
011	−0.5
100	0.5
101	1.5
110	2.5
111	3.5

Figure 5. Quantization using a uniform scalar quantizer. Encoding (left) and decoding (right).

The above quantization schemes can be jointly applied for improved results. For instance, K-means clustering (vector quantization) may be first applied to determine the set of quantized values (centroids) and, then, these values may be converted to a fixed-point representation.

LPDNN currently supports quantization of both weights and activations at a per-layer basis using (a) round-to-nearest to convert floating-point to fixed-point or (b) K-means clustering with a fixed-point representation of the quantized values using the previous scheme. The quantization engine in LPDNN enables design space exploration through per-layer and end-to-end sensitivity analysis of these quantization schemes to the overall accuracy of the model. A single layer and variable type, i.e. weights or activations, can be quantized while keeping all other layers in single-precision (32-bit) floating-point format to analyse the influence of quantization techniques and the sensitivity of a layer or group of layers within a network. The pipeline process allows the selection of the number of iterations for each parameter search to trade-off between the analysis time and performance. As shown in Figure 6, the pipeline contains two main phases: 1) Layer Analysis and 2) Network Space Exploration. In the Layer Analysis phase, the quantization technique is selected. The objective of this phase is to search for the distribution details and optimum fixed-point parameters, e.g. Integer and Fractional Lengths [IL, FL], for several bit widths. In order to minimize the quantization error, the search is performed automatically for activations and weights independently in each layer. All the quantization parameters are written onto the configuration file which is used to infer and compare the custom and baseline floating-point implementation. In the second phase, the Network Space Exploration is carried out taking as input the configuration file with the optimum parameters per layer from the Layer Analysis phase. The objective of the Network Space Exploration is to determine the sensitivity of each layer to bit-width scaling and the set of possible configurations when a group or all layers of the network are quantized. More details are provided in [33].

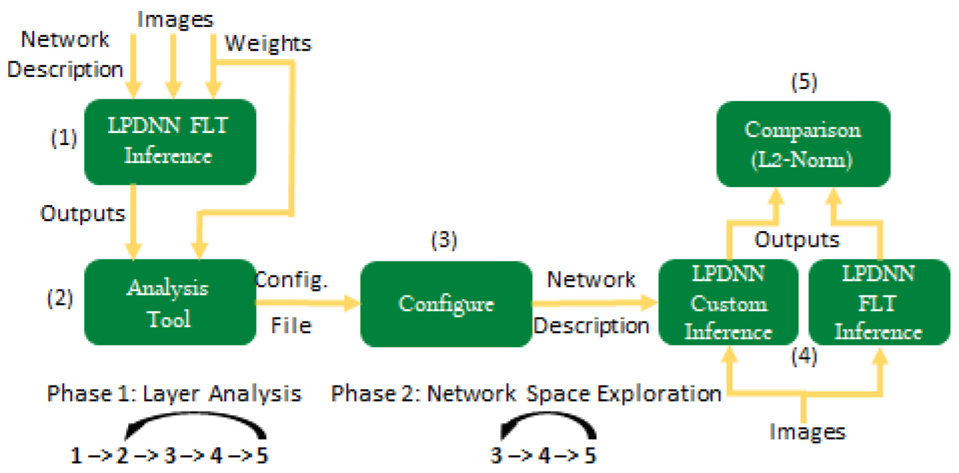


Figure 6. Quantization pipeline in LPDNN

2.4.4 Benchmarking

The purpose of model benchmarking is to provide both task-level and target platform information on the performance of a model. LPDNN currently provides tools to measure target platform performance of DNN inference in terms of speed, memory usage, and storage at multiple granularities. Specifically, LPDNN currently provides tools to individually benchmark different BLAS libraries, DNN layers that pose performance bottlenecks, e.g. Convolutional and Fully-Connected, using all available plugins of LPDNN (see Section 2.3), as well as end-to-end inference performance using all available plugins. These tools will be used to evaluate LPDNN in Section 4.

3. Integration of DNNs in AI applications through LPDNN

In this section, we demonstrate how the LPDNN framework can be used to build an AI application using a DNN model. Specifically, we will demonstrate use of the LPDNN API to perform image classification.

