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.
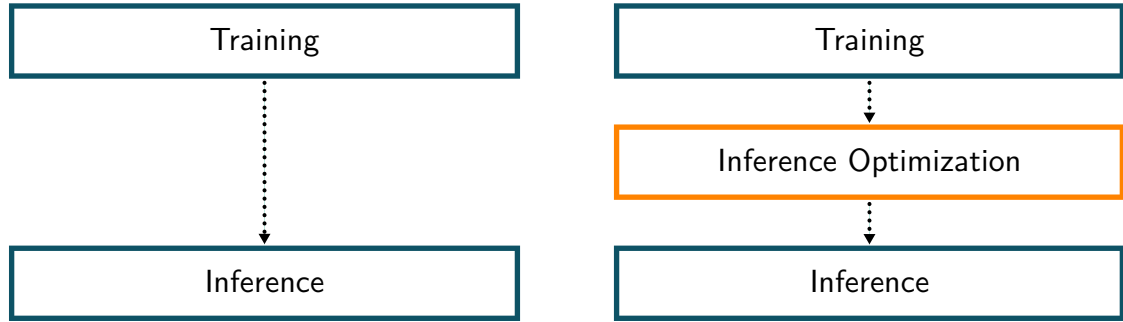
2.2 Inference Optimization

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 $0.98\times$ to $3.5\times$ on CPU [4, p. 9] and $1.6\times$ to $3.8\times$ 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)
```

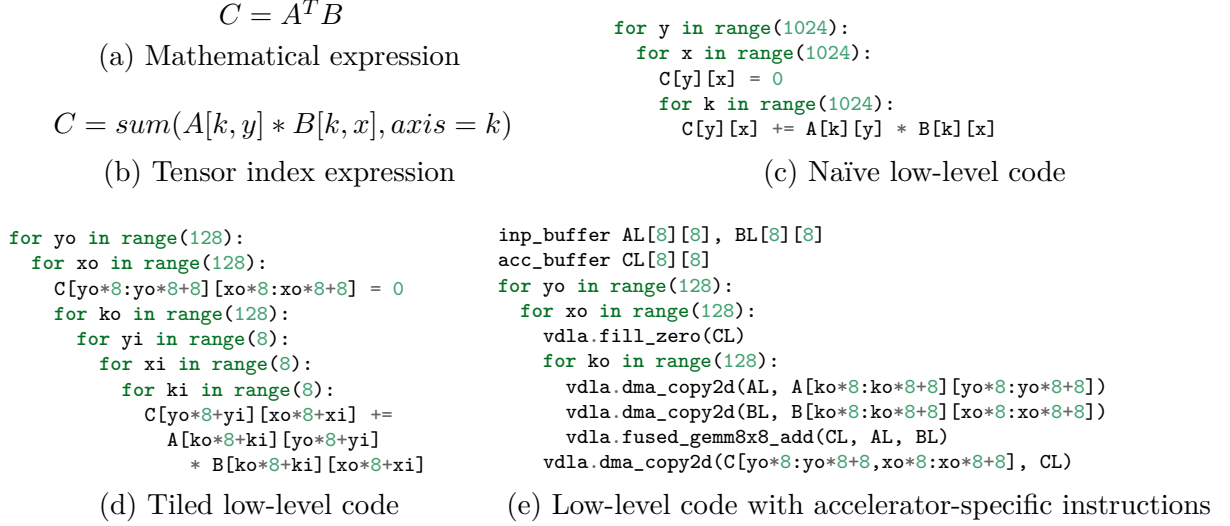(e) Low-level code with accelerator-specific instructions

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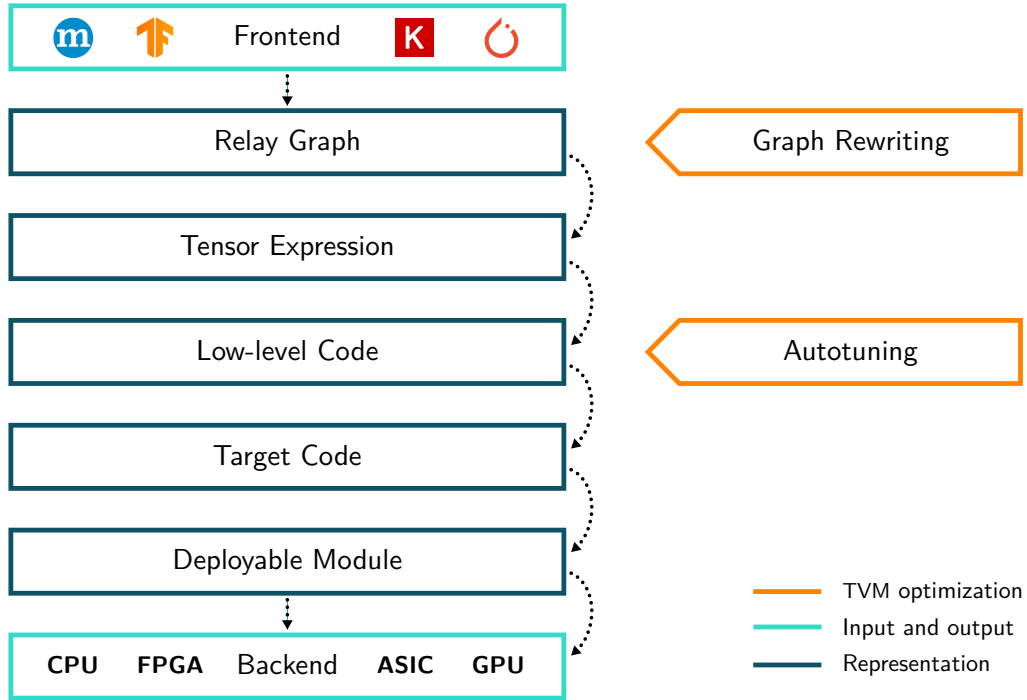[2] `https://github.com/dmlc/tvm/`

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 execution of the autotuning process for one model a *job*. The component that executes the autotuning logic for one job the *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. 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.[3]

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. In case the target hardware is different from the client hardware, cross-compilation is necessary. 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. This is only done after a sufficient number of new profiling data has been collected, so this stage might be skipped in some iterations.

