Towards Rapid Exploration of Heterogeneous TinyML Systems using Virtual Platforms and TVM's UMA

Samira Ahmadifarsani
Rafael Stahl
Philipp van Kempen
samira.ahmadifarsani@tum.de
Chair of Electronic
Design Automation, Technical University of Munich
Munich, Germany

Daniel Mueller-Gritschneder
Ulf Schlichtman
daniel.mueller@tum.de
Chair of Electronic
Design Automation, Technical University of Munich
Munich, Germany

Abstract

The rapid setup of deep learning compilation toolchains for heterogeneous TinyML systems with a processor and dedicated ML accelerator is still at an early stage. Here, achieving the most optimal combination of targets for a TinyML application on ultra-low-power edge devices demands additional benchmarking solutions to estimate the final performance.

Apache TVM's Universal Modular Accelerator (UMA) interface as an easy-to-use API is a promising speed-up approach to this scope. In this paper, we integrate a simple custom dedicated accelerator into TVM using UMA to offload the quantized convolution operators in order to demonstrate such an approach. Furthermore, we leverage MLonMCU tool and its capability of virtual prototyping to estimate and explore the performance improvement achieved by the accelerator.

CCS Concepts: • **Computer systems organization** → **Embedded systems**; • **Computing methodologies** → **Machine learning**.

Keywords: TVM, UMA, TinyML, Virtual prototyping

ACM Reference Format:
Samira Ahmadifarsani, Rafael Stahl, Philipp van Kempen, Daniel Mueller-Gritschneder, and Ulf Schlichtman. 2023. Towards Rapid Exploration of Heterogeneous TinyML Systems using Virtual Platforms and TVM's UMA. In *Workshop on Compilers, Deployment, and Tooling for Edge AI (CODAI '23), September 21, 2023, Hamburg, Germany*. ACM, New York, NY, USA, 5 pages. https://doi.org/10.1145/3615338.3618121

1 Introduction

Tiny machine learning (TinyML) is an emerging field at the intersection of embedded systems and machine learning. TinyML aims to bring machine learning inference to ultra-low-power edge devices, typically under a milliwatt. For running the inference of a neural network on such devices, energy efficiency is a critical factor that has to be considered. However, the growing size of networks poses challenges in terms of computational effort and memory requirements. To deploy the TinyML models effectively, optimization and quantization techniques are necessary due to the limited computational and memory resources available on edge devices.

One of the main techniques for improving the power and performance efficiency of TinyML systems is hardware acceleration. In recent years various deep learning accelerators emerged, including Google's EdgeTPU, ARM's Ethos-U NPUs as well as VTA [8]. Some of these accelerators employ custom compilation toolchains that are limited to support only one deep learning framework, and others do not focus on the software toolchain [10]. Thus, involving dedicated accelerators to offload specific operations makes model deployment more complicated.

In [10], the state-of-art deep learning (DL) compilers such as TensorFlow Lite (TFLite), TensorFlow XLA, Glow and TVM were evaluated in terms of support for heterogeneous platforms. TVM stood out as the most comprehensive solution with its Bring-Your-Own-Codegen (BYOC) flow [2]. TVM compiler has gained significant traction as an efficient compiler framework for the deployment of deep learning models. It allows the optimization of neural networks across various hardware targets, ranging from high-performance GPUs to resource-constrained microcontrollers. Concerning hardware accelerators, it provides us with BYOC flow, which allows developers to target new libraries and accelerators from TVM's high-level IR.

VTA and ARM Ethos-U are the important accelerators that have been integrated into TVM. However, the limited number of advanced accelerators being integrated may be due to the complexities involved in the TVM's integration process [7]. The Universal Modular Accelerator (UMA) [6] serves as a unified infrastructure to address this challenge by simplifying the integration of accelerators into TVM using Python APIs. The UMA group introduced a mock-accelerator named Vanilla used as a proof-of-concept example, demonstrating how to integrate a new accelerator into TVM using UMA APIs. The

CODAI '23 , September 21, 2023, Hamburg, Germany

ACM ISBN 979-8-4007-0337-9/23/09.
https://doi.org/10.1145/3615338.3618121

example showcases the required patterns and transformations to offload the single-precision floating-point (FP32) convolution layers. However, the introduction of more complex examples, especially for the execution of quantized models, is still under development.

