Hewlett Packard Enterprise

DHBW
Duale Hochschule
Baden-Württemberg

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

Project Thesis II / T2000

for the study program
Computer Science

at the
Baden-Wuerttemberg Cooperative State University Stuttgart

by
Dominik Stiller

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

Declaration of Authorship

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

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

Sindelfingen	September 3rd, 2019	
Place	Date	Dominik Stiller

Abstract

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

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

Contents

Acronyms

ANN artificial neural network

CNN convolutional neural network

GPU graphics processing unit

ML machine learning

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 feeds sample data to the model 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. 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]. CNNs apply convolution and pooling to a region of the input tensor in a sliding fashion, so entries only interact with other entries in their neighborhood. Convolution applies one or more kernels 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 and padding. CNNs are an important tool in state-of-the-art computer vision applications.

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 performing the layer's computation, and finally an output node. The edges describes how data flows between the vertices.

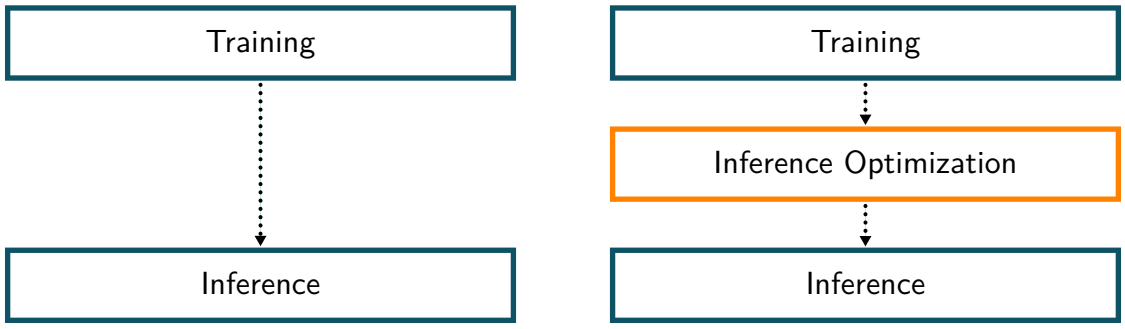
2.2 Inference Optimization

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 an important field. Reduction of the inference time has a number of advantages:

- less hardware is required to achieve the same inference rate
- higher inference rate can be achieved with same hardware
- real-time applications are made possible, 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 optimization, they could save over 4 h 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), tensor processing units or field-programmable gate arrays are used to speed up both training and inference. Every device has different features such as specialized instructions, memory size and layout, cache access, and parallelization support. However, generic models cannot make full use of accelerator capabilities and fall short of leveraging the full potential. Consequently, 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 the traditional machine workflow, the trained model is deployed as-is(Figure 1a). Inference optimization adds an additional step, turning the trained model into a functionally equivalent but



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

Figure 1: Machine learning workflow

optimized version for inference (Figure 1b). In this step, we first apply high-level transformations that rewrite the computation graph by 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.

Convolution operations are computationally very intensive and make up the majority of state-of-the-art CNNs, such as Inception[6] and ResNet[7]. 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.

The model determines what should be calculated, but it does not specify how it is calculated. The actual implementation offers lots of optimization potential. There is always a generic default implementation which is the straightforward way of performing the calculation. However, it does not consider shared memory between threads or cache access patterns, which can have a significant adverse effect on performance [8]. 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 tiling sizes, loop unrolling factors and thread numbers, any of which could be the best one but is very much specific to the target device [5, p. 2]. Finding the optimal such combination is the goal of inference performance optimization.

2.3 Manual Optimization

state of the art cuDNN and TensorRT and Intel MKL, taken as baseline requires deep knowledge of target device, usually provided by vendor limitations - no support for new de-

vices - no support for unconventional shapes - no support for new graph-level optimizations elaborate limitations high-level optimization need to wait until vendor provides low-level support

2.4 Automated Optimization

vendor-agnostic and does not require expert knowledge Enables innovation by enabling high-level optimization and fostering experimentation with unconventional layers, not supported by manual frameworks describe autotuning process on high level definition of search space (loop unrolling, tiling, threads) Problem: search space is very large (billions), and any one of them could be the best one for one target device

impossible to try all autotuning frameworks have some solution to explore search space rapidly look at TVM and TC

has same or even better performance than hand-optimized libraries show numbers

There are two frameworks that implement autotuning

2.4.1 TensorComprehensions

does not use machine learning

2.4.2 TVM

using machine learning TVM is framework that proposed and implements autotuning

import from many frontends, compilation for many backends has own graph-level and tensor operator-level representation calls target-specific compiler

define autotuning job, task first extraction of tasks schedules as abstraction with knobs details of autotuning process Profiling repeated multiple times RPC allows autotuning logic to run on powerful server, but profiling to happen on target device with figure

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. (visited on Sep. 1, 2019).

[3] Seagate, "Smart manufacturing moves from autonomous to intelligent: Inside project athena: Seagate's internal ai edge platform," 2019. [Online]. Available: %5Curl%7Bhttps://www.seagate.com/www-content/enterprise-storage/it-4-0/images/cs595-1-1901-seagate-athena.pdf%7D (visited on Sep. 1, 2019).

[4] Y. Liu, Y. Wang, R. Yu, M. Li, V. Sharma, and Y. Wang, "Optimizing cnn model inference on cpus," 2019. [Online]. Available: %5Curl%7Bhttp://arxiv.org/pdf/1809.02697v3%7D.

[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. [Online]. Available: %5Curl%7Bhttps://arxiv.org/pdf/1802.04799.pdf%7D (visited on Jun. 5, 2019).

[6] C. Szegedy, Wei Liu, Yangqing Jia, Pierre Sermanet, Scott Reed, Dragomir Anguelov, Dumitru Erhan, Vincent Vanhoucke, and Andrew Rabinovich, "Going deeper with convolutions," 2015. [Online]. Available: %5Curl%7Bhttps://www.cv-foundation.org/openaccess/content_cvpr_2015/papers/Szegedy_Going_Deeper_With_2015_CVPR_paper.pdf%7D (visited on).

[7] K. He, X. Zhang, S. Ren, and J. Sun, *Deep residual learning for image recognition*, [Online]. Available: %5Curl%7Bhttps://arxiv.org/pdf/1512.03385.pdf%7D (visited on).

[8] Y. Hu, *Optimize deep learning gpu operators with tvm: A depthwise convolution example*, 2017. [Online]. Available: %5Curl%7Bhttps://tvm.ai/2017/08/22/Optimize-Deep-Learning-GPU-Operators-with-TVM-A-Depthwise-Convolution-Example%7D.

Glossary

target device

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