electronics

MDPI

Article

An Integrated Analysis Framework of Convolutional Neural Network for Embedded Edge Devices †

Seung-Ho Lim *, Shin-Hyeok Kang, Byeong-Hyun Ko, Jaewon Roh, Chaemin Lim and Sang-Young Cho

Division of Computer Engineering, Hankuk University of Foreign Studies, Yongin 17035, Korea; adam16377@gmail.com (S.-H.K.); devko@kakao.com (B.-H.K.); harryroh2003@gmail.com (J.R.); chaemin.lim.hufs@gmail.com (C.L.); sycho@hufs.ac.kr (S.-Y.C.)

* Correspondence: slim@hufs.ac.kr

† This paper is an extended version of our paper published in ICCE 2022.

Abstract: Recently, IoT applications using Deep Neural Network (DNN) to embedded edge devices are increasing. Generally, in the case of DNN applications in the IoT system, training is mainly performed in the server and inference operation is performed on the edge device. The embedded edge devices still take a lot of loads in inference operations due to low computing resources, so proper customization of DNN with architectural exploration is required. However, there are few integrated frameworks to facilitate exploration and customization of various DNN models and their operations in embedded edge devices. In this paper, we propose an integrated framework that can explore and customize DNN inference operations of DNN models on embedded edge devices. The framework consists of the GUI interface part, the inference engine part, and the hardware Deep Learning Accelerator (DLA) Virtual Platform (VP) part. Specifically it focuses on Convolutional Neural Network (CNN), and provides integrated interoperability for convolutional neural network models and neural network customization techniques such as quantization and cross-inference functions. In addition, performance estimation is possible by providing hardware DLA VP for embedded edge devices. Those features are provided as web-based GUI interfaces, so users can easily utilize them.

Keywords: edge device; DNN; GUI interface; inference engine; ONNX; customization; DLA VP

Citation: Lim, S.-H.; Kang, S.-H.; Ko, B.-H.; Roh, J.; Lim, C.; Cho, S.-Y. An Integrated Analysis Framework of Convolutional Neural Network for Embedded Edge Devices. *Electronics* **2022**, *11*, 1041. https://doi.org/10.3390/electronics11071041

Academic Editors: József Sütő, Stefan Oniga, Alin-Sasa Tisan and Fernando Morgado-Dias

Received: 7 February 2022
Accepted: 24 March 2022
Published: 26 March 2022

1. Introduction

Recently, deep neural networks (DNN)-based applications have been widely used with excellent performance in various fields. Those DNN-based applications are also emerging in edge devices of IoT systems [1]. In the case of DNN applications in the IoT system, training is mainly performed in the server and inference operation is performed on the edge device. The embedded edge devices still take a lot of loads in inference operations due to low computing resources. Generally, DNN has a complex network structure composed of hundreds of layers and millions of parameters to increase accuracy, and computation for training and inference with DNN is based on a lot of high-order floating points of data. Learning or training of DNN, which requires a lot of computing resources, is usually performed in a high-performance system, and only inference is performed in embedded edge devices by bringing the DNN model generated by training results.

The embedded edge devices, which have limited resources, suffer from executing a high complexity network and parameter resolution. Appropriate customization schemes should be considered for edge devices to perform DNN inference properly on the device side. There are two approaches to efficiently optimize DNN in edge devices: optimizing the network at software algorithm level, and using a hardware Deep Learning Accelerator (DLA), and these two approaches are mixed. For software optimization, a lightweight technique to reduce the complexity of the DNN network has been studied such as network

layer compression, weight pruning, and filter quantization methods [2–9]. The hardware Deep Learning Accelerator has multiple-pipelined Multiply–Accumulate (MAC) modules and pipelined data path and buffer structure for fetching and processing input/weight data from external memory to accelerate DNN operations [10–17].

The availability of deep learning models in the form of open sources, frameworks, and cloud APIs has enabled many developers to easily create applications combined with DNNs [18,19]. In addition, visualized DNN platform has been provided so that general users can easily apply training a DNN model to their appropriate domains. Although there are several analysis frameworks of DNN networks for embedded edge devices [20–23], there are few integrated tools and frameworks to explore and customize those technologies to optimize specific neural network applications in edge devices.

In this paper, we designed and implemented an integrated framework to analysis and explore architecture for customizing neural network applications on edge devices. In particular, the framework is specialized for Convolutional Neural Network (CNN) models and their applications such as image classification and object detection, since CNN is the one of the most useful among various neural network algorithms and CNN-based object detection applications with camera input on edge devices are an active area of use. The framework provided in this paper is an integrated system for customizing convolutional neural network models in embedded edge devices. It provides integrated interoperability for convolutional neural network models and network customization techniques such as quantization and cross-inference functions. In addition, performance estimation is possible by providing hardware DLA VP for embedded edge devices. The framework provides three main parts: the front-end GUI part, the inference engine part, and the DLA Virtual Platform (VP) part. In the front-end part, a web-based GUI interface is provided for users to run and simulate various deep learning models. In the inference part, it provides integrated interoperability for various convolutional neural network models with ONNX [24] format to provide open standard interface for existing neural network models, so various convolutional neural network models and operations are applied and compared with the framework. In addition, it provides various customization interfaces including parameter quantization and editing parameters, as well as crosswise and stepwise inference analysis of network layer. In the last part, our framework provides hardware DLA VP implemented in SystemC. The DLA VP is based on RISC-V embedded processor and has a dedicated acceleration module for convolutional neural network, so hardware exploration is possible by executing the customized neural network models through DLA platform and inferring the performance of the deep learning model on edge device.

This is an extended version of the previous paper published as an abstract at ICCE 2022 [25], in which this paper is focused on the integrated framework with GUI interface, inference engine, and hardware virtual platform for edge device which is also our previous work [15], and additionally gives experimental results. The organization of this paper is as follows. The background and related work of this paper are described in Section 2, and details of the developed framework are explained in Section 3. Section 4 shows experimental results and Section 5 concludes this paper.

2. Background and Related Work

2.1. Optimization and Acceleration for Embedded Edge Devices

There are several DNN customization tools and platforms. Most of the tools, frameworks, and libraries are aimed at advancing and optimizing the training and inference operations for advanced neural network models in high-performance GPUs or Cloud Computing systems. However, they have few functions and features for exploration and customization of neural network models for embedded edge devices. That is, most platforms do not provide detailed information, parameters, and performance metrics inside embedded edge devices, such as quantization, parameter pruning, and cross-layer inference with different parameter formats. In addition, there are few ways to estimate the

detailed internal operations of deep neural network with running on embedded edge device platform.

There have been several studies to reduce the complexity of neural networks by pruning unimportant network connections; quantizing and applying Huffman conding [2]; using vector quantization to CNN [3]; studying of quantization, coding, pruning, and sharing techniques for image instance retrieval [4]; or by applying filter-wise quantization [5]. Some studies tried to optimize network model architecture by applying complex-optimized computation methods [6,7]. Ref. [8] showed some techniques that can be used to reduce computational cost on computer architecture and instruction level uses.

NVDIA developed a deep learning accelerator architecture called NVDLA to accelerate neural network operation with dedicated hardware module for neural network operations [10]. Eyeriss [11] designed a flexible accelerator architecture for deep neural networks on edge devices. There were studies which proposed deep learning accelerator architecture as a co-processor of RISC-V embedded CPU by designing corresponding instructions for the accelerator [12,13]. In [14], they investigated the extension of embedded processor architecture to accelerate deep neural networks with in-pipeline hardware and related instructions. Ref. [15] developed dedicated deep learning accelerator as a controller module of embedded processor. Ref. [16] showed customized object detection system using YOLO with RISC-V based hardware accelerator, and [17] showed an 8-bit quantized inference model with NVDLA accelerator.