3.1 Image Classification with LPDNN

The following example assumes a Caffe model, e.g. AlexNet, has been converted to LPDNN's internal format using the model importing tool described in Section 2.4.1, and is provided through the automatically generated "lpdnn_netspec.h/.c" and "lpdnn_model.h/.c" files.

```
///
/// Basic usage of the LPDNN API for image classification
///
#include "lpdnn_api.h"
#include "lpdnn_blob.h"
#include "lpdnn_dtypes.h"
#include "lpdnn_netspec.h"
#include "lpdnn_model.h"

// Initialise LPDNN
LpdnnInit();

// Network parameters from automatically generated "lpdnn_netspec.h" and "lpdnn_model.h"
const LpdnnNetParams* netSpec = &lpdnn_network_spec;
const void* netModel = &lpdnn_network_model;

// Create and initialize LPDNN network
LpdnnNetHndl net= LpdnnNetAlloc("AlexNet");
LpdnnNetInit(net, netSpec, netModel);

// Get input blob
LpdnnBlobHndl inputBlob = LpdnnNetInputBlobGet(net);

// Initialise input blob data with the image for classification
void* inputData = LpdnnBlobDataGet(inputBlob);
…

// Perform inference
LpdnnNetInfer(net, inputBlob);

// Get inferred result
LpdnnConstBlobHndl output = LpdnnNetGetLayerOutput(net);
float* result = LpdnnBlobDataGet(output);

// Use result
…

// Cleanup
LpdnnNetFree(app->net);
LpdnnDispose();
```

4. Evaluation of LPDNN on Selected Platforms

In this section, we will evaluate the performance of LPDNN on three platforms, including the Firefly-RK3399 and NVIDIA Jetson TX2, for a set of DNN models that will be used in the demonstrators of the project. Specifically, we will benchmark the following models:

- MobileNet [7]
- SphereFace [34]

The evaluation process is incremental. First, we evaluate low-level BLAS libraries that are typically used to accelerate deep learning computations in order to determine the best candidate for every platform. In the following, we benchmark Convolutional layer implementations that are available in LPDNN through different plugins (see Section 2.3). These layers typically dominate the execution time of DNN inference, therefore, highly optimized implementations are essential to achieving high overall performance. Finally, we report end-to-end inference performance using different plugins.

4.1 Experimental Setup

In the following experiments, all software involved has been cross-compiled for the target platform using GCC-6.4 with the -O3 optimization flag. For every benchmark, we run 10 iterations and report metrics using the average execution time over all iterations. Table 2 provides an overview of the evaluation platforms. We have highlighted in red the hardware that has been used for benchmarking. Since, LPDNN is currently stable for single-threaded execution in CPU mode, we use one CPU core for our experiments. We set the CPU affinity using the "taskset" command-line tool available on both platforms.

Table 2. Evaluation platforms

Platform	Firefly-RK3399	NVIDIA Jetson TX2
CPU	2 x ARM Cortex-A72 @ 2.0 GHz, 4 x ARM Cortex-A53 @ 1.5 GHz	2 x NVIDIA Denver2@ 2.0 GHz 4 x ARM Cortex-A57 @ 2.0 GHz
GPU	ARM Mali T860MP4	NVIDIA Pascal 1.3 GHz

4.2 Evaluation of BLAS Libraries

In the following, we evaluate multiple GEMM implementations from different BLAS libraries, including publicly available open-source Atlas [15], OpenBLAS [16] and BLIS [17], as well as the ARM-GEMM library that has been developed as part of Bonseyes by ARM Ltd. GEMM is typically used to implement Convolutional layers, so high-performance implementations are essential.

