Hewlett Packard Enterprise

DHBW
Duale Hochschule
Baden-Württemberg

A load-aware scheduler for large-scale neural network autotuning

Project Thesis II / T2000

for the study program
Computer Science

at the
Baden-Wuerttemberg Cooperative State University Stuttgart

by
Dominik Stiller

Submission Date	September 12, 2019
Project Period	18 Weeks
Company	Hewlett Packard Enterprise
Corporate Supervisor	Junguk Cho
University Supervisor	Prof. Dr. Bernd Schwinn
Matriculation Number, Course	4369179, TINF17A

Declaration of Authorship

I hereby declare that the thesis submitted with the title *A load-aware scheduler for large-scale neural network autotuning* is my own unaided work. All direct or indirect sources used are acknowledged as references.

Neither this nor a similar work has been presented to an examination committee or published.

Sindelfingen	September 3rd, 2019	
Place	Date	Dominik Stiller

Abstract

Real-time computer vision applications with deep learning-based inference require hardware-specific optimization to meet stringent performance requirements. However, this approach requires vendor-specific libraries developed by experts for some particular hardware, limiting the set of supported devices and hindering innovation. The deep learning compiler stack TVM is developed to address these problems. TVM generates the optimal low-level implementation for a certain target device based on a high-level input model using machine learning in a process called autotuning.

In this paper, we first explore the capabilities and limitations of TVM's autotuning implementation. Then, we develop a scheduler to orchestrate multiple, parallel autotuning jobs on shared computation resources such as CPUs and GPUs, allowing us to minimize resource idle time and job interference. Finally, we reflect our design choices and compare the efficiency of our approach with the default, scheduler-less design.

Contents

Acronyms

ANN artificial neural network

CNN convolutional neural network

DL deep learning

GPU graphics processing unit

ML machine learning

RPC remote procedure call

TC TensorComprehensions

List of Figures

List of Tables

List of Source Codes

1 Introduction

AI is increasing in popularity AI is used in many different areas Users aren't experts Existing products for easier setup and deployment of training and inference infrastructure by offering AI infrastructure as a service

1.1 Problem

Common applications like industrial monitoring or autonomous driving require real-time performance accelerator hardware with device-specific model optimizations needed Currently manual optimization Requires deep knowledge -> not easy for non-expert users

required: automated inference performance optimization (autotuning) To offer it as a service so it can be used by a larger audience requires it to be scalable Current autotuning does not scale well To the best of our knowledge, there is no existing solution.

1.2 Scope

In this paper, we design and develop the prototype of a central, load-aware scheduler to solve this problem This scheduler controls multiple jobs that share computation resources to enable large-scale artificial neural network autotuning First step, develop framework to examine capabilities and limitations of autotuning in different configurations on multiple accelerator devices Allows us to find properties which we can leverage to parallelize autotuning Design and create a working proof-of-concept implementation Evaluate our scheduler design and compare with default implementation Propose an Autotuning as a Service architecture as base for future work

Thesis: Controlling the execution of multiple jobs by a load-aware scheduler makes large-scale autotuning more efficient in terms of - autotuning completion time - resulting inference performance and - hardware requirements

dont improve autotuning process itself, but propositions are made in future work dont develop actual autotuning as a service product, but propose an architecture

Project was conducted by Hewlett Packard Labs

2 Background

Machine learning (ML) has become an important sub-field of computer science. It emulates human-like learning using mathematical models, so predictions can be made about new data in the future. Rather than explicitly programming how to make those predictions, the developer provides sample data during *training*. Once the accuracy of the trained model is sufficient, it can be used for *inference*. The model can be thought of as the approximation of a function mapping from the input data to some output, e.g., a label for classification, or a numerical value for regression [1, p. 164].

2.1 Artificial Neural Networks

While there are a variety of ML models in use today, artificial neural networks (ANNs) are among the most powerful and flexible, due to their ability to represent complex functions [1, p. 163]. They find application in fields as diverse as image and speech recognition, movie recommendations and medical diagnosis.

ANNs are composed of multiple layers, with the output of one layer being the input of another layer. The first layer receives the input data, and the last layer produces the final output. With an increasing number of layers, or *depth* of the network, more complex functions can be approximated. These deep networks are subject of the ML sub-field of deep learning (DL). All layers perform some computation given a set of trained or specified parameters and the input. Both parameters and inputs are tensors, a higher-dimensional generalization of vectors and matrices. Traditional ANNs feature only fully-connected layers with some activation function.

Grid-like data such as time series (1D) or images (2D) benefit from additional layers found in convolutional neural networks (CNNs) [1, p. 326]. This makes CNNs an important tool in state-of-the-art computer vision applications. CNNs apply convolution and pooling to a region of the input tensor in a sliding fashion, so values only interact with other values that are located in their neighborhood. Convolution applies one or more kernel matrices to the input, which are element-wise multiplied with the current region and then summed

up into a single output value. Pooling averages or finds the maximum of the region as output value. Both operations support a variable stride (step size) and padding.

While neural network models logically consist of a series of layers, machine learning frameworks usually represent them in a computation graph. The computation graph's first vertex is the input node, followed by a number of tensor operators with their parameters performing the layer's computations, and finally an output node. The edges describes how data flows between the vertices.

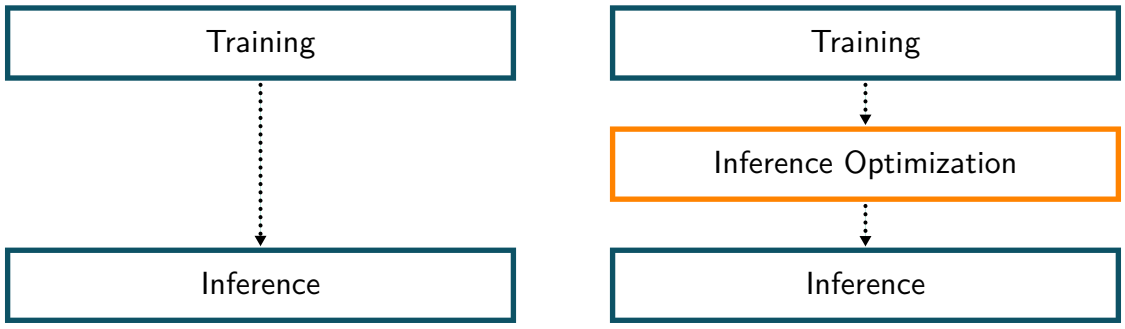
2.2 Inference Optimization

Typically, the amount of inferences heavily outweighs the amount of trainings, since training only needs to be done once (albeit model re-training is usually done periodically when new training data is available). For this reason, while training takes longer by several orders of magnitude, speeding up inference has a larger impact and is a worthwhile endeavor. Reduction of the inference time has a number of advantages:

- less hardware is required to achieve the same inference rate
- a higher inference rate can be achieved with same hardware
- real-time applications are facilitated, e.g., autonomous driving, industrial monitoring

In real-time applications with a high inference rate, even small improvements in inference performance (in the order of milliseconds) can be critical to guarantee the required throughput. For example, a major hard drive manufacturer detects defects in their products using a CNN-based smart manufacturing solution [2, p. 11]. They perform inference on 3 million images every day, so if they could only save 5 ms per image due to some performance optimization, that would amount to over 4 h less total inference time every day [3]. Alternatively, they could save costs by needing less servers that are equipped with expensive accelerator devices.

Accelerator devices such as graphics processing units (GPUs), ASICs like tensor processing units or FPGAs are used to speed up both training and inference. However, generic ML models cannot make full use of accelerator capabilities and fall short of leveraging the full potential. Every device has different features such as specialized instructions, memory size and layout, cache access, and parallelization support. This means that models need to be attuned to the *target device* to achieve the best inference performance. But even if no special accelerator devices are used but only a conventional CPU, adapting to the specific architecture can yield great performance benefits [4, p. 1]. In a traditional machine workflow, the trained model is deployed as-is (Figure 1a). Inference optimization adds an



(a) Traditional without inference optimization (b) Improved with inference optimization

Figure 1: Machine learning workflow

additional step, turning the trained model into a functionally equivalent but optimized version before inference (Figure 1b). In this step, we first apply high-level transformations that rewrite the computation graph by, for example, fusing tensor operators, pre-computing constant parts or transforming the data layout in memory [5, p. 1–3]. More importantly, however, we can change the low-level implementation of tensor operators.

The model determines what tensor operators are calculated, but it does not specify how they are calculated. Deliberately choosing the actual implementation offers great optimization potential. There is always a generic naïve implementation, which is the straightforward way of performing the calculation. However, it does not consider, e.g., memory sharing between threads or cache access patterns, which can have a significant adverse effect on performance [6]. Techniques such as loop unrolling, reordering and tiling as well as multi-dimensional threading and tensor compute instructions can help leverage the accelerator's capabilities, but there is an abundance of combinations of settings for these techniques, the best of which is very much specific to the target device [5, p. 2]. Finding the optimal such combination is the key aspect of tensor operator optimization.

Convolution operators are computationally very intensive and make up the majority of modern CNNs, such as Inception[7] and ResNet[8]. Therefore, tensor operator optimization should focus on convolution over other types like pooling and fully-connected. It is not possible to optimize convolutions in general, but we need to optimize for every distinct parameter set that is present in the computation graph, i.e. combination of input shape, kernel shape, padding, and stride. This means that the effort increases with a higher variety of layer configurations.

2.3 Manual Optimization

Optimized implementations for tensor operators with a specific parameter set are provided by accelerator vendors in libraries like cuDNN for NVIDIA GPUs and Intel Math Kernel Library for Intel CPUs. The vendors possess the hardware-specific knowledge to write good implementations by hand, but human expertise is required for this approach. While state-of-the-art, manual optimization has a number of inherent shortcomings:

- slow support for new devices
- slow support for new graph-level optimizations
- no support for unconventional shapes
- vendor lock-in

These limitations hinder innovation, which is undesirable in a field so fast-evolving and relatively young as DL. Researchers need to make a choice between avoiding devices, high-level optimizations and new network architectures that are not supported by those predefined operator libraries, and using unoptimized implementations [5, p. 1].

2.4 Automated Optimization

Automated tensor operator optimization, or *autotuning*, overcomes these shortcomings by eliminating the need for human experts. Vendor-agnostic frameworks can discover good implementations regardless of hardware, model or graph optimizations. This enables innovation by fostering experimentation with novel or unconventional layers and high-level transformations that are not supported by manual libraries. Autotuning can achieve the same, in some cases even better inference performance than state-of-the-art vendor-provided operator libraries. Compared to these libraries, autotuning delivers speedups of 0.98× to 3.5× on CPU [4, p. 9] and 1.6× to 3.8× on server-class GPUs [5, p. 10] for commonly used CNNs. Even a slightly worse performance is impressive considering that no domain-specific expert knowledge has been applied but only a few hours of autotuning.

Autotuning works by exploring the space of possible implementations in an organized fashion. Functionally equivalent implementations can be generated by a *schedule* which defines a series of parametrized transformations that can are applied to the naïve implementation. The *search space* is defined by the set of permutations of parameter settings. The settings control, for example, loop unrolling factors, loop order, loop tiling sizes and thread numbers, and can usually be adjusted in steps of powers of 2 [5, p. 5] [9, p. 16]. One specific combination of settings, i.e. one element of the search space, is called *configuration*.

$$C = A^T B$$

(a) Mathematical expression

$$C = sum(A[k, y] * B[k, x], axis = k)$$

(b) Tensor index expression

```
for y in range(1024):
  for x in range(1024):
    C[y][x] = 0
    for k in range(1024):
      C[y][x] += A[k][y] * B[k][x]
```

(c) Naïve low-level code

```
for yo in range(128):
  for xo in range(128):
    C[yo*8:yo*8+8][xo*8:xo*8+8] = 0
    for ko in range(128):
      for yi in range(8):
        for xi in range(8):
          for ki in range(8):
            C[yo*8+yi][xo*8+xi] +=
              A[ko*8+ki][yo*8+yi]
                * B[ko*8+ki][xo*8+xi]
```

(d) Tiled low-level code

```
inp_buffer AL[8][8], BL[8][8]
acc_buffer CL[8][8]
for yo in range(128):
  for xo in range(128):
    vdla.fill_zero(CL)
    for ko in range(128):
      vdla.dma_copy2d(AL, A[ko*8:ko*8+8][yo*8:yo*8+8])
      vdla.dma_copy2d(BL, B[ko*8:ko*8+8][xo*8:xo*8+8])
      vdla.fused_gemm8x8_add(CL, AL, BL)
    vdla.dma_copy2d(C[yo*8:yo*8+8,xo*8:xo*8+8], CL)
```

(e) Low-level code with accelerator-specific instructions

Figure 2: Expressions and low-level code for transposed matrix multiplication [5, p. 4]

Defining the values a setting can take is done manually for each class of target devices, but the search is guided by an algorithm that proposes candidate configurations. This is necessary since the size of the search space makes the brute-force approach of trying all configurations infeasible. As an example, the search space size for a ResNet-18 on an NVIDIA GPU exceeds 172 million possible configurations, any one of which could be the best. ML-based or genetic algorithms help with rapid convergence to a decent, or ideally the best configuration without need of exhausting the whole search space.

