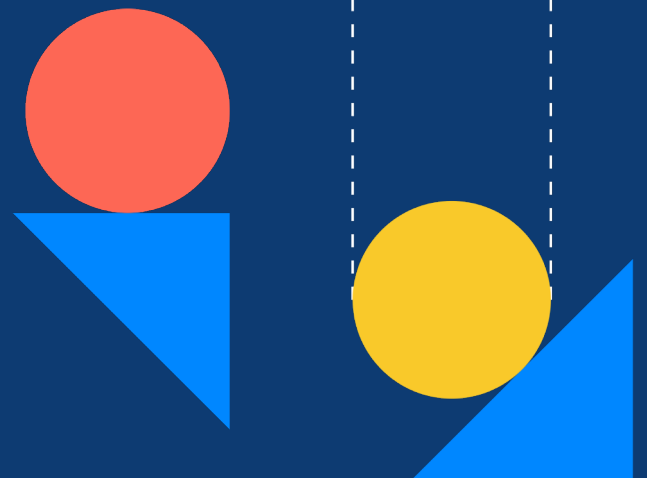
How to use Apache TVM to optimize your ML models

Faster inference in the cloud and at the edge

Sameer Farooqui
Product Marketing Manager, OctoML



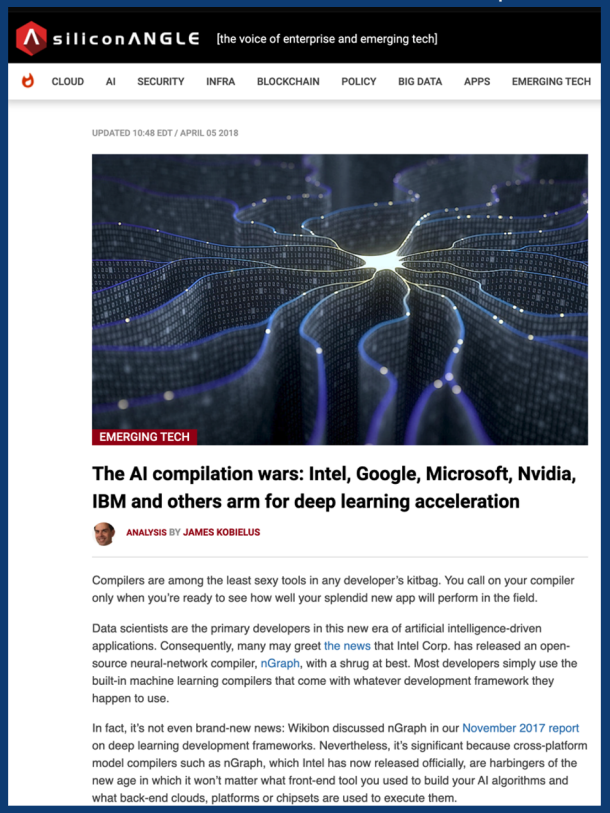
OctoML

Faster Artificial Intelligence Everywhere

Optimizing Deep Learning Compiler

siliconANGLE

April 2018



siliconANGLE [the voice of enterprise and emerging tech]

CLOUD AI SECURITY INFRA BLOCKCHAIN POLICY BIG DATA APPS EMERGING TECH

UPDATED 10:48 EDT / APRIL 05 2018

EMERGING TECH

The AI compilation wars: Intel, Google, Microsoft, Nvidia, IBM and others arm for deep learning acceleration

ANALYSIS BY JAMES KOBIELUS

Compilers are among the least sexy tools in any developer's kitbag. You call on your compiler only when you're ready to see how well your splendid new app will perform in the field.

Data scientists are the primary developers in this new era of artificial intelligence-driven applications. Consequently, many may greet the news that Intel Corp. has released an open-source neural-network compiler, nGraph, with a shrug at best. Most developers simply use the built-in machine learning compilers that come with whatever development framework they happen to use.

In fact, it's not even brand-new news: Wikibon discussed nGraph in our November 2017 report on deep learning development frameworks. Nevertheless, it's significant because cross-platform model compilers such as nGraph, which Intel has now released officially, are harbingers of the new age in which it won't matter what front-end tool you used to build your AI algorithms and what back-end clouds, platforms or chipsets are used to execute them.