4.2.1 Firefly-RK3399

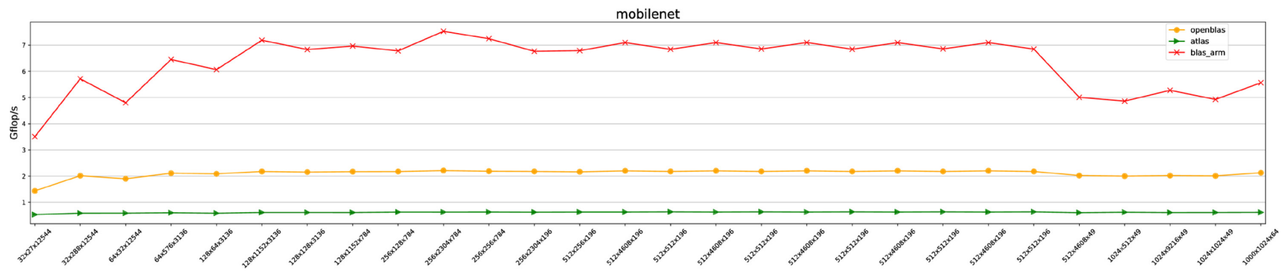


Figure 7. GEMM performance for MobileNet on Firefly-RK3399

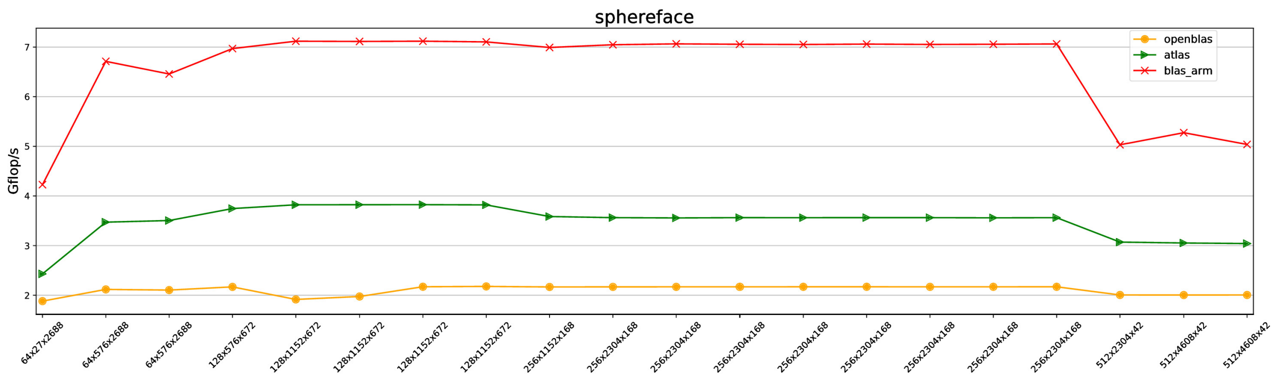


Figure 8. GEMM performance for SphereFace on Firefly-RK3399

4.2.2 NVIDIA Jetson TX2

It is noteworthy that on this platform we also benchmarked the GPU using the cuBLAS library [23].

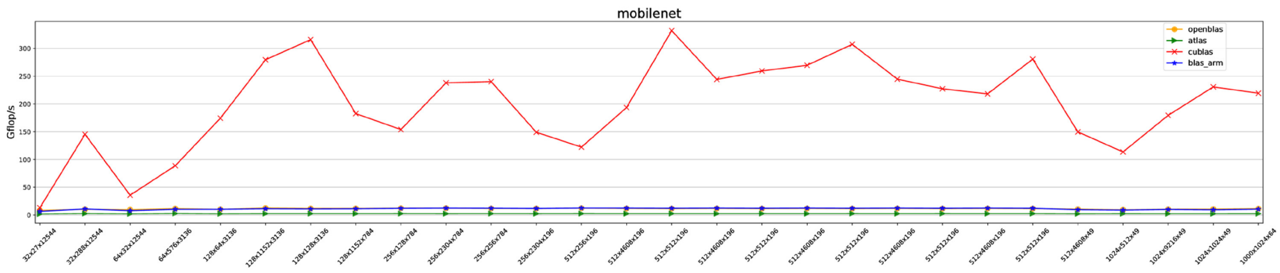


Figure 9. GEMM performance for MobileNet on NVIDIA Jetson TX2

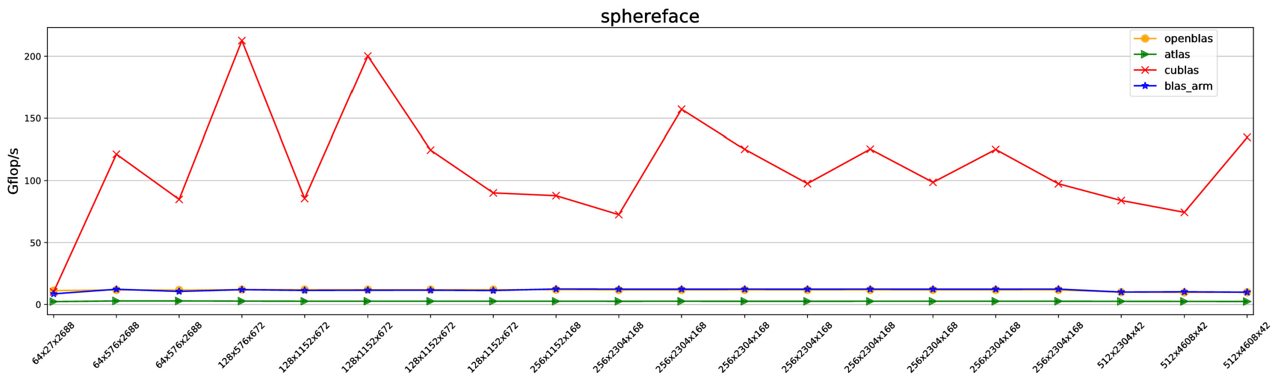


Figure 10. GEMM performance for SphereFace on NVIDIA Jetson TX2

4.3 Evaluation of Convolutional Layers

In the following we benchmark Convolutional layer implementations available in LPDNN through different plugins. In particular, we report performance for cpu_blas, cpu_nnpack and cpu_sparse whenever applicable. For the cpu_blas plugin we benchmark the im2col lowering method using the ARM-GEMM library (see Section 2.3.1.1.2). For the cpu_sparse plugin we have randomly inserted zero weights into the model so that every layer is 75% sparse.