In this paper, we investigate TVM's UMA interface and its capabilities to support heterogeneous TinyML platforms and quantized neural networks (QNNs). We start from the Vanilla floating point example and expand it for offloading QNN convolution layers. Secondly, we use MLonMCU [11], an end-to-end benchmarking tool for virtual prototyping and estimate performance before the availability of actual hardware accelerators. For this we set up a virtual platform using the plugin mechanism of the ETISS instruction set simulator and profile how many instructions can be offloaded from the processor by rapidly prototyping a quantized convolution accelerator.

2 Background

2.1 TVM Compiler

TVM is a compiler stack for deep learning that is open-source in nature. Its primary purpose is to take model definitions provided by DL frameworks as input and generate efficient code implementations for a wide range of DL hardware as output. To achieve this, TVM follows a two-step process. Firstly, it parses the model graph into the high-level intermediate representations[1] (IR), and subsequently converts them into low-level IRs, which can be targeted at diverse backends such as CPUs, GPUs, FPGAs, and machine learning ASICs. Through this approach, TVM offers graph-level and operator-level optimizations, ensuring performance portability for deep learning workloads across different hardware backends.

To support dedicated accelerators, TVM incorporates the BYOC mechanism. This mechanism enables hardware backend providers to implement their own code generators and register them as Relay backend compilers. By doing so, they can leverage a framework that integrates their codegen tools in a plug-and-play manner [2]. BYOC allows to benefit from hardware-agnostic optimizations. Additionally, developers can take advantage of flexible interfaces to annotate and partition computational graphs using different strategies, as well as apply hardware-specific optimizations to the partitioned graphs, thereby further enhancing performance.

Introducing the UMA interface aims to simplify the integration of external accelerators into TVM by utilizing the BYOC infrastructure. In subsequent sections, we will provide a detailed explanation of UMA's functionality and its usage.

2.2 UMA Interface

UMA aims to establish a unified infrastructure for seamlessly integrating external accelerators into TVM. By leveraging UMA, developers gain access to a set of predefined file structures, Python interface classes, and an API that facilitates

[1] called Relay

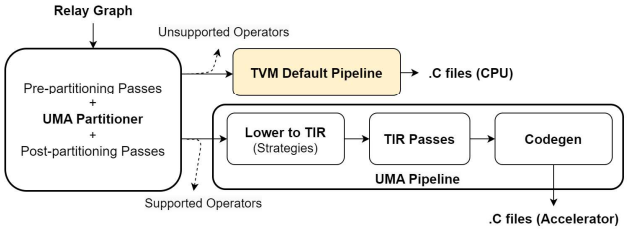

Figure 1. UMA interface.

the integration process of accelerators. This unified infrastructure enables the straightforward offloading of specific operator patterns to on-chip accelerators while the remaining operators in the model are executed on the CPU.

UMA command line interface generates several Python file structures for integrating a new accelerator into TVM. One of the key components is the backend class, which serves as the main file structure for the accelerator. This backend class allows users to add necessary patterns, Relay passes, Tensor intermediate representation (TIR) passes, and strategies for operators that can be offloaded to the accelerator.

The patterns and passes enabled within the backend class are utilized by two UMA components, the UMA partitioner and the UMA pipeline. These components are embedded into TVM's graph partitioning and pipeline. As depicted in Fig. 1, the UMA partitioner takes the Relay graph and identifies supported and unsupported operators based on the specified patterns. Unsupported operators follow the default TVM flow, while the UMA pipeline processes supported operators.

Within the UMA pipeline, the partitioned Relay functions are lowered to TIR using the Tensor Operator Inventory (TOPI) or custom schedules. Subsequently, a user-defined TIR pass is applied to the TIR function to construct the external call function. Finally, TVM's code generator generates inference C code from the TIR, encompassing both the functions invoked by the accelerator and the processor.