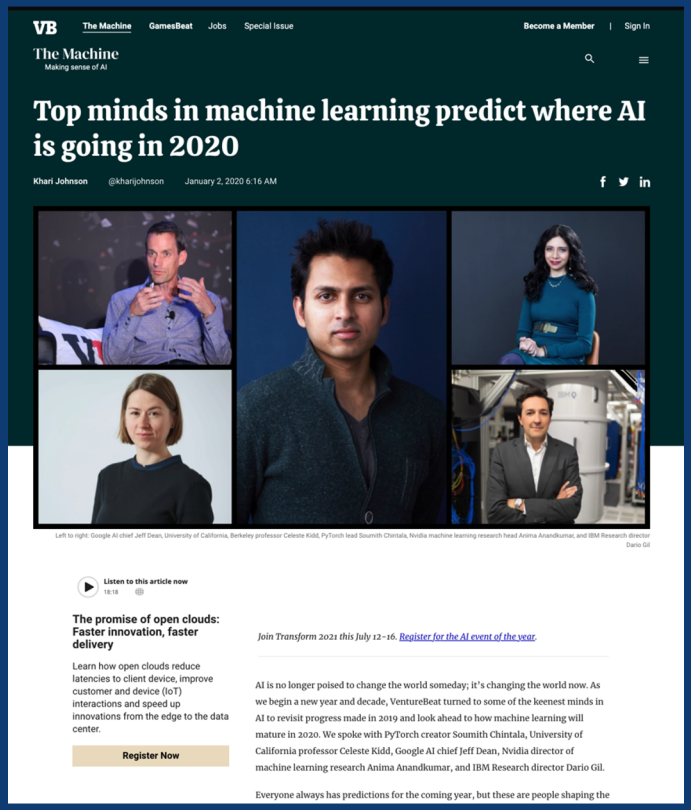
Read the article

Quotes from article:

- "...cross-platform model compilers [...] are **harbingers of the new age** in which it won't matter what front-end tool you used to build your AI algorithms and what back-end clouds, platforms or chipsets are used to execute them."
- "Cross-platform AI compilers **will become standard components** of every AI development environment, enabling developers to access every deep learning framework and target platform without having to know the technical particular of each environment."
- "...within the next two to three years, the AI industry will **converge around one open-source cross-compilation** supported by all front-end and back-end environments"