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

---

[3]In TVM's implementation, the selection of the next candidates is actually performed at the end of the iteration after updating the model, and the first stage just picks a batch from the already selected configurations. However, it is more logically simpler to think of the candidate selection using simulated annealing as part of the first stage. Furthermore, it is not wrong due to the iterative nature of autotuning.
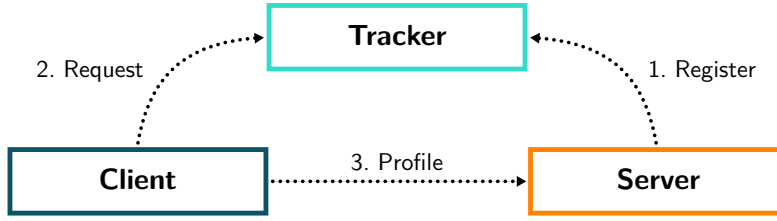
Figure 5: TVM's RPC architecture

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 capa-
bilities 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
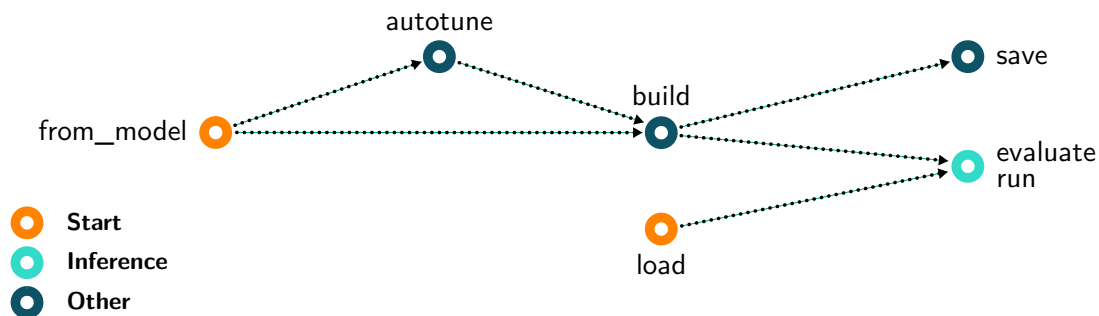exactly this is done differs even inside the same framework. SimpleTVM provides

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 if it runs too long. 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

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 one on two mobile-grade CPUs (Intel Core i5-5300U and i5-7300U), a server-grade CPU (Intel Xeon E5-2650 v3) and a high-end GPU (NVIDIA Tesla K80). The same two images were used as model input in all cases.