Currently, the only accelerator integrated into TVM using UMA is the Vanilla accelerator [5], a C-coded implementation of 2D convolution. UMA's Vanilla example serves as a reference template for the backend structure, allowing developers to offload the convolution layer of a model with FP32 data types and same-padding.

However, many common accelerators such as NPUs, typically process 8-bit integer (INT8) data due to its performance and memory efficiency benefits. For this reason, we leverage and assess UMA with a quantized Deep Neural Network (DNN) model targeted at an INT8 virtual custom accelerator.

2.3 Quantized Deep Learning Models in TVM

Initially, TVM was primarily designed to optimize FP32 computations and did not have built-in support for optimizing pre-quantized INT8 models. To address this limitation, the QNN dialect was introduced [4], extending TVM's internal representation with a quantization context. The QNN dialect acts as a higher-level IR layered on top of the graph-level IR.

In the QNN dialect, new operators specific to quantized operations are added without any graph- or tensor-level optimizations. Indeed, these QNN operators are lowered to a sequence of existing operators within the TVM compiler, which already have graph- and tensor-level optimizations [4]. Fig. 2 illustrates how the QNN dialect fits into the TVM framework.

When a developer adds a new QNN operator, they also provide a description of how this operator can be lowered to a sequence of existing Relay operators. These QNN operators typically correspond to the quantized operators defined in the deep learning framework being used. A framework parser is responsible for parsing the framework model and producing a framework-agnostic graph, which consists of a mixture of QNN and Relay operators. Subsequently, the QNN Canonicalize pass converts the QNN operators into a sequence of Relay-only operators using the sequence for the conversion. From this point onward, the existing TVM infrastructure can be reused for further optimizations and code generation.

Fig. 3 illustrates the QNN flow for a specific example of a quantized TFLite Conv2D within TVM. The TFLite Conv2D operation is first parsed and represented as a sequence of QNN and Relay operators, and then the QNN operators are broken down into several Relay operators after the Canonicalization pass [4]. The result of parsing the TFLite quantized conv2d operator is a sequence of QNN Conv2D, Relay bias addition, Relay clipping, and QNN requantize operator. QNN conv2d computation only handles the quantized tensors and adjustments for zero points. The QNN requantize operator applies the quantization scale parameters and converts the tensor values back to the INT8 datatype, ensuring the values are properly quantized to the desired datatype for further processing or inference.

3 Approach

In this section, we utilize UMA to offload a portion of a TFLite model onto a rapidly prototyped accelerator while the remaining parts are processed on a RISC-V processor core. We initially evaluate UMA with an FP32 model using the Vanilla example. Then, we outline the necessary modifications to enable the execution of a quantized model. The UMA backend for the quantized version of Vanilla is available on GitHub[2].

[2]https://github.com/tum-ei-eda/TVM-UMA/tree/quantized_vanilla_example

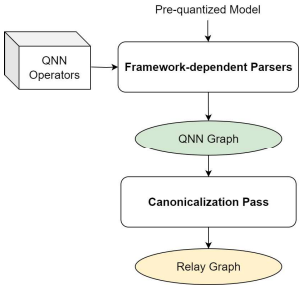

Figure 2. QNN Dialect in TVM.

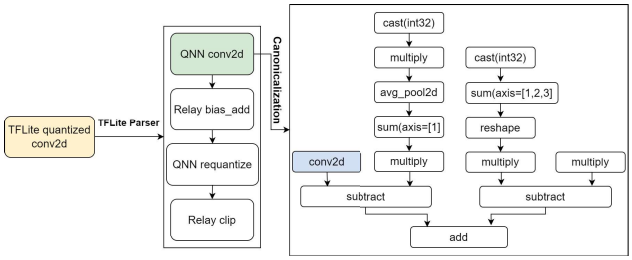

Figure 3. QNN Conv2D compilation.

3.1 Integrating New Accelerators

UMA offers an easy-to-use API for integrating new hardware accelerators into TVM. It provides a Python interface that generates the initial code structure for an accelerator, including the backend class.