Figure 2 provides an example of how different configurations affect the generated low-level code. The operator functionality is some mathematical calculation, in our example a transposed matrix multiplication (2a). Before autotuning, that functionality is specified in a tensor expression language, which describes how to compute each element of the output tensor from the input tensors using a concise notation (2b). Note that this notation is implicit, meaning that it does not prescribe implementation details. The autotuning framework then makes the computation explicit by applying a schedule with specific parameters from the configuration to the operator's default code. The simple but naïve reference code can be used to check the correctness after complex transformation (2c). The low-level code is an intermediate representation that allows transforms, e.g. tiling for memory locality (2d) or accelerator-specific instructions for buffers and specialized tensor operators (2e). The specific tiling factors and buffer sizes can be varied and are determined by the applied configuration [5, p. 4 ff.] [9, p. 9 ff.].

Since the low-level code is only an intermediate representation, target-specific code, e.g., LLVM assembly for CPU or a CUDA kernel for NVIDIA GPUs, needs to be generated. The appropriate compiler then builds that code, possibly in parallel for multiple configurations in a batch, after which the implementation can be executed. For autotuning, the execution

time is then profiled on the target device to evaluate the performance. The profiling result is then stored alongside the implementation and fed back to the algorithm that selects candidate configurations. This allows the algorithm to improve its proposals for the next batch [9, p. 15 f.]. The iterative autotuning process can be stopped when a sufficiently fast implementation has been found or no better one has been discovered in a long time. Then, the full computation graph can be used for inference with the best implementations that have been found in the autotuning process for all operators.

There are two frameworks that implement autotuning, which will be described now.

2.4.1 TensorComprehensions

TensorComprehensions[1] (TC) has been developed by Facebook's AI Research team and comprises three main components: a language to express tensor computations (similar to Figure 2b), an optimizing compiler to generate efficient GPU code from expressions, and an autotuner that finds good implementations and stores them in a compilation cache. It uses a polyhedral compiler to reason about and manipulate the loop structures of an implementation [9, p. 3]. However, only tensor-operators are considered, the framework is designed to be independent of computation graphs [9, p. 4].

Autotuning in TC starts from configurations that worked well for similar expressions, and some predefined strategies. The autotuner determines the configuration parameters and admissible value ranges. Then, a genetic algorithm generates a batch of candidate configurations. The value for each configuration parameter is randomly selected from one of three parents that are selected probabilistically based on their fitness. Furthermore, there is a low probability of mutation, which means that a random value is assigned to some parameters. Configurations are then compiled in parallel and profiled on an available GPU. A fitness value inversely proportional to the execution time is assigned to the configuration and stored in the autotuning database. Then, the process starts anew by selecting the next candidates using the updated database. This is repeated for a set amount of time [9, p. 15 f.].

2.4.2 TVM

TVM[2] started as a research project at the University of Washington but is now supported and used by a large open-source community and companies like Amazon and Facebook. Unlike TC, which only represents and optimizes tensor operators, TVM is an end-to-end

[1] `https://github.com/facebookresearch/TensorComprehensions`
[2] `https://github.com/dmlc/tvm/`

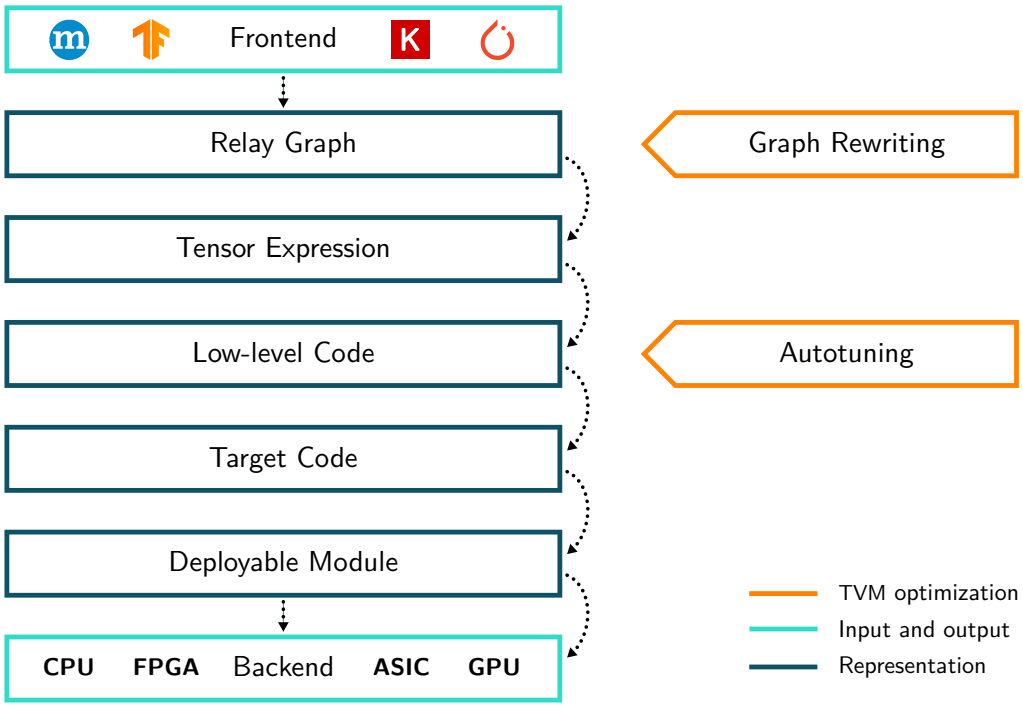


Figure 3: Levels of abstractions in TVM flow

DL compiler stack. It can import whole models from a frontend framework and build minimal, optimized modules that can be deployed to backends like CPUs, GPUs or FPGAs. Figure 3 shows how the layers of the stack provide different levels of abstraction.