2.2. Frameworks for Neural Networks

As Artificial Intelligence (AI) and deep neural networks-based applications show excellent performance in many fields, many tools and frameworks have been developed to build various deep learning network models and their operations for learning and inference in a variety of real fields [19]. Deep neural network tools and frameworks that are widely used for deep neural network research and development include Caffe [26], Theano [27], CNTK [28], TensorFlow [29], PyTorch [21], and DIGITS [22]. Typical tools or platforms make it easy for users to use DNN models and to use a visualization interface to get a graphical picture of the model and its behavior. Each of them has different criteria such as programming language, usage method, development environment, interface, supported CPUs and GPUs, maturity level, supported neural network model, and so on.

Caffe [26] is a widely used deep learning library and framework developed by the Berkeley AI Research Center. Caffe has high expressiveness, modularity, and execution speed. As for the expression method, plain text schema can be used instead of code for the neural network model expression. Modularity is high so that new tasks can be easily added to the existing model. In addition, it provides an environment where multiple GPUs can be used. Theano [27] is a deep learning to and library developed by the Montreal Institute for Learning Algorithms at the University of Montreal. It is provided in the form of a Python library and is mainly used to manipulate and evaluate mathematical expressions for neural network models. There are high-level frameworks running windows system. It has overhead due to the compilation process to load Python modules and has a complex structure to extend models. CNTK [28] is an open-source deep learning toolkit developed by Microsoft. It is a toolkit that allows users to graphically run deep neural network models step-by-step on various deep neural network models and analyze the results. It can be used as a library for Python, C#, and C++ programs, or it can be used as a standalone type. Another distinct feature of CNTK is that it supports ONNX format, so interoperability between frameworks is high. Although these tools are suitable for designing complex neural network models in general-purpose or high computing systems supporting GPUs, they lack features for customizing and optimizing models for embedded edge devices.

TensorFlow [29] is another kind of open-source programming framework that was firstly developed by the Google Brain team for internal Google use for developing deep learning and machine learning applications, and was later released as open source. TensorFlow can be used in various programming languages such as Python, JavaScript, C++,

and Java, and it also supports various GPUs. TensorFlow Lite [20], also called TensorBoard, is a suite of tools that support running TensorFlow models on mobile, embedded, and IoT devices. The main components are an interpreter that can run the quantization model and a converter that converts the TensorFlow model into an efficient format that the interpreter can use. However, it does not provide a framework environment for executing and analyzing neural network operations in real embedded edge devices. PyTorch [21] is a machine learning library based on torch which is developed by Facebook. PyTorch also provides an interface to increase operation speed through various GPUs and supports implementation of neural network models and operations in Python. The PyTorch framework supports tools for quantization for parameter optimizations. It supports dynamic quantization, static quantization, and quantization-aware learning, and supports inference on quantized models to perform performance comparison with the original model. Apache TVM [30] is a compiler tool and framework that optimizes the DNN model for various hardware including CPUs, GPUs, microcontrollers, and FGPAs. TVM supports block sparsity, multi-bit level quantization, memory planning, various processor compatibility, and Python programming-based prototyping. DIGITS [22] is a deep learning GPU training system developed by NVIDIA. DIGITS uses a simplified text file format rather than a programming language to describe neural network models and parameters, supports multi-GPU-based neural network operation, and it provides a web-based interactive user interface for user convenience. Pico-CNN [23] is another kind of deep learning inference framework for embedded systems providing C source code with standard C libraries. It supports pre-trained ONNX models to make C code and deploy various neural network models with C code. It is not GUI-based, and does not support embedded architecture such as accelerator.

3. Integrated Framework for Edge Devices

The purpose of the framework is to provide a GUI environment for optimizing deep learning models on edge devices. For this, the framework provides three main functions: First, a standard interface is applied to enhance interoperability between deep learning models implemented in various previous frameworks in a single engine. Second, a hardware Virtual Platform (VP) and its interfaces are provided for inferring the performance of a deep learning model in a DLA. Third, for user convenience, a web-based graphical user interface is provided. The overall architecture of the framework is shown in Figure 1, which is composed of three parts: the GUI platform part, the inference engine part, and the DLA VP part.

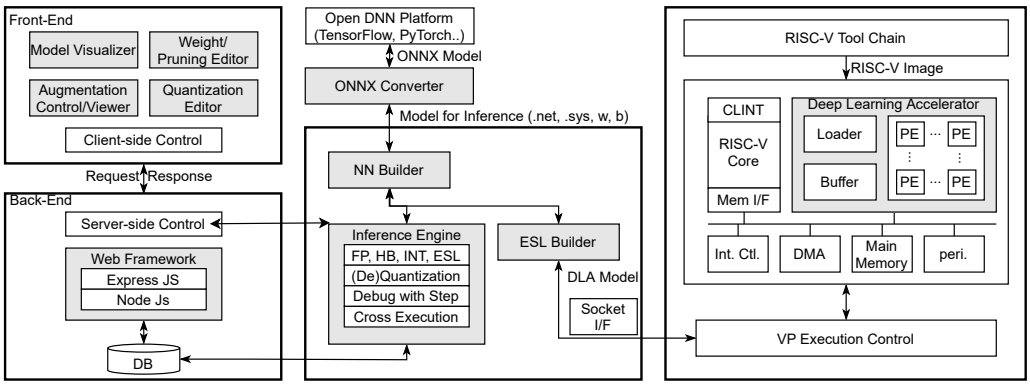


Figure 1. The overall architecture of the framework.

3.1. GUI Platform

The GUI interface is responsible for receiving the configurations of neural network model from the user, executing the neural network model through the inference engine or virtual platform, and displaying the results graphically. Figure 2 shows the overall structure of the GUI platform composed of the front-end and back-end. Specifically, for

the optimization of the deep learning operation of the embedded edge device, the GUI interface provides various interfaces to run the functions provided by the inference engine described above, such as list of the supported neural network models, file interface, settings of quantization level including FP, INT, and Hybrid for inferencing, stepwise and crosswise inferencing of each layer, as shown in the left part of the Figure 2. In addition, it also provides visualization of the neural network model and inference processes, the editing interface of weight parameters of the neural network model, data augmentation, and file management functions.

To provide those features, the back-end part of GUI platform consists of five modules as shown in the right part of Figure 2: Router and Event Listener (REL), Process Manager (PM), File Manager (FM), Session Manager (SM), and Auxiliary Utility Manager (AUM). REL is responsible for the main request processing function as the manager of the platform. Since all functions of the platform, such as file management, inference selection, weight editing, image augmentation, and image visualization, are visually expressed in a web browser, it performs event service routines according to various requests from other parts. The back-end event listener receives the user's configuration for the neural network model through internal socket interface from the front-end part, and REL executes the neural network operation of the model to the interface engine based on this information. The inference engine event listener interacts with the inference engine through socket communication. It receives the results of the inference engine and stores the results in the file storage space through the file manager.

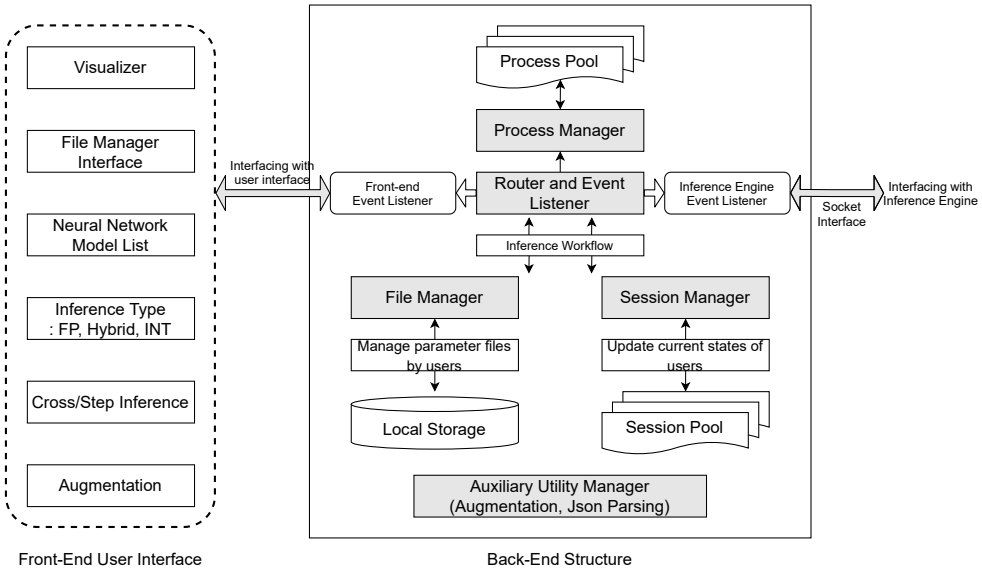


Figure 2. Structure of the GUI platform part composed of the front-end and back-end.