Venture Beat

Jan 2020



Read the article

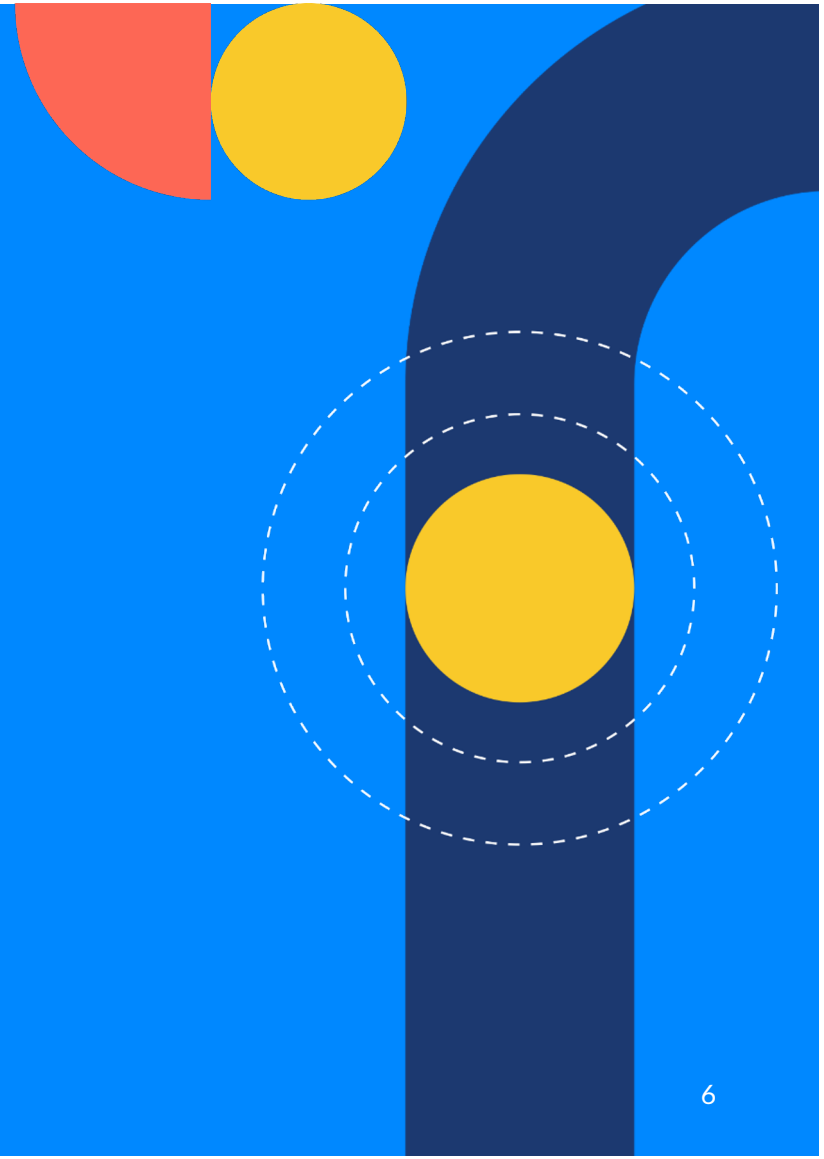
Quote from Soumith Chintala:
(co-creator of PyTorch and distinguished engineer at Facebook AI)

"With PyTorch and TensorFlow, you've seen the frameworks sort of converge. The reason quantization comes up, and a bunch of other lower-level efficiencies come up, is because **the next war is compilers for the frameworks — XLA, TVM, PyTorch has Glow, a lot of innovation is waiting to happen,**" he said.

"For the next few years, you're going to see … how to quantize smarter, how to fuse better, how to use GPUs more efficiently, [and] how to automatically compile for new hardware."