Within the backend class, developers can utilize various functions to register patterns, strategies, Relay passes, TIR passes, and the codegen. Additionally, the codegen component integrates the driver or interfaces specific to the hardware accelerator directly into the generated inference code. Table I outlines the deployment flow and specifies the stages where each registered function is applied.

Vanilla is a mock-accelerator, which the UMA group introduced and used as a proof-of-concept example to show how to integrate a new accelerator into TVM using UMA's API. Vanilla is designed with a MAC (Multiply-Accumulate) array but does not have any internal memory. It offers a C interface that enables the execution of FP32 Conv2D operations, including same-padding. As a result, Vanilla can only process Conv2D layers, while the CPU handles all other layers.

To begin the integration process, we generate the code structure for the Vanilla backend using the UMA command-line interface [5]. This step generates the necessary code skeletons for the Vanilla backend, which includes the registration of a pattern, a TIR pass, and the UMA codegen. The corresponding Relay operator for FP32 Conv2D layers is nn.conv2d, which the pattern specifies for the partitioner.

The input DNN model used in the paper is a TFLite model, which by default follows the NHWC (channels-last) memory layout. However, the Vanilla accelerator processes data in the NCHW (channels-first) layout. Consequently, a TIR pass is created specifically to search for NCHW blocks within the TIR function and replace them with the corresponding external function. Therefore, we register a layout conversion pass as a relay pass to be applied on Conv2D layers during the pre-partitioning phase. The convert layout pass sets up the infrastructure to change the data layout of the graph with a minimal number of data layout transforms [3]. In the case of this example, the pass inserts three layout transforms to convert the input feature, kernel, and bias data to the channels-first layout. Similarly, another layout transform is inserted to convert the output data back to the channels-last layout for the remaining parts of the model.

Table 1. The Registered Functions of a Backend Class

	Pre-Partitioning	Partitioning	Post-Partitioning	Relay-to-TIR Lowering	TIR-to-Runtime Lowering
_register_relay_pass	✓		✓		
_register_pattern		✓			
_register_operator_strategy				✓	
_register_tir_pass				✓	
_register_codegen					✓

After adding the convert layout pass to the Vanilla backend, it can be registered as a target to offload Conv2D layers with stride one and same-padding features within the TFLite models.

3.2 Offloading QNN Operators

This section describes how to use the UMA interface to run quantized models on a custom accelerator. Our custom accelerator is a modified version of Vanilla, which can process QNN Conv2D followed by a bias addition operator.

To construct the backend for this custom accelerator, the initial step is defining the necessary pattern to annotate and partition the supported operators. Fig. 4 illustrates the Conv2D layer of a quantized TFLite model after the convert layout pass. Consequently, the pattern specifies a composite function consisting of the qnn.conv2d operator followed by a bias addition operator.

After the partitioning process, any unsupported Relay and QNN operators are handled using the default TVM flow, which includes the canonicalization pass to lower QNN operators to existing Relay operators. While it is possible to activate the canonicalization pass in the UMA pipeline, the Conv2D relay operator within the qnn.conv2d definition after canonicalization is only a component of the overall result calculation and is not fitted on the custom accelerator, as depicted in Fig. 3.

Without canonicalizing, UMA lower cannot lower down qnn.conv2d to TIR because it has no specified schedules and implementations in TOPI. In other words, TVM requires the user to provide a schedule, which is a description of how the computation should be performed. To address this, we add a strategy that specifies the computation and the schedule to be used. In this case, the strategy is the TOPI implementation for the qnn.conv2d operator, explicitly targeting the Hexagon processor.

Once the lowering process to TIR is completed, the TIR pass is responsible for extracting essential data and parameters from the generated TIR representation to interface the generated code and the custom accelerator through the registered C/C++ file in the codegen. This includes information such as input feature, kernel, and bias data pointers, input and kernel dimensions, as well as quantization zero points. In the Vanilla example, the TIR pass is designed to handle the computation block of the nn.conv2d operator. However, for the QNN Conv2D operator, the TIR function depends on the specified strategy, requiring adjustments to the pass. Specifically, modifications are made to the pass to extract data from the computation blocks based on the Hexagon strategy, and provide the zero-point values for the quantized computation.

