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
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.

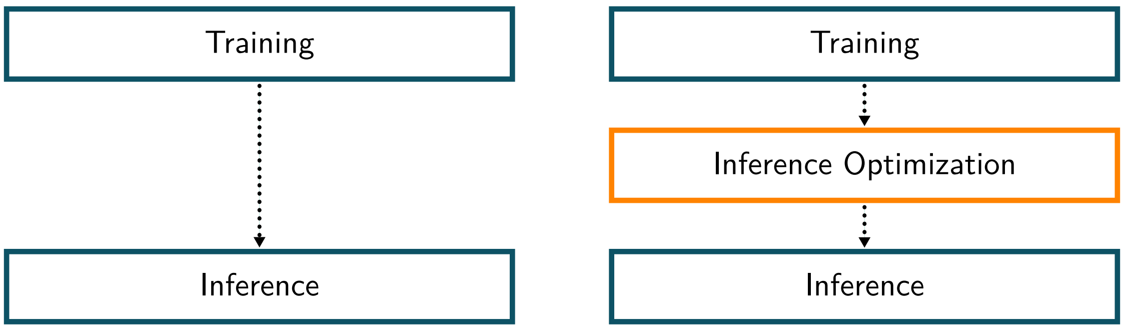
2.2 Inference Optimization

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 and functionally equivalent implementations in an iterative fashion. This *search space* is defined per-operator by the set of permutations of *knob* settings, which result in different implementations when being applied to a template. These knobs control, for example, loop unrolling, loop order, loop tiling and threading and can usually be adjusted in steps of powers of 2 [5, p. 5] [9, p. 16]. One specific combination of knob settings, i.e. one element of the search space, is called *configuration*. Defining the knobs is done manually for each class of target devices, but the

$$C = A^T B$$

(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 schedule

```
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 schedule

```
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) Schedule with accelerator-specific instructions

Figure 2: Expressions and schedules for transposed matrix multiplication [5, p. 4]

search is guided by an algorithm that proposes a batch of candidate configurations. This is necessary since the size of the search space makes the brute-force approach of trying all configurations infeasible. For 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 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 the configuration to the operator's template, which lowers the tensor expression to a *schedule*. A simple but naïve reference schedule can be used to check the correctness of more complex schedules (2c). The schedule 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 schedule 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 a schedule [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 DL compiler stack. It can import whole models from a frontend framework and build

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

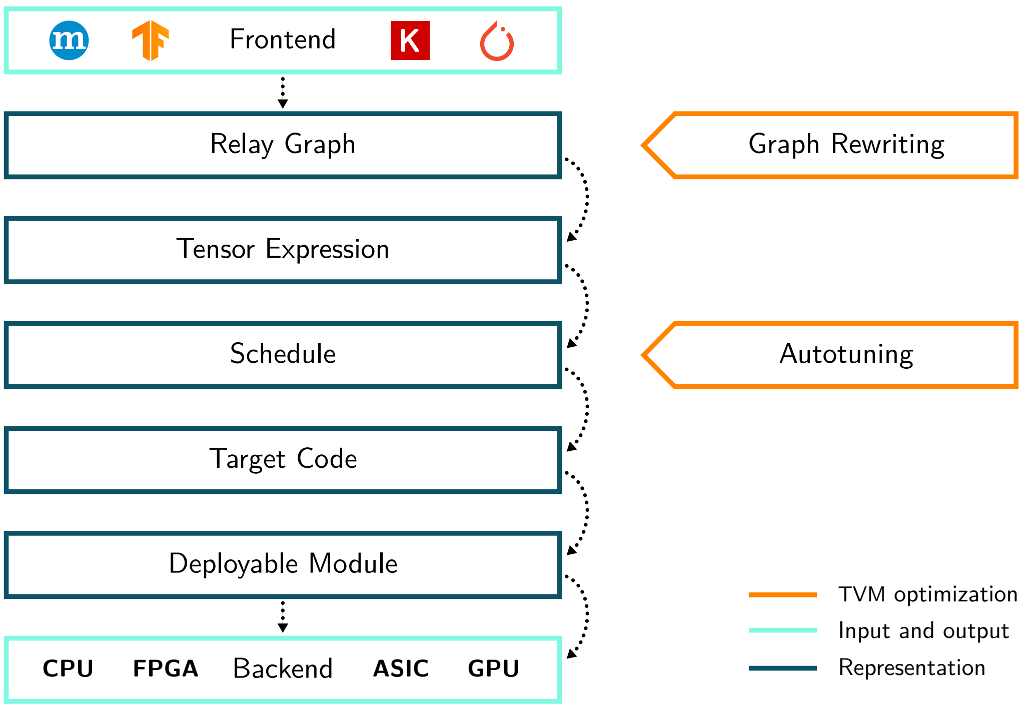


Figure 3: Levels of abstractions in TVM flow

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, which are expressive enough to describe convolutions and other matrix and vector operations. TVM comes with tensor expressions for common computations used in DL such as various activation functions, pooling, and linear transformation.

Lowering to schedule, then target code calls target-specific compiler

autotuning using machine learning

define autotuning job, task first extraction of tasks schedules as abstraction with knobs details of autotuning process Profiling repeated multiple times Schedule intermediate representation RPC allows autotuning logic to run on powerful server, but profiling to happen on target device with figure transfer learning using invariant representation Describe saved modules

In this project, we use TVM because of the novel, machine learning-based approach Using (commit id) with a few modifications to support measurements (check what else we changed)

3 Using TVM

We want to explore capabilities and limitations of TVM Want to be able to quickly test different scenarios (models, configurations, hardware)

3.1 SimpleTVM

We created a simpler interface for TVM, called SimpleTVM Using TVM follows the same workflow every time created wrapper for simpler usage of TVM expose easy, chainable interface Makes it easy for researchers who are new to TVM to get started

Created automated benchmarking framework superb enable automated testing of different configurations to be able to run multiple configurations without human intervention

Docker container to be able to easily deploy TVM with all dependencies on any server

3.2 Parameters

Autotuning with TVM has a plethora of parameters that can affect both the autotuning process itself and the result. Setting these parameters properly requires knowledge of how TVM works as well as the hardware.

List of most important parameters

Number of trials Number of iterations, tradeoff between autotune time vs inference performance, converges

Profiling timeout attune to target device

Batch size how many configurations are selected and built, usually number of cores, also default. not same as model batch size

Transfer learning between tasks always, but between jobs? For experiments

3.3 Capabilities

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.

Glossary

target device

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