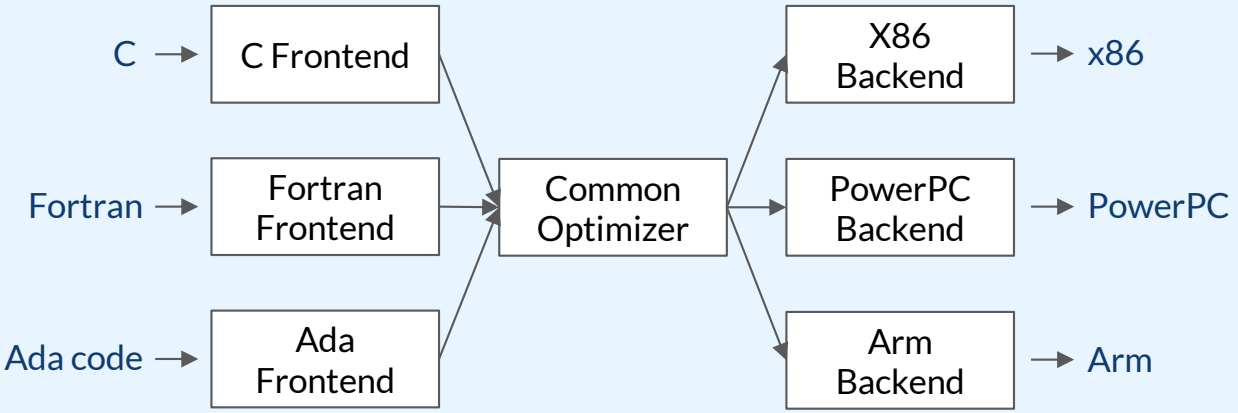
This Talk

- What is a ML Compiler?
- How TVM works
- TVM use cases
- OctoML Product Demo

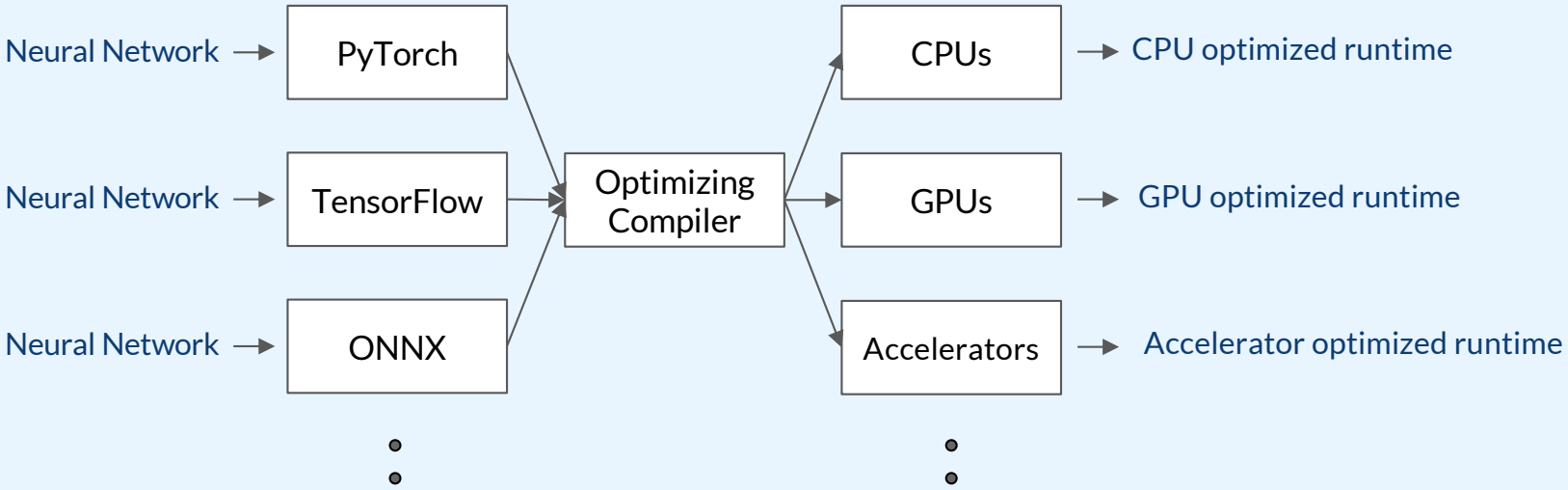


Classical Compiler

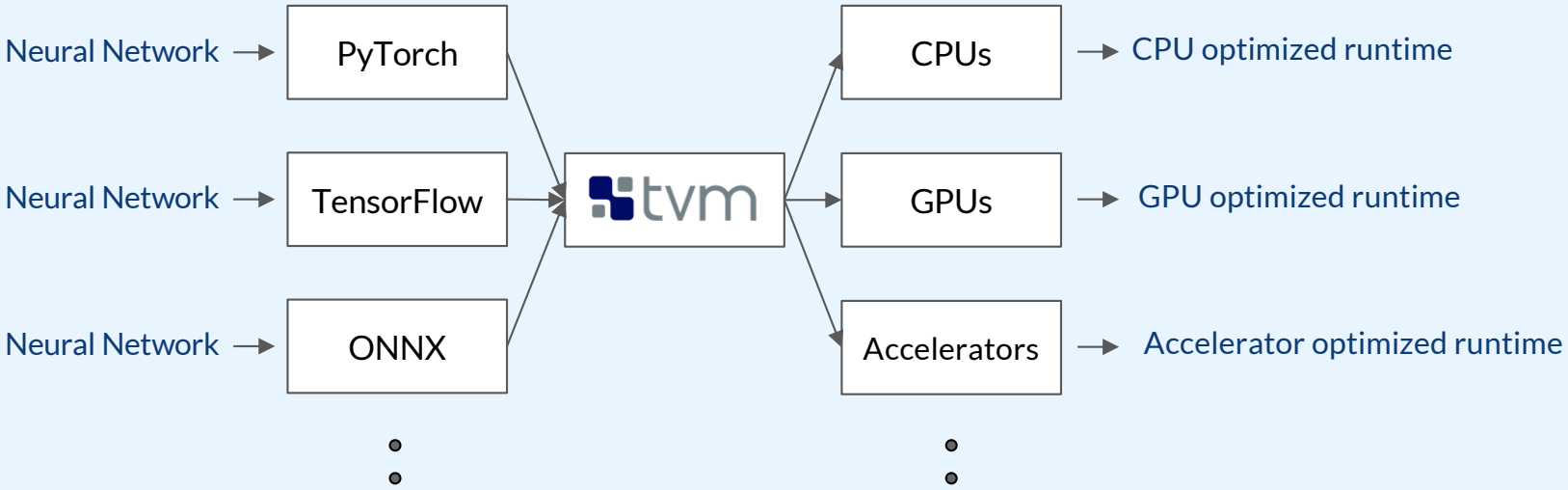
Classical Compiler



Deep Learning Compiler

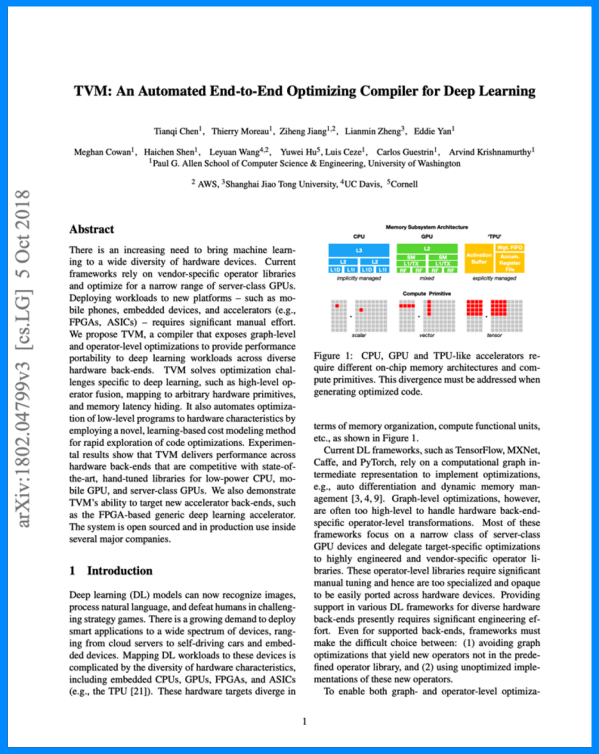


Deep Learning Compiler



TVM: An Automated End-to-End Optimizing Compiler for Deep Learning

Feb 2018



TVM: An Automated End-to-End Optimizing Compiler for Deep Learning

Tianqi Chen[1], Thierry Moreau[1], Ziheng Jiang[1,2], Lianmin Zheng[3], Eddie Yan[1]

Meghan Cowan[1], Haichen Shen[1], Leyuan Wang[4,2], Yuwei Hu[3], Luis Ceze[1], Carlos Guestrin[1], Arvind Krishnamurthy[1]

[1]Paul G. Allen School of Computer Science & Engineering, University of Washington

[2] AWS, [3]Shanghai Jiao Tong University, [4]UC Davis, [5]Cornell

Abstract

There is an increasing need to bring machine learning to a wide diversity of hardware devices. Current frameworks rely on vendor-specific operator libraries and optimize for a narrow range of server-class GPUs. Deploying workloads to new platforms – such as mobile phones, embedded devices, and accelerators (e.g., FPGAs, ASICs) – requires significant manual effort. We propose TVM, a compiler that exposes graph-level and operator-level optimizations to provide performance portability to deep learning workloads across diverse hardware back-ends. TVM solves optimization challenges specific to deep learning, such as high-level operator fusion, mapping to arbitrary hardware primitives, and memory latency hiding. It also automates optimization of low-level programs to hardware characteristics by employing a novel, learning-based cost modeling method for rapid exploration of code optimizations. Experimental results show that TVM delivers performance across hardware back-ends that are competitive with state-of-the-art, hand-tuned libraries for low-power CPU, mobile GPU, and server-class GPUs. We also demonstrate TVM's ability to target new accelerator back-ends, such as the FPGA-based generic deep learning accelerator. The system is open sourced and in production use inside several major companies.