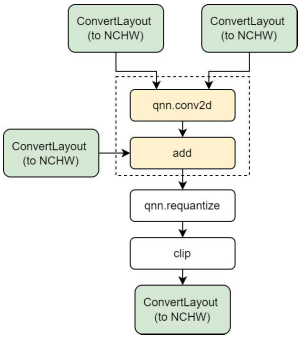

Figure 4. ConvertLayout transforms.

3.3 Virtual Prototyping

MLonMCU is a comprehensive benchmarking tool that facilitates the deployment of machine learning models on various targets. In this paper, we utilize MLonMCU to evaluate the deployment flow using the TVM and UMA interface.

MLonMCU supports the ETISS target, which is utilized to simulate a 32-bit RISC-V microcontroller [9]. ETISS features a plugin mechanism, allowing for the inclusion of new functionality into the simulation loop. Leveraging this capability, we integrate the custom accelerator as a memory-mapped peripheral within ETISS. This setup enables us to simulate the target architecture, comprising a CPU and the custom accelerator, using a RISC-V ISS, a closely-coupled peripheral, and a shared memory.

Fig. 5 depicts the virtual prototype of the target architecture, explicitly showcasing the components of the virtual accelerator, as an ETISS plugin. The accelerator comprises a controller, computation module, register interface (RegIF), and load/store unit (LSU). The accelerator controller handles the writing and reading callbacks to and from the register interface, which stores data, kernel, bias and result addresses, dimensions, quantization zero points, and a control bit. Moreover, it manages the execution of a QNN Conv2D and bias addition. The LSU is responsible for loading input feature, kernel, and bias data from memory and storing the resulting data into accelerator buffers.

4 Evaluation

In the section, we evaluate the deployment of the quantized TFLite models on the virtual platform, including our custom accelerator dedicated to a QNN Conv2D and a bias addition. As described in the previous section, we employ the MLonMCU tool to evaluate the virtual prototype. MLonMCU uses TVM to optimize, compile and generate C codes of inference functions. Additionally, the UMA API is integrated into the MLonMCU package to enable the utilization of the new accelerator integrated into TVM.

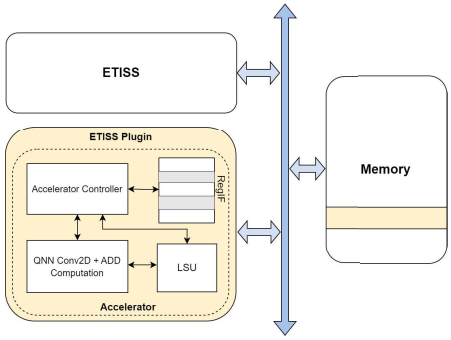

Figure 5. The model of target architecture.

In our study, we focused on three quantized TFLite models from the MLPerf Tiny benchmark [1]. These models heavily rely on convolutional layers and represent use cases for keyword spotting (aww), visual wake words (vww), and image classification (resnet). Notably, the MAC ratio of supported convolutional layers by the accelerator is 79%, 83%, and 77% for resnet, vww, and aww, respectively. Moreover, most of the convolutional layers in the aww and vww models are depth-wise separable convolutions, allowing their pointwise convolution to be efficiently offloaded to the accelerator.

The models are subjected to two different experiments. In the first case, MLonMCU employs TVM's default compilation flow to optimize and compile the model and execute the generated inference only on the CPU. For the second case, the accelerator is integrated using UMA to offload the composite function of qnn.conv2d and bias addition. In this configuration, the accelerator acts as a memory-mapped peripheral for the CPU, and the call function in the inference function just handles the interaction with the register interface of the accelerator to offload the composite functions. Table II demonstrates the resulting CPU instruction count of the simulated RISCV target for each case obtained from the MLonMCU report. The evaluation clearly shows that offloading the supported operators to the accelerator led to a significant reduction in the number of instructions, with values of 75%, 65%, and 63% for resnet, vww, and aww models, respectively. Additionally, the simulation time for the execution of the inference function on ETISS, while the convolution layers are offloaded to the accelerator is shown in Table II.

5 Conclusion

This paper presented a combination of virtual prototyping and TVM's UMA to explore the heterogenous TinyML systems. Although the UMA interface as an easy-to-use API is at its early stages, it provides interfaces to facilitate the process of integrating the kernel libraries and frameworks of the hardware accelerators. We investigated its Vanilla example and expanded its functionality to enable the execution of quantized models on a virtual accelerator. Moreover, we utilized the MLonMCU tool to estimate the performance improvement of inference achieved by offloading the models with varying computational complexity to the integrated accelerator. Overall, this combination of TVM's UMA and rapid prototyping using MLonMCU looks promising for enabling the exploration and simulation of new hardware architectures, as well as the profiling and debugging of full software stacks on simulated heterogeneous platforms tailored for TinyML applications.

Table 2. Evaluation Result

Model	CPU Instruction Count			Simulation Time
	CPU	*CPU + Acc.*	*Saved*	
resnet	5.73e07	1.44e07	75%	0.68s
vww	4.12e07	1.46e07	65%	1.33s
aww	1.43e07	5.34e06	63%	0.54s

Acknowledgments

This work has been developed as part of the Eureka PENTA project 2021028 ECOMAI funded by the German Ministry of Education and Research (BMBF) (ref no. 16ME0571).

References

[1] Colby Banbury, Vijay Janapa Reddi, Peter Torelli, Jeremy Holleman, Nat Jeffries, Csaba Kiraly, Pietro Montino, David Kanter, Sebastian Ahmed, Danilo Pau, et al. 2021. Mlperf tiny benchmark. *arXiv preprint arXiv:2106.07597* (2021).

[2] 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).

[3] Animesh Jain. [n. d.]. *Convert Layout Pass.* https://tvm.apache.org/docs/arch/convert_layout.html

[4] Animesh Jain, Shoubhik Bhattacharya, Masahiro Masuda, Vin Sharma, and Yida Wang. 2020. Efficient execution of quantized deep learning models: A compiler approach. *arXiv preprint arXiv:2006.10226* (2020).

[5] M. J. Klaiber, P. P. Bernardo, and C. Gerum. 2022. *Making your Hardware Accelerator TVM-ready with UMA.* https://tvm.apache.org/docs/tutorial/uma.html

[6] M. J. Klaiber, P. P. Bernardo, and C. Gerum. 2022. *UMA: Universal Modular Accelerator Interface.* https://github.com/apache/tvm-rfcs/blob/main/rfcs/0060_UMA_Unified_Modular_Accelerator_Interface.md

[7] Chen Liu, Matthias Jobst, Liyuan Guo, Xinyue Shi, Johannes Partzsch, and Christian Mayr. 2023. Deploying Machine Learning Models to Ahead-of-Time Runtime on Edge Using MicroTVM. *arXiv preprint arXiv:2304.04842* (2023).

[8] Thierry Moreau, Tianqi Chen, Luis Vega, Jared Roesch, Eddie Yan, Lianmin Zheng, Josh Fromm, Ziheng Jiang, Luis Ceze, Carlos Guestrin, et al. 2019. A hardware–software blueprint for flexible deep learning specialization. *IEEE Micro* 39, 5 (2019), 8–16.

[9] Daniel Mueller-Gritschneder, Keerthikumara Devarajegowda, Martin Dittrich, Wolfgang Ecker, Marc Greim, and Ulf Schlichtmann. 2017. The extendable translating instruction set simulator (ETISS) interlinked with an MDA framework for fast RISC prototyping. In *Proceedings of the 28th International Symposium on Rapid System Prototyping: Shortening the Path from Specification to Prototype.* 79–84.

[10] Max Sponner, Bernd Waschneck, and Akash Kumar. 2021. Compiler toolchains for deep learning workloads on embedded platforms. *arXiv preprint arXiv:2104.04576* (2021).

[11] Philipp van Kempen, Rafael Stahl, Daniel Mueller-Gritschneder, and Ulf Schlichtmann. 2023. MLonMCU: TinyML Benchmarking with Fast Retargeting. *arXiv preprint arXiv:2306.08951* (2023).