Automated Framework for Deep Learning Model Deployment on Edge Devices with Hardware Accelerator

Hoang Tran Viet
Viettel Group
Hanoi, Vietnam
hoangtv23@viettel.com.vn

Quang Le Hoang Minh
Viettel Group
Hanoi, Vietnam
quanglhm@viettel.com.vn

Du Tran Ngoc
Viettel Group
Hanoi, Vietnam
dutn1@viettel.com.vn

Thang Nguyen Minh
Viettel Group
Hanoi, Vietnam
thangnm35@viettel.com.vn

Trung Dong Quang
Viettel Group
Hanoi, Vietnam
trungdq8@viettel.com.vn

Kien Nguyen Trung
Viettel Group
Hanoi, Vietnam
kiennt453@viettel.com.vn

Abstract

Edge computing devices have seen exponential growth globally in recent years, thanks to their advantages such as enhanced data privacy, energy efficiency, and reduced response latency. Thus, deploying deep learning models on edge devices has emerged as an inevitable trend. However, major challenges are optimizing computation and memory usage on resource-constrained hardware while guaranteeing inference accuracy. To confront these challenges, we propose an automated deep learning framework for the deployment of deep learning models on edge devices having a hardware accelerator. Our proposed framework is a self-contained design flow consisting of model training from scratch, model compression techniques such as structural pruning, integer-only quantization, and firmware generation for a hardware accelerator using a hardware-specific compiler. Evaluations show that the proposed framework enhances the performance of deep learning models on an edge device with a hardware accelerator by up to 40 times faster than the complete processor-based inference while ensuring an accuracy tolerance of 3% compared to the corresponding full-precision model.

CCS Concepts

• **Software and its engineering** → *Compilers*; • **Computing methodologies** → Neural networks.

Keywords

Deep Learning, TVM-BYOC, Model Compression, AI Accelerator

ACM Reference Format:
Hoang Tran Viet, Quang Le Hoang Minh, Du Tran Ngoc, Thang Nguyen Minh, Trung Dong Quang, and Kien Nguyen Trung. 2025. Automated Framework for Deep Learning Model Deployment on Edge Devices with Hardware Accelerator. In *Proceedings of ACM SAC Conference (SAC'25).* ACM, New York, NY, USA, Article 4, 8 pages. https://doi.org/xx.xxx/xxx_x

1 Introduction

In recent years, Deep learning (DL) has been transforming key sectors, driving faster innovation, and profoundly improving efficiency across industries. DL models are usually deployed in a cloud-based using powerful computing servers. In this architecture, edge devices send data to a central cloud system for centralized DL inference. This approach places a computational burden on the network infrastructure due to a huge inference load that requires a large amount of data transferred between the cloud system and the edge devices. One solution for lessening the burden of inference and latency is to deploy DL models on edge devices [21], [22]. However, DL model development must meet the constraints of low power consumption on edge devices while achieving satisfactory performance.

To fulfill the constraints, dedicated DL hardware accelerators for edge devices have been developed. These accelerators can perform arithmetic computations between tensors parallelly. DL model optimization techniques are also applied such as structural pruning, quantization, and optimized firmware compilation process for hardware accelerators. However, these techniques are often implemented separately, thus making model development time-consuming. Furthermore, developers need to carry out steps manually, such as determining the pruning portion of the model's layers, integrating quantization steps, and refining the structure of the model's computation graph before compilation. Hence, it is necessary to design a self-contained optimization framework that automates all separated steps. In this paper, we propose a self-contained automatic framework that contains model training, optimization, and firmware generation. The key components of our framework include a joint training and compression pipeline that performs one-shot training from scratch, combining structural pruning and integer-only quantization; a customized open-source automatic compiler for the hardware accelerator's software development kit (SDK) to compile the compressed model into firmware; and the design of computational modules for DL hardware accelerators using the developed SDK.

Our paper is organized as follows: Section 1 introduces the background and the motivation of the framework. Section 2 presents related research. Section 3 describes our proposed framework. Section 4 evaluates our proposal both in software and hardware aspects. Section 5 concludes the paper.

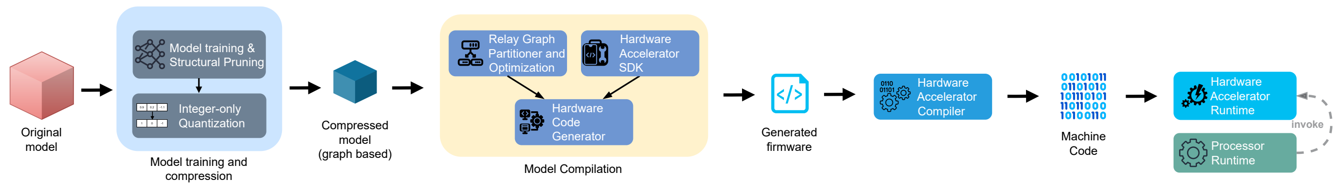
Original model
Model training & Structural Pruning
Integer-only Quantization
Model training and compression
Compressed model (graph based)
Relay Graph Partitioner and Optimization
Hardware Accelerator SDK
Hardware Code Generator
Model Compilation
Generated firmware
Hardware Accelerator Compiler
Machine Code
Hardware Accelerator Runtime
invoke
Processor Runtime

Figure 1: A process of deploying a DL model to an accelerated edge device.

