Exploring Machine Learning Compiler - Apache TVM

YUHUA CHENG*, Chalmers University of Technology, Sweden

Apache TVM is an open-source deep learning compiler stack that aims to optimize the deployment of machine learning models on a wide range of hardware platforms. It addresses the challenge of optimizing models for diverse hardware, from CPUs to accelerators. TVM acts as a deep learning compiler, transforming high-level model descriptions into efficient code for specific targets. Besides TVM there are many other machine learning compiler as well like PyTorch's Glow, Google's XLA etc. But TVM currently supports the most diverse hardwares and has the most users range from developers to vendors. In this report, we try to use TVM for some practical projects while in the mean time study the internal details of it. During these projects, we found that Apache TVM is a powerful tool for optimizing neural network on heterogeneous hardware systems. It can help to bridge the gap between algorithm developer and high performance engineer, free hardware designers from part of the burden of implementing efficient deep learning libraries. What's more, the ecosystem of TVM can help to democratize access to AI hardware and bring applications using AI to everyone.

CCS Concepts: • **Computer systems organization → Parallel architectures**; • **Software and its engineering → Parallel programming languages**; • **Hardware → Emerging languages and compilers**.

Additional Key Words and Phrases: High Performance Computing, Heterogeneous Systems, Machine Learning Compiler, Machine Learning

ACM Reference Format:
Yuhua Cheng. 2023. Exploring Machine Learning Compiler - Apache TVM. *ACM Trans. Arch. Code Optim.* 00, 00, Article 00 (2023), 15 pages. https://doi.org/XXXXXXX.XXXXXXX

1 INTRODUCTION

As highlighted by Lattner [14], the post Moore's law and Dennard scaling era has ushered in a daunting challenge: the development of an efficient software stack for modern hardware architectures. This challenge extends to a wide array of hardware platforms, ranging from traditional CPUs and GPUs to specialized TPUs, ASICs, and various combinations thereof. The complexity deepens as not only are hardware types proliferating, but the landscape of applications is also rapidly expanding. These applications include convolution neural networks, graph neural networks, recurrent neural networks, and transformer models, among others. Efficiently mapping these diverse applications to the multitude of hardware options available presents a formidable task, resulting in an intricate NxN combination problem. This challenge underscores the critical need for innovation in software development and optimization techniques to harness the full potential of contemporary hardware, enabling the seamless execution of AI workloads across an evolving and heterogeneous computing landscape.

It's in this landscape machine learning compilers like TVM [4] emerges as a promising solution. Characterized by it's two level IR design inspired from LLVM [15] and Halide [17], TVM is able to optimize machine learning models for various hardware targets and providing promising results. It

Author's address: Yuhua Cheng, yuhua@chalmers.se, Chalmers University of Technology, Gothenburg, Sweden, 41296.

1544-3566/2023/00-ART00 $15.00
https://doi.org/XXXXXXX.XXXXXXX

has already been supported by lots of hardware vendors such as NVIDIA, ARM, AMD etc. Thus, TVM appears to be a promising tool to investigate into.

In this report, we try to study the details of TVM in a practical way and explore the full potential of it. Our contributions are summarized as follows:

1) We used TVM for optimizing computations from kernel operator to whole neural network.
2) We demonstrated how to use TVM runtime under C++.
3) We compared the performance of TVM to TensorRT, and showed the time/memory/power efficiency of TVM.
4) We added a new codegen component into TVM library which achieved 30% speedup compared to Arm Compute Lib [2].
5) We summarized all the challenges faced when using TVM and provided insights for future research.

The report is organized as follows: section 2 provides an overview of TVM compilation flow and section 3 introduces the development environments setup for following sections. Section 4 shows how to use TVM's Auto-Scheduler [20] to generate efficient kernel for matrix multiplication. While section 4 is using TVM for a small operator, in section 5 we try to use TVM to optimize a whole CNN model and deploy it using TVM runtime. In section 6 we will explore the performance between TVM and vendor library especially TensorRT with more detail. In section 7, we showed how to support existing codegen of accelerators through TVM's BYOC [6]. In the final section, we comment the pros and cons of TVM based on our experience in this project course and provide directions for future work.

2 OVERVIEW OF TVM COMPILATION FLOW

Like many other AI compilers, TVM has multi-level IRs for abstracting optimization at different levels. Relay IR, the high level IR of TVM, is responsible for import AI models from other frameworks like PyTorch and TensorFlow Lite into Relay expression and perform graph level optimizations such as common sub-expression elimination, operator fusion, layout conversion etc. After finish graph level optimization, the Relay IR is lowered into TensorIR which is the low level IR of TVM. In TensorIR, optimizations depend on hardware like loop scheduling, multi-threading, double buffering are performed. One of the powerful tool within TVM is the Auto-Scheduler (ANSOR) [20] which can generate efficient machine learning kernels for a specific hardware automatically. While ANSOR can generate high performance computational kernels the tuning time is pretty long, Bring Your Own Codegen (BYOC) [6] is another tool which can help TVM leverage the existing optimized vendor libraries or accelerators. In the subsections, we will introduce Relay IR, ANSOR and BYOC in model detail.

2.1 RelayIR

Neural Network can be expressed as Directed Acyclic Graph (DAG) where operations are represented as nodes and data dependencies are represented as directed edges in the graph. Relay IR is the graph representation of neural network in TVM. One of it's function is for importing neural network expression from other framework like TensorFlow and PyTorch. You can use the frontend API of Relay to import models from existing supported framework, or use it to support your own deep learning framework. Example of showing how to import model into Relay IR from PyTorch can be seen from List-1. If your check the source code for importing PyTorch model [1], you will find there is a map that records how operators in PyTorch is mapped to operators defined in Relay IR. Using this way, you can implement you own mapping logic to support your own framework.