4.3.1 Firefly-RK3399

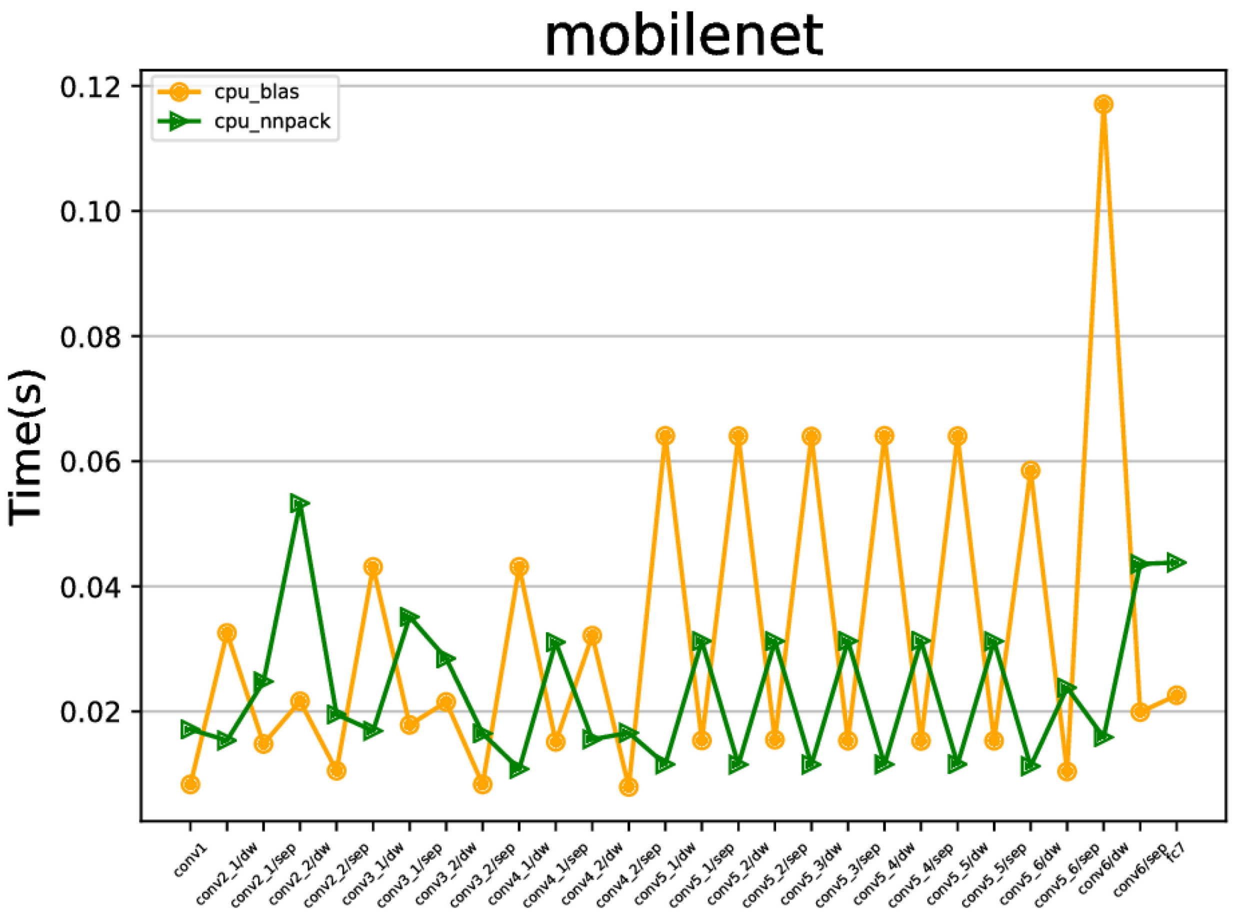


Figure 11. Convolutional layers of MobileNet with different LPDNN plugins on Firefly-RK3399