1 Introduction

Deep learning (DL) models can now recognize images, process natural language, and defeat humans in challenging strategy games. There is a growing demand to deploy smart applications to a wide spectrum of devices, ranging from cloud servers to self-driving cars and embedded devices. Mapping DL workloads to these devices is complicated by the diversity of hardware characteristics, including embedded CPUs, GPUs, FPGAs, and ASICs (e.g., the TPU [21]). These hardware targets diverge in terms of memory organization, compute functional units, etc., as shown in Figure 1.

Figure 1: CPU, GPU and TPU-like accelerators require different on-chip memory architectures and compute primitives. This divergence must be addressed when generating optimized code.

Current DL frameworks, such as TensorFlow, MXNet, Caffe, and PyTorch, rely on a computational graph intermediate representation to implement optimizations, e.g., auto differentiation and dynamic memory management [3, 4, 9]. Graph-level optimizations, however, are often too high-level to handle hardware back-end-specific operator-level transformations. Most of these frameworks focus on a narrow class of server-class GPU devices and delegate target-specific optimizations to highly engineered and vendor-specific operator libraries. These operator-level libraries require significant manual tuning and hence are too specialized and opaque to be easily ported across hardware devices. Providing support in various DL frameworks for diverse hardware back-ends presently requires significant engineering effort. Even for supported back-ends, frameworks must make the difficult choice between: (1) avoiding graph optimizations that yield new operators not in the predefined operator library, and (2) using unoptimized implementations of these new operators.

To enable both graph- and operator-level optimiza-

1

Read the paper

- "There is an increasing need to bring machine learning to a wide diversity of hardware devices"
- TVM is "a compiler that exposes *graph-level* and *operator-level* optimizations to provide performance portability to deep learning workloads across diverse hardware back-ends"
- "Experimental results show that TVM delivers performance across hardware back-ends that are **competitive with state-of-the-art, hand-tuned libraries** for low-power CPU, mobile GPU, and server-class GPUs"

Relay: A High-level Compiler for Deep Learning

April 2019

Relay: A High-Level Compiler for Deep Learning

Jared Roesch, Steven Lyubomirsky, Marisa Kirisame, Logan Weber, Josh Pollock, Luis Vega, Ziheng Jiang, Tianqi Chen, Thierry Moreau, and Zachary Tatlock

{jroesch, sslyu, jerry96, weberlo, joshpol, vegaluis, ziheng, tqchen, moreau, ztatlock}@cs.uw.edu
Paul G. Allen School of Computer Science & Engineering
University of Washington

arXiv:1904.08368v2 [cs.LG] 24 Aug 2019

Abstract

Frameworks for writing, compiling, and optimizing deep learning (DL) models have recently enabled progress in areas like computer vision and natural language processing. Extending these frameworks to accommodate the rapidly diversifying landscape of DL models and hardware platforms presents challenging tradeoffs between expressivity, composability, and portability. We present Relay, a new compiler framework for DL. Relay's functional, statically typed intermediate representation (IR) unifies and generalizes existing DL IRs to express state-of-the-art models. The introduction of Relay's expressive IR requires careful design of domain-specific optimizations, addressed via Relay's extension mechanisms. Using these extension mechanisms, Relay supports a unified compiler that can target a variety of hardware platforms. Our evaluation demonstrates Relay's competitive performance for a broad class of models and devices (CPUs, GPUs, and emerging accelerators). Relay's design demonstrates how a unified IR can provide expressivity, composability, and portability without compromising performance.

1. Introduction

Deep learning (DL) has radically transformed domains like computer vision and natural language processing (NLP) [36, 56]. Inspired by these successes, researchers and companies are continually experimenting with increasingly sophisticated DL models and developing specialized hardware backends. DL frameworks for writing, optimizing, and compiling DL models reduce the complexity of these tasks, which in turn accelerates DL research and product development.

Popular DL compiler intermediate representations (IRs) offer different tradeoffs between expressivity, composability, and portability [1, 33, 50, 52, 5, 38]. Early frameworks adopted IRs specialized for then-state-of-the-art models and/or emerging hardware accelerators. As a result, non-trivial extensions require patching or even forking frameworks [27, 47, 52, 41, 55, 38, 51]. Such *ad hoc* extensions can improve expressivity while maintaining backwards compatibility with existing execution mechanisms. However, they are difficult to design, reason about, and implement, often resulting in modifications that are mutually incompatible.

Let us consider a hypothetical scenario that exemplifies IR design tensions in DL compilers. Suppose a machine learning engineer wants to write an Android app that uses sentiment analysis to determine the moods of its users. To maintain privacy, the app must run completely on-device, i.e., no work can be offloaded to the cloud. The engineer decides to use a variant of TreeLSTM, a deep learning model that uses a tree structure [46]. Unfortunately, current frameworks' IRs cannot directly encode trees, so she must use a framework extension like TensorFlow Fold [26].

Suppose that after adapting the model to run on her phone, the out-of-the-box performance of her model on her particular platform is not satisfactory, requiring her to optimize it. She chooses to employ *quantization*, an optimization that potentially trades accuracy for performance by replacing floating-point datatypes with low-precision ones. Although researchers have developed a variety of quantization strategies, each of which makes use of different bit-widths, rounding modes, and datatypes, our engineer must use a strategy supported by existing frameworks [15, 14, 34]. Unfortunately, frameworks only provide support for a small number of strategies, and supporting new quantization strategies is non-trivial. Each combination of operator, datatype, bit-width, and platform requires unique operator implementations. Optimizations like operator fusion exacerbate this combinatorial explosion, further increasing the number of unique implementations required. Furthermore, if a framework doesn't have specific support for the target phone model she cannot take advantage of specialized deep learning instructions or coprocessors [3].

The scenario above highlights the three-pronged *extensibility challenge* for DL IRs:

1. *Expressivity*: It should be straightforward to write models involving control flow, first-class functions and data structures (e.g., trees, graphs, and lists).
2. *Composability*: It should be straightforward to add and compose new optimizations with existing ones (e.g., quantization, operator fusion, and partial evaluation).
3. *Portability*: It should be straightforward to add new hardware targets (e.g., TPU, Inferentia) [20, 2].

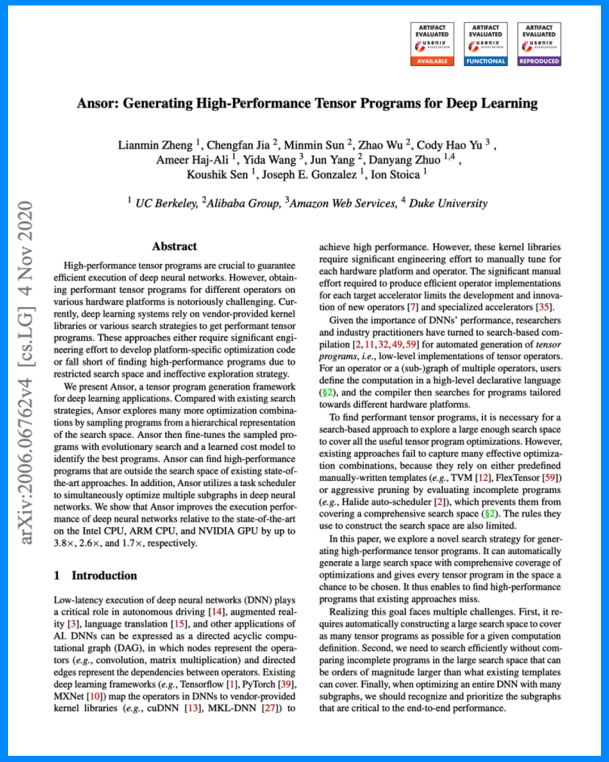
Previous IRs have struggled to address these challenges, treating each component of the framework as a disconnected

[Read the paper]

- Relay is "**a high-level IR** that enables end-to-end optimization of deep learning models for a variety of devices"
- "Relay's functