Integrating TVM and NVDLA

Dan Wu (A0219183A), Shivam Aggarwal (A0219046H)

December 10, 2020

1 Introduction

As the number of DNN models and hardwares keeps increasing, the deployment and optimization of new DNN models on a specific hardware is not an easy thing. To solve this problem, Chen et al. proposed TVM [1], an open deep learning compiler stack aiming to bridge the gap between productivity-oriented deep learning frameworks and performance or efficiency-oriented hardware backends. It can provide performance improvement and portability to deep learning workloads across diverse back-ends. Currently, TVM supports back-ends like CPU, GPU, FPGA, and a handful of accelerators. TVM also has an extension called Versatile Tensor Accelerator (VTA)[2], a programmable accelerator that exposes a RISC-like programming abstraction to describe compute and memory operations at the tensor level. NVDLA is an open-source DNN accelerator and can achieve industry-level DNN inference throughput. It is configurable at the build time to meet different performance, power, and area trade-offs, targeting embedded systems and IoT devices with limited power budgets. At present, TVM does not support NVDLA as backends. We prove there is a need to integreate TVM and NVDLA, then propose a feasible workflow to achieve this goal.

2 Motivation

At present, although TVM stack supports diverse hardwares, NVDLA is not one of them. There is a need to integrate the two mainstream stacks due to the following three reasons. First, NVDLA only supports Caffe as the frontend support, but TVM supports more mainstream frontends like Tensorflow, Pytorch, etc. Second, NVDLA is a mature hardware and has been used in industry. It is far more complicated than VTA. Third, NVDLA's compiler toolchain consists of only 13 different optimization methods and almost half of them are not implemented yet in the current release, while TVM software stack supports more optimizations in diverse levels, like high-level graph and operator optimizations and low-level loop tiling. In general, the optimization in TVM stack is much more stronger than the one in NVDLA. Therefore, we intend to integrate these two open-source projects so that characteristics of TVM, like richer optimization and wider front-end support, can be introduced to the industry-level DNN inference accelerator NVDLA.

3 Related work

Recently, there has been growing interest in bridging the gap between machine learning frameworks and underlying hardware. Improving the deployment efficiency of DL models on various hardware backends, including CPUs, FPGAs, and specialized hardware accelerators such as NVDLA is an active research theme during the last several years. ONNC (Open Neural Network Compiler) one such effort, is a compilation framework designed specifically for deep learning accelerators (DLAs) and supports the NVDLA compilation framework. It can compile a model into an executable NVDLA loadable file. ONNC is built on top of LLVM and supports ONNX (Open Neural Network Exchange) intermediate representation. ONNX is a unified open format for representing various deep neural networks and allows users to port models across diverse DL frameworks.

In contrast, our work focus on utilizing the TVM compiler stack that not only supports the ONNX representation but also offers a diverse set of optimizations, including both graph-level and operator-level. In general, TVM is a more mature compilation framework widely accepted by both academia and industry. Additionally, TVM provides a blueprint to support accelerators such as VTA (Versatile Tensor Accelerator). TVM and VTA form an end-to-end hardware-software deep learning system stack that includes hardware design, drivers, and a JIT runtime. Similarly, TVM offers the BYOC (Bring Your Own Codegen) framework to support heterogeneous hardware backends. Related works include Xilinx's Vitis AI, the development stack for hardware-accelerated AI inference on Xilinx FPGA platforms. They recently released an integrated workflow using TVM's BYOC flow. Our work intends to integrate the compilation flow of NVDLA with TVM using the BYOC framework.

4 NVDLA

In the first part, we will introduce the whole workflow of NVDLA. As the target of this project is to integrate the optimization methods and front-end support of TVM to the NVDLA software, we will mainly focus on the compiler and keep the NVDLA runtime unchanged. So we will introduce the details of the NVDLA compiler in the second part.