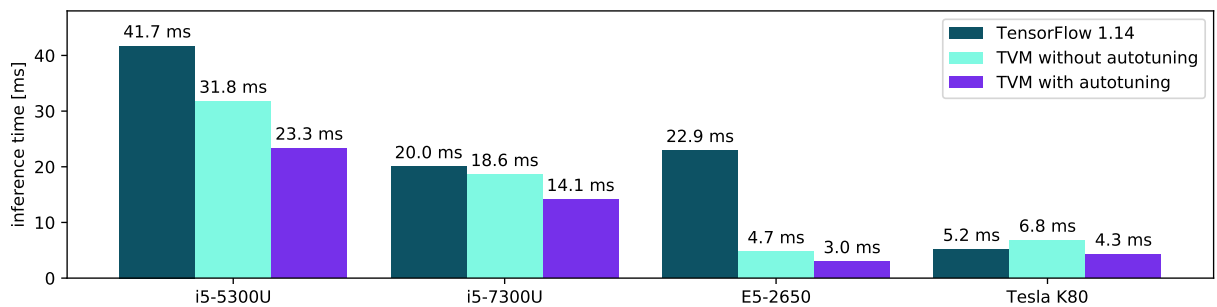

Figure 7: Inference performance with TensorFlow and TVM

In all cases does TVM with autotuning show performance improvements over TensorFlow. Especially on CPU, inference speed improves by 44% (18.4 ms) for the i5-5300U and by 30% (5.9 ms) for the i5-7300U. But also for the GPU, the autotuned version is 17% (0.9 ms) faster, albeit measurement inaccuracies are possible on this small time scale. Nonetheless, even a similar performance is impressive considering that no human expert

knowledge was required and autotuning took less than 6 h for CPU and less than 12 h for GPU (due to the much larger search space). Further results for a wider variety of devices and models, including recurrent neural networks, are provided in [11] and show similar improvements.Add analysis for Xeon

TVM performs better than TensorFlow on CPU even without autotuning. Graph-level optimizations alone are enough to result in faster inference by 24% (9.9 ms) for the i5-5300U and by 7% (1.4 ms) for the i5-7300U. Since only a series of pre-defined transformation passes are applied to the Relay program, graph-level optimization is performed in a matter of seconds. However, non-autotuned TVM cannot keep up with TensorFlow on GPU, it is 31% (1.6 ms) slower.

These results show that TVM on par with manual optimization, at least for our limited evaluation scenarios. Furthermore, TVM is under active development and can be expected to show further performance improvements in the future.

3.4 Limitations

While the autotuning results are promising, we found that autotuning process suffers from some fundamental restrictions inherent to the current design which limit its efficiency. For all real measurements in this section, we evaluated autotuning of a ResNet-18 (12 convolutional layers, 1 fully-connected layer) with 2000 iterations per task on two machines with two Intel Xeon E5-2650 v3 CPUs and four Tesla K80 GPUs, one machine for client and target device each.

3.4.1 Resource Utilization

Since stages in autotuning depend on results of the previous stages (configurations are required for building, executables are necessary for profiling, time measurements are used to update the model), they need to be executed in sequence. Because profiling runs on the target device, the result is a lot of idle time on both the client and the target device. This sub-optimal resource[1] utilization is exemplified in Figure 8.

Measurements with our test setup showed that, for a total autotuning completion time of 14.5 h, the client is idle for 6.6 h (45%) and the target device for 7.9 h (55%). Since computation resources, especially DL accelerators, are rather costly, we want to minimize resource idle time. If existing resources are utilized better, it might not be necessary to

---

[1]We define *resource* as a machine that executes some stage in the autotuning process, e.g., the target device or the hardware that the client runs on.

Figure 8: Resource utilization during autotuning

acquire new hardware. If the individual occurences of idle time are long enough, it might be possible to use it for other computations in between. Indeed, the mean execution time for each stage is as follows: 8.3 s for building, 39.5 s for profiling, and 44.0 s for updating the model and selecting the next batch. This is long enough to reasonably assume that resource utilization can be improved by sharing the device. However, interference must be prevented.