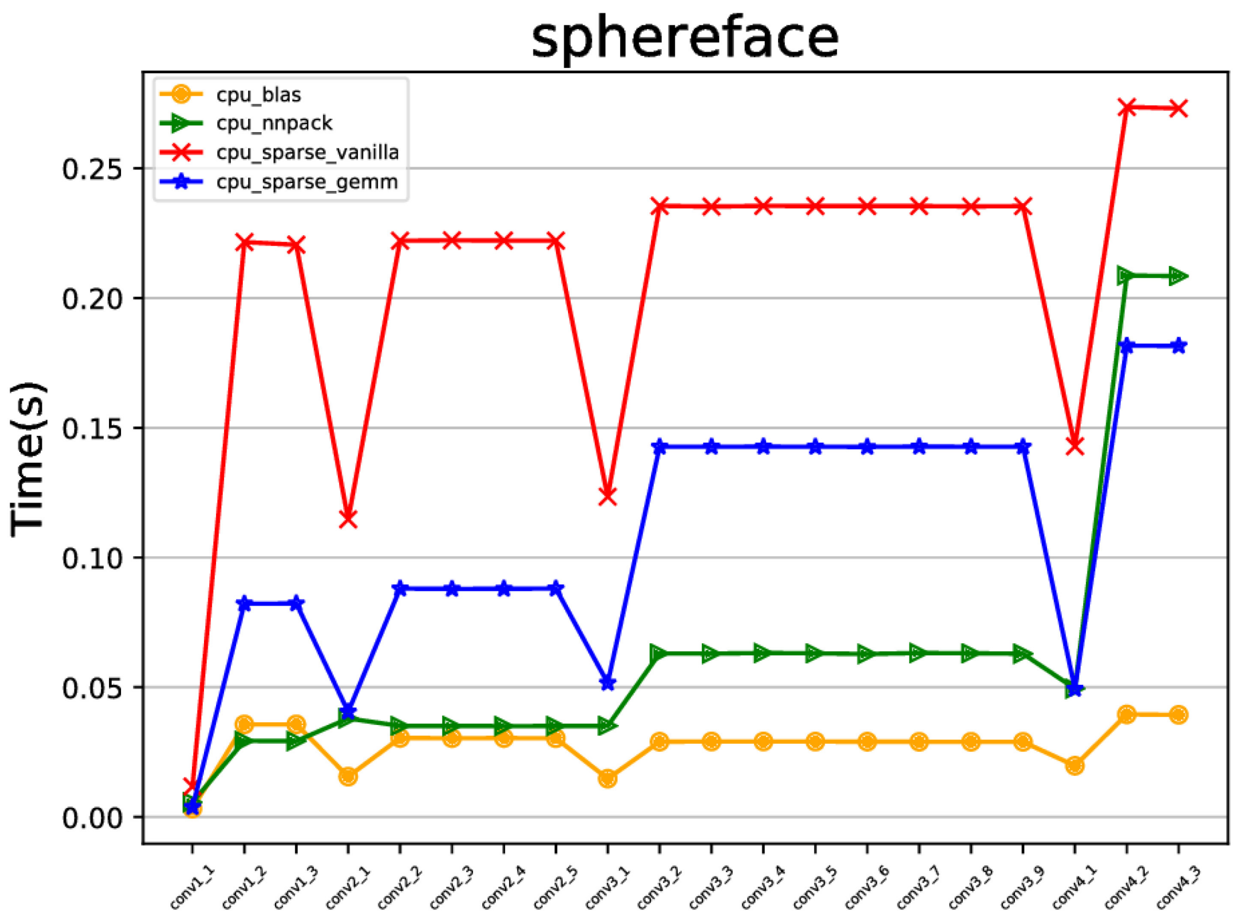


Figure 12. Convolutional layers of SphereFace with different LPDNN plugins on Firefly-RK3399

4.3.2 NVIDIA Jetson TX2

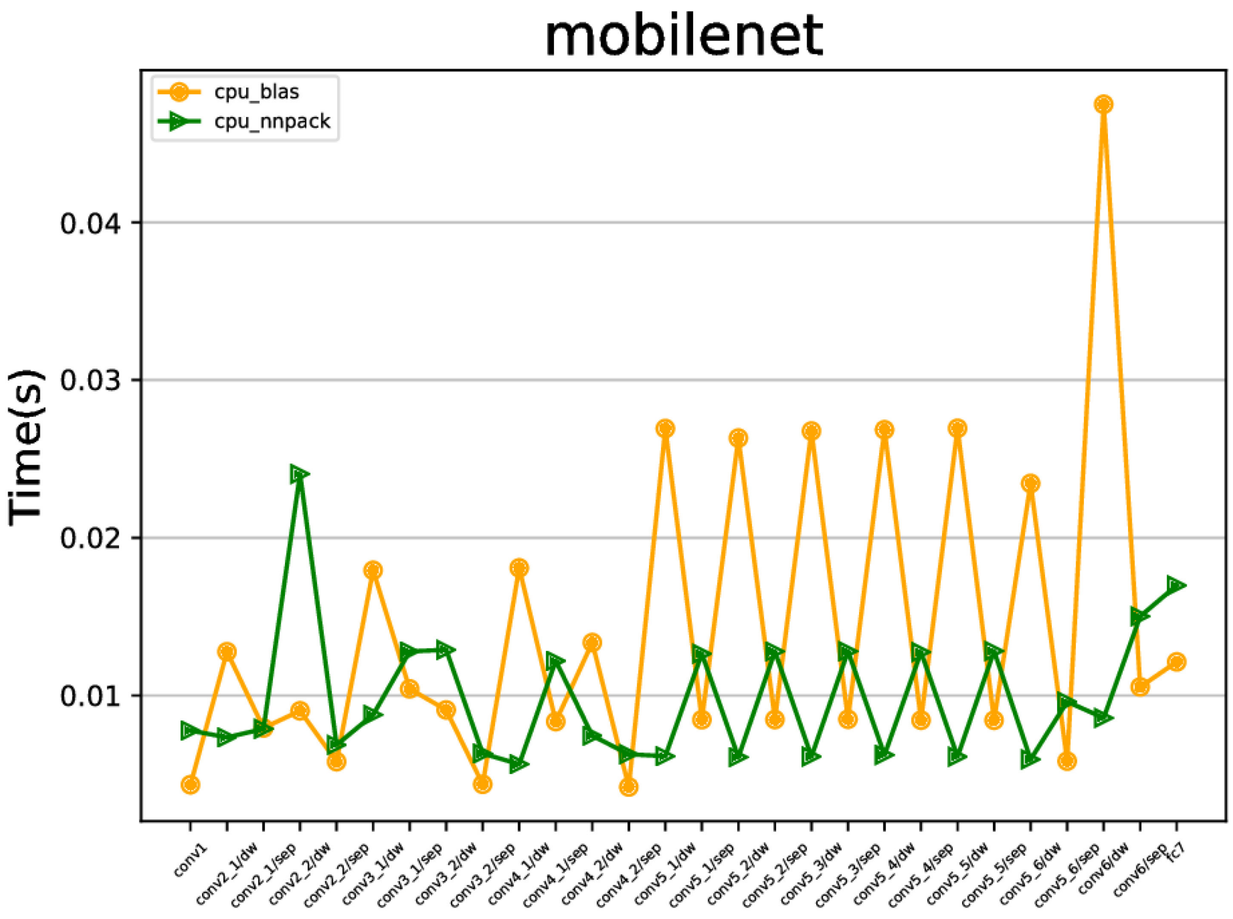


Figure 13. Convolutional layers of MobileNet with different LPDNN plugins on NVIDIA Jetson TX2

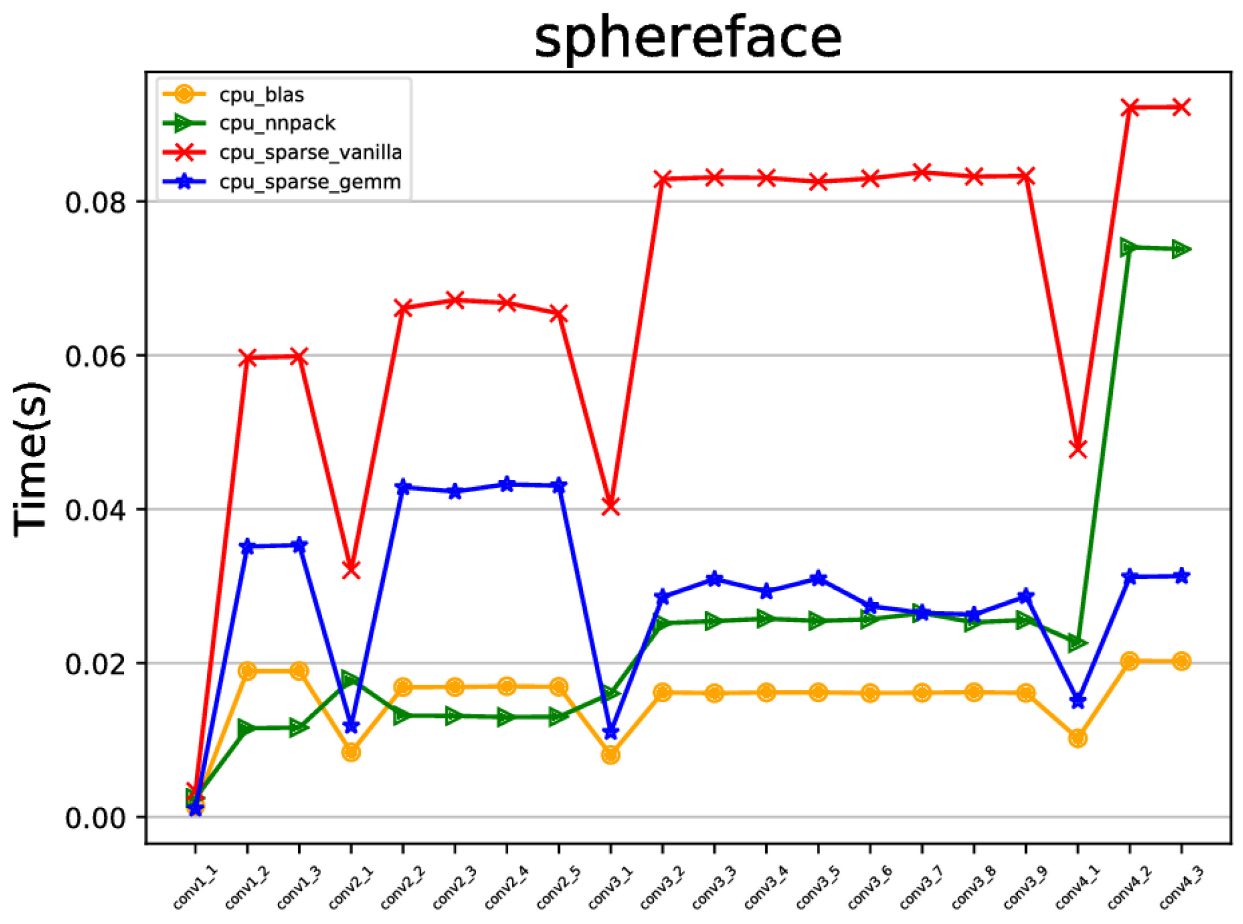


Figure 14. Convolutional layers of SphereFace with different LPDNN plugins on NVIDIA Jetson TX2

4.4 Evaluation of End-to-End Inference

In the following we report the execution time in seconds of end-to-end inference (batch size = 1) using a single plugin for all layers. For cpu_blas, cpu_nnpack and gpu_cudnn, whenever a layer is not available in the plugin we use the implementation available in cpu_vanilla.

4.4.1 Firefly-RK3399

Table 3. Execution time (secs) of inference on Firefly-RK3399

Model	cpu_vanilla	cpu_blas	cpu_nnapck
MobileNet	19.46	0.58	-
SphereFace	32.10	0.49	0.96

Table 4. Execution time (secs) of inference on NVIDIA Jetson TX2

Model	cpu_vanilla	cpu_blas	cpu_nnapck	gpu_cudnn
MobileNet	20.31	0.56	-	0.34
SphereFace	36.03	0.47	0.50	0.20

References

[1] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, Manjunath Kudlur, Josh Levenberg, Rajat Monga, Sherry Moore, Derek G. Murray, Benoit Steiner, Paul Tucker, Vijay Vasudevan, Pete Warden, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. 2016. TensorFlow: a system for large-scale machine learning. In Proceedings of the 12th USENIX conference on Operating Systems Design and Implementation (OSDI'16). USENIX Association, Berkeley, CA, USA, 265-283.

[2] NVIDIA TensorRT, https://developer.nvidia.com/tensorrt

[3] Caffe2, https://caffe2.ai/

[4] Sergey Ioffe and Christian Szegedy. 2015. Batch normalization: accelerating deep network training by reducing internal covariate shift. In Proceedings of the 32nd International Conference on International Conference on Machine Learning - Volume 37 (ICML'15), Francis Bach and David Blei (Eds.), Vol. 37. JMLR.org 448-456.

[5] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E. Hinton. 2012. ImageNet classification with deep convolutional neural networks. In Proceedings of the 25th International Conference on Neural Information Processing Systems - Volume 1 (NIPS'12), F. Pereira, C. J. C. Burges, L. Bottou, and K. Q. Weinberger (Eds.), Vol. 1. Curran Associates Inc., USA, 1097-1105.

[6] Liu, W., Anguelov, D., Erhan, D., Szegedy, C., Reed, S., Fu, C. Y., & Berg, A. C. (2016). SSD: Single shot multibox detector. In Computer Vision - 14th European Conference, ECCV 2016, Proceedings (Vol. 9905 LNCS, pp. 21-37). (Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics); Vol. 9905 LNCS). Springer Verlag. DOI: 10.1007/978-3-319-46448-0_2.

[7] Howard, A.G., Zhu, M., Chen, B., Kalenichenko, D., Wang, W., Weyand, T., Andreetto, M., & Adam, H. (2017). MobileNets: Efficient Convolutional Neural Networks for Mobile Vision Applications. CoRR, abs/1704.04861.

[8] Simonyan, Karen & Zisserman, Andrew. (2014). Very Deep Convolutional Networks for Large-Scale Image Recognition. arXiv 1409.1556.

[9] C. Szegedy et al., "Going deeper with convolutions," 2015 IEEE Conference on Computer Vision and Pattern Recognition (CVPR), Boston, MA, 2015, pp. 1-9. doi: 10.1109/CVPR.2015.7298594

[10] Iandola, F.N., Han, S., Moskewicz, M.W., Ashraf, K., Dally, W.J., & Keutzer, K. (2016). SqueezeNet: AlexNet-level accuracy with 50x fewer parameters and<0.5MB model size.

[11] He, K., Zhang, X., Ren, S., & Sun, J. (2016). Deep Residual Learning for Image Recognition. 2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR), 770-778.

[12] Chollet, Francois. (2017). Xception: Deep Learning with Depthwise Separable Convolutions. 1800-1807. 10.1109/CVPR.2017.195.

[13] Yangqing Jia. 2014. Learning Semantic Image Representations at a Large Scale. Ph.D. Dissertation. EECS Department, University of California, Berkeley. http://www2.eecs.berkeley.edu/Pubs/TechRpts/2014/ EECS-2014-93.html

[14] A. Vasudevan, A. Anderson, and D. Gregg, "Parallel multi-channel convolution using general matrix multiplication," in 28th IEEE International Conference on Application-specific Systems, Architectures and Processors, ASAP 2017, Seattle, WA, USA, July 10-12, 2017, 2017, pp. 19–24.