2 Related Works

2.1 Model Compression and Compilation Pipelines

Intel's Distiller [24], an open-source Python library that provides tools for model compression, such as pruning and quantization techniques. The library allows DL developers to design custom compression pipelines for specific applications. In [12], Distiller's pruning method is used as the sparse compression algorithm to reduce memory usage and enhance inference speed on a hardware accelerator. However, Distiller does not consider compiling a DL model to the firmware. In [23], the authors provide a way to deploy DL models to edge devices by integrating quantization techniques, which are based on the RISC-V Packed Single Instruction, specifically, Multiple Data extension (P extension) for DL computations. This method introduces a 16-bit integer type and saturation instructions to replace the 32-bit floating-point type in the RISC-V's P extension. The authors also introduce an auto-tuning mechanism to select the optimal values for fixed-point processing. The research shed light on the potential of combining quantization and hardware optimization to improve inference on edge devices. However, the work optimized model compression only for a RISC-V-based hardware accelerator. In [11], the authors adapt TVM, an open-source compiler for DL models (i.e., DL compiler), for performing DL inferences on a DL accelerator [4]. Nevertheless, the project does not have a compression mechanism to optimize resource utilization in the hardware accelerator. Since edge devices often require a resource-effective runtime to facilitate DL models, the DL compiler must perform a sophisticated migration of quantized models to the hardware accelerator.

2.2 Model Pruning

Structural pruning removes an entire structured group of parameters, thereby reducing the size of DL models. Modern hardware accelerators employ structural pruning[1] since it is a resource-effective approach for implementation [6]. Unfortunately, this structural pruning usually requires complex node-wise constraints for the model's computation graph. When dealing with complex DL model designs, in which computational dependencies extend across multiple groups of layers, the pruning complexity grows exponentially. This is because all the layers in the related groups must be pruned together to maintain the structural integrity of the model. Holistic Filter Pruning (HFP) [7] allows a precise pruning of both parameters and matrix multiplications in DL models. HFP calculates the pruning rates based on model complexity and uses a gradient-based algorithm to distribute the sub-pruning rates accordingly. The model is then fine-tuned to guarantee accuracy while maintaining the pruned structure. DepGraph [8] focuses on a static pruning approach for a multi-stage training pipeline. The approach interprets DL model into a group of dependent layers by using a recursive graph traversal. To maintain the model's sparsity, the authors employ a group-level important criterion that can be learned through training, which safely removes unimportant groups with minimal performance degradation. Only Train Once (OTO) [3] uses a structural pruning method for DL models by generalizing network dependencies and applying grouping strategies. OTO provides automatic pruning and fine-tuning models with limited modifications of the model and its source code. Its framework is a one-shot joint training and pruning process that involves constructing a trace graph, grouping vertices into partitions for structural pruning, and applying a gradient descent-based algorithm to update parameters and generate a structural sparsity. The output model has a smaller size but inference performance is guaranteed. Instead of training and pruning sequentially as in considered previous works, our proposed framework can train and prune from scratch simultaneously, while ensuring fulfilled accuracy. In addition, to automate the DL deployment, the proposed framework does not require any intermediate modification, thus reducing development time.

[1]In a DL model, neural nodes are connected via structural parameters, consisting of weight and bias tensors. When a node is pruned, the parameters related to the pruned node are also eliminated. Therefore, the pruning process physically reduces the size of a DL model.

2.3 Model Quantization

The quantization process discretizes floating point parameter and activation tensors to decrease the computational complexity, which in turn reduces the energy consumption [16]. In our paper, a integer-only quantization is chosen to transform DL models such that they can be utilized by fixed-point accelerators and fulfill stringent resource constraints.

Integer-only quantization method uses an affine mapping from floating point numbers x to integers x_q as follows

$$x_q = round\left(\frac{x}{S} + Z\right), \tag{1}$$

where *round* is a rounding function, and scale factor S is the step size for quantization. In hardware, scale factor S can be considered as a combination of integer multiplication and right bit-shifting. Zero-point Z is an integer value and is used to ensure that the actual zero value is accurately represented during quantization. At the inference, parameter and activation tensors are in fixed point format. Specifically, the 8-bit fixed-point quantization is adopted as the baseline precision level [9]. Where the weight and activation tensors are quantized as *uint8* tensors, and the bias tensors are converted to *int32* tensors. We note that quantization scheme (1) uses quantization parameters S and Z for all integer values in a tensor. The quantization algorithm is also applied for all of the DL model's computational functions.

Quantization can induce information loss during the process and degrade accuracy. Quantization error exists because elements in a tensor can have significantly different magnitudes. So, in [15], authors propose a Cross-Layer Equalizer (CLE) to normalize DL model's parameter distributions across layers before quantization. To improve the precision of quantized values, AdaRound [14] is applied as a post-processing step to improve the precision. The algorithm fine-tunes parameter rounding decisions through a learning-based optimization so that it reduces errors from a trivial rounding-to-nearest method. Both CLE and AdaRound are integrated into the quantization pipeline.

2.4 Model Compilation

An implementation of a DL model on a hardware accelerator necessitates a DL compiler that can translate a high-level machine learning model into optimized executable code. Apache TVM [4] is an open-source DL compiler that provides various DL frameworks such as ONNX, TensorFlow, TFLite, and PyTorch. It optimizes DL models and generates binary executable files for various processor types and enables heterogeneous computing across multiple hardware components. To optimize DL models across diverse platforms, TVM generates a unified graph-based representation, so-called *Relay*, using different Python-based parsers for each framework [17]. Relay defines the data flow of machine learning operators and is represented in Python code, text format, or as an Abstract Syntax Tree (AST). The AST is a graph-based format and can be optimized through a graph-based optimization procedure, which includes removing redundant nodes, graph simplification, constant folding, and layout transformation.