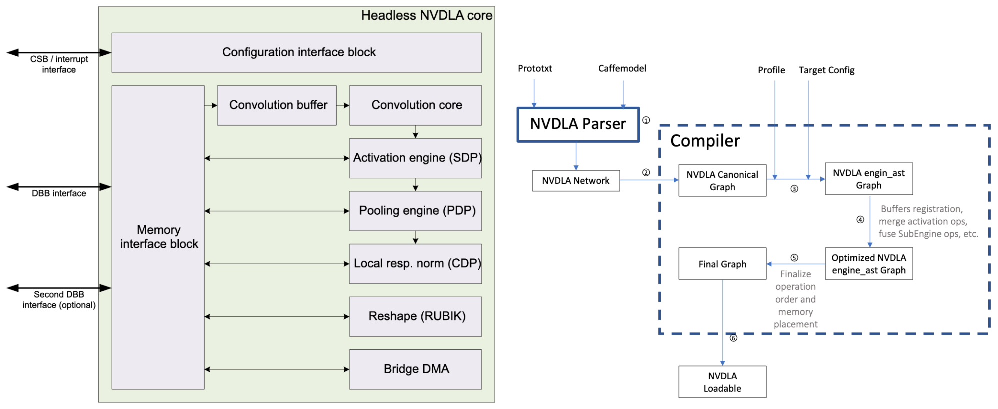

Figure 1: Left:NVDLA hardware architecture. Right: Build process of NVDLA.

4.1 Workflow

NVDLA takes a trained caffe model as input, parse and compile the model to generate a loadable file. The loadable file indicates when and where should these operations be executed on the NVDLA hardware. Then the loadable file and test images are feeded to the NVDLA runtime to get the inference result. NVDLA runtime consists of two parts - the user-mode driver and kernel-mode driver.

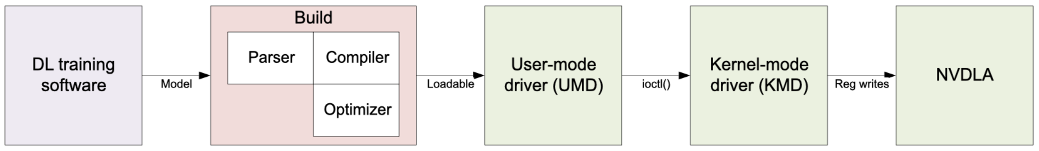

Figure 2: Workflow of NVDLA.

4.2 NVDLA Compiler

Figure 1 portraits how official NVDLA compiler generates the loadable file. First, the parser loads the prototxt file, describing the structure of a model, and a Caffemodel file, storing the weights. The parser outputs a *Network*, which is a list of neural network layers. Then, in step 2, the official NVDLA compiler transform the *Network* to a *Canonical Graph*. The nodes in *Canonical Graph* represent operations, edges indicate the data transfer. After that, in step 3, given the name of profile and configuration of targeted hardware, NVDLA compiler transforms the *Canonical Graph* to an *Engine_ast Graph*. In *Engine_ast Graph*, each node (operation) will be mapped to a specific compute module in hardware. In step 4, the compiler will do optimizations on *Engine_ast Graph* and finalize operation order and memory replacement in step 5. In the last step, the final graph will be emitted to a loadable file.

5 System Design

The proposed workflow is shown in Figure 3. First, TVM performs frontend compilation to translate frontend languages such as PyTorch, Tensorflow, and Caffe to the intermediate representation using the existing Relay compiler. Then, TVM's Bring Your Own Codegen (BYOC) framework infrastructure is used to convert the Relay IR into a json file containing the neural network information. Then the modified and rebuilt NVDLA compiler accepts the json file as input and outputs loadable file. Finally, the loadable file and test image are fed to the NVDLA runtime for model inference. This integration will enable the development on the NVDLA architecture using a diverse set of neural network frontends and equip NVDLA with high-level graph optimization. We will introduce BYOC framework and changes in NVDLA compiler in the following sections.

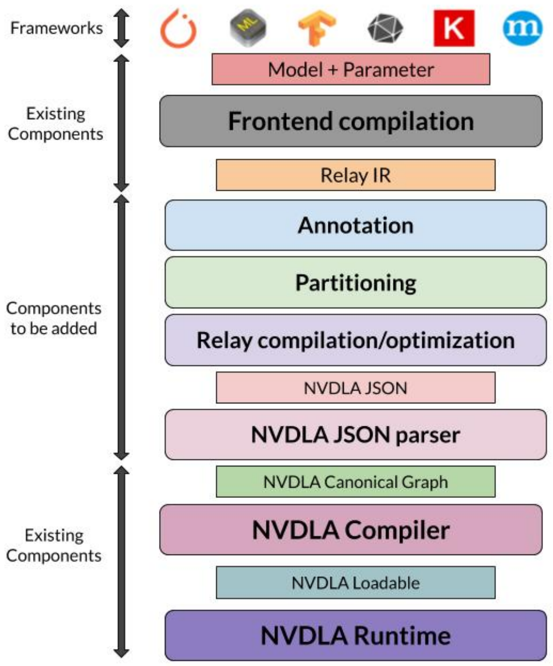

Figure 3: Proposed end-to-end system flow

5.1 Bring Your Own Codegen To TVM (BYOC)

5.1.1 Graph Annotation

As a first step, taking the Relay-generated intermediate graph, TVM annotates the nodes that can be offloaded to the NVDLA architecture. We implement a whitelist of supported operators that can be implemented on our target hardware. We can intuitively specify which Relay operators are supported by NVDLA with the BYOC API. For example, we use the following code snippet to build a rule saying that our NVDLA codegen supports Conv2D:

```
@tvm.ir.register_op_attr("nn.conv2d", "target.nvdla")
def _nvdla_conv2d_wrapper(attrs, args):
  return True
```

5.1.2 Graph Transformation

During the second step, the TVM compiler stack automatically transforms and optimizes the graph based on the annotations. We create a Relay function with an attribute *Compiler* to indicate that this Relay function should be entirely offloaded to the accelerator. We can add optimizations such as merging "regions" in the graph to reduce the data transfer, but we will skip such optimizations for now.

5.1.3 Relay compilation/optimization

This is the most important step in the whole compilation flow. In this step, TVM sequentially sends every Relay function with *Compiler=NVDLA* to codegen. The codegen compiles the Relay function to the JSON representation. Once all the functions are translated into a JSON representation, a single JSON file will be generated by the TVM export_library Python API. We can customize the serializer to generate any forms in JSON we like as long as our JSON runtime could interpret them. One such example:

```
node = {
    "op": "kernel",
    "name": op_name,
    "inputs": [[_, 0, 0] for _ in range(len(inputs))],
    "attrs": {
        "num_inputs": str(len(inputs)),
        "num_outputs": "1",
        "shape": [[list(shape)]],
        "dtype": [[dtype]],
    }
}
```

5.2 Modified NVDLA compiler

As NVDLA compiler and runtime is open-source, we modify and build our own compiler and keep the original runtime. To be specific, we changed the original input interface, enable it accept json file as input. After parsing the json file, preprocessing is also needed due to the difference in network representation between Relay and NVDLA. For example, in Relay IR, input and bias are stored in the same level of neural network layers. However, in NVDLA canonical graph, weight is just an attribute of neural network operation and inputs are represented by edges. To get the NVDLA canonical graph from the preprocessed representation, we rewrite most node generation functions and the whole graph generation function. After the correct canonical graph is generated, we reuse the rest functions in native NVDLA compiler. The total lines of code we added in compiler is ∼1k lines.

6 Experiment

Execution Setup. For compilation, we used Ubuntu 16.04 on a PC with Intel(R) Core(TM) i7-9700 CPU at 3.00GHz. For model inference, we run native NVDLA runtime on NVDLA hardware simulator on SystemC. The simulator has system Linux nvdla 4.13.3 on Arch64 at 62.5MHz.

Inputs(workload). As we haven't integrate too much optimization functions of TVM stack into the pipeline in this project, the primary indicator is correctness. To ensure the functional correctness of our pipeline, we choose a simple test case - one convolution layer on MNIST data set, with kernel shape (6, 1, 3, 3).

Evaluation Process. We integrated NVDLA into Apache TVM using a two-step process.

First, we generate an NVDLA-specific JSON representation using the TVM stack. We implement a list of supported operators that can be implemented on NVDLA. TVM automatically annotates the nodes in the graph that can be offloaded to the NVDLA architecture, partitions the graph based on the target hardware configuration, and generates the JSON representation. JSON file consists of nodes representing different network operators such as convolution, pooling, etc., and corresponding weight values if any. We evaluated our generated codegen using a variety of deep learning models with different frontend frameworks such as PyTorch, MxNet, Tensorflow, etc. Our approach can correctly translate all TVM supported frameworks and can directly support TVM's Relay intermediate representation. Second, we feed the json file into modified NVDLA compiler and generate the loadable file. Finally, run the loadable with test image to get the inference result. As NVDLA runtime will automatically round the values into integers, we compared the output of NVDLA runtime and true value after rounding. The two outputs are the same, which proves the functional correctness of our pipeline.

Index	result
NVDLA Compilation Time	0.023s
NVDLA Inference Time	4.37s
Functional Correctness	✓

Table 1: Performance of work flow. Time is shown with average of 10 test cases.

We also analyze additional programming effort required to integrate NVDLA into the TVM stack. We only require an additional 20-30 lines of code to run any new ML model using TVM-supported frontends on NVDLA. However in traditional setting, we need to spend a lot of time to rewrite a model from one frontend into another. If the model we want changes fast on diverse frontends, then it will be a big cost.

7 Challenges

- Diverse workflow design
  Ideally, we want integrate both high-level graph optimization and low level operator optimization of TVM stack into NVDLA compiler. But lower we go, more difficult the task is. To get a prototype and prove the feasibility of this integration, we choose to transform from Relay IR to NVDLA canonical graph.

- Different network operator representation
  In Relay, bias and inputs are stored as network layers, dense layer only accepts 2d tensor as input. But in NVDLA, bias is a parameter of layer, inputs are represented as edges and dense layer accepts 4d tensor. Efforts are need to find out the transformation rule.

- Strict Check in NVDLA compiler
  As official NVDLA compiler is for industrial use, it has a lot of checks to enable users run a real neural network with abundant error log support. However, this mechanism hinder us from running a single layer test for certain kind of layers.

- Non-Debuggable NVDLA runtime

Although NVDLA runtime is open-source, the hardware simulator does not support debug mode and the loadable file is binary. Even if the loadable file is successfully generated, we still need to run model inference on NVDLA simulator without error and compare the output with true value to test whether the compilation works well or not. If these two results are different, we need to go through the whole compilation process as we only know there is a bug but have no idea why and where.

8 Work distribution & Experience

Our project includes two independent system components: TVM and NVDLA. Since the two system designs were completely new to us, we spent the first half of the semester to read and understand various components of the project.

- Shivam: I looked into the TVM documentation and codebase, including Python API. Next, I looked at the VTA architecture and its integration with TVM, including JIT Runtime and build process of the compiler and additional hardware intrinsic transformation passes such as *CPUAccessRewrite*, *InjectALUIntrin* etc. Finally, I explored the BYOC framework and studied existing works such as Xilinx Vitis AI and ARM's Compute Library that are already combined with the TVM stack using BYOC infrastructure. I successfully implemented the annotation phase, graph partitioning phase and the final JSON generation in TVM using BYOC API. We are able to generate NVDLA-intrinsic JSON representation for a diverse set of neural networks such as LeNet, AlexNet, etc using different ML frameworks including PyTorch, Tensorflow, MxNet, etc.

- Dan: The official document of NVDLA does not fully match with the source code, I spent more than one week to build the simulator and correctly run model inference on it. To add and rewrite the functions in NVDLA compiler, I read a large part of compiler source code ($\sim$ 4k lines), figuring out the detailed compilation process, especially how the canonical graph is generated and how optimizations are executed. I also compare the Relay IR and Canonical Graph for 3 classical networks to figure out the representation transformation rule. Then I rewrite the first and second steps in NVDLA compiler, including new interface for NVDLA compiler to read from JSON file, new function to transform relay-like IR to canonical graph-like IR, new function to build a canonical graph from the transformed IR. The most tough thing is that loadable file is binary while the simulator does not support debug mode. When something goes wrong, I need to debug the whole compilation process rather than just keep an eye on the code I write, due to unknown memory pollution in the later process.

