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

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 Deep Learning

Machine learning

Consists of training and inference

2.1 Artificial Neural Networks

general structure with layers

layers: fully-connected, batch norm

basically a function mapping from input (image, text, data) to result (e.g. classification)

trend towards more layers to create more complex function which are more powerful

many layers: deep neural network

2.2 Convolutional Neural Networks

more layers: pooling and convolutions

important for computer vision

2.3 Computation Graph Representation

In its essence, machine learning is a series of calculations, mostly on matrices and vectors

While neural network models 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 (convolutions, pooling) and finally an output node.

The edges describes how data flows between the vertices.

3 Inference Optimization

The traditional machine learning workflow consists of training and inference.

The number of inferences heavily outweighs the number of trainings, since an unlimited number of inferences can be made once a model is trained, albeit model re-training is done periodically to improve accuracy.

Thus, the performance optimization of inference is an important field.

Reduction in inference time has advantages

- need less hardware to perform same number of inferences

- higher number of inferences with same hardware

- enables real-time applications such as autonomous driving or industrial monitoring

Especially in real-time applications, a lot of inferences are being made. every saved ms makes big difference

Look closer at Seagate

use of specific accelerator devices

Generic models perform poorly because they dont make full use of accelerator capabilities

Not only need capable accelerator, but also model that is attuned to leverage its full potential

3.1 Tensor Operator Optimization

additionally to traditional training and inference deep learning workflow, we introduce inference performance optimization to meet real-time requirements

include graphic showing train-inference vs train-optimize-inference to optimize for minimal inference time of whole network, we need to optimize implementation of every layer, or rather the respective tensor operator with the specific parameters (shape, padding, stride) in the computation graph

especially conv2d, because there are many and they are very computationally intensive

Dense not so important because less computational intensive (number?)

WHAT should be calculated is determined by model

HOW it should be calculated is not specified, so actual implementation can change

simple default implementation, but also loop unrolling, tiling, threads, example code?

It is important to note that the optimal implementation (regarding speed, memory usage) is very much dependent on target device

memory sharing and data reuse

3.2 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 devices

- 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

3.3 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

3.3.1 TensorComprehensions

does not use machine learning

3.3.2 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)

4 Using TVM

We want to explore capabilities and limitations of TVM

Want to be able to quickly test different scenarios (models, configurations, hardware)

4.1 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

4.2 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

4.3 Capabilities

Inference improvement vs default tvm and TF

Good in tradeoff with autotuning

Use numbers from paper and own numbers

4.4 Limitations

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

4.4.1 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

4.4.2 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

4.4.3 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

5 Autotuning Scheduler

Enabling controlled parallel autotuning is necessary to solve those problems

necessitates central scheduler that orchestrates all jobs

5.1 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

5.1.1 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

5.1.2 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

5.2 Implementation

Figure with autotuning procedure with scheduler

Since TVM only provides a python interface, we are using python 3.5

5.2.1 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

5.2.2 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

5.2.3 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

5.3 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

6 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

6.1 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

6.2 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

7 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

7.1 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