The top layer in the TVM stack is Relay. Relay is an intermediate model representation that enhances traditional computation graphs with concepts of functional programming to form a more powerful language. Relay supports shape-dependent tensor types and automatic differentiation, which is essential for DL training [10, p. 61]. Additionally, a runtime to execute Relay programs in various programming languages is provided and needs to be present whenever executing TVM-based models. Relay programs can be created programmatically or from a textual source code. More convenient for users, however, is the import from diverse frontends, including TensorFlow, Keras, PyTorch and MXNet, which enables the use and optimization of existing models. Graph-level optimization in TVM is pass-based, with each pass inspecting or rewriting the syntax tree of the Relay program in some way. Standard passes are provided and perform, for example, automatic differentiation, type inference, operator fusion or tensor layout transformations [5, p. 3]. Beyond that, writing custom passes is facilitated by an extensible design.

Next in the stack is a tensor expression language, which has similar features as TC's. It allows user to describe computation rules that generate a tensor without specifying loop structures and other details. The rules are composed of primitive mathematical operations like addition and multiplication and are expressive enough to describe tensor, matrix and

vector operations. TVM comes with tensor expressions for common computations used in DL such as various activation functions, convolution, pooling, and matrix multiplication [5, p. 4 f.]. The tensor expression language is used to describe the functionality of tensor operators from the model. In the usual TVM workflow, the required operators are extracted from the Relay graph and matched with existing tensor expressions, so there is no need to write them manually.

Implicit tensor expressions need to be mapped to explicit, backend-independent loq-level code. TVM, again, uses a pass-based design, which is different from TC's polyhedral approach. Basic transformations called schedule primitives are combined into schedules that are applied to the naïve straightforward implementation to, for example, change loop structures and thread binding. This design is based on the Halide language for image processing, which works with similar multi-dimensional data as DL, but enhances it with more primitives to optimize accelerator performance. TVM leverages nested and cooperative parallelism to make effective use of GPU memory structure by enabling data reuse across threads through shared memory regions. This is done in a special memory scoping pass. TVM also equips the low-level code with hardware-specific instructions through a tensorization pass which matches computations patterns with a corresponding intrinsic from the target (such as general matrix multiply), making it extensible for new hardware architectures. A latency hiding pass introduces explicit management of fine-grained synchronization for memory and computation instructions on specialized DL accelerators [5, p. 5 f.]. Default schedule templates are provided for every hardware type, but users can create their own templates to incorporate their knowledge of the backend.

Low-level code cannot be executed, but it can directly be converted to target-specific code and then compiled for the target device. Backend-specific code generators create the source files, which are then built by the respective compiler and packed into a module which contains the implementation of all tensor operators in the model. This module can be deployed along with a JSON description of the Relay graph and a parameter file containing the weights for all operations. The TVM runtime (300 kB to 600 kB) needs to be installed on the target system to execute the model. However, a full DL framework is not required, making TVM modules very lightweight to integrate into applications.

While TCs's autotuner is guided by a genetic algorithm, TVM uses a ML-based cost model to predict the performance of an implementation. Specifically, gradient boosted trees are used because of their advantage in training and prediction speed over neural network-based models. Since the model is queried frequently, the inference overhead must be smaller than the profiling it seeks to replace. While profiling can be in the order of seconds, the gradient boosted trees model performs prediction in 0.67 ms on average. Model training time also needs to be considered; the cost model is updated periodically as more configurations have

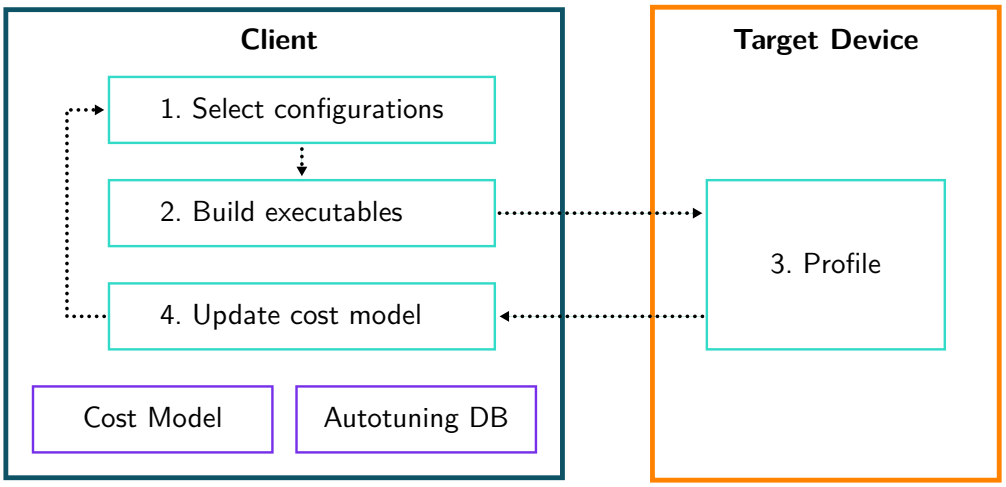


Figure 4: Iterative autotuning process in TVM

been explored, which improves the accuracy for further predictions with more experimental trials. This learning-based approach is preferable to static, predefined cost models for every new hardware target, which is infeasible due to the increasing complexity of modern accelerators [5, p. 8 f.]. Pure low-level code cannot be used as input for the cost model, we need to encode it into a vector space first. This encoding needs to be a transferable representation which is invariant between programs to make the cost model effective. Encoding works by extracting context features from each loop level, including memory access count, reuse ratio of each memory buffer and loop annotations such as "unroll" or "parallel". Furthermore, context relation features enable generalization across different loop nest patterns [11, p. 4].