The inference engine operates separately from the GUI platform part, and multiple users can run the inference engine at the same time from the individual user interface. When running the inference engine, a separate child process is executed in the process manager to handle the individually separated running of inference engine. The process manager also examines the inference engine process by referring to the session. If an abnormal termination or malfunction of the inference engine process occurs, it performs post-processing of the inference engine to be killed safely.

The file manager manages the file directory on a per-user basis so that multiple users can use the inference engine at the same time. The file manager stores the neural network model files and weight files uploaded by the user in the file directory, and the user can edit and upload/download the weight files via file manager. The session manager manages the user's state using a session and updates the user's state whenever the state changes. Sessions are also used by the process manager to check the normal state of a process. When

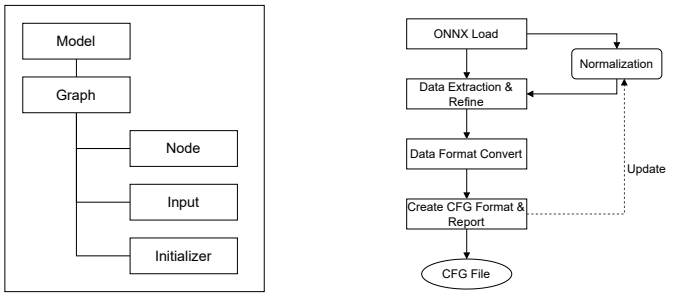
a state change is detected in the event routine handled by the event receiver, the session manager updates the state of the session.

Finally, in the Auxiliary Utility Manager, there are two auxiliary functions that support the platform: data augmentation and JSON parser. For data computation of neural networks, the platform supports a data augmentation function that increases the number of training images by transforming images for training [31]. The JSON parser converts the model file and weight file into JSON format. The JSON-formatted models and weight files are used to check the number of neural network layers and information of each layer in the visualization process of the neural network.

3.2. Inference Engine

The structure of the inference engine part is composed of ONNX Converter, NN builder, ESL builder, and Inference Engine. In general, since the DNN framework uses an individual model representation and storage method, it is difficult to use a DNN model developed in one framework, in another framework. To increase the interoperability of models between frameworks, an open form of intermediate representation is required. ONNX [24] provides framework-independent model representations for sharing models developed in various frameworks. To increase interoperability between frameworks through ONNX, the model developed in a certain framework must be first converted into an ONNX model, and the ONNX model needs to be converted again into a model suitable for another framework.

ONNX is described as Google's Protobuf [32] and supports about 200 operations corresponding to the layers of the neural network. The definition of a model in a neural network is not simply a list of data, but requires additional description of what the data means. Figure 3a shows the main components of ONNX, in which important information in ONNX model configuration are Node, Input, and Initializer, shown in the figure. Node generally acts as a layer in a neural network, Input represents dimension information of a matrix, and Initializer stores weights. Each element is a hierarchical object structure, and a model is constructed by grouping related objects into a doubly linked list. Our framework provides an ONNX converter, which converts the ONNX model into deep learning models executable in our inference engine. The ONNX converter uses the ONNX model to create networks, weights, bias, etc. according to the format suitable for NN builder. Figure 3b shows the operation flow of the implemented ONNX converter. It consists of four steps: ONNX Load, data extraction and purification, conversion to specific model format, and CFG format creation. The ONNX loader plays the role of loading the ONNX model file. After loading the ONNX model file, in the data extraction and purification process, it traverses all the nodes, inputs, and initializers inside the ONNX model to extract information about the layer type, name matrix size, and so on. During extraction, the ONNX model data is normalized to be fixed as the proper model input format. The extracted and refined information is converted into a form that can be managed collectively by putting the entire layer information in one internal object. After that, it is finally converted into a CFG file format that can be used in the inference engine.



(**a**) Basic ONNX structure composition and model description

(**b**) ONNX converter and operation flow

Figure 3. Basic ONNX structure composition and model description, and developed ONNX converter and its operation flow.

The NN builder performs NN operations through the inference engine using the generated configuration of network model by ONNX converter. The inference engine provides model information such as network layer and weight parameter to the back-end web interface so that the user can monitor and change it at the GUI platform part. To explore and customize the network models for edge device, the inference engine provides features such as parameter quantization, layer-cross execution, layer-stepwise execution, and parameter editing. The quantization method applied in the inference engine is the combination of Log2 distribution method and the KL diffusion method, which was applied to NVDLA architecture [17]. Log2 distribution is a method that chooses the largest area which has scattered values in 8 levels (8 bits) among the values obtained by taking logarithmic values of the floating-point data, and KL diffusion technique is a method of minimizing the difference in the amount of information between two transformed data by applying the KL diffusion formula to obtain cross entropy.

The inference engine supports three inference methods: floating point-based inference, hybrid inference, and integer (quantized) inference. The hybrid method means that quantization of input/output parameters is applied for the convolution layer, but floating-point operation is performed on the remaining layers. The general inference operation proceeds with network input, network constructing, preprocessing, inference operation, outputting the result and saving to a file, and saving image. The common flow of the three inference methods is the same, but it differs in the preprocessing part and the file storage part. In the case of floating-point inference, only batch normalization is performed in preprocessing, but in the case of hybrid inference, batch normalization and quantization are performed. In the case of integer inference, the whole quantization of the image is also performed. When performing inference, for each individual layer, the input, weight, and output results for the layer are stored as separate files, so the contents are used to analyze the performance of each inference.

Another feature supported by the inference engine is cross-inference. The cross-inference refers to a method of proceeding with step-by-layer inference by applying different inference methods between layers. Cross-inference is made at breakpoints on a layer-by-layer basis, like breakpoints in program debugging. The inference engine uploads the inference model before inference starts and sets the basic format of inference for each layer. Figure 4 shows an example of cross-inference. In the figure, the breakpoint is set to layer 3. Layers from 0 to 2 are designated as float-point inference, layer 3 and 4 are designated as integer inference, and layer 5 is designated as float-point inference again. In this case, when the inference engine starts inference, it performs float-point inference in layers 0, 1 and 2 and stops. In layer 3, integer data is generated by performing quantization operation on the weight parameters that were specified as float-point values, and then neural network operation is performed on the generated integer data. In layer 5, de-quantization is performed to make float-point inference again.

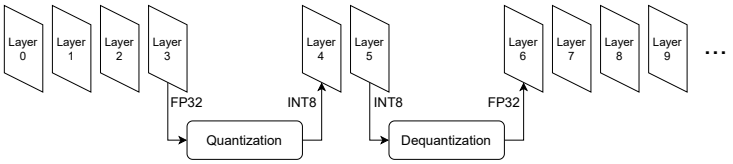


Figure 4. Cross inference procedure with quantization and de-quantization.

The Inference Engine and NN builder also have an interface that delivers the neural network model to the ESL (Electronic System Level) builder. We implemented a hardware virtual platform at the system level to analyze various performance issues for executing neural network operations on edge device hardware. The ESL builder is responsible for creating a network models for executing neural network operations on this virtual platform. When the inference engine performs neural network operation for each network layer, the user selects whether to perform it through the inference engine (actually, this is done by CPUs and GPUs of server running inference engine) or through hardware DLA VP. If the user chooses to execute the neural network operation with the hardware DLA VP, the inference engine delivers the network model and parameters to the ESL builder. The ESL builder converts the received model and parameters into a specific format of model and parameters so that they can be executed in the VP, then transmits them to the VP through the socket interface. When the corresponding neural network operation is completed, the VP sends the results back to the ESL builder. Then, the ESL builder sends the corresponding result back to the front-end through via inference engine and displays the result on the GUI screen.

3.3. DLA Platform Part

We provide a function with Virtual Platform (VP) to estimate and predict the performance of the NN model according to the hardware configuration of embedded edge device. As shown on the right side of Figure 1, the hardware VP operates as an independent module. The virtual platform basically includes an embedded CPU Core, a TLM 2.0 Bus module, an Interrupt controller, and a Memory interface with DMA (Direct Memory Access) controlling. The CPU core module supports RISC-V's RV32IM instruction set [33,34] and provides interrupt handling and a system call interface to software running on a virtual platform.

There are various edge device platforms widely used in IoT systems, which are generally composed of a low-spec embedded processor, memory and interfaces. Among embedded processors used in the research area, RISC-V [33,34] is an emerging processors since it provides well structured and expandable instruction set architecture for edge device, as well as it is on open source architecture. So, in the framework, a RISC-V-based edge device system is modeled and configured as a virtual platform, and based on this platform, we developed DLA VP to provide dedicated deep learning accelerator that can accelerate convolutional neural network operations. Recently, the RISC-V Virtual Platform was developed based on SystemC [35], which is efficient for system verification in a relatively short time. It was designed with a generic bus system using TLM 2.0 around RISC-V RV32IM core, so it is very expandable and configurable platform that can extend other TLM-connected modules for verifying special functions in the RISC-V VP environment. We developed a DLA platform based on the RISC-V VP platform [35] by integrating the Deep Learning Accelerator (DLA) module into the virtual platform. As shown in Figure 1, it is connected to the TLM 2.0 Bus through the target port and is allocated to a part of the address range of the RISC-V CPU core. The development of DLA VP platform was previously published, and we have integrated the DLA VP into the framework in this paper. Please refer to the details [15].

The DLA module is composed of several modules to perform neural network operations such as convolution, activation, and pooling, as shown in Figure 5. The internal data structure of the DLA module includes GSFR (Global Special Function Register) and Buffers. GSFR is the memory-mapped register area that the RISC-V CPU core accesses by being

mapped to the CPU's address range. An application running on the VP can execute neural network models and individual operation with the DLA module by setting configurations of the network model to the GSFR area. The register values that can be set are as follows. For convolution settings, filter size, stride length, parameter type, and bias type can be set. The filter size can be set from 1×1 to 3×3, and the stride length can be set. In addition, the width, height, and size for the image are set, and the locations of the memory address of image and parameters such as weight and bias value are set in register set. The registers for activation and pooling configuration also exist within the register set. We can specify the activation type as a register value, in which the activation types supported by DLA are ReLU and leaky ReLU. The pooling type also can be specified as a register value, and pooling supports min, max, and average pooling. The pooling sizes supported by the DLA are 2×2 and 3×3. The internal Buffer is a buffering space used to temporarily buffering data for neural network operations. The buffer is subdivided into image, parameter, temp, CPIPE and APIPE data buffers according to the characteristics of the data.

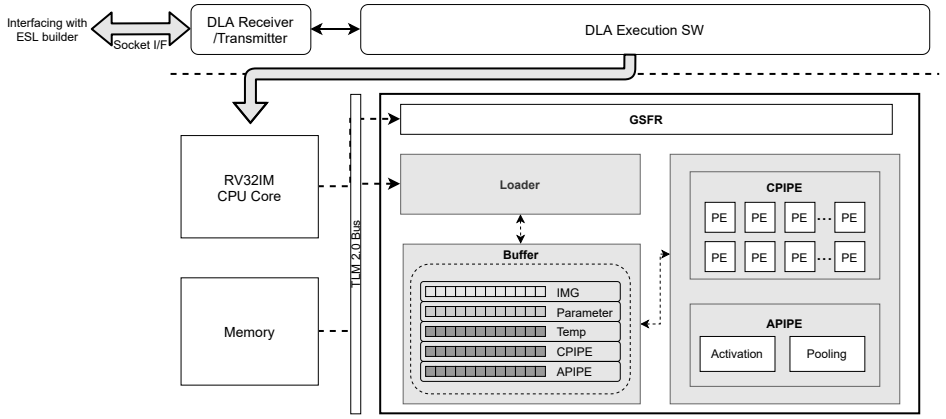


Figure 5. DLA architecture consists of GFSR, Buffer, Loader, and PIPE modules.

Th other Internal modules include the loader and PIPE modules. The loader module is an interface module that is connected to the outside of the DLA module. Between DLA and memory, all data for neural network operation is transmitted and received through the loader module through DMA operation. The loader module reads data from the main memory to process deep learning operations and writes back the resulting data to main memory. The PIPE module further divides into CPIPE and APIPE modules. The CPIPE module performs the convolution operation with loaded image data and weight parameters, and the APIPE module performs the activation or pooling operations. The element that performs the actual neural network operation in PIPEs is the Processing Element (PE) module. For pipelined and parallel neural network operation, vector-based MAC is performed by grouping groups of 4 units.

The interworking between the ESL builder and the DLA platform is also described in Figure 5. When the ESL builder transmits a specific neural network model and parameters converted in the format executed on the DLA platform to the DLA platform through the socket interface, the DLA Receiver/Transmitter of the DLA platform receives the corresponding information. This information is then delivered to the DLA execution SW, and it performs neural network operations through the DLA module. To execute neural network operation in the DLA module, a RISC-V-compiled software is required. The DLA execution SW is the software compiled by RISC-V toolchain, and thus can be executed on the RISC-V core. After completing the neural network operation, the results are then sent back to ESL builder via the DLA Receiver/Transmitter.

4. Experiments

In the framework architecture, the front-end and back-end are implemented in JavaScript as a web framework based on react, inference engine is implemented in C, and the DLA VP

for edge device simulation is implemented in SystemC. Since the front-end GUI platform provides a web environment, users use the framework through the web interface. Using the framework implemented in this paper, users may derive and analyze the results by changing and applying various factors affecting performance and overhead of various neural network models through a GUI interface, and users may estimate hardware performance such as memory buffer and operation time by running it on the hardware DLA VP. Interface features available to users include model conversion and visualizer, file manager, inference settings such as normal inference, stepping inference, cross inferences, and execution with DLA VP platform.

4.1. GUI Interface of Front-End

The initial interface of the framework and the arrangement of interfaces for each function are shown in Figure 6. As shown in the figure, the initial interface has four elements according to each function: user login, visualizer, file manager, and inference. The user login is an interface for user management, and the visualizer is a user interface that graphically displays information about the network model being used. File manager is an interface to manage neural network models in the framework. As shown in the figure, the users may create, delete, and manage neural network models through the file manager interface. The inference interface executes inference operations of the selected model among several models in the framework. This GUI interface provides configuration options to set model selection, cross inference, inference parameter type, and stepper option for layers, so users can try to perform inference operation according to the configuration.

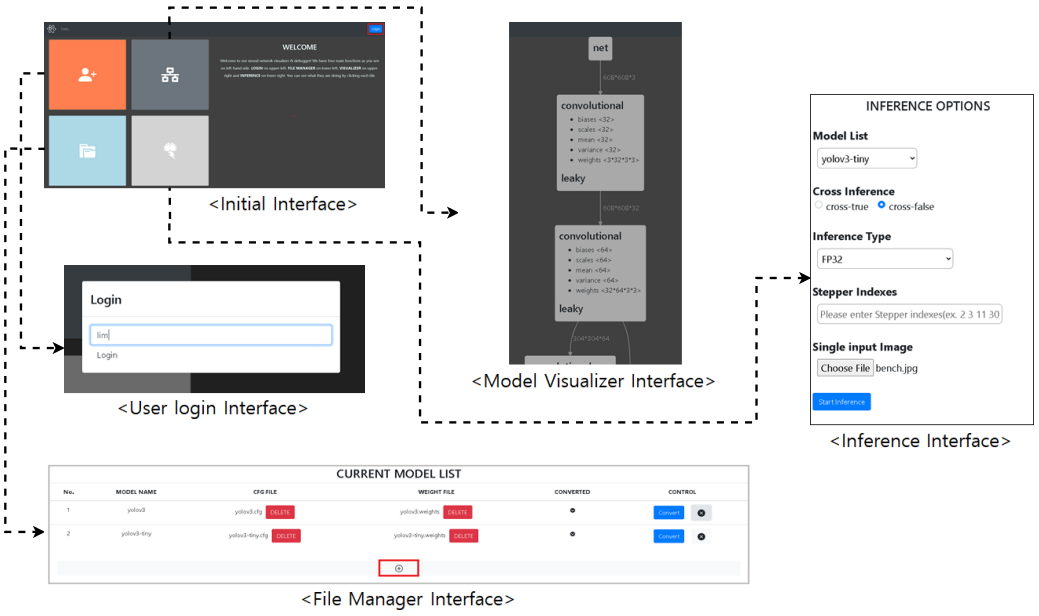


Figure 6. The initial interface of the framework and the arrangement of interfaces for each function: user login, visualizer, file manager, and inference.

4.2. Inference Engine

To run various neural network models in the inference engine or hardware DLA VP provided in one platform, the model needs to be converted into an executable configuration in the engine, which is performed by the ONNX converter and NN builder provided in our framework. Figure 7 shows the conversion progress and results of the neural network model using the file manager interface that performs model conversion. As shown in the figure, the file manager interface has the neural network models used in the framework as a list and provides a button to add a neural network model at the bottom of the GUI. When the button is clicked, an empty model is added to the list. After that, the user sets the model by inputting the CFG file and the parameter file for the model. The CFG file

and parameter file are configuration and parameter information of the model converted by ONNX convert. After that, when the user clicks the 'Convert' button on the right, the model is converted into a model format that can be used in our framework.

Figure 7. File manager inferface and converting procedure of a specific neural network model.

The neural network models in the list are run in the inference engine or DLA platform, and the results can be analyzed graphically through the GUI screen. User interface of the framework running inference is shown in Figure 8. The left part of the figure describes user interface for configuring network model to execute inference, the configurations include selecting model, checking to options of cross inference, selecting parameter type, and checking to option of stepper inference mode. As shown in left part of the figure, the framework provides users with an interface to select the neural network model from the model list. For the selected model, it helps to explore and customize the model or a specific layer of the model through interfaces such as inference method, inference type with quantization, and stepper index. First, you can set the inference type to either cross-true or cross-false. Cross-true is a method that can specify different inference types for each layer of a neural network model, but cross-false is a method that cannot specify differently. The inference type that can be specified is a floating-point or integer type. In the stepper indexes field, you put a specific layer on which to perform neural network operations. Then, an image is selected as a target for neural network operation. In addition, it is possible to select whether the corresponding layer setting is performed in the software-based inference engine or the hardware DLA VP platform through the button shown in the upper middle part of Figure 8.

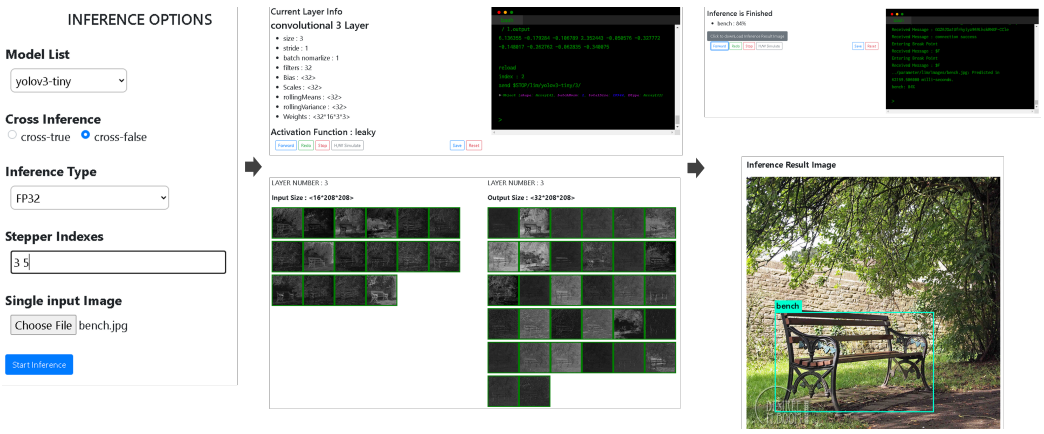
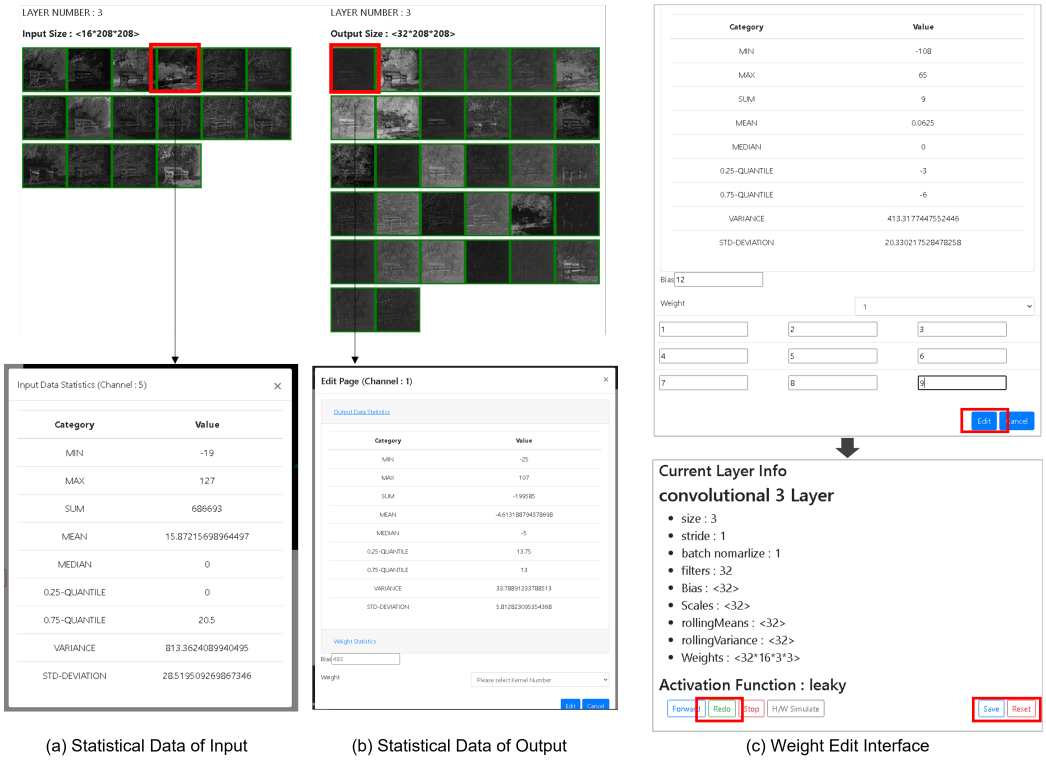


Figure 8. GUI interfaces to setting inference options, running inference through inference engine or hardware DLS VP, and its resulting display.

After the setting for inference, when you click the blue button 'start inference', the layer information being executed is displayed such as filter size, stride size, number of filters, batch size, means and variances, and inference is processed according to the setting, as shown in the upper middle part of Figure 8. The user can also identify the inference procedure being performed through the console screen. When the inference is finished, a message indicating that the inference has been completed is displayed, and the results created by the layer execution is displayed for each channel according to the settings, which

is shown in the lower middle part of Figure 8. After performing the operation on the current layer, the user proceeds with the desired operation by using one of the four buttons: forward, redo, stop, and H/W simulate, shown in the middle of the figure. The forward button proceeds inference to the next layer, and the redo button performs the current layer again. The stop button stops and ends the inference. The H/W simulate button lets the framework perform the HW simulation using the HW platform for the current layer.

We can analyze the inference operation on a specific layer with the statistical information of the input and output as well as the resulting images. As an example of the result, Figure 9 shows the results of performing the inference operation on layer 3 of the yolov3-tiny neural network model. As shown in the figure, for a specific layer of the neural model, it can be analyzed by displaying the inference operations according to the configuration of the layer. In addition, individual statistical analysis of the input data and output data of the corresponding layer is possible. As shown in Figure 9a,b, layer 3 of yolov3-tiny is a convolution operation that generates 32 output channels for 16 input channels having a size of 208 × 208, so the framework displays the result as an image for each channel and individual statistical information is identified by clicking the image of the corresponding channel in both input and output.



(a) Statistical Data of Input (b) Statistical Data of Output (c) Weight Edit Interface

Figure 9. The results of performing inference of specific layer of selected neural network model.

After analyzing the results and image information of the corresponding layer, we can perform the inference again with changing parameter values such as weight and bias. If you click the 'redo' button after changing some bias or weight values through the parameter editing GUI interface, inference operation of the corresponding layer is performed again, as shown in Figure 9c. With those features, you can perform the customization procedure and analysis like quantization or editing parameter values.

As an example of application to the framework, an object detection application called Darknet [36], which is widely used for object detection in embedded edge devices, is applied to our framework by applying the model converting of Darknet yolov2, yolov3, and yolov3-tiny into our framework. We performed various features such as visualization, file management, inference, and so on, and it was identified that most of the features provided by the framework operated normally. The summary of applying features for several

versions of Darknet object detection deep neural network application to our framework is shown in Table 1. As shown in the table, it is identified that most of the features provided by the framework are operated normally. The features include file management, inference method including hybrid inference, stepping inference, and cross inference with different type of parameters, and visualization of the network model. In addition, the framework also has augmentation features. The features such as visualizer, file manager, and augmentation are functional parts, and the functions operate normally. For the inference, it indicates whether object detection is performed properly through the framework. In the case of yolo2 and yolov3, inference results were not obtained properly by the cross inference, which we need to analyze the networks with parameters of cross inferencing. For the case of yolo3-tiny, all the features of our framework were properly executed. The inference operations with hybrid, stepping, and cross settings give proper results, and we could analyze the inference results.

Table 1. Summary of features in the framework with several versions of Darknet object detection deep neural network applications.

Features	yolo2	yolo3	yolo3-Tiny
File Manager	O	O	O
Steping Inference (FP, HYB, INT)	△	△	O
Cross Inference	X	X	O
Visualizer	O	O	O
Augmentation	O	O	O

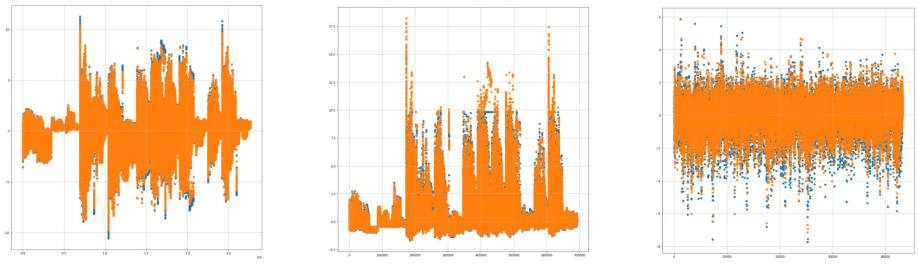
4.3. Results of DLA VP

The hardware DLA VP platform can simulate the usability of an edge device for neural network operation, and through this, it provides a direction for estimating performance and customizing the actual edge device. First, to check whether the implemented hardware DLA VP normally executes the neural network operation, we performed specific neural network operation such as Convolution Neural Network (CNN) operation, activation, and pooling which corresponds to a specific layer of the yolov3-tiny neural network model on our platform. Specifically, the neural network operations for layers 0, 1, and 13 of the yolov3-tiny neural network models are executed on both the inference engine and the hardware DLA VP, and the results are analyzed. Layer 0 performs the convolution operation using 16 3×3 filters on 416×416 image with 3 channels to output 416×416 image with 16 channels, and layer 1 performs max pooling operation on 416×416 image with 16 channels to output 208×208 image with 16 channels. Layer 13 generates an output of 13×13 image with 256 channels by performing convolution operation using 256 13×13 filters on 13×13 image with 1024 channels.

Figure 10 shows the distribution of image pixel values resulting of each neural network operation of layer 0, 1, and 13. In the figure, the blue point is the distribution of the result performed by the inference engine, and the orange point is the distribution of the neural network operation output performed in the hardware DLA VP platform. As shown in the figure, most of the pixels of each neural network operation overlap, indicating that the neural network operation performed by the hardware DLA VP platform is appropriate. In addition, when performing the neural network operation in the inference engine, we set the parameters with 32-bit floating point, while we set the parameters with 8-bit integer when performing neural network operation in hardware DLA VP to see the quantization effects on the neural network operations. From the figure, it is identified that the results of neural network operations overlap in most areas, which means that parameter simplification using quantization does not significantly reduce the accuracy of neural network computation on edge devices. When we performed a quantitative similarity check using the cosine similarity method [37], it was confirmed that the similarity was at least 85%. In summary,

from the experiments, we identify that the implemented hardware DLA VP platform normally executes deep neural network operations, and users can perform simulation on the edge device using the hardware DLA VP platform for a specific layer of the neural network model through the GUI interface and analyze the results by comparing the results of inference engine.

Next, we estimated the performance of various metrics for the hardware DLA VP platform with neural network operations. If you look at many existing DLAs, in the detailed implementation the DLA may differ from one another; however, the components of DLA are similar in general. Although our DLA VP could not represent all other DLAs or embedded edge devices, the quantitative analysis of performance with our DLA VP can be used as a reference for designing neural network applications in embedded edge devices. The purpose of DLA VP is to figure out how much performance effect is derived if various neural network models and operations are executed on an embedded edge device, and to give insight to optimize the neural network model for embedded edge devices.



(**a**) Inference Engine vs. DLA VP platform for yolo-tiny3 layer 0

(**b**) Inference Engine vs. DLA VP platform for yolo-tiny3 layer 1

(**c**) Inference Engine vs. DLA VP platform for yolo-tiny3 layer 13

Figure 10. Comparison of each pixel values of inference operation with inference engine and DLA VP platform that execute layers 0, 1, and 13 of Darknet yolov3-tiny: (**a**) CNN of 416 × 416 size with 16 filters, (**b**) max pooling of 416 × 416 size, (**c**) CNN of 13 × 13 size with 1024 filters. In the figure, pixel values of inference engine are plotted with blue points while pixel values of DLA VP are plotted with orange points.

Specifically, our DLA is based on the RISC-V processor, and the DLA controller architecture consists of the Loader, CPIPE, and APIPE to perform neural network operations. Each module is divided into sub-modules in detail, and we define operation times at ESL levels by analyzing the detailed architecture and data flow performed in DLA for each module. Table 2 shows the sub-modules within the DLA controller, and summarizes the operation times consumed by each sub-module of DLA. Of course, operation times are changeable manually according to modeling, and performance estimation may also vary according to the changed values. As shown in the table, the Loader module is internally composed of Router, Data, and Requester modules. The CPIPE module consists of con2CPIPE, 4 PEs, and CPIPEDone, and the APIPE module consists of con2APIPE, 4 PEs, and APIPEDone. The PE modules in CPIPE and APIPE are modules that perform convolution or activation/pooling operations. In this experiment, it was set for 4 PEs to perform parallel operation with the data path outside, which may be changed by DLA configurations.

Table 2. Configuration parameters of sub-modules within the DLA for hardware simulation.

(Unit)				**Loader Module**			
Memory Delay per Byte				Router	Data	Requester	Sum
5				11	43	10	64
CPIPE				APIPE			
con2CPIPE	4 PEs	CPIPEDone	Sum	con2APIPE	4 PEs	APIPEDone	Sum
555	1114	67	1736	208	399	100	707

The performance metrics provided by the hardware DLA VP include the amount of data transmitted to the DLA, the number of calls for each module, and the estimated execution time of each module according to a specific neural network operation. These metrics can be monitored through the GUI interface. Based on those settings of operation times, we performed the convolution neural network operation and the subsequent max pooling operation for layer 0 and layer 13 of yolov3-tiny on the hardware DLA VP, and estimated the performance with those metrics. The experimental results for each performance metric are described in Table 3. As shown in the table, we can compare and analyze the performance metrics estimated when performing neural network computations in hardware DLA VP with respect to the configuration of specific neural network computations. Specifically, when performing neural network operation, according to the configuration such as image size, the number of channels, and the number of outputs, we get inspiration about what is the distribution of the transferring amount through the data buffer, and what is the distribution of the call count of each internal module in the DLA module in detail. From the experimental results, we can compare the performance metrics such as relative amount of data buffer or parallelism levels of PEs running neural network operation by comparing the configurations of layers like layer 0 consisting of large images with small channels and outputs while layer 13 consists of small images with large channels and outputs. According to this quantitative analysis, we can have insight into determining how much to size the data buffer and how many PEs to perform neural network calculations according to the layer configuration of the neural network model. By doing this, more optimized neural network application will be possible by determining in real time how much buffer is required or how to parallelize the PE on an embedded edge device.

Table 3. Simulation results for the configured parameters. It executes layer 0 and 13 of yolov3-tiny with hardware DLA VP.

Metrics	Layer 0 (416 × 416 with 3 Channels, 16 Outputs)			Layer 13 (13 × 13 with 1024 Channels, 256 Outputs)		
Amount of Data transfer(KB)	IMG		2028	IMG		169
	Para.		1.125	Para.		6144
	Temp		5408	Temp		338
	CPIPE		2704	CPIPE		42.25
	APIPE		2704	APIPE		169
Call Counts per Each Module(#)	Loader		34	Loader		276
	CPIPE		359,424	CPIPE		3,407,872
	APIPE		232,960	APIPE		6656
Estimated Execution Time (unitx10^6)	Loader		65.76704	Loader		35.13472
	CPIPE	CON2CPIPE	199.48032	CPIPE	CON2CPIPE	1891.36896
		4PEs	400.398336		4PEs	3796.369408
		CDONE	24.081408		CDONE	228.327424
	APIPE	CON2APIPE	48.45568	APIPE	CON2APIPE	1.384448
		4PEs	92.95104		4PEs	2.655744
		ADONE	23.296		ADONE	0.6656

5. Conclusions

Since DNN operation generally requires a lot of computing resources, a high-performance computing system is mainly used for DNN-based applications. However, recently, applications using DNN in low-spec computer systems such as embedded edge devices are increasing. One of the instances is IoT systems. DNN operation can be divided into training and inference. In the case of DNN application in the IoT system, training is mainly performed in the server and inference is performed on the edge device. However, edge devices still take a lot of loads in inference operations due to low computing resources. A lot of research has been done to make the DNN operation work normally in the embedded edge device. In terms of the DNN algorithm, research has been conducted mainly on applying algorithm enhancement such as neural network layer compression, parameter quantization, or editing in neural networks. On the hardware side, there was a study adding an independent Deep Learning Accelerator for hardware acceleration in edge devices. However, there are few integrated frameworks that can explore and customize inference operations for various DNN models to optimize DNN in edge devices.

In this paper, we developed an integrated framework that can explore and customize inference operations of various CNN models on embedded edge devices. The framework provided in this paper consists of the GUI interface part, inference engine part, and hardware DLA VP part. As a key feature, by applying the ONNX-based standard model expression, various existing DNN models can be integrated and compared into our framework and provide integrated interoperability for various neural network models. The inference engine has several neural network customization techniques for embedded edge devices such as hybrid quantization, stepper and cross-inference functions. In addition, hardware performance estimation is possible using the hardware DLA VP platform. Since all interfaces are provided as web-based GUIs, users utilize them through the web. Currently, the framework and HW DLA VP have limitations in that they operate on only CNN models. There are some cases in which normal results were not obtained for specific

features. In further work, we plan to expand the framework to be able to operate on a wider field of DNN inference operations with edge devices.

Author Contributions: Conceptualization, S.-H.L. and S.-Y.C.; methodology, S.-H.L.; software, S.-H.K., B.-H.K., J.R. and C.L.; validation, S.-H.K., B.-H.K., J.R. and C.L.; formal analysis, S.-H.L., S.-H.K., B.-H.K., J.R. and C.L.; investigation, S.-Y.C.; resources, S.-H.K., B.-H.K., J.R. and C.L.; data curation, S.-H.L., B.-H.K., J.R. and S.-Y.C.; writing—original draft preparation, S.-H.L.; writing—review and editing, S.-H.L. and S.-Y.C.; visualization, S.-H.L.; supervision, S.-H.L. and S.-Y.C.; project administration, S.-H.L. and S.-Y.C.; funding acquisition, S.-H.L. All authors have read and agreed to the published version of the manuscript.

Funding: This work was supported by the MSIT (Ministry of Science and ICT), Korea, under the National Program for Excellence in SW, supervised by the IITP (Institute of Information & Communication Technology Planning & Evaluation)(2019-0-01816). This work was funded by Genesys Logic. Inc. This work was supported by the National Research Foundation of Korea (NRF) grant funded by the Korean government (MSIT) (NRF-2021R1F1A1048026). This work was supported by Hankuk University of Foreign Studies Research Fund.

Conflicts of Interest: The authors declare no conflict of interest.

Sample Availability: Samples of the compounds are available from the authors.

Abbreviations

The following abbreviations are used in this manuscript:

DNN	Deep Neural Network
GUI	Graphic User Interface
DLA	Deep Learning Accelerator
ONNX	Open Neural Network Exchange
VP	Virtual Platform
MAC	Multiply-Accumulate
AI	Artificial Intelligence
CNTK	Microsoft Cognitive Toolkit
DIGITS	Deep Learning GPU Training System
GPU	Graphica Processing Unit
CPU	Central Processing Unit
NN	Neural Network
ESL	Electronic System Level
TVM	Tensor Virtual Machine
FP	Floating Point
INT	Integer
REL	Router and Event Listener
PM	Process Manager
FM	File Manager
SM	Session Manager
AUM	Auxiliary Utility Manager
JSON	JavaScript Object Notation
CFG	Configuration
RISC-V	Reduced Instruction Set Computer
TLM	Transaction-level Modeling
GSFR	Global Special Function Register
ReLU	Rectified Linear Unit
PE	Processing Element
CNN	Convolution Neural Network

References

1. Sundaravadivel, P.; Kesavan, K.; Kesavan, L.; Mohanty, S.P.; Kougianos, E. Smart-log: A deep-learning based automated nutrition monitoring system in the IoT. *IEEE Trans. Consum. Electron.* **2018**, *64*, 390–398. [CrossRef]

2. Han, S.; Mao, H.; Dally, W.J. Deep Compression: Compressing Deep Neural Networks with pruning, trained quantization and Huffman coding. *arXiv* **2015**, arXiv:1510.00149.
3. Gong, Y.; Liu, L.; Yang, M.; Bourdev, L. Compressing deep convolutional networks using vector quantization. *arXiv* **2014**, arXiv:1412.6115.
4. Chandrasekhar, V.; Lin, J.; Liao, Q.; Morère, O.; Veillard, A.; Duan, L.; Poggio, T. Compression of Deep Neural Networks for Image Instance Retrieval. In Proceedings of the Data Compression Conference, Snowbird, UT, USA, 4–7 April 2017; pp. 300–309.
5. Kim, H.; Jo, G.; Lee, H.; Shin, D. Filter-Wise Quantization of Deep Neural Networks for IoT Devices. In Proceedings of the 2021 IEEE International Conference on Consumer Electronics, Las Vegas, NV, USA, 10–12 January 2021.
6. Howard, A.; Zhu, M.; Chen, B.; Kalenichenoko, D.; Wang, W.; Weyand, T.; Andreetto, M.; Adam, H. MobileNets: Efficient Convolutional Neural Networks for Mobile Vision Applications. *arXiv* **2017**, arXiv:1704.04861.
7. Chollet, F. Xception: Deep Learning with Depthwise Separable Convolutions. In Proceedings of the 2017 IEEE Conference on Computer Vision and Pattern Recognition (CVPR), Honolulu, HI, USA, 21–26 July 2017; pp. 1800–1807.
8. Vanhoucke, V.; Senior, A.; Mao, M.Z. Improving the speed of neural networks on CPUs. In Proceedings of the Deep Learning and Unsupervised Feature Learning NIPS Workshop, Granada, Spain, 12–17 December 2011; Volume 1, pp.1–8.
9. Kim, E.; Lee, K.H.; Sung, W.K. Recent Trends in Lightweight Technology for Deep Neural Networks. *Korean Inst. Inf. Sci. Eng.* **2020**, *38*, 18–29.
10. NVIDIA Corporation. Nvdla Open Source Project. 2018. Available online: http://nvdla.org/primer.html (accessed on 30 July 2020).
11. Chen, Y.; Yang, T.; Emer, J.; Sze, V. Eyeriss v2: A Flexible Accelerator for Emerging Deep Neural Networks on Mobile Devices. *IEEE J. Emerg. Sel. Top. Circuits Syst.* **2019**, *9*, 292–308. [CrossRef]
12. Wu, N.; Jiang, T.; Zhang, L.; Zhou, F.; Ge, F. A Reconfigurable Convolutional Neural Network-Accelerated Coprocessor Based on RISC-V Instruction Set. *Electronics* **2020**, *9*, 1005. [CrossRef]
13. Li, Z.; Hu, W.; Chen, S. Design and Implementation of CNN Custom Processor Based on RISC-V Architecture. In Proceedings of the 2019 IEEE 21st International Conference on High Performance Computing and Communications; IEEE 17th International Conference on Smart City; IEEE 5th International Conference on Data Science and Systems (HPCC/SmartCity/DSS), Zhangjiajie, China, 10–12 August 2019; pp. 1945–1950.
14. Porter, R.; Morgan, S.; Biglari-Abhari, M. Extending a Soft-Core RISC-V Processor to Accelerate CNN Inference. In Proceedings of the 2019 International Conference on Computational Science and Computational Intelligence (CSCI), Las Vegas, NV, USA, 5–7 December 2019; pp. 694–697.
15. Lim, S.-H.; Suh, W.W.; Kim, J.-Y.; Cho, S.-Y. RISC-V Virtual Platform-Based Convolutional Neural Network Accelerator Implemented in SystemC. *Electronics* **2021**, *10*, 1514. [CrossRef]
16. Zhang, G.; Zhao, K.; Wu, B.; Sun, Y.; Sun, L.; Liang, F. A RISC-V based hardware accelerator designed for Yolo object detection system. In Proceedings of the 2019 IEEE International Conference of Intelligent Applied Systems on Engineering (ICIASE), Fuzhou, China, 26–29 April 2019.
17. Migacz, S. NVDLA 8-bit Inference with TensorRT. In Proceedings of the GPU Technology Conference, San Jose, CA, USA, 8–11 May 2017.
18. Nguyen, G.; Dlugolinsky, S.; Bobáket, M.; Tran, V.; Garcia, A.; Heredia, I.; Malik, P.; Hluchy, L. Machine Learning and Deep Learning Frameworks and Libraries for Large-scale Data Mining: A Survey. *Artif. Intell. Rev.* **2019**, *52*, 77–124. [CrossRef]
19. Erickson, B.J.; Korfiatis, P.; Akkus, Z.; Kline, T.; Philbrick, K. Toolkits and Libraries for Deep Learning. *J. Dig. Imaging* **2017**, *30*, 400–405. [CrossRef] [PubMed]
20. TensorBoard: TensorFlow's Visualization Toolkit. Available online: https://www.tensorflow.org/tensorboard (accessed on 1 August 2020).
21. Paszke, A.; Gross, S.; Massa, F.; Lerer, A.; Bradbury, J.; Chanan, G.; Killeen, T.; Lin, Z.; Gimelshein, N.; Antiga, L.; et al. PyTorch: An Imperative Style, High-Performance Deep Learning Library. In Advances in Neural Information Processing Systems 32 (NeurIPS 2019), Vancouver, Canada, 8–14 December 2019.
22. Yeager, L.; Bernauer, J.; Gray, A.; Houston, M. *Digits: The Deep Learning GPU Training System*; ICML AutoML Workshop: Lille, France, 11 July 2015.
23. Pico-CNN. Available online: https://github.com/ekut-es/pico-cnn (accessed on 1 August 2020).
24. ONNX. Available online: https://github.com/onnx/onnx (accessed on 1 July 2020).
25. Lim, S.H.; Kang, S.H.; Ko, B.H.; Roh, J.; Lim, C.; Cho, S.Y. Architecture Exploration and Customization Tool of Deep Neural Networks for Edge Devices. In Proceedings of the 40th IEEE International Conference on Consumer Electronics, Las Vegas, NV, USA, 7–9 January 2022.
26. Jia, Y.; Shelhamer, E.; Donahue, J.; Karayev, S.; Long, J.; Girshick, R.; Guadarrama, S.; Darrell, T. Caffe: Convolutional Architecture for Fast Feature Embedding. *arXiv* **2014**, arXiv:1408.5093.
27. Theano Development Team. Theano: A Python framework for fast computation of mathematical expressions. *arXiv* **2016**, arXiv:1605.02688.
28. Seide, F.; Agarwal, A. CNTK: Microsoft's Open-Source Deep-Learning Toolkit. In Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, San Francisco, CA, USA, 13–17 August 2016.

29. Abadi, M.; Agarwal, A.; Barham, P.; Brevdo, E.; Chen, Z.; Citro, C.; Corrado, G.S.; Davis, A.; Dean, J.; Devin, M.; et al. TensorFlow: Large-Scale Machine Learning on Heterogeneous Systems. *arXiv* **2016**, arXiv:1603.04467. Available online: https://www.tensorflow.org/ (accessed on 1 March 2020).
30. Apache TVM. Available online: https://tvm.apache.org (accessed on 1 December 2021).
31. Sharp, High Performance Node.js Image Processing. Available online: https://sharp.pixelplumbing.com/ (accessed on 1 December 2021).
32. Google, Protocol Buffers. Available online: https://developers.google.com/protocol-buffers (accessed on 1 December 2021).
33. Waterman, A.; Asanovi'c, K. *The RISC-V Instruction Set Manual; Volume I: User-Level ISA*; SiFive Inc. and CS Division, EECS Department, University of California: Berkeley, CA, USA, 2017.
34. Waterman, A.; Asanovi'c, K. *The RISC-V Instruction Set Manual; Volume II: Privileged Architecture*; SiFive Inc. and CS Division, EECS Department, University of California: Berkeley, CA, USA, 2017.
35. Herdt, V.; Groβe, D.; Le, H.M.; Drechsler, R. Extensible and Configurable RISC-V Based Virtual Prototype. In Proceedings of the 2018 Forum on Specification and Design Languages (FDL), Munich, Germany, 10–12 September 2018; pp. 5–16.
36. Redmon, J.; Farhadi, A. Yolo9000: Better, faster, stronger. In Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR), Honolulu, HI, USA, 21–26 July 2017.
37. Cosine Similarity. Available online: https://en.wikipedia.org/wiki/Cosine_similarity (accessed on 1 December 2020).