[15] R. Clint Whaley and Jack J. Dongarra. 1998. Automatically tuned linear algebra software. In Proceedings of the 1998 ACM/IEEE conference on Supercomputing (SC '98). IEEE Computer Society, Washington, DC, USA, 1-27.

[16] Xianyi, Z., Qian, W., & Chothia, Z. (2012). OpenBLAS. URL: http://xianyi.github.io/OpenBLAS, 88.

[17] Field G. Van Zee and Robert A. van de Geijn. 2015. BLIS: A Framework for Rapidly Instantiating BLAS Functionality. ACM Trans. Math. Softw. 41, 3, Article 14 (June 2015), 33 pages. DOI: https://doi.org/10.1145/2764454

[18] Intel® Corporation. 2013. Intel® Math Kernel Library. Retrieved from http://software.intel.com/en-us/intel-mkl.
[19] Vasilache, Nicolas & Johnson, Jeff & Mathieu, Michael & Chintala, Soumith & Piantino, Serkan & Lecun, Yann. (2014). Fast Convolutional Nets With fbfft: A GPU Performance Evaluation.
[20] Shmuel Winograd. Arithmetic complexity of computations, volume 33. Siam, 1980.
[21] Lavin, A., & Gray, S. (2016). Fast algorithms for convolutional neural networks. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition* (pp. 4013-4021).
[22] Chetlur, Sharan & Woolley, Cliff & Vandermersch, Philippe & Cohen, Jonathan & Tran, John & Catanzaro, Bryan & Shelhamer, Evan. (2014). cuDNN: Efficient Primitives for Deep Learning.
[23] NVIDIA CUBLAS, https://developer.nvidia.com/cublas
[24] Jiecao Yu, Andrew Lukefahr, David Palframan, Ganesh Dasika, Reetuparna Das, and Scott Mahlke. 2017. Scalpel: Customizing DNN Pruning to the Underlying Hardware Parallelism. In Proceedings of the 44th Annual International Symposium on Computer Architecture (ISCA '17). ACM, New York, NY, USA, 548-560. DOI: https://doi.org/10.1145/3079856.3080215.
[25] Y. Saad, Iterative Methods for Sparse Linear Systems, Society for Industrial and Applied Mathematics, Philadelphia, PA, 2003
[26] Yangqing Jia, Evan Shelhamer, Jeff Donahue, Sergey Karayev, Jonathan Long, Ross Girshick, Sergio Guadarrama, and Trevor Darrell. 2014. Caffe: Convolutional Architecture for Fast Feature Embedding. In Proceedings of the 22nd ACM international conference on Multimedia (MM '14). ACM, New York, NY, USA, 675-678. DOI: https://doi.org/10.1145/2647868.2654889.
[27] Andrew Anderson and David Gregg. 2018. Optimal DNN primitive selection with partitioned boolean quadratic programming. In Proceedings of the 2018 International Symposium on Code Generation and Optimization (CGO 2018). ACM, New York, NY, USA, 340-351. DOI: https://doi.org/10.1145/3168805.
[28] Darryl D. Lin, Sachin S. Talathi, and V. Sreekanth Annapureddy. 2016. Fixed point quantization of deep convolutional networks. In Proceedings of the 33rd International Conference on International Conference on Machine Learning - Volume 48 (ICML'16), Maria Florina Balcan and Kilian Q. Weinberger (Eds.), Vol. 48. JMLR.org 2849-2858.
[29] PyTorch, https://pytorch.org/
[30] Suyog Gupta, Ankur Agrawal, Kailash Gopalakrishnan, and Pritish Narayanan. 2015. Deep learning with limited numerical precision. In Proceedings of the 32nd International Conference on International Conference on Machine Learning - Volume 37 (ICML'15), Francis Bach and David Blei (Eds.), Vol. 37. JMLR.org 1737-1746.
[31] Han, Song & Mao, Huizi & Dally, William. (2016). Deep Compression: Compressing Deep Neural Networks with Pruning, Trained Quantization and Huffman Coding.
[32] Jacob, Benoit & Kligys, Skirmantas & Chen, Bo & Zhu, Menglong & Tang, Matthew & Howard, Andrew & Adam, Hartwig & Kalenichenko, Dmitry. (2017). Quantization and Training of Neural Networks for Efficient Integer-Arithmetic-Only Inference.
[33] Miguel de Prado Escudero, Maurizio Denna, Luca Benini and Nuria Pazos. (2018). QUENN: QUantization Engine for low-power Neural Networks.
[34] Liu, W., Wen, Y., Yu, Z., Li, M., Raj, B., & Song, L. (2017, July). Sphereface: Deep hypersphere embedding for face recognition. In *The IEEE Conference on Computer Vision and Pattern Recognition (CVPR)* (Vol. 1).