3.4.2 Scalability

In preparation for enabling autotuning on a larger-scale, we need to examine the scalability of the current design. Scalability in this section refers to the ability to run an arbitrary number of autotuning jobs at the same time without sacrificing efficiency and result quality. We define objectives, that a scalable solution should satisfy:

1. High inference performance, since it is the ultimate goal of autotuning

2. Low amount of required hardware, since additional devices are costly

3. Low autotuning time, since autotuning takes long

These objectives are listed in order of priority. Good inference performance is the primary objective. It obviates the need to buy new hardware. Furthermore, there usually is a large amount of inferences which makes a longer autotuning time negligible over time. Rapid autotuning is nonetheless desirable.

We compare two setups for scaling that are possible using only the components that TVM comes with by default, schematically depicted in Figure 9. For the sake of simplicity, we only show depict two jobs, but this generalizes to any higher number. The evaluation of both setups with regard to the previously defined objectives is summarized in Table 1.

Dedicated resources In the first setup (Figure 9a), each job is run on its own set of resources. This means they are completely independent and do not affect each other. Both the resulting inference performance and the autotuning time are as good as

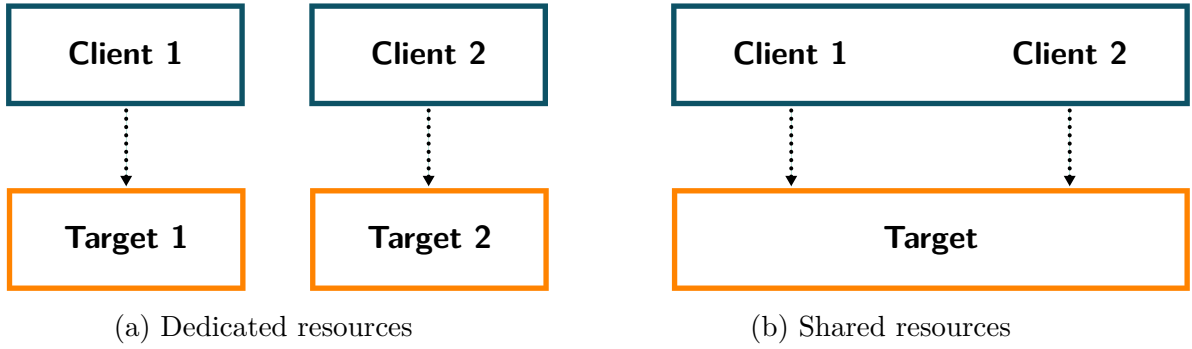(a) Dedicated resources   (b) Shared resources

Figure 9: Setups for scaling autotuning

possible, since they are equal to single-job autotuning. However, additional resources need to be acquired for every new job, which is not an economically feasible approach on a larger scale. Adding resources on demand from a cloud computing platform such as Amazon EC2 or Azure VM might work for the client machine, but since the actual target device needs to be used for profiling, which is likely to not be available on those cloud platforms, this is not a satisfactory solution. Alternatively to having separate sets of machines, the same machiens could be used to run the jobs in sequence. This trades off the amount of hardware that is required for autotuning time. With a higher number of jobs, this will take too long.

Shared resources In the second setup (Figure 9b), all jobs run in parallel on the same resource. To share the target device, we launch one RPC server for each client per GPU in our test setup. Due to the resource sharing, only one set of machines is required, which is good in terms of cost. However, this is very probable to lead to interference, when multiple jobs execute a stage on the same resource simultaneously. Interference has a detrimental effects on both inference performance and autotuning time. Interference on the client will slow down compute-intensive stages like building or model updates (50%–70% CPU usage). Modern CPUs are able to parallelize using multiple cores, but because building and model training uses all cores, time-sharing between multiple processes needs to be employed by the kernel. Process idle times and context switching overhead increase the autotuning time considerably. Assigning dedicated CPU cores per job will also make the process slower due to decreased resources per job and will not work when scaling up. Even more critical is interference on the target device. This distorts the profiling results, so good implementations might seem to perform bad if another job profiles on the same target device in parallel. Deceiving measurements will also lead to an inaccurate cost model. Since good inference performance is the prime objective of large-scale autotuning, this setup is also not satisfactory.

Setup	Hardware required	Inference performance	Autotuning time
Dedicated resources	2x	High	Low
Shared resources	1x	Low	High
Optimum	1x	Low	Low

Table 1: Evaluation of setups regarding scalability objectives

Both setups do not meet all objectives we set for large-scale autotuning. Especially when scaling up to more parallel jobs, efficiency deteriorates significantly. We can conclude that the current implementation and architecture of autotuning in TVM does not scale well.

Having regarded possible scaling approaches and the reasons for their benefits and shortcomings, we can formulate two features for an optimal solution to satisfy the objectives:

- Resources must be shared and utilized fully before adding new servers to minimize the required hardware for cost saving

- Interference must be prevented to guarantee a high inference performance and low autotuning time

3.4.3 Similar Problems

Our literature review was not successful in finding a solution to scale up autotuning. However, we can generalize the problem statement; we are looking for a solution to share available resources optimally between multiple tasks that are partially idle due to some dependency. From this point of view, we find two papers solving a similar problem.

[12] increases parallelism of a hierarchy of tasks and subtasks on multiprocessor platforms. Subtasks have control and data dependencies, but tasks are independent of each other. They employ a mix of exact and heuristic scheduling algorithms at design-time. Their scheduler interleaves sub-tasks of different tasks while respecting the dependencies between the subtasks of a single task. The result is a 37% shorter execution time with increased resource utilization. The hierarchical structure of tasks and subtasks is similar to jobs and stages in autotuning.

[13] enable sharing of GPU cores by multiple kernels. In current GPU architectures, concurrently launched kernels use separate cores. However, interleaving of code from multiple kernels on the same core allows them to minimize core idle time introduced by memory latency. They increase the throughput of benchmarking applications by 7%.

[12] and [13] improve parallelism for multiprocessing units while we want to improve parallelism for distributed machines. Nonetheless is the interleaving approach relevant.

4 Autotuning Scheduler

Interleaving of the stages of multiple jobs is our key concept for enabling large-scale autotuning. [12] uses a design-time scheduler to create a program with good concurrency. We need to dynamically schedule incoming jobs, so the schedule cannot be predetermined. We need an additional component in the autotuning architecture that orchestrates running jobs. In this chapter, we describe the design and implementation of our central scheduler that controls stage execution.

4.1 Design

Our scheduler possesses the two features that have been determined to be imperative for optimal large-scale autotuning:

- Computation resources are shared between jobs. This facilitates good resource utilization since the idle time of one job can be leveraged to execute another job. This is called *interleaving* and saves hardware and costs as a result. However, stage dependencies of a single job must be maintained.

- Interference between jobs is prevented. This guarantees that inference performance and autotuning time are as good as possible. The scheduler needs to check if the resource that will be used by the next stage is free before execution. This might necessitate the postponing of stage executions if the stage is ready before the resource becomes free.

These two features not only make it match the optimal solution, but also do they solve the problem of bad resource utilization of single-job autotuning by leveraging that shortcoming.

In case multiple jobs are ready, the scheduler needs to decide which one to run. The simplest approach is to use a round-robin algorithm, which iterates over the jobs in the order they were started and picks the first one that is ready. More sophisticated approaches might apply some logic to decide on a job which would maximize resource utilization but keep the average autotuning time low. However, we choose the round-robin algorithm

for our first version. It is easy to implement and works reasonably well for an arbitrary number of jobs, which allows us to proof our concept.
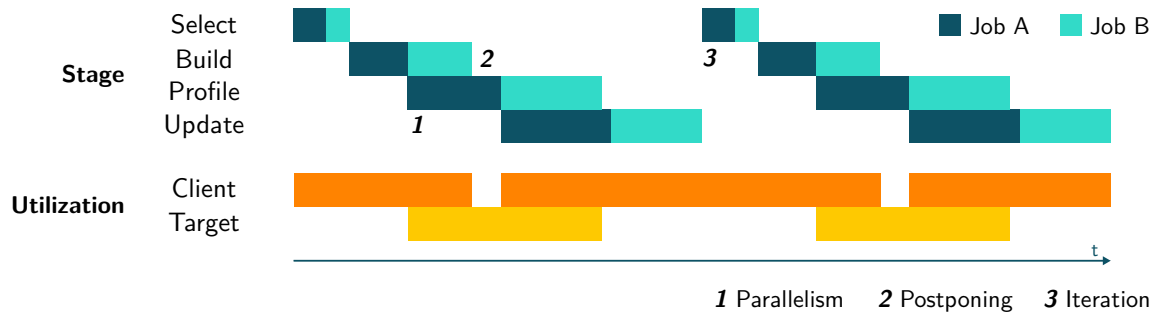

Figure 10: Interleaving of multiple autotuning jobs

Figure 10 illustrates round-robin-based interleaving with an example of two jobs. Significant events are marked with numbers. Job A and Job B are started at the same time, but assume that the scheduler knows first about A. The first stage of A is executed, then the first stage of B. Once B finishes, the round-robin algorithm decides it is A's turn again and executes its second stage. However, a more sophisticated algorithm might have decided that it is better to let B execute first. Once A finishes the seconds stage, the client machine is free and B can execute the second stage. At the same time, A is ready to execute the third stage which will run on the target device. Since the target device is not in use, A can execute the profiling there in parallel to B's building since they use separate resources (*1*). Building does not take as long as profiling, so A is ready to profile before B finishes its profiling stage. Therefore, A's third stage is postponed until the target device is free (*2*). A's profiling and B's update model can, once again, execute simultaneously since they use distinct resources. After one iteration of all four stages, the process starts anew with the first stage (*3*). This continues until both jobs are done. In a real scenario, new jobs might appear while other jobs are already running. The round-robin scheduler simply adds them to its list of jobs and includes them in the interleaving. Note how the resource utilization in Figure 10 is much improved over the single-job autotuning in Figure 8 due to overlapping and sharing. Especially on the client device, utilization has almost been maximized since three of the four stages use the client.check if is consistent with new scheduler algorithm

4.1.1 Autotuning Decomposition

The default autotuning process is monolithic and can be regarded as a blackbox from the outside (Figure 11a). This means, the autotuning loop can be started, and it does not finish until the whole job is completed. Once a stage finishes, the next one is executed immediately. However, the scheduler needs to be able to control the execution of the

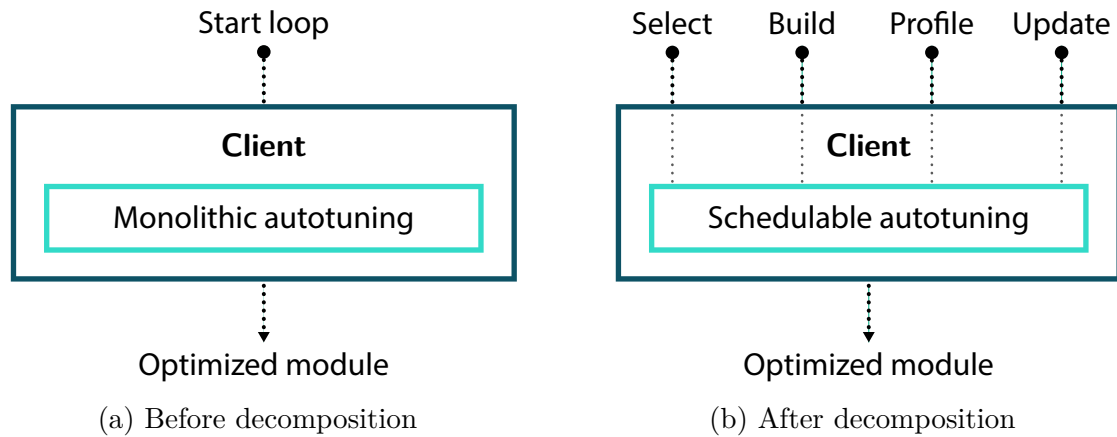(a) Before decomposition   (b) After decomposition

Figure 11: Client interface before and after decomposition

individual stages because it needs to prevent interference by means of delayed execution. This necessitates the decomposition of the autotuning process into schedulable units, corresponding to the stages (Figure 11b). The client does not execute any of the stages on its own. Rather, it provides an interface to execute schedulable units and waits for an external trigger to do so. The autotuning loop can now run in another component, such as the scheduler.

4.1.2 Scheduling Algorithm

To fulfill the task of interleaving while preventing interference, the scheduler needs to know four pieces of information for every job:

- Whether the current stage is done; lets scheduler know if the job is ready for next stage

- Whether the whole job is done; lets scheduler know if the job needs to be considered in the future

- Which resource the current stage runs on; lets scheduler know which resources are free and which are in use

- Which resource the next stage will run on; lets scheduler postpone stage execution to prevent interference

We call this *load-awareness*; being aware of the state of jobs and resources as well as their relationships. Theoretically, this allows the scheduler to work not only with TVM's autotuning but any process that can provide this information. Designing our scheduler to be agnostic of the underlying process also simplifies the algorithm.

```
1 queues = {r: [] for r in resources}
2 while True:
3     # Phase 1: Round-robin scheduling
4     for job in jobs:
5         if job.is_stage_done():
6             mark_as_free(job.previous_resource)
7             if job.is_complete():
8                 jobs.remove(job)
9             else:
10                queues[job.next_resource].enqueue(job.next_stage)
11    # Phase 2: Execution
12    for resource, queue in queues:
13        if is_free(resource) and len(queue) > 0:
14            stage = queue.dequeue()
15            stage.execute()
16            mark_as_busy(resource)
```

Listing 2: Interleaved scheduling pseudocode

We present the algorithm for interleaved scheduling in Listing 2. Each job exposes an interface that provides the four pieces of information necessary for scheduling. The scheduler keeps a list of jobs in the order in which they were registered. Furthermore, it keeps a list of resources which are marked as free or busy. Each resource also has a queue which contains stages that are ready and will use that resource (Line 1). The algorithm is an infinite loop (Line 2), with each iteration consisting of two phases: scheduling and execution. In the first phase, the scheduler checks for each job if the current stage is done (Lines 4–5). Busy jobs are not regarded further. If the stage is done, the resource that was used is marked as free (Line 6). If the stage was the last stage in the job, the job can be removed from the list of jobs (Line 7–8). Otherwise, the job is ready to continue and the next stage is added to the queue of the resource that it will run on (Line 9–10). The iteration over all jobs in order is what effectively makes this round-robin scheduling. In the second phase, the scheduler iterates over each resource and the corresponding queue (Line 12). If the resource is free and there are pending stages for that resource, the first stage in the queue is dequeued and executed (Lines 13–15). The respective resource needs to be marked as busy (Line 16).

Additionally to interleaving, our scheduler supports two other strategies for executing multiple jobs, which will be used in the evaluation for comparison.

Sequential scheduling (Listing 3) works similar to single-job autotuning without a scheduler. Jobs do not run in parallel, but the next job is only started when the previous one finishes. This renders consideration of resource free/busy state unnecessary, and stages do not need

```python
1  while True:
2      job = jobs.next()
3      if not job:
4          continue
5
6      while not job.is_complete():
7          if job.is_stage_done():
8              job.next_stage.execute()
9      jobs.remove(job)
```

Listing 3: Sequential scheduling pseudocode

```python
1  while True:
2      current_jobs = jobs
3      if len(current_jobs) < 2:
4          continue
5
6      while not any([j.is_complete() for j in current_jobs]):
7          if all([j.is_stage_done() for j in current_jobs]):
8              [j.next_stage.execute() for j in current_jobs]
9      jobs.remove_all(current_jobs)
```

Listing 4: Synchronous scheduling pseudocode

to be postponed. However, since it is controlled by the scheduler, we can calculate the overhead introduced by adding a scheduler component.

Synchronous scheduling (Listing 4) forces parallel execution of the same stage of multiple jobs on the same resource, making it the exact opposite of interleaved scheduling. Postponed stage execution is applied here to guarantee full interference. We use this strategy to evaluate the worst case effect of interference. However, this only works for equal jobs, since there needs to be symmetry between stages of all jobs.

4.2 Implementation

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

since only proof-of-concept, very specific to make it work quickly and non-flexible/fault-tolerant Leverage SimpleTVM only supports one client server and target device server at the moment

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

overhead of scheduling

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, use ideas from [12] 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.

[12] Z. Ma, F. Catthoor, and J. Vounckx, "Hierarchical task scheduler for interleaving subtasks on heterogeneous multiprocessor platforms."

[13]   M. Awatramani, J. Zambreno, and D. Rover, "Increasing GPU throughput using kernel interleaved thread block scheduling," 2013.

Glossary

target device

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