Autotuning in TVM is an iterative process as seen in Figure 4. We call the component that executes the autotuning logic *autotuning client*. Profiling the implementation requires execution on the actual target device. This can make autotuning a distributed process, if the client is another device. We call the execution of the autotuning process for one model a *job*. However, autotuning is not performed for a whole model at once, but rather for a set of *tasks* which correspond to autotunable tensor operators with a specific configuration (shapes, padding stride). These tasks need to be extracted from the model before starting the process for each of them. Autotuning consists of four stages that depend on each other, making it necessary to execute them in sequential order. Understanding the stages and their dependencies is key for enabling large-scale autotuning.

Initialization At the start of each task, profiling results from previous jobs are loaded from a global autotuning database, a file that contains data from all previous jobs along with information about the target and configuration. The loaded results are passed to the cost model for transfer learning. This yields good cost model from

the beginning and improve in quality over time. Then the autotuning loop can be launched.

1. **Select candidate configurations** At the start of each iteration, a batch of candidate configurations that have a promising performance is selected using the cost model. A simple strategy such as enumerating and running every configuration through the model, then selecting the top performers is impracticable with large search spaces. Rather, candidates are selected using parallel simulated annealing, which is a heuristic optimization algorithm that trades off finding an exact optimum for a much improved speed. Additionally, exploration is ensured by random selection of some configurations. If no training data exist yet, random candidates are picked.

2. **Build executables** The client combines the batch of configurations from the previous stage with the schedule template, then applies the schedule to the tensor expression for the operator of the current task. The resulting low-level code is then translated into backend-specific code and compiled. The result is a tar file that contains everything that is necessary to run the executable, namely the compiled tensor operator itself and backend-specific code such as the CUDA driver library for NVIDIA GPUs.

3. **Profile on target device** Since the cost model's prediction of the implementation's performance are not completely reliable, the real performance needs to be evaluated on the target device. The tar files from the build stage are uploaded to the target device. Then the implementation is profiled by running the executable a number of times with random data. The measured execution times are averaged and returned to the client, which stores the results in the autotuning database for this job. . Parallel profiling on the same computation resource should be avoided to guarantee accurate results.

4. **Update cost model** The cost model is updated with the measurements from the profiling stage to improve the proposed configurations in the next iteration.

Finalization After a certain number of trials, the loop is stopped. The best configuration that was discovered can now be used to build a faster implementation of the tensor operator that was optimized. Usually, the best configuration is also written into a separate database that contains only the best known configurations. The autotuning database for this job is merged with the global one. This concludes the autotuning process in TVM.

The target device is usually specialized for DL workloads. Therefore, it is desirable to run the client on a machine that features a strong CPU to accelerate the compute-intensive build and profile stages. This distribution across multiple machines requires an remote

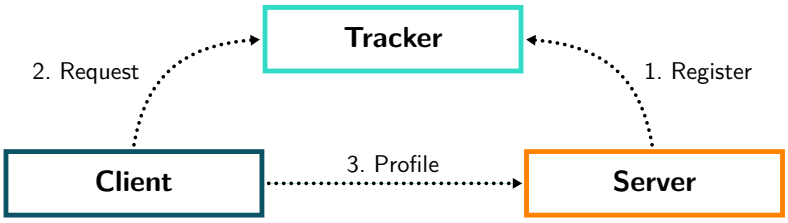


Figure 5: TVM's RPC architecture

procedure call (RPC) infrastructure that makes it possible to profile on a different server. TVM's RPC architecture (Figure 5) comprises three components:

Client A client runs an autotuning job and and is responsible for selecting the candidate configurations, building the executables and updating the model with the profiling results. This means that the client contains both the cost model and the autotuning database. It also controls the profiling, but the actual execution is happening on servers.

Server A server can receive and execute TVM modules, which basically makes it an RPC-enabled TVM runtime that runs on the target device. The interface on the client side does not change because TVM transparently handles remote execution like local execution. A server has a *device key*, which is an arbitrary identifier for a certain device type, but usually is it based on the accelerator's name. Multiple servers can have the same device key if they run on identical target devices.

Tracker A tracker keeps a list of servers to help clients discover unused servers for profiling. The tracker matches incoming requests from clients with free servers using a FIFO-based scheduling algorithm. Scheduling is implemented using a queue for servers and a heap for requests. Requests can have a priority.

TVM's RPC is enabled by two distinct protocols. The control plane protocol is used for communication involving the tracker, namely server registration and requests from clients. The data plane protocol facilitates remote execution on a server and is initiated by clients.

First, the tracker is started and listens on the first free port between 9190 and 9199. Then, one or multiple servers are started which bind to ports between 9091 and 9199. They register with the tracker by transmitting the device key, and the address and port which clients can use to connect. The tracker puts them in a queue, with separate queues for every device key. At this point, clients can request a server with a specific device key. The tracker matches the free server with that device key that registered first with the request that has the highest priority, then the one that was received first. If requests have the same priority, scheduling degrades to simple FIFO scheduling, and the request heap

effectively becomes a queue. Once a client has acquired a server, it is marked as busy in the tracker and the client initiates a connection to the server to use its TVM runtime for profiling.

Since autotuning works in batches, usually not a single but multiple servers are requested to run profiling in parallel. This can speed up profiling if multiple target devices are available. For example, if a machine is equipped with 4 GPUs of the target device type, 4 RPC servers can be launched on that machine, with each one being assigned to a different one of the GPUs.

In this project, we use TVM instead of TC because of the novel, machine learning-based approach, which promises better results than a genetic algorithm due to better guidance by the cost model. We are using the TVM version from June 11, 2019 (commit 8f219b9) for comparable results throughout the project. We made some modifications:

- Add decomposed version of autotuner with separate methods for stages
- Add time measurement for autotuning stages
- Add loading of autotuning records from multiple files
- Fix Tensorflow import for models including PlaceholderWithDefault

3 Using TVM

For our end goal of enabling large-scale autotuning, we need to explore the current capabilities and limitations of TVM first, especially with regard to the execution of multiple autotuning jobs simultaneously. The modern DL landscape is very diverse in terms of models and hardware, so to evaluate TVM in a diverse range of scenarios is crucial for gaining a proper understanding. To this end, we developed a framework that enables us to a large number of experiments rapidly.

3.1 SimpleTVM

Since using TVM follows a similar flow every time, we created SimpleTVM which exposes the individual steps through a convenient interface. This makes it easy for researches who are new to TVM to get started. FSince a lot of the experiments include benchmarking, time measurements are taken for most steps and automatically saved in a *benchmarking context*. The flow of SimpleTVM has some dependencies, which are enforced. For example, a model needs to be imported before building. The interface including possible flows is depicted in Figure 6. The methods that are exposes are now regarded closer.

`from_model` Loading the Relay representation for the model is the beginning of a TVM flow. To that end, TVM supports the import from various frontends. Before the import of the model, however, it needs to be loaded and prepared for import. How exactly this is done differs even inside the same framework. SimpleTVM provides

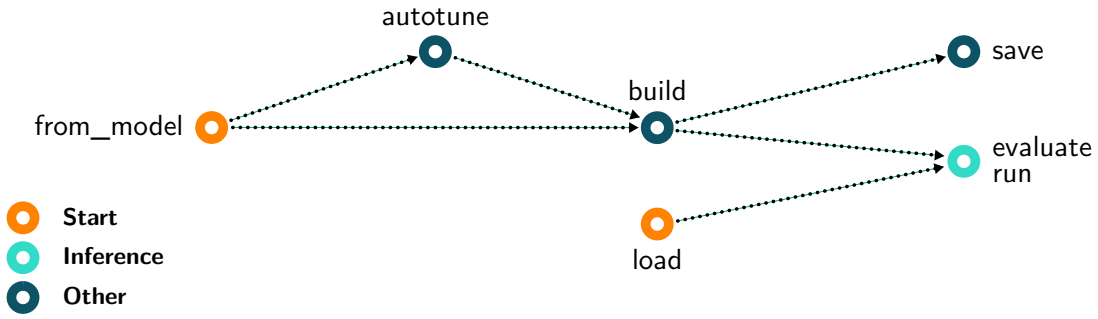


Figure 6: Interface and flow of SimpleTVM

a unified import interface for TVM testing models, TensorFlow saved models (.pb files) and TensorFlow hub models. `from_model` can easily be extended to support more frontends.

`autotune` Once a model has been imported, autotuning can be run for each of the tensor operators. This step is optional, since TVM falls back to a default implementation if no records for that tensor operator exist in the autotuning database. Since this step is rather complex, there is a plethora of configuration options, the most important of which are exposed in `autotune`'s interface. SimpleTVM is designed to get new users started quickly so there are default values, but more experienced users can adjust the values according to their circumstances.

`build` Before a TVM model can be executed, the target-specific executable needs to be build from the Relay model as described in the previous chapter. When building, SimpleTVM can either automatically use records from the global autotuning database, or the results from a specific autotuning job can be used.

`save` After building, the library containing the operators can be saved along with the graph description and weights.

`load` Beyond starting from an imported model, SimpleTVM can also load a previously saved TVM modules, which makes it possible to use a model that has been autotuned earlier. Saved modules can only be loaded if they have been built on the same device.

`run` Inference can be run using this method. It accepts and returns a NumPy array. The input can, for example, be an image. However, loading the image and preparing it for inference, e.g., scaling and normalizing, needs to be done by the user.

`evaluate` To profile the performance, `evaluate` runs inference on random data multiple times, then averages the measured times. However, in contrast to the profiling stage of autotuning, not the performance of individual tensor operators but the whole model is measured.

An example of how SimpleTVM is typically used is presented in Listing 1. First, the `BenchmarkingContext` is created (Line 1), which stores information about the current run such as the run id (a 32-character alphanumeric identifier for this execution of SimpleTVM), target device, measured times, the loaded model and the target device key to send to the tracker. When using a CPU as target device, the CPU architecture should be specified so TVM can select the proper hardware-specific tensor instructions. The benchmarking context is passed to the `SimpleTVM` object (Line 2). Here, the address of the RPC tracker can be specified for distributed autotuning. If the address is not specified, autotuning will create a local tracker and server to perform autotuning on the same device as the client.

```
1 ctx = BenchmarkingContext('cpu', device_key='i7', cpu_arch='skylake')
2 tvm = SimpleTVM(ctx, rpc_tracker=('tracker', 9190))
3 tvm.from_model('mobilenet.pb', output_name='out', output_size=10)
4 tvm.autotune().build().save().evaluate()
5 ctx.save()
6
7 # Saved model can be used later to run inference
8 tvm.load('run_id')
9 prediction = tvm.run(data)
```

Listing 1: Typical SimpleTVM flow for CPU including autotuning

Next, a model is imported (Line 3). Since the name of the output layer and the size of the output vector differs, it needs to specified explicitly. SimpleTVM's concise, chained syntax is used to autotune, build, save and evaluate the model (Line 4). For the sake of brevity, default parameters are leveraged, but the user can customize the actual calls to TVM functions by providing more parameters. Finally, the benchmarking context is saved (Line 5). This enables analysis at a later point, e.g., to examine the autotuning process or the inference performance measured by the evaluation. Note that this step is distinct from the saving of the TVM module. At a later point and usually by another application, the saved module which is identified by the run id can be loaded back. Then it can be used run inference on any data.

Additionally to SimpleTVM, we developed an automated benchmarking script called *superb*. superb is short for "super benchmark" because it allows testing of different configurations without human intervention, so it performs benchmarking on a higher level than SimpleTVM's mechanisms. The user can specify the values for all parameters that should be tested. superb enumerates all possible combinations, effectively determining the n-ary product set of all value lists, then executes SimpleTVM once with each configuration. Additionally, it sets up the required servers and the tracker. The results from all configurations are collected and can then be processed by another script. This script evaluates the resulting inference performances, aggregates some information and writes them into a file, enabling further analysis with other tools such as Jupyter notebooks.

All SimpleTVM-related files are stored in the "~/.tvm-benchmark" directory. This includes the autotuning databases of currently running jobs, the global autotuning database and a file containing only the best known configurations. There are subdirectories for each SimpleTVM run with a log file for debugging, the saved benchmarking context and the autotuning log file for this run, if applicable. In another subdirectory, the results of superb experiments are collected with a csv file containing the aggregate information like mean autotuning time and the mean execution time for each stage. Finally, all saved modules are saved in a directory named after their run id.

Since we want to test TVM on a variety of machines, we created Docker images to be able to easily deploy TVM with all dependencies on any server. The GPU version also includes the CUDA libraries, and a helper script for using the images mounts some folders into the container and sets up the environment. The Docker images in conjunction with SimpleTVM and superb form the foundation for our experiments.

3.2 Parameters

Autotuning with TVM offers a plethora of configuration options that affect both the autotuning process itself and the result. Setting these parameters to adequate values for the given job and hardware requires knowledge of how TVM works, but in some cases it is a matter of trial and error. However, guidelines and descriptions of the most important parameters can help. All of the following parameters can be specified when using SimpleTVM

Number of trials This determines the number of configurations to try for each autotuning task. A higher number will generally result in a better inference performance since the search space can be explored more, but this results in an increased autotuning completion time. However, the result starts to converge to the optimum after about 500 iterations, so there is a limit to the inference performance that can be achieved. Especially with CPUs, that have a small search space compared to GPUs, there might not even be more options to try. Practically, the optimal result can be expected with the number of trials set to 2000.

Profiling timeout This determines the time after which the profiling for one configuration is killed. Since every tensor operator has a different computational intensity and performance varies across types of hardware, this timeout needs to be adjusted accordingly. A high profiling timeout will allow longer execution, which drives up total autotuning completion time and might not yield better results since long-running implementations are not good and can safely be killed. A low profiling timeout might also kill of good implementations. It should be noted that the optimal timeout does not depend on the actual execution time, since profiling runs the implementation multiple times and might even dynamically adjust the number of executions. In practice, a low timeout should be set first. If the log shows too many timeout errors, the timeout can be increased. 5 seconds seems to be a good value for GPU target devices, while 20 seconds or more are appropriate for CPU autotuning.

Batch size This determines how many configurations are selected and built in parallel for every autotuning iteration. This can speed up autotuning considerably, especially if a

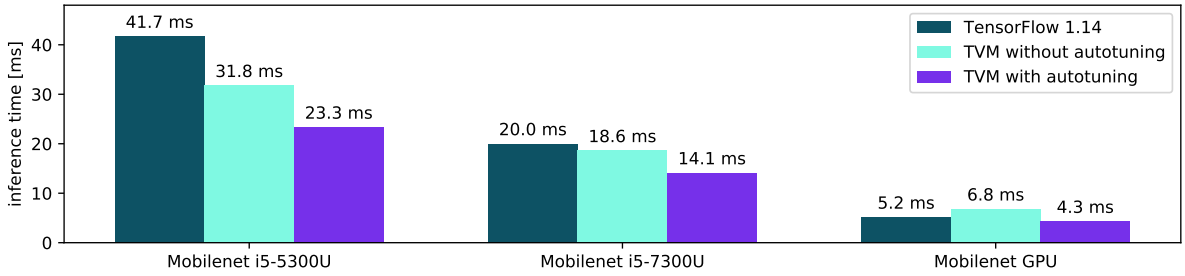


Figure 7: Inference performance with TensorFlow and TVM

large number of CPU cores are available on the client to run many compiler processes in parallel. The number of cores is also the default value. For larges batches, the model is updated and queried less frequently, but in general there seems to be no detrimental effect of having a high batch size. It should be notes that this is not the same as the batch size of the model, which would change the shape of the tensor operators.

Transfer learning This determines whether or not transfer learning is used between jobs. According to this setting, the data from the global autotuning database might be used to train the cost model at the start of each task. Between tasks, there is always transfer learning. Usually, transfer learning should be enabled for the most optimal inference performance results. However, we disable transfer learning for experiments to guarantee a fair comparison between earlier and later ones.

3.3 Capabilities

Using SimpleTVM and our knowledge about proper parameter settings as foundation, we evaluated how TVM performs in comparison to state-of-the-art manual tensor operator libraries. We use TensorFlow 1.14 as baseline since it is a popular framework for DL applications. cuDNN is enabled for GPU. Autotuning with TVM was executed with 2000 trials, so the numbers should represent the optimal implementation. For evaluation, we test a Mobilenet with a batch size of on two mobile-grade CPUs (Intel Core i5-5300U and i5-7300U) and a server-grade GPU (NVIDIA Tesla K80). Additionally, we test a heavier ResNet-18 on the GPU.

Inference improvement vs default tvm and TF Good in tradeoff with autotuning Use numbers from paper and own numbers

3.4 Limitations

TVM suffers from some fundamental restrictions, which cannot be changed in the current design.

3.4.1 Resource Utilization

We noticed lots of resource idle time due to synchronous design Show figure from poster Want to minimize idle time because edge resources are limited (define edge) Due to dependencies of stages, cannot be changed for a single job

3.4.2 Scalability

Our goal is to enable large-scale autotuning for our AaaS, autotune multiple models at the same time

objectives: Be able to run an arbitrary number of autotuning jobs while 1. maximizing inference performance: ultimate goal of autotuning 2. minimize hardware requirements: save cost 3. minimizing autotuning time: make autotuning worth the effort in order of priority State that autotuning time is not as crucial since it is rendered negligible by a large amount of inferences

With default tvm, there are two possible setups Include figure with two setups Include table with three experiments here

1. two completely separate autotuning jobs running independently on additional dedicated servers, one autotuning runner per server Pros: good autotuning and inference time, because they don't affect each other Cons: Costly because we need multiple sets of the same hardware, bad hardware utilization not an economically feasible approach. We cannot simply use machines from a PaaS provider since actual target device needs to be used Alternatively, we could use the same server and run them in sequence, trading off hardware required (halved) for autotuning time(doubled)

2. two autotuning runners sharing the same server Pros: only one set of hardware Cons: - interference drives up autotuning time Explain interference Autotuning takes long (in our tests anywhere between 3 and 36 hours, depending on hardware and network size) Especially update model takes 64% longer when two jobs are running simultaneously, very CPU intensive (50-70%) - results in worse inference performance because profiling is distorted (show numbers), as we saw most important

In both setups, we do not meet all objectives Gets worse the more jobs we add AaaS is not possible efficiently with current implementation and architecture of autotuning in TVM, does not scale well

Ideally: Prevent interference, because it affects autotuning time and inference performance Minimize hardware required by utilizing available hardware fully before adding new servers for cost reasons

However, there does not seem to be any solution yet

3.4.3 Similar Problems

In general, problem can be formulated as follows: How can resources be shared optimally between multiple tasks that are partially idle?

Add two examples

4 Autotuning Scheduler

Enabling controlled parallel autotuning is necessary to solve those problems necessitates central scheduler that orchestrates all jobs

4.1 Design

general idea: (1) Share computation resources to minimize idle time by interleaving stages -> use idle time of one job to execute another job. Allows us to save on hardware, since we maximize resource utilization (2) make sure to keep dependencies and prevent interference, postpone execution of some stages until resource is free -> ideal solution from previous chapter include figure from poster

since only proof-of-concept, very specific to make it work quickly and non-flexible/fault-tolerant Leverage SimpleTVM

4.1.1 Scheduling Algorithm

to keep scheduler algorithm simple, we designed it to be agnostic of stages scheduler needs to know - knows which job will use which resource - knows which resource is currently available we call this load-aware theoretically, could work for any application that supports this interface (e.g. TC?)

allows for variable strategies to compare different designs show scheduling pseudocode

4.1.2 Autotuning Decomposition

Necessary step before implementation Show figure Default TVM: Procedure is monolithic Start runner and loop does not stop until its finished We want to be able to control the execution of individual stages

Decompose monolith into separate units for stages This allows us to control when which stage is being executed Necessary for scheduler Runner does not do anything on its own but waits for commands

4.2 Implementation

Figure with autotuning procedure with scheduler Since TVM only provides a python interface, we are using python 3.5

4.2.1 RPC

We want clients to live in different processes, docker containers, possibly physical servers (why?) requires RPC infrastructure consisting of scheduler and clients different from TVM RPC infrastructure clients register to scheduler describe endpoints

4.2.2 Components

Show whole stack, denote what happens in scheduler, what happens in runner Show which communication is in-process and which is RPC JobManager negotiates between autotuning stages interface and simple scheduler interface, keeps track at position in autotuning show abstract scheduler and client interface

4.2.3 Challenges

initially wanted to run scheduler and clients in one multi-threaded process without RPC to get results quickly not possible due to python global interpreter lock

evaluation of design choices takes long because autotuning is a slow process, created MockJob for debugging of scheduler

4.3 Autotuning as a Service

imagine autotuning as a service where users can submit their trained model and receive an optimized version according to SLA Describe as a service More sophisticated scheduler, requires moving more autotuning logic from client to scheduler Make client stateless

Keep trained model and update it every n new entries to skip transfer learning time for every task Check currently known best configurations and see if SLA is already met before actually starting autotuning Automatically set up autotuning infrastructure Split jobs on task and search space level to parallelize more - make better use of unused resources - faster autotuning, e.g. for paying customers

5 Evaluation

evaluation environment: 125 GB RAM Intel Xeon E5-2650 v3, 2.30 GhZ with avx2 instructions 4x Tesla K80 GPU

Python 3.5 on Ubuntu 16.04

5.1 Results

Comparison of interleaved design vs synchronous and sequential in terms of autotuning time and inference time hardware and network specifications

Evaluation only with limited set of hardware and models, general statement requires more experiments

compare with thesis from introduction

5.2 Limitations

Very rudimentary scheduler Predictive scheduler using times for task to make scheduling more intelligent Requires more control in scheduler, not only simplified interface Add Knows which job is in which stage and how long is each stage estimated to take to load-awareness running update model and build of one job directly after another will probably decrease waiting time, since that job can then already use the target device, so there is less target device idle time Believe that more and heterogenous jobs that vary significantly in complexity will enable better resource utilization and less wait time, given a more intelligent scheduler

6 Conclusion

Describe results Only used scheduler for TVM, but should work for TC as well because it also has stage dependencies Enabled large-scale autotuning with only small sacrifices in autotuning time, thesis holds for our limited set of tests

6.1 Future Work

More intelligent scheduler algorithm Get rid of tracker and let scheduler assign servers

After best approach is found from prototype, make into mature product to enable real-time DL applications for everybody

Bibliography

[1] Ian Goodfellow, Yoshua Bengio, and Aaron Courville, *Deep Learning*. MIT Press, 2016.

[2] Lyve Data Labs, "Seagate edge RX: A smart manufacturing reference architecture solution," 2019.

[3] Seagate, "Smart manufacturing moves from autonomous to intelligent: Inside project athena: Seagate's internal AI edge platform," 2019.

[4] Y. Liu, Y. Wang, R. Yu, M. Li, V. Sharma, and Y. Wang, "Optimizing CNN model inference on cpus," 2019.

[5] T. Chen, T. Moreau, Z. Jiang, L. Zheng, E. Yan, M. Cowan, H. Shen, L. Wang, Y. Hu, L. Ceze, C. Guestrin, and A. Krishnamurthy, *TVM: An automated end-to-end optimizing compiler for deep learning*, Feb. 12, 2018.

[6] Y. Hu, *Optimize deep learning GPU operators with TVM: A depthwise convolution example*, 2017.

[7] C. Szegedy, Wei Liu, Yangqing Jia, Pierre Sermanet, Scott Reed, Dragomir Anguelov, Dumitru Erhan, Vincent Vanhoucke, and Andrew Rabinovich, "Going deeper with convolutions," 2015.

[8] K. He, X. Zhang, S. Ren, and J. Sun, *Deep residual learning for image recognition*,

[9] N. Vasilache, O. Zinenko, T. Theodoridis, P. Goyal, Z. DeVito, W. S. Moses, S. Verdoolaege, A. Adams, and A. Cohen, *Tensor comprehensions: Framework-agnostic high-performance machine learning abstractions*, Feb. 13, 2018.

[10] J. Roesch, S. Lyubomirsky, L. Weber, J. Pollock, M. Kirisame, T. Chen, and Z. Tatlock, "Relay: A new IR for machine learning frameworks," in *Proceedings of the 2nd ACM SIGPLAN International Workshop on Machine Learning and Programming Languages*, ser. MAPL 2018, New York, NY, USA: Association for Computing Machinery, 2018, pp. 58–68, ISBN: 9781450358347.

[11] T. Chen, L. Zheng, E. Yan, Z. Jiang, T. Moreau, L. Ceze, C. Guestrin, and A. Krishnamurthy, *Learning to optimize tensor programs*, May 21, 2018.

Glossary

target device

the device that inference will be performed on; usually an accelerator located on the edge