[1] https://github.com/apache/tvm/blob/main/python/tvm/relay/frontend/pytorch.py

```
# load pretrained pytorch model
input_shape = [1, 3, 227, 227]
input_tensor = torch.randn(input_shape)
shape_list = [("input", input_shape)]
model = getattr(torchvision.models, model_name)(weights=AlexNet_Weights.DEFAULT)
model.eval()
scripted_model = torch.jit.trace(model, input_tensor).eval()
# import model from pytorch into TVM Relay IR
mod, params = relay.frontend.from_pytorch(scripted_model, shape_list)
```

Listing 1. Import PyTorch model into Relay

```
# transform data layout from other formation into NHWC
mod = relay.transform.ConvertLayout({"nn.conv2d": ["NHWC", "default"]})
```

Listing 2. Layout transform in Relay

```
with tvm.transform.PassContext(opt_level=3):
    lib = relay.build(mod, target=target, params=params)
```

Listing 3. Relay transform level

After the model is imported into Relay IR, all kinds of optimizations related to graph and can reduce the overhead of it can be applied. For example, common optimizations familiar to us from programming language compiler like constant folding, common sub-expression elimination or optimizations related to neural network like layout transformation are included in Relay. Due to design reasons, different framework usually have different data layout, in image tasks for TensorFlow the data layout is NHWC, while for PyTorch it's NCHW where N is the batchsize dimension, C is the channel dimension and H, W and the two dimension of your image. In Relay, you can convert all different data layouts into the same format to guarantee uniform expression in later stage optimizations or convert between different data layout due to subgraphs are offloaded to backend with mismatching data layout. Optimizations in Relay are often called transforms. For example, layout transform optimization on convolution operator can be done like List-2. Each transform has an attribute called optimization level in it, the optimization level will decide whether this transform will be applied under a certain optimization context. For example, in List-3, the 'opt_level' is 3, which is the highest level of optimization defined in TVM, then all the transforms with optimization level less or equal than the level in current context will be applied to the model. If we use a lower optimization level in the context, for example 0, then all transforms with optimization level larger than 0 will not be applied in the context unless you apply them explicitly. You can refer to [1] for more details on the pass infrastructure of Relay. Input and output of Relay transforms are wrapped as tvm.IRModule, as can been seen from Figure-1. IRModule is the basic block for passing information between different optimization levels among TVM, it helps design optimizations across high level and low level. All optimizations can be achieved at Relay level is listed in the documentation of TVM on Relay transform[2].

Since at Relay IR level the granularity of operators is higher and it's easy to recognize operators, it's common to map operators to vendor libraries or user designed optimized library at Relay stage. For example, offloading operators like convolution to TensorRT happens at Relay level. This is used in BYOC as can be seen from Figure-2.

[2]https://tvm.apache.org/docs/reference/api/python/relay/transform.html

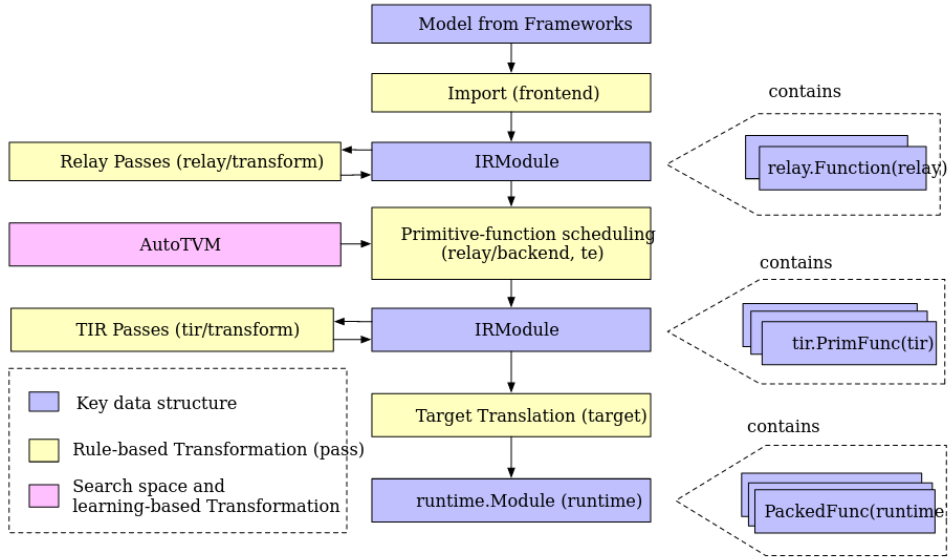

Fig. 1. TVM Design Architecture

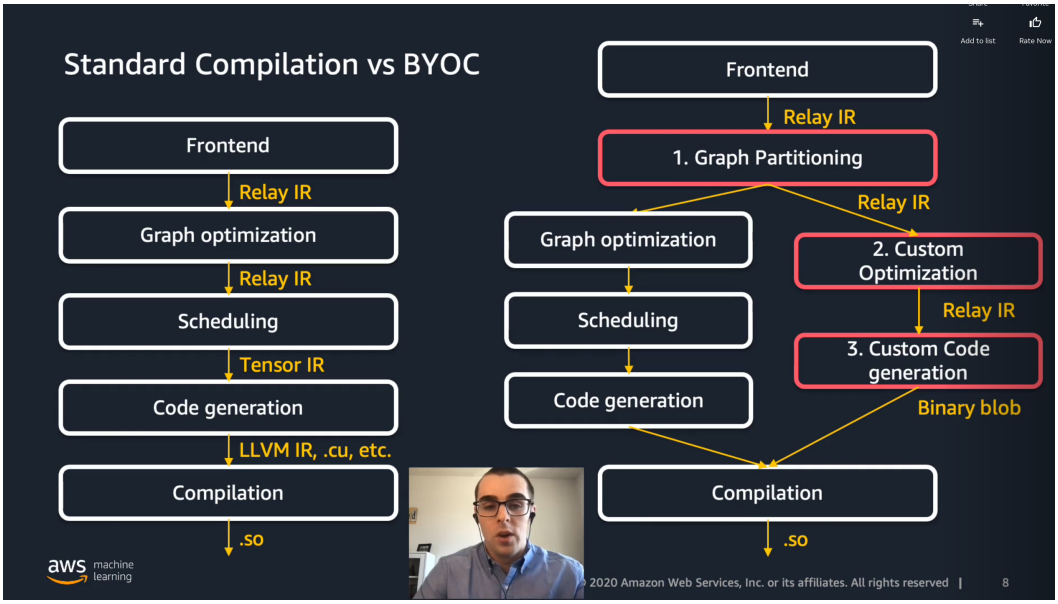

Fig. 2. TVM BYOC Compilation Flow [16]

The design of Relay IR is elegant and expressive, however, Relay IR does not support dynamic shape so well. The third generation of TVM high level IR Relax [21] aims to solve this and will replace Relay in the future work. It might be interesting to compare the Relay with Relax in the future. Another big player in providing unified IR for machine learning compiler is MLIR. MLIR

has a large ecosystem as well and has been used by lots of chips designers to design their own compiler due to MLIR's modular design. For example, AMD just acquired NodAI who uses MLIR as the IR infrastructure of their compiler to expanding AI software capabilities [3]. But now TVM does not support exporting its own Relay IR to MLIR which limits its usability. If Relay IR can support directly offloading to MLIR, it will broaden TVM's use cases. We believe this direction is of huge potential and promising, because it can bring two of the largest players of machine learning compiler together.

The process of convert neural network from Relay IR to TensorIR is called lowering stage. For now, not too much effort is put into investigating TensorIR in the project course. It can be one of the direction for future work as well.

2.2 ANSOR

In TVM the definition of the computation and the implementation of the computation are decoupled. The decouple of computation definition and implementation inherits the idea from Halide [17]. TVM can generate many valid optimized implementations for each hardware [4] under the same computation definition. Implementations are represented as schedules in TVM in the format of tiling size, loop unrolling factor, parallelization etc. Developer can choose to use the schedule primitives to generate implementations and pick the most optimized manually or use the powerful tool ANSOR (Auto-Scheduler) [20] to automatically find optimized implementation. ANSOR is an automated schedule optimizer consists of a schedule proposer and machine learning based cost model. The former one proposes valid schedules by search the parameter space. The latter one is used to evaluate the proposed schedule/implementation on target hardware and help find local/global optimal configuration of the schedule search space with gradient tree boosting method. The whole overflow of ANSOR can be seen from Figure-3. By using ANSOR, developers can optimize an operator computation definition and achieve efficiency on pair with vendor library. There is a RPC mechanism built into ANSOR as well, to help optimize code on embedded devices through remote procedural call. The tuning log is kept in a database, which will be reused in the tuning process and for getting the optimal result. However, it might be possible to build a permanent database for different model hardware pair to reduce the long tuning time. For example, if you have the performance evaluation result in the database for MobileNetv2 on Raspberry Pi 4B, next time you can just avoid running MobileNetv2 on it again by fetching result from database. It's possible to share the database within a group or a company to improve the tuning time efficiency of ANSOR.

Powerful as it is, the long tuning time is always the challenges shared in TVM community. Future work can be using better heuristic to guide the search progress than the current tree boosting method. Or rather than using loop-oriented scheduling strategy, we can use task mapping paradigm as proposed in [9] which reduces the tuning time by 11x compared to ANSOR. In conclusion, it's worthwhile and of great value to explore methods to reduce the long tuning time in ANSOR.

2.3 BYOC

While ANSOR can help to generate efficient implementations for AI operators automatically, due to the huge search space, the tuning time is pretty long. On the other hand, there already lots of vendor libraries specifically designed for according hardware such as TensorRT for NVIDIA GPU, ARM Compute Lib for ARM devices. These libraries are already optimized for us to use. To make fully use of existing software stack, BYOC (Bring Your Own Codegen) [6] is introduced by TVM ecosystem partners. BYOC is a tool for integrating existing codegen (library, accelerator etc) into TVM, enabling mutual benefits for both TVM community and vendor libraries. TVM can utilize the highly optimized vendor libraries reducing the search time, and vendor library can enjoy the whole ecosystem of TVM. The whole process of BYOC overflow can been seen from Figure-4. Besides

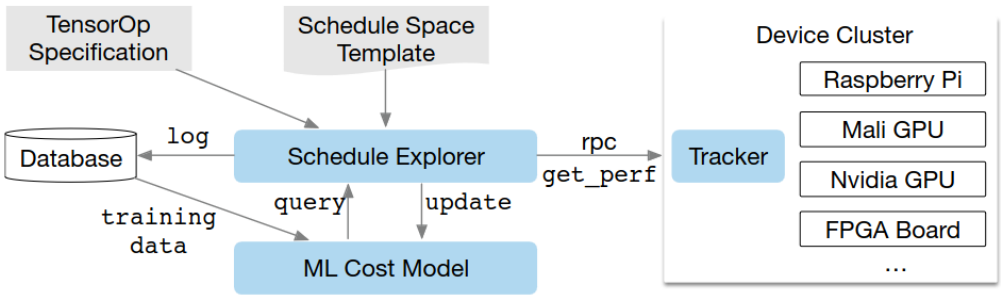

Fig. 3. Auto-Scheduler Overview [20]

reducing codegen time, with more and more new types of operators appearing, most of the time vendor library is slow to support. So it's pretty common you have a model, most the operators in it are supported by vendor libraries like TensorRT, but one or two operators you designed are not supported. In this case, you can offload the supported operators to vendor library, while keep the unsupported operators within TVM stack which will tune them for host or another device execution. Another thing when using BYOC to pay attention to is that when you are offloading operators to another codegen, memory traffic might be introduced because of copying tensors between different codegens. If the memory overhead does not compensate the speedup of the computation, you will want to leave the operators within TVM itself as well. These situations are the reason you want to have host codegen and device codegen just as show in Figure-4.

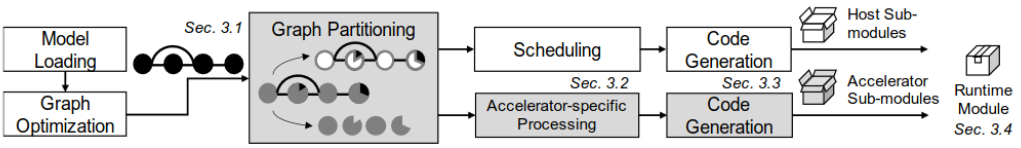

Fig. 4. Bring Your Own Codegen Overview [6]

So for accelerator developers, they can use this mechanism to only offload supported operators to their device while leave the rest of the network to TVM which will tune the unsupported operators. Upon BYOC, UMA (Universal Modular Accelerator Interface) is proposed to create a unified infrastructure for easily integrating external accelerators into TVM [13] which provides a more general interface for integrate accelerators than BYOC.

More details on how to use BYOC to support a neural network inference engine designed for mobile phone can be found in Section 7.

3 DEVELOPMENT ENVIRONMENT

All the developments in this report are performed on a 16 cores, i7-11850H machine equipped with RTX3070 Mobile NVIDIA GPU. The operating system used is Ubuntu 22.04, the TVM version used is v0.13.0, the NVIDIA driver used is 530.30.02 and the CUDA version is 12.1. To benchmark the performance on ARM device in section 6, experiments are done on a Raspberry Pi 4B. To setup

```
@tvm.script.ir_module
class MyMatMulModule:
  @T.prim_func
  def main(A: T.Buffer[(M, K), "float32"],
           B: T.Buffer[(K, N), "float32"],
           C: T.Buffer[(M, N), "float32"],
           ):
    T.func_attr({"global_symbol": "main", "tir.noalias": True})
    for i, j, k in T.grid(M, N, K):
      with T.block("C"):
        vi, vj, vk = T.axis.remap("SSR", [i, j, k])
        with T.init():
          C[vi, vj] = 0.0
        C[vi, vj] = C[vi, vj] + A[vi, vk] * B[vk, vj]
```

Listing 4. Matrix Multiplication with TVM Script

the development environment for TVM, we follow the instructions in the official documentation [10]. In most of the cases, we compile the library within a docker container which makes it easy for later management like configuring for different build options etc.

4 AUTOMATICAL CUDA CODE GENERATION

In this section we use TVM to automatically generate kernels for matrix multiplication, which are the basic blocks of many neural network computation. Generally if we want to develop matrix kernels using CUDA, attention needs to be paid to things like how to utilizing shared memory, how to set block size, and more advanced techniques like register tiling, warp scheduling. The whole progress is complex and requires deep understanding of CUDA programming model and GPU architecture. TVM can be used to solve this challenge by introducing the ANSOR (Auto Scheduler) [20].

In TVM, we first define the computation using TVM's internal domain specific language TVM Script, then based on the computation definition we can have a bunch of different schedules of the computation which represent different implementations utilizing different high performance tricks from software or/and hardware. As you can see from List-4, the abstraction of matrix multiplication in TVM Script is more like mathematical expression. After we finish the computation definition, we can choose to schedule the computation using primitives defined in TVM as shown in List-5, or we can just call the ANSOR API of TVM and provide the hardware type to it. For the latter, based on the hardware type here, TVM can recognizes it's shared memory size, number of streaming multiprocessors etc. Then with the help of the searching algorithms, it can produce the optimal implementation for us. In this project, I compared TVM's auto search result with shared memory tiling strategy implemented with TVM schedule primitives. The manually implemented shared memory tiling strategy achieve 1336.6 GFLOPS, while TVM's auto scheduler achieved 3107.5 GFLOPS. There are still some gaps compared to highly optimized kernel defined in NVIDIA's library, but already a huge improvement given how little effort it requires from developers.

The source code for this section is stored in GitHub repository[3]. Readers can follow the tutorial in the jupyer notebook to understand all the steps.

Future exploration directions may be how to generate other types of efficient kernels for like graph algorithms, sparse computation, transformers etc. Another path can be studying if using the

[3]https://github.com/digital-nomad-cheng/matmul_cuda_kernel_tvm

```
sch = tvm.tir.Schedule(MyMatMulModule)
block_C = sch.get_block("C")
# get loop index
i, j, k = sch.get_loops(block=block_C)
# divide outer loop for blocking
i0, i1 = sch.split(i, [None, 128])
# bind outer loop index with CUDA block
sch.bind(i0, "blockIdx.x")
# bind inner loop index with thread within a block
sch.bind(i1, "threadIdx.x")
sch.mod.show()
```

Listing 5. Navie implementation using TVM schedule primitive

Table 1. Performance between TVM and MNN on MobileRetinaFace, time averaged for 1000 runs

	graph opt	graph opt + ANSOR	MNN
Time(ms)	5324	4853	4485
Speedup	-	9%	15.7%

abstraction of TVM can achieve the same or better result with kernels defined in vendor libraries like NVIDIA's cublas [7].

5 OPTIMIZE AND DEPLOY RETINAFACE

The previous section is optimizing a simple operator kernel. In this section, we try to use ANSOR to optimize a face detection model designed for mobile devices which composed of multiple operators. The model is adapted from RetinaFace [8] with mobilenetv2 as feature extraction backbone for efficiency, we will call it MobileRetinaFace for convenience. The whole pipeline of using ANSOR for a model is almost the same as tuning a single operator. But due to the increasing number of layers to tune, the total time increased a lot in this case - almost six hours on the development machine. As we discussed before the long tuning time is always one of the challenges hold developers back from using it. By using vendor library, you have the expert high performance engineers do the heavy load ahead of time, so when you are using the library you don't need to do this. But when using TVM, you need to use the machine to do the heavy load - the search process. The extra part in this section than last section is using TVM's runtime to deploy a machine learning model with C++. After tuning the whole model, it's easy to export a runtime library in the format of shared library. When deploying, simply use TVM's runtime system to load the shared library, feed input and finally get output. To compare the efficiency of TVM with other neural network library, we test another library MNN [11] on the development machine which is a highly optimized neural network inference engine. For TVM, we compare two versions, one version only performs graph level optimizations without ANSOR optimization. In this case, after graph level optimizations, the operator will be lowered using the predefined schedule strategy which may not be optimal for hardware. The other version, is combining graph level optimizations and ANSOR. The result is listed in Table-1.

We can see from the table that using the default graph level optimizations, the efficiency of TVM is good enough. And although we tuned TVM pretty long, the efficiency increase from ANSOR is not too much. MNN achieved the best efficiency among all three which is not surprising given there is a team of expert engineers work on it. The reason ANSOR does not provide too much performance increase might be because the operators used in MobileRetinaFace are pretty common

Table 2. Performance between TensorRT and ANSOR on MobileNetv2

	Power(W)	Memory(MB)	Time(ms)
Baseline	67	170	1.61
TensorRT	79	246	0.51
ANSOR	79	170	0.49

and the default schedule strategy defined in TVM is more or less approaching optimal. Besides, the size of MobileRetinaFace is super lightweight for our development machine. Our assumption is for operators that are not well optimized or running on hardware with more limited resources, the advantage of ANSOR will become more obvious.

Another issue for our current implementation is that in image tasks, there are usually lots of pre-processing and post-processing steps like resize, color space transform, normalization etc. TVM is used to optimize the neural network itself, but not these pre-processing and post-processing steps while MNN optimizes them as well. Besides, you need to deal with tensor copy from/TVM after/before these steps. These all will bring the whole efficiency of the system down which can be one of the direction for engineering optimization in the future.

The source code for this section is stored in GitHub repository[4], readers can check the repository for more implementation details including how to use ANSOR and how to use TVM runtime under C++.

6 PERFORMANCE COMPARISON BETWEEN TENSORRT AND ANSOR

In this section, we will further investigate ANSOR's performance on NVIDIA GPU compared to NVIDIA's library TensorRT from memory footprint, power consumption and time efficiency perspectives. For edge devices, time efficiency is not the only concern of development, other resources like memory and power are constrained as well. We want to study if TVM can still compete with vendor libraries from all these perspectives. Two models are used in the experiment. One of the model used is MobileNetv2 the most used light weight backbone for mobile devices. The other one is YOLOv8 [12], the most recent object detection model. We only choose the nano version of YOLOv8 here, because other versions will take too long time to optimize with TVM on the development machine which makes it unpractical to do experiments. For comparison, we have a baseline which is model only applied with graph level optimizations in TVM. Both ANSOR and TensorRT optimizations are applied based on the baseline version. For using ANSOR upon NVIDIA GPU, the only difference compared to last section is to change the target type from LLVM for CPU to CUDA for GPU. TensorRT is supported through BYOC in TVM already, so we will use TensorRT through the TVM API which will only offload supported computation intensive operators to TensorRT. According to the source code of TVM where class IsComputeIntensiveGraph is implemented[5], only convolution, dense, batch matmul, sum, max, min, mean and prod operation are offloaded. The running time is averaged for 100 runs. The power consumption and memory footprint are recorded using 'nvidia-smi' tool of NVIDIA which will sample the device at the period of 0.2 seconds on the development device. and The results are kept in Table 2 for MobileNetv2 and Table 3 for YOLOv8 nano.

As we can see from both Table 2 and Table 3, for both models, ANSOR outperforms TensorRT on every aspect. Although the time efficiency is almost the same for ANSOR and TensorRT, ANSOR reduces the memory usage a lot, up to 40% reduction in memory usage for YOLOv8 nano case.

[4]https://github.com/digital-nomad-cheng/RetinaFace_TVM
[5]https://github.com/apache/tvm/blob/main/python/tvm/relay/op/contrib/tensorrt.py

Table 3. Performance between TensorRT and ANSOR on YOLOv8 nano

	Power(W)	Memory(MB)	Time(ms)
Baseline	80	190	7.25
TensorRT	80	328	3.07
ANSOR	80	194	2.99

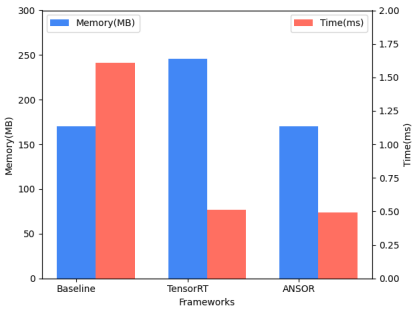

Fig. 5. Performance of MobileNetv2

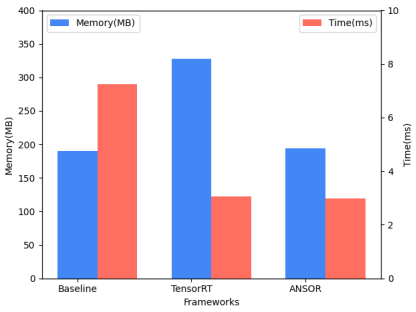

Fig. 6. Performance of YOLOv8 nano

TensorRT achieved 2-3x time efficiency compared the one only applied graph level optimization, but it also increase the memory traffic a lot.

Since we are using TensorRT through TVM, the accuracy might not be correct due to the tensor copy between TVM and TensorRT. This need further investigation, for example, offload the whole model to TensorRT and evaluate the metrics.Another thing to note is when are are considering the power consumption, the tuning progress of ANSOR itself used lots of power as well. It takes 6 hours to generate the result in table for MobileNetv2 and 5 and a half hour for YOLOv8 nano.

In the future it might be worthwhile to explore using ANSOR to optimize the operators that are not offloaded to TensorRT and evaluate the performance combined with TensorRT. It might also be interesting to see the performance difference when using lower bit of data for computing like float16 or int8 which TensorRT is famous for.

All the source code for this section is kept Github repository as well[6], reader can use it as reference for re-implementation.

7 INTEGRATE A NEW CODEGEN - NCNN

After the previous three sections, in this section we will try to make changes to TVM codebase itself, which requires a deeper understanding of TVM. In this part, we try to leverage TVM's BYOC to integrate a new codegen designed for ARM devices into TVM. The codegen we choose here is ncnn [18] - a highly optimized neural network inference engine designed for mobile devices. ncnn has been used in products with tens of millions users such as WeChat. The benchmark baseline here is Arm Compute Lib [2], another neural network engine from ARM. Arm Compute Lib is integrated into TVM through BYOC already. By studying how Arm Compute Lib is integrate into TVM, we can develop the code for ncnn in a similar way. There are several documentations in TVM showing how to use BYOC as a developer as well, both [5] and [19] are good materials for understanding BYOC.

[6] https://github.com/digital-nomad-cheng/tvm_project_course/tree/main/schedule

```
    def _register_extern_op_helper(op_name, supported=True):
        @tvm.ir.register_op_attr(op_name, "target.ncnn")
        def _func_wrapped(expr):
            return supported

        return _func_wrapped
    _register_extern_op_helper("nn.dense")
    _register_extern_op_helper("nn.conv2d")
```

Listing 6. Register supported ncnn operators

```
    def conv_pattern():
        """Create a conv pattern
        Returns
        -------
        pattern : dataflow_pattern.AltPattern denotes the convolution pattern.
        """
        pattern = is_op("nn.conv2d")(wildcard(), is_constant())
        pattern = pattern.optional(lambda x: is_op("nn.bias_add")(x, is_constant()))
        pattern = pattern.optional(is_op("nn.relu"))
        return pattern
```

Listing 7. Pattern matching for Conv2d, optionally with bias add and ReLU in BYOC

TVM will first label those operators supported by custom codegen and partition those operators from the whole computation graph. For those unsupported operators, you can still use the ANSOR to optimize the performance on host side. And for those supported operators, they will be dispatched to your own codegen. That said, there are several steps to use BYOC: 1) annotate the operators you want to offload to the codegen, 2) parse the information of offloaded operators from Relay graph node to JSON node in the codegen, 3) implement the runtime of the codegen and use the JSON node information to initialize the runtime. We will go through these steps one by one with code examples.

Given a computation graph composed of different operators like convolution, fully connected and pooling, we need to recognize which operator can be offloaded to the third party codegen. List-6 shows the code snippets of supporting dense layer and convolution layer in ncnn codegen. The supported Relay operators will be annotated with target codegen name prefix - ncnn here.

Besides registering supported operators, we want to merge some operators together as well. For example, we want to optionally match bias add and/or ReLU with convolution because then we can fuse the bias add and ReLU computation into the convolution, the optimization is called operators fusion. Operators fusion can reduce the data traffic by utilizing data locality of these operators. According to [4], the fused model can be 1.4x faster than unfused model at most. Operator fusion can be achieved by TVM's pattern matching mechanism as shown in List-7. This code section will tell TVM to merge all the convolution operators optionally with bias add and/or ReLU layer coming after. The overview of the operator fusion and annotation process can be seen in Figure-7.

For the second step, after we registered the supported operators, these operators in the format of Relay IR will be passed to the custom codegen. In the custom codegen, we need to parse the Relay graph node into JSON node. List-8 shows part of the codegen implementation for ncnn, it's parsing the Relay graph nodes into JSON nodes depends on the node type. The supported operators should match the ones you registered at Python side.

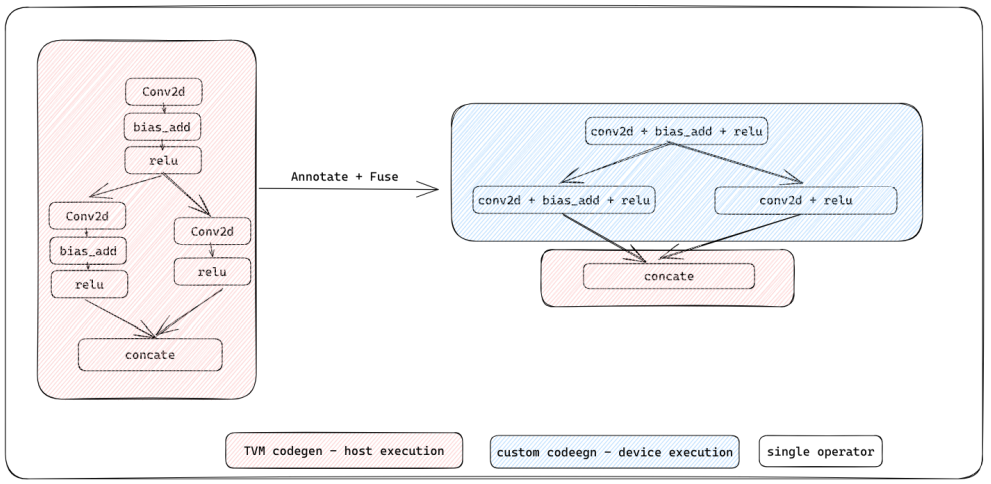

Fig. 7. BYOC fuse and annotate operators

```
std::vector<JSONGraphNodeEntry> VisitExpr_(const CallNode* cn) override {
  if (cn->op.as<OpNode>()) {
    return JSONSerializer::VisitExpr_(cn);
  }
  if (!cn->op.as<FunctionNode>()) {
    LOG(FATAL) << "NCNN JSON runtime does not support calls to "
               << cn->op->GetTypeKey();
  }
  auto fn = cn->op.as<FunctionNode>();
  auto comp = fn->GetAttr<String>(attr::kComposite);
  ICHECK(comp.defined()) << "NCNN Json runtime only supports composite functions.";
  const std::string name = comp.value();
  std::shared_ptr<JSONGraphNode> json_node;
  if (name == "ncnn.dense") {
    json_node = CreateCompositeDenseJSONNode(cn);
  } else if (name == "ncnn.conv2d") {
    json_node = CreateCompositeConvJSONNode(cn);
  }
  else {
    LOG(FATAL) << "Unrecognized NCNN pattern: " << name;
  }
  return AddNode(json_node, GetRef<Expr>(cn));
}
```

Listing 8. Codegen of ncnn, parsing information from Relay graph node into JSON node

For the third step, we need to implement the ncnn runtime. These JSON nodes from step 2) then will be parsed at runtime when we initialize the ncnn runtime. The parsed JSON node information can also be exported as a JSON file to support adaptability for other framework/accelerators which is implemented here[7], at the request of Imsys AB. Normally, we only need to initialize the engine

[7]https://github.com/digital-nomad-cheng/tvm_project_course/blob/main/byoc/alexnet_ncnn_codegen.py

Table 4. Performance between Arm Compute Lib and ncnn on AlexNet, time averaged for 100 runs

	Arm Compute Lib	ncnn
Time(s)	0.125	0.085
Speedup	-	30%

once in which we cached all the layers and tensors for later usage to reduce the memory overhead for creating inference layers, input/output tensors and loading weights from memory. Then at inference time, we just feed input into these cached tensors and forward the cached layers to get the result. You can reference here[8] for the whole implementation of the ncnn runtime which supports dense, convolution and reshape layer.

After we integrate ncnn into TVM through BYOC, we compared its performance with Arm Compute Lib on Raspberry Pi 4b with AlexNet model. The input image size is 227x227, the result is average for 100 runs. The result is kept in Table 4. To our surprise, the time efficiency of ncnn is much better than Arm Compute Lib. 0.085 seconds for ncnn compared 0.125 seconds for Arm Compute Lib, a 30% speedup. In this section, we showed how easy it is to integrate a new codegen into TVM's compilation flow and improve the overall efficiency of our machine learning system.

There are still some problems in our current ncnn BYOC integration remain to be optimized in the future. First, the current implementation still needs to copy input from tvm to ncnn input tensor and copy output from ncnn into tvm output which introduces memory traffic. Future work might be using TVM to manage the memory of ncnn tensor so that memory copy can be avoided. Avoiding memory copy might further increase the performance of ncnn runtime a lot especially when the supported operators are fragmented in the whole graph. The second issue is the current implementation only supports dispatch one layer to ncnn runtime at a time. For example if convolution is followed by dense layer, we have to launch the ncnn runtime two times. This introduce extra memory overhead. In the future, efforts can be put into dispatch subgraph containing multiple layers to minimize the data communication cost between host and cost.

The whole source code for integration of ncnn into TVM using BYOC is provided as a GitHub repository[9].

8 CONCLUSION

In this report, we explored machine learning compilation framework TVM in a practical way. We used TVM for optimizing neural networks for different hardware and added an existing neural network codegen into TVM. Through these project, we have a deeper understanding of the IRs and tools in TVM. Multilevel IR empowers TVM to optimize computation graph of neural network in a hierarchical way. ANSOR helps TVM to generate efficient kernel libraries automatically. BYOC can bring existing software and hardware stack into TVM ecosystems. But these are some limitation of TVM as well, like the long tuning time of ANSOR, performance gap compared to vendor library/compilers, complicate to support new operators and extra memory traffic introduced when using TVM etc. Possible future ideas to research on are listed as following: 1) improving the long tuning time of ANSOR based on the most recent research such as [9], 2) using BYOC to offload computation to FPGA, 3) optimizing the existing ncnn integration by supporting more operators and reducing memory traffic, 4) exploring TVM's performance on lower bit computation and support quantization mechanism, 5) researching on how to bridge TVM IR with MLIR, 6) using

[8] https://github.com/digital-nomad-cheng/tvm/blob/ncnn_codegen/src/runtime/contrib/ncnn/ncnn_runtime.cc
[9] https://github.com/digital-nomad-cheng/tvm

TVM to generate other computation pattern kernels such graph algorithms, sparse computation, CFD problems, and evaluating the performance.

9 ACKNOWLEDGMENTS

This work was done for the DAT085 project course at Chalmers University of Technology. It's under the supervise of several parties including Chalmers University of Technology and Imsys AB. Professor Pedro Petersen Moura Trancoso was the supervisor at Computer Science Engineering of Chalmers. Dag Helmfrid, Dávid Juhász and Mohammad Riazati were industry supervisors from Imsys AB.

REFERENCES

[1] Apache TVM. 2023. Relay pass infrastrucutre. https://tvm.apache.org/docs/arch/pass_infra.html.

[2] ARM. 2023. ARM Compute Library. https://github.com/ARM-software/ComputeLibrary.

[3] Vamsi Boppana. 2023. AMD Completes Acquisition of Nod.ai. https://community.amd.com/t5/ai/amd-completes-acquisition-of-nod-ai/ba-p/639781.

[4] Tianqi Chen et al. 2018. TVM: An Automated End-to-End Optimizing Compiler for Deep Learning. (2018). arXiv: 1802.04799 [cs.LG].

[5] Zhi Chen and Cody Yu. 2020. How to Bring Your Own Codegen to TVM. https://tvm.apache.org/2020/07/15/how-to-bring-your-own-codegen-to-tvm.

[6] 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. (2021). arXiv: 2105.03215 [cs.LG].

[7] NVIDIA Corporation. 2023. CUDA Toolkit Documentation - cuBLAS. (Oct. 15, 2023). https://developer.nvidia.com/cublas.

[8] Jiankang Deng, Jia Guo, Yuxiang Zhou, Jinke Yu, Irene Kotsia, and Stefanos Zafeiriou. 2019. RetinaFace: Single-stage Dense Face Localisation in the Wild. (2019). arXiv: 1905.00641 [cs.CV].

[9] Yaoyao Ding, Cody Hao Yu, Bojian Zheng, Yizhi Liu, Yida Wang, and Gennady Pekhimenko. 2023. Hidet: Task-Mapping Programming Paradigm for Deep Learning Tensor Programs. In Association for Computing Machinery, New York, NY, USA. ISBN: 9781450399166. DOI: 10.1145/3575693.3575702.

[10] Apache Software Foundation. 2023. Apache TVM Documentation - Install. (Oct. 15, 2023). https://tvm.apache.org/docs/install/index.html.

[11] Xiaotang Jiang et al. 2020. MNN: A universal and efficient inference engine. *Proceedings of Machine Learning and Systems*, 2, 1–13.

[12] Glenn Jocher, Ayush Chaurasia, and Jing Qiu. 2023. YOLO by Ultralytics. Version 8.0.0. (Jan. 2023). https://github.com/ultralytics/ultralytics.

[13] Michael J. Klaiber. 2022. UMA: Universal Modular Accelerator Interface. https://discuss.tvm.apache.org/t/rfc-uma-universal-modular-accelerator-interface/12039.

[14] Chris Lattner. 2021. ASPLOS Keynote: The Golden Age of Compiler Design in an Era of HW/SW Co-design. YouTube. https://www.youtube.com/watch?v=4HgShra-KnY.

[15] Chris Lattner and Vikram Adve. 2004. LLVM: A compilation framework for lifelong program analysis & transformation. In *International symposium on code generation and optimization, 2004. CGO 2004.* IEEE, 75–86.

[16] Trevor Morris. 2021. Achieve Best Inference Performance on NVIDIA GPUs by Combining TensorRT with TVM Compilation Using SageMaker Neo. https://www.nvidia.com/en-us/on-demand/session/gtcspring21-s32214/.

[17] Jonathan Ragan-Kelley, Connelly Barnes, Andrew Adams, Sylvain Paris, Frédo Durand, and Saman Amarasinghe. 2013. Halide: a language and compiler for optimizing parallelism, locality, and recomputation in image processing pipelines. *Acm Sigplan Notices*, 48, 6, 519–530.

[18] Tencent. 2023. ncnn. Version 2023.10.16. https://github.com/Tencent/ncnn.

[19] Apache TVM. 2023. Bring Your Own Codegen To TVM. https://tvm.apache.org/docs/dev/how_to/relay_bring_your_own_codegen.html.

[20] Lianmin Zheng et al. 2020. ANSOR: Generating High-Performance Tensor Programs for Deep Learning. (2020). arXiv: 2006.06762 [cs.LG].

[21] Denise Kutnick Ziheng Jiang Yuchen Jin. 2022. Relax: Co-Designing High-Level Abstraction Towards TVM Unity. https://discuss.tvm.apache.org/t/relax-co-designing-high-level-abstraction-towards-tvm-unity/12496.

Received 25 October 2023; revised 25 October 2023; accepted 25 October 2023