9 Discussion

In the previous system design, as shown in Figure 4, we focus on the VTA and TVM stack that constitutes a blueprint for an end-to-end accelerator-centric deep learning system, consisting of drivers, a JIT runtime, and an optimizing compiler stack based on TVM. We proposed to build a JIT compiler similar to the existing VTA JIT compiler. This JIT compiler translates the output of TVM tensor IR into the NVDLA loadable file. We thought that the actual implementation would be very similar to the existing NVDLA compiler, effectively mapping each network operation onto the targeted functional block of the NVDLA implementation. In our new design, we intend to re-use existing parts of the project, such as the NVDLA Runtime. Here, we build both the compiler as well as the JIT runtime module from scratch. We realized several challenges with this compilation flow:

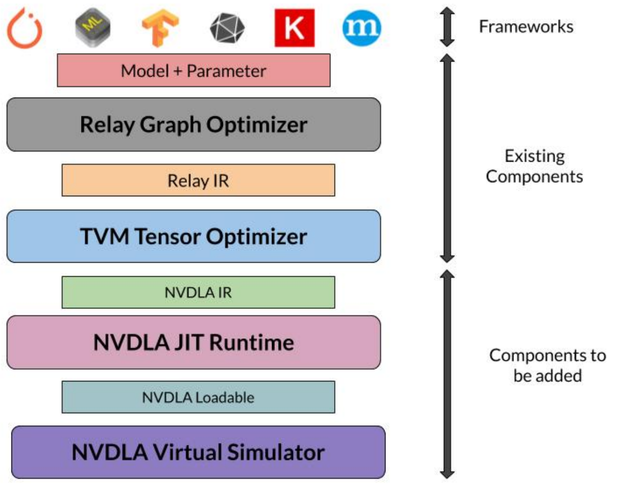

Figure 4: Earlier version of the integration of NVDLA into TVM

- Firstly, this approach is more suited for VTA-like accelerators that can be tightly coupled within the TVM stack.
- Secondly, the NVDLA architecture executes the target neural network layer-by-layer on respective execution engines such as SDP. We realize that this system design requires more time and effort since we need to re-build the whole compilation flow.
- NVDLA's compiler, runtime APIs and the loadable file have very intricate design. We need to reverse engineer the loadable file to understand the complete structure. Several existing systems such as Columbia University's ESP and ONNC's toolchain integrate NVDLA using its existing compilation toolchain.
- TVM's BYOC framework supports external codegen/runtime such as Xilinx Vitis AI and ARM's ETHOS and enables integration with hardware backend providers such as TensorRT with using the TVM stack.

10 Future Work

Firstly, our current approach focuses on a two-step process, where we first generate an intermediate JSON representation and then utilize it to emit an NVDLA loadable using NVDLA's default compilation flow. We intend to integrate the two frameworks in an end-to-end fashion to build and deploy applications on the NVDLA hardware quickly. We can follow the Xilinx Vitis AI's approach using BYOC flow to compile TVM inside a docker container with NVDLA. Additionally, we are also interested in integrating TVM with CGRA based accelerators such as HyCUBE. Coarse-Grained Reconfigurable Arrays (CGRAs) are emerging hardware accelerators that support word-level reconfigurability, and offer better energy-efficiency as compared to FPGAs. HyCUBE is one such CGRA based accelerator with reconfigurable interconnect to enable single-cycle communications between distant functional units proposed very recently by our research group. It achieves significant performance improvement with upto $1.5X$ and $3X$ performance-per-watt compared to a CGRA with standard NoC and a CGRA with neighbor-to-neighbor connectivity,respectively. We plan to utilize the NVDLA-TVM flow to integrate HyCUBE with the TVM compiler stack. This approach will improve the programmability on CGRA based platforms.

References

[1] Tianqi Chen, Thierry Moreau, Ziheng Jiang, Lianmin Zheng, Eddie Yan, Haichen Shen, Meghan Cowan, Leyuan Wang, Yuwei Hu, Luis Ceze, Carlos Guestrin, and Arvind Krishnamurthy. TVM: An automated end-to-end optimizing compiler for deep learning. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, pages 578–594, Carlsbad, CA, October 2018. USENIX Association. ISBN 978-1-939133-08-3. URL `https://www.usenix.org/conference/osdi18/presentation/chen`.

[2] T. Moreau, T. Chen, L. Vega, J. Roesch, E. Yan, L. Zheng, J. Fromm, Z. Jiang, L. Ceze, C. Guestrin, and A. Krishnamurthy. A hardware–software blueprint for flexible deep learning specialization. *IEEE Micro*, 39(5):8–16, 2019.