TVM's BYOC [5] connects TVM and external runtime for different hardware accelerators. The key feature of BYOC is that it enables seamless hooks of the kernel library, compiler, and framework of the target hardware accelerator into TVM, thus allowing DL models to perform inference on the hardware accelerator. TVM can also optimize the execution for a DL hardware accelerator using BYOC. The idea is that TVM decomposes the Relay computation graph into partitions and offloads specific parts to the accelerator, while the other parts remain for the host processor using TVM's native compilation process. Therefore, TVM gives developers the freedom to invoke the external runtime and handle memory transfer between the processor and the accelerator.

3 Methodology

Our proposed automated DL deployment framework for a hardware accelerator in an edge device consists of two stages. In the model compression stage, an automatic structural pruning scheme removes structural groups of parameters based on heuristic criteria and a quantization pipeline that includes compression methods, such as AdaRound and CLE. Next, in the model compilation stage, parts of the model's computation graph are offloaded to a DL hardware accelerator's code generation process. The code generation

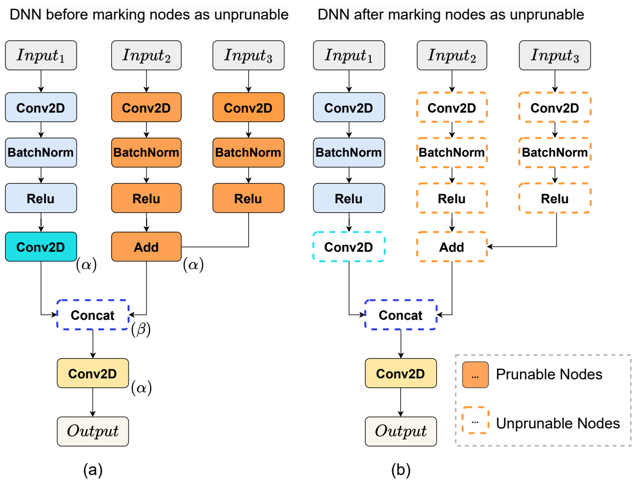

Figure 2: An example of the node marking process performed by the DFS algorithm. Nodes of the same color form a partition. (a) the computation graph before the marking process, (b) the computation graph after the marking process.

contains details of acceleration variables, arithmetic computations, and memory allocation on the target's runtime. In other words, the model's computational and data flow is realized on the hardware accelerator.

3.1 Model Compression Stage

The compression stage uses structural pruning as follows. Based on a gradient descent algorithm, a structural sparsity is formed by learning data properties, which efficiently saves training time because our training and pruning processes are performed simultaneously. We note that, in conventional methods, pruning often occurs after a training process and the pruning process takes several fine-tuning times to recover the desired accuracy. Thus, training effort is performed redundantly. However, it arbitrarily prunes the parent nodes, so their common child nodes may not operate properly. This is when the child nodes are operators that require operands with the same tensor's channel dimension (i.e., so-called α nodes), such as addition, subtraction, multiplication, convolution, etc. For nodes that do not require operands having the same tensor's channel dimension (i.e., so-called β nodes), such as split or concatenation, their parent nodes can be pruned differently. But, if the β node also has a direct or even indirect α node, we cannot prune randomly, thus posing pruning burdens. To address such a pruning problem, i.e., when a β node has as a direct or indirect α node, we propose AutoOTO algorithm to trace the parents of β nodes and their corresponding partitions, as shown in Fig.2). A partition is defined as a zero-invariant group, which contains nodes having the same level of pruning. If any parameter values of a zero-invariant group are zeros, their corresponding output values to the next layer are zeros as well [3]. So, instead of manually searching a pattern of α-after-β, our algorithm detects the pattern and marks

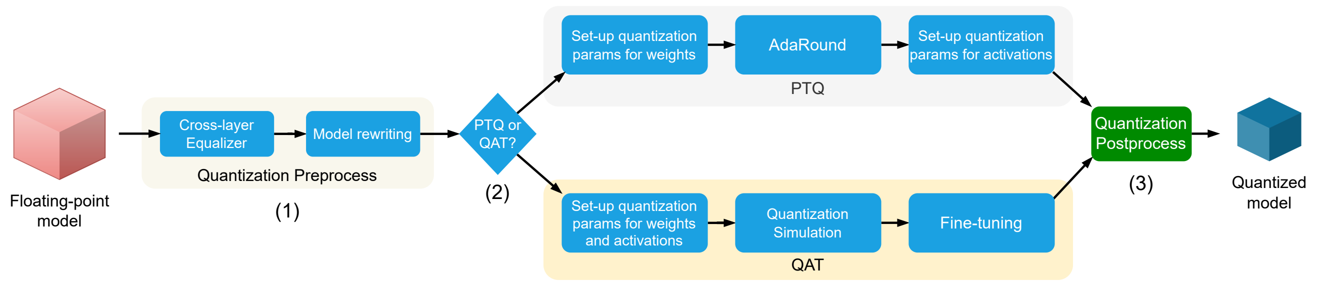

Figure 3: Model quantization pipeline.

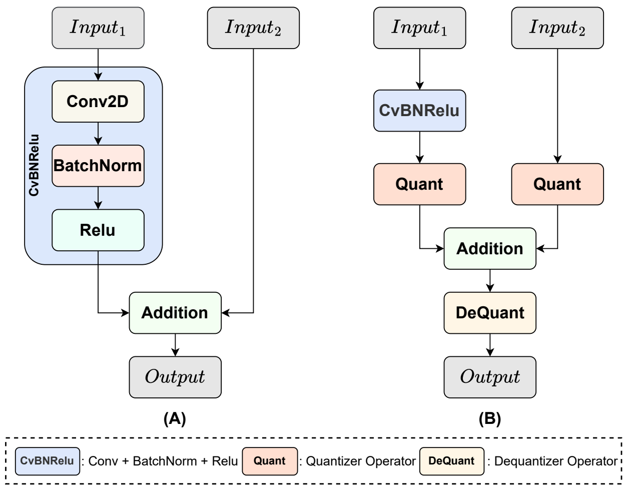

Figure 4: (a) the original model before the quantization, (b) the model after the post-processing.

all the parent nodes of all β nodes to fulfill the requirement, which is the tensor's channel dimension of a parent node's output and child node's input must be similar.

The algorithm can automatically trace and mark all node partitions as *unprunable* (e.g., the Conv2d node and a partition containing Add nodes). We note that the shape of unprunable nodes' parameter tensors (i.e., weight and bias tensors) must remain unchanged throughout the pruning process and fulfill the prerequisite requirement. Therefore, unprunable nodes are still trained but cannot be pruned. After obtaining the pruned model, quantization is performed based on a pipeline of modules' processing.

The model quantization pipeline consists of the following modules. For the quantization pre-process step (step (1) in Fig. 3) the model is imported into the CLE module to balance the parameter distribution across layers of the full precision model, thus making the quantization easier. CLE is important for models having separated layers and for per-tensor quantization, which favors our framework. Next, the *model rewriting* module traces the model and represents it as a computation graph to eliminate its inactive nodes. Then, the module inserts Quantizers and Dequantizer nodes, modifies operators that are not natively supported by the quantization framework, and implements operator fusion during the forward pass. The node insertion and operator fusion are illustrated in Fig. 4. Quantizer converts tensors from full-precision to fixed-point type, while Dequantizer does the reversed way. The module's outputs are the source code of the modified model.

After that, the model can be quantized using one of two quantization regimes, so-called Post-training Quantization (PTQ) and Quantization-Aware Training (QAT), the step is depicted (2) in Fig. 3. Users opt to choose one of them to perform a model quantization.

The PTQ regime involves three steps. First, a layer-wise mean square error (MSE) criterion is used to compute quantization parameters S and Z in equation (1) for all parameter tensors. We consider a set of quantization parameters to be optimal when its de-quantized tensor has the smallest MSE value compared to the original tensor. Then, AdaRound is applied to select the optimal rounded parameter values layer-by-layer. Finally, the quantization parameters are determined for all output activation tensors, layer-wise MSE criterion is used in this step to set these parameters, which requires several calibration iterations.

The QAT regime also involves three steps. First, we need to compute quantization parameters for both parameter and activation tensors, because a better initialization will help faster training and improve final accuracy. We employ layer-wise MSE criteria to generate initial quantization parameters. After that, all of the quantization parameters and parameter tensors are made learnable by the quantization process. The process consists of simulating the fixed-point operation as a floating point in the forward pass, while in the back-propagation process, the tensor values are kept as a floating point for the gradient-based update. Finally, the training process is performed with a smaller learning rate to make it converge to a better-quantized model.

Since the DL layer' structure provided by the output of QAT and PTQ is incompatible with the SDK for firmware generation, it is necessary to create a structure that the SDK favors. In our paper, we use the TFlite. After the optimal quantization model is obtained, the output computation graph has to contain operators in the operator-oriented format to comply with integer-only quantization regulations for efficient hardware acceleration. Thus, we convert the model to TFLite format using the *direct conversion module*. The module acts as a post-processing step, depicted as step (3) in Fig. 3. The idea behind this is that the module maps the operators of the model from the development environment format to TFLite equivalents and applies graph-level optimization iterations. The

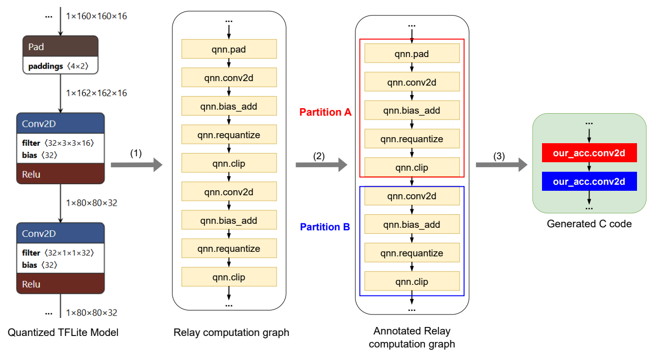

Figure 5: Three steps of the DL model compilation flow. (1) the Relay parsing process and the optimization passes. (2) pattern clustering and graph annotation. (3) graph partition, region merging and offloads composite regions into corresponding functions on the SDK.

Algorithm Pseudo-code for the Accelerator code generation

```
Class VisitNode:
function VisitExpr(Node)
    if Node.type == VarNode then
        res = VisitExpr_VarNode(Node)
    else if Node.type == TupleNode then
        res = VisitExpr_TupleNode(Node)
    else if Node.type == CallNode then
        res = VisitExpr_CallNode(Node)
    end if
    return res
end function

function VisitExpr_VarNode(Node)
    return Node.name
end function

function VisitExpr_TupleNode(Node)
    outs = []
    for field in Node.fields do
        outs.append(VisitExpr(field))
    end for
    return outs
end function

function VisitExpr_CallNode(Node)
    func_name = Node.kComposite
    args = GetArgumentNames(Node)
    return GenerateBody(Node.body, func_name, args)
end function

function GetArgumentNames(call)
    args_name = []
    for arg in call.args do
        res = VisitExpr(arg)
        args_name.append(res.name)
    end for
    return args_name
end function
```

optimization procedure includes constant folding, optimizing consecutive tensor transformations, and no-op fusion or elimination. The output is a TFLite model, an input for the compilation stage to generate executable code.

3.2 Model Compilation Stage

The compilation stage transforms a compressed DL model into firmware for the hardware accelerator. To adapt different model architecture designs, a compilation process for DL models often copes with various data flows between operators, even though DL models limit the number of operators. Without an automation process, the task of connecting the modules makes the firmware generation for the DL model time-consuming. Manual resource management for each operator reduces the flexibility of model deployment.

We propose an automatic model compilation framework that adapts to various models and reduces deployment time while ensuring inference accuracy. The proposed framework starts to convert the quantized TFLite model into Relay. Then, the Relay graph is partitioned into sub-graphs for the BYOC mechanism to generate accelerator code. The generated firmware from BYOC is compiled using the hardware accelerator's compiler to generate the machine code. The details of the proposed model compilation flow are illustrated in Fig. 5.

To generate firmware from a quantized TFLite model, the TFLite model is transformed into Relay using the Relay parser. It is done by mapping each TFLite operator to the corresponding Relay operator. Next, a series of graph-level optimizations are applied directly to the Relay representation to optimize the inference performance. During the conversion from TFLite to Relay, a single TFLite operator can be transformed by TVM into a group of Relay operators. Therefore, the Relay nodes are then virtually grouped into different clusters based on predefined patterns, so that the clusters belong to the corresponding computational modules. Once the clusters are formed, they are annotated with a *target* hardware platform to indicate the corresponding code generation process for the platform. Finally, the Relay graph is partitioned into different clusters, and operations in each cluster are merged into one composite module. And, the module is processed by the corresponding code generation process for the chosen target hardware platform to generate the final firmware. This firmware contains implemented modules of the hardware accelerator's SDK.

The details of firmware generation based on Relay are described as follows. Relay TVM can be divided into four main groups: VarNode, ConstNode, CallNode, and TupleNode. VarNode represents input tensors, ConstNode stores constant tensors such as weight and bias tensors, CallNode represents operators, whose inputs are either VarNodes or other CallNodes, and TupleNode holds tuples of VarNodes, ConstNodes, or CallNodes. We note that ConstNodes are ignored because constant values, i.e., weight and bias tensors, tensor width, tensor height, and tensor depth are automatically extracted from the TFLite model, and then converted into files used for configuring each operator in the generated firmware, so-called as configuration files.

Table 1: Experiments of model pruning method on object detection tasks.

Method	YOLOv5				YOLOv8				YOLOv9			
	Params.	FLOPs	mAP50	mAP	Params.	FLOPs.	Acc	mAP	Params.	FLOPs	Acc	mAP
Baseline	-	-	65.6%	42.4%	-	-	70.8%	50.9%	-	-	73.4%	51.0%
HFP	55.2%	67.2%	61.4%	39.0%	75.2%	68.8%	67.2%	47.1%	82.0%	79.9%	68.8%	48.7%
DepGraph	60.3%	72.2%	63.2%	40.2%	80.5%	75.6%	70.1%	49.6%	84.0%	82.2%	72.2%	48.9%
Ours	**53.0%**	**66.8%**	**65.1%**	**41.7%**	**70.0%**	**64.5%**	**70.5%**	**50.7%**	**80.0%**	**79.7%**	**72.8%**	**50.3%**

The firmware generation process is presented in Algorithm 1. First, the input node is passed to the general processing module VisitExpr for identifying the type of the input node. The identified node is forwarded to the appropriate VisitExpr module corresponding to that node type. If the input node is a VarNode, it is processed by the VisitExpr_VarNode module to generate the input variable's name in the firmware. For an input node of type TupleNode, the VisitExpr_TupleNode module sequentially processes each node in the tuple, invoking VisitExpr_VarNode or VisitExpr_CallNode depending on the node's type. If the input node is CallNode, it is processed by the VisitExpr_CallNode module. Since the argument of this node is the output of the node immediately before it, the node can be found by recursively traversing through the GetArgumentNames module to get the name of the input for this node. After obtaining the input name for the node, the GenerateBody function is used to generate declarations for a configuration file, SDK's module calls with the corresponding configurations, and memory allocations for each module, in the output firmware.

Memory should be allocated efficiently since edge devices have limited resources. The memory allocation for modules of SDK has two types, allocation for modules that are computed only one time and allocation for modules that must be stored, which is analogous to the memory management in a residual-based architecture. For the former type, memory will be released immediately after they are computed so that other modules can reuse the memory. For the latter type, memory is only released when the last module using it has finished computations.

At the end of the compilation stage, the generated firmware is compiled into a machine code via the accelerator compiler. In Fig.1, TVM is responsible for compiling the DL model from the TFLite format into the firmware. The compiled firmware has the main module that invokes other modules written in the accelerator's SDK. We note that the SDK's modules are processed following the same order as the ones in the TFLite's computation graph. The SDK's modules are represented similarly to the layers in DL models, and the computations in these modules are implemented following the integer-only quantization scheme and the per-tensor quantization level. Our accelerator's SDK supports most layers of the CNN family. The convolution module is implemented using Data Reuse techniques and SIMD instructions, enabling the processing of 64 operations simultaneously. The main challenge is to construct the Softmax and Sigmoid modules, both of which involve division and exponential computations. For the division operator, we apply the Newton-Raphson approximation method [1] thanks to its accuracy. For the exponential module, a Taylor series approximation is applied for its computational efficiency.

Table 2: Experiments of model pruning method on classification tasks.

Method	MobileOne			MobileNetv2			EfficientViT		
	Params.	FLOPs	Acc.	Params.	FLOPs.	Acc	Params.	FLOPs	Acc
Baseline	-	-	80.0%	-	-	71.8%	-	-	79.3%
HFP	52.1%	58.0%	76.4%	62.7%	67.4%	68.7%	**58.2%**	**62.6%**	76.0%
DepGraph	72.5%	80.2%	**79.6%**	70.8%	75.3%	68.9%	64.6%	69.3%	72.6%
Ours	**25.0%**	**31.5%**	79.2%	**30.0%**	**34.7%**	**70.2%**	68.1%	72.2%	**78.7%**

4 Experiments

4.1 Experimental Setups

To evaluate the effectiveness of the proposed framework for developing and deploying models on edge devices, we conduct object detection and image classification tasks. For the image classification comparison, we consider different DL model architectures, for example, CNN-based models such as MobileOne [20], MobileNetV2 [19], and the Transformer-based model such as EfficientViT [2]. For the object detection task, we perform experiments on a series of tiny models from the YOLO family [10], such as YOLOv5, YOLOv8, and YOLOv9. The ImageNet [18] dataset and the COCO2014 [13] dataset of human class are used to train the models for the classification and object detection tasks, respectively.

The model training is conducted on an Nvidia Tesla V100 GPU and public source code. In the pruning evaluations, all the considered pruning algorithms are configured such that the channel dimensions of parameter and activation tensors are an order of eight to optimize memory efficiency on the AI accelerator. We use the ZCU102 development kit as an edge device, which has a quad-core Arm® Cortex®-A53 processor and an AI accelerator designed to offer one tera operation per second of *uint8* and *int32* arithmetic inference. The experiments consist of two parts. First, we implement DL models on the proposed model compression pipeline and evaluate their effectiveness. Second, we demonstrate the efficiency of the compressed model accelerated on the edge device compared to the original floating-point model, which is hosted by the processor. We consider two evaluation criteria: execution speed and accuracy. We also compare the inference performance among the original model, the compressed model, and the optimized model for the edge device.

4.2 Experimental Results of Pruning Methods

We compare our results to those obtained from other pruning methods such as HFP [7], Torch-Pruning [8], and our AutoOTO method. As depicted in Table 1 and 2, our integrated compression method gives superior results compared to HFP and DepGraph for the classification and detection tasks, respectively. The models optimized by our AutoOTO achieve higher pruning ratios but gain better accuracy than HFP and DepGraph methods. Only for MobileOne model,

Table 3: Experiments of model compression flow on image classification tasks.

Model	Params.	Pruning	Quant.	Acc.	Acc. Loss
MobileOne	100%	-	-	80.1%	-
	25%	Ours	-	79.2%	-
	25%	Ours	PTQ	77.9%	2.2%
	25%	Ours	QAT	78.6%	**1.5%**
MobileNetv2	100%	-	-	72.0%	-
	30%	Ours	-	70.2%	-
	30%	Ours	PTQ	69.1%	2.9%
	30%	Ours	QAT	69.9%	**2.1%**
EfficientViT	100%	-	-	79.4%	-
	68%	Ours	-	78.7%	-
	68%	Ours	PTQ	76.7%	2.7%
	68%	Ours	QAT	77.6%	**2.2%**

Table 4: Experiments of model compression flow on object detection tasks.

Model	Params.	Pruning	Quant.	mAP50	mAP	mAP Loss
YOLOv5	100%	-	-	65.6%	42.4%	-
	53%	Ours	-	65.1%	41.7%	-
	100%	-	QAT	64.2%	40.9%	**1.5%**
	53%	Ours	PTQ	61.4%	37.7%	4.7%
	53%	Ours	QAT	63.7%	38.6%	3.8%
YOLOv8	100%	-	-	70.8%	50.9%	-
	70%	Ours	-	70.5%	50.7%	-
	100%	-	QAT	70.2%	50.3%	**0.6%**
	70%	Ours	PTQ	67.1%	48.8%	2.1%
	70%	Ours	QAT	69.1%	49.3%	1.6%
YOLOv9	100%	-	-	73.4%	51.0%	-
	80%	Ours	-	72.8%	50.3%	-
	100%	-	QAT	73.2%	50.7%	**0.7%**
	80%	Ours	PTQ	71.2%	48.9%	2.1%
	80%	Ours	QAT	72.6%	50.1%	0.9%

results show that DepGraph's accuracy is marginally higher than the AutoOTO's. However, DepGraph owns a significantly higher number of parameters and FLOPs than AutoOTO. Furthermore, AutoOTO allows a fast model development with only a single training from scratch, while HFP and DepGraph require a multi-stage training pipeline.

4.3 Performance of The Proposed Framework

Table 3 and 4 show the results of image classification and object detection experimental, respectively. We observe that image classification models experience limited accuracy loss after the pruning process. However, accuracy is still satisfied around the desired value, especially for MobileOne and MobileNetV2 models. For EfficientViT, because of its complex architecture, its parameters are pruned less than MobileOne and MobileNetV2 models but it retains a similar accuracy level.

For the object detection task, after a pruning process, AutoOTO keeps 70% and 80% of the parameters required for YOLOv8 and YOLOv9, respectively. At such a level of pruning, QAT flow still guarantees acceptable accuracy, which proves the effectiveness of our proposed compression method for resource-constrained edge devices. The performance of PTQ is less than that of QAT in all

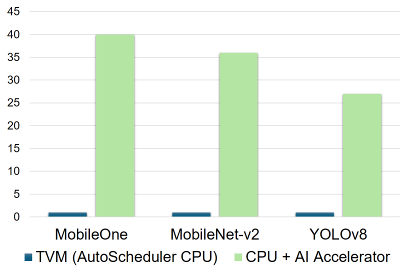

Figure 6: Performance improvements by the framework

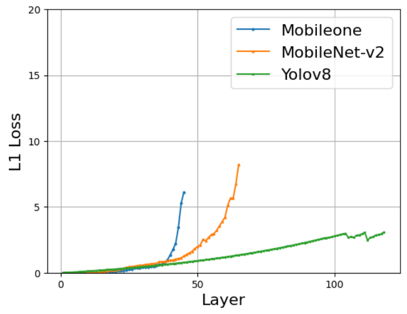

Figure 7: The accumulated error between compressed and deployed model.

experiments. For example, we can prune up to 53% number of YOLOv5's parameters with a trade-off of 0.7% mAP only. However, the model is sensitive to the quantization process, resulting in a large decrement in accuracy.

4.4 Experiments on Edge Device Deployment

We conduct experiments with MobileOne, MobileNetv2, and YOLOv8 models, and compare inference performance on both the processor and hardware accelerator versus a baseline, which is the complete processor-based inference of the full-precision DL model. For a fair comparison, we used TVM AutoScheduler to generate an optimized code of the model for the Arm processor in our development kit. As depicted in Fig.6, considered models achieve faster inference speed than the baseline, especially, MobileOne can perform inference up to 40 times faster than the baseline. Table 5 shows that MobileOne, MobileNetv2, and YOLOv8 models induce marginal performance loss compared to the baseline. That is the highest loss is only at 3.03% belonging to MobileNetv2.

For the model compilation and inference on the edge device, we compare the accumulated error of the TFLite model and the model deployed on the hardware accelerator. The error is computed using the L1 Loss function. Fig.7 illustrates the L1 Loss per layer for three models, MobileOne, MobileNetV2, and YOLOv8. MobileOne consists of only a series of fused convolutional layers connected in series, while MobileNetV2 includes residual connections, and YOLOv8 is a highly branched model. The errors are measured with the *uint8* scale, so the possible range of L1 Loss can be in a range of $[0, 255]$, however, the largest error is only 8. It can be observed that, when the number of layers rises, the L1 loss of the deployed models

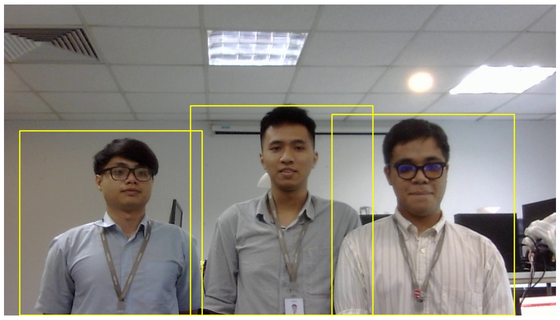

Figure 8: A demonstration of human detection using the Yolov8 model running on an edge device with a hardware accelerator.

deviates from the TFLite, and the accumulated error increases. It is because, during the inference, optimized models round values based on 32-bit operations, whereas TFLite rounds values using 64-bit operations of the processor. We also observe that models with multiple branches (or skip connections), such as MobileNetV2 or YOLOv8, obtain better rounding errors. The reason is that branches do not use multiplication operators, which can induce more rounding errors than other operators.

Table 5: Performance loss between TFLite and deployed model.

Model	Metric	Performance Loss on Edge	Total Performance Loss
MobileOne	Acc.	0.40%	1.54%
MobileNetv2	Acc.	1.03%	3.03%
YOLOv8	mAP	0.21%	1.81%

5 Conclusion

In this paper, we propose a self-contained automated framework for deploying DL models on edge devices having a hardware accelerator. The proposed framework includes model compression and model compilation pipeline to optimize the model for resource-constrained edge devices. The framework offers a seamless method for structural pruning and quantizing DL models into integer-only formats to facilitate the usage of hardware accelerators. We show that the models are successfully compiled into the firmware for hardware accelerators using our customized SDK. We prove the effectiveness of our approach based on different evaluations. The results show that the framework can accelerate hardware inference performance by up to 40 times compared to a complete processor-based inference at an acceptable accuracy loss.

References

[1] Saba Akram and Quarrat Ul Ann. 2015. Newton raphson method. *International Journal of Scientific & Engineering Research* 6, 7 (2015), 1748–1752.

[2] Han Cai, Junyan Li, Muyan Hu, Chuang Gan, and Song Han. 2023. EfficientViT: Lightweight multi-scale attention for on-device semantic segmentation. *arXiv preprint arXiv:2205.14756* 2 (2023).

[3] Tianyi Chen, Luming Liang, DING Tianyu, Zhihui Zhu, and Ilya Zharkov. 2023. OTOv2: Automatic, Generic, User-Friendly. In *International Conference on Learning Representations*.

[4] Tianqi Chen, Thierry Moreau, Ziheng Jiang, Lianmin Zheng, Eddie Yan, Haichen Shen, Meghan Cowan, Leyuan Wang, Yuwei Hu, Luis Ceze, et al. 2018. {TVM}: An automated {End-to-End} optimizing compiler for deep learning. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*. 578–594.

[5] Zhi Chen, Cody Hao Yu, Trevor Morris, Jorn Tuyls, Yi-Hsiang Lai, Jared Roesch, Elliott Delaye, Vin Sharma, and Yida Wang. 2021. Bring your own codegen to deep learning compiler. *arXiv preprint arXiv:2105.03215* (2021).

[6] Hongrong Cheng, Miao Zhang, and Javen Qinfeng Shi. 2024. A survey on deep neural network pruning: Taxonomy, comparison, analysis, and recommendations. *IEEE Transactions on Pattern Analysis and Machine Intelligence* (2024).

[7] Lukas Enderich, Fabian Timm, and Wolfram Burgard. 2021. Holistic filter pruning for efficient deep neural networks. In *Proceedings of the IEEE/CVF winter conference on applications of computer vision*. 2596–2605.

[8] Gongfan Fang, Xinyin Ma, Mingli Song, Michael Bi Mi, and Xinchao Wang. 2023. Depgraph: Towards any structural pruning. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*. 16091–16101.

[9] Benoit Jacob, Skirmantas Kligys, Bo Chen, Menglong Zhu, Matthew Tang, Andrew Howard, Hartwig Adam, and Dmitry Kalenichenko. 2018. Quantization and training of neural networks for efficient integer-arithmetic-only inference. In *Proceedings of the IEEE conference on computer vision and pattern recognition*. 2704–2713.

[10] Glenn Jocher, Jing Qiu, and Ayush Chaurasia. 2023. *Ultralytics YOLO*. https://github.com/ultralytics/ultralytics

[11] Chao-Lin Lee, Chun-Ping Chung, Sheng-Yuan Cheng, Jenq-Kuen Lee, and Robert Lai. 2023. Accelerating AI performance with the incorporation of TVM and MediaTek NeuroPilot. *Connection Science* 35 (10 2023). https://doi.org/10.1080/09540091.2023.2272586

[12] Hui-Hsin Liao, Chao-Lin Lee, Jenq-Kuen Lee, Wei-Chih Lai, Ming-Yu Hung, and Chung-Wen Huang. 2021. Support Convolution of CNN with Compression Sparse Matrix Multiplication Flow in TVM. In *50th International Conference on Parallel Processing Workshop* (Lemont, IL, USA) *(ICPP Workshops '21)*. Association for Computing Machinery, New York, NY, USA, Article 17, 7 pages. https://doi.org/10.1145/3458744.3473352

[13] Tsung-Yi Lin, Michael Maire, Serge Belongie, James Hays, Pietro Perona, Deva Ramanan, Piotr Dollár, and C Lawrence Zitnick. 2014. Microsoft coco: Common objects in context. In *Computer Vision–ECCV 2014: 13th European Conference, Zurich, Switzerland, September 6-12, 2014, Proceedings, Part V 13*. Springer, 740–755.

[14] Markus Nagel, Rana Ali Amjad, Mart Van Baalen, Christos Louizos, and Tijmen Blankevoort. 2020. Up or down? adaptive rounding for post-training quantization. In *International Conference on Machine Learning*. PMLR, 7197–7206.

[15] Markus Nagel, Mart van Baalen, Tijmen Blankevoort, and Max Welling. 2019. Data-free quantization through weight equalization and bias correction. In *Proceedings of the IEEE/CVF International Conference on Computer Vision*. 1325–1334.

[16] Markus Nagel, Marios Fournarakis, Rana Ali Amjad, Yelysei Bondarenko, Mart Van Baalen, and Tijmen Blankevoort. 2021. A white paper on neural network quantization. *arXiv preprint arXiv:2106.08295* (2021).

[17] Jared Roesch, Steven Lyubomirsky, Logan Weber, Josh Pollock, Marisa Kirisame, Tianqi Chen, and Zachary Tatlock. 2018. Relay: A new ir for machine learning frameworks. In *Proceedings of the 2nd ACM SIGPLAN international workshop on machine learning and programming languages*. 58–68.

[18] Olga Russakovsky, Jia Deng, Hao Su, Jonathan Krause, Sanjeev Satheesh, Sean Ma, Zhiheng Huang, Andrej Karpathy, Aditya Khosla, Michael Bernstein, Alexander C. Berg, and Li Fei-Fei. 2015. ImageNet Large Scale Visual Recognition Challenge. *International Journal of Computer Vision (IJCV)* 115, 3 (2015), 211–252. https://doi.org/10.1007/s11263-015-0816-y

[19] Mark Sandler, Andrew Howard, Menglong Zhu, Andrey Zhmoginov, and Liang-Chieh Chen. 2018. Mobilenetv2: Inverted residuals and linear bottlenecks. In *Proceedings of the IEEE conference on computer vision and pattern recognition*. 4510–4520.

[20] Pavan Kumar Anasosalu Vasu, James Gabriel, Jeff Zhu, Oncel Tuzel, and Anurag Ranjan. 2023. Mobileone: An improved one millisecond mobile backbone. In *Proceedings of the IEEE/CVF conference on computer vision and pattern recognition*. 7907–7917.

[21] Fangxin Wang, Miao Zhang, Xiangxiang Wang, Xiaoqiang Ma, and Jiangchuan Liu. 2020. Deep learning for edge computing applications: A state-of-the-art survey. *IEEE Access* 8 (2020), 58322–58336.

[22] Xiaofei Wang, Yiwen Han, Victor CM Leung, Dusit Niyato, Xueqiang Yan, and Xu Chen. 2020. Convergence of edge computing and deep learning: A comprehensive survey. *IEEE Communications Surveys & Tutorials* 22, 2 (2020), 869–904.

[23] Chun-Chieh Yang, Yi-Ru Chen, Hui-Hsin Liao, Yuan-Ming Chang, and Jenq-Kuen Lee. 2023. Auto-tuning Fixed-point Precision with TVM on RISC-V Packed SIMD Extension. *ACM Trans. Des. Autom. Electron. Syst.* 28, 3, Article 33 (March 2023), 21 pages. https://doi.org/10.1145/3569939

[24] Neta Zmora, Guy Jacob, Lev Zlotnik, Bar Elharar, and Gal Novik. 2019. Neural network distiller: A python package for dnn compression research. *arXiv preprint arXiv:1910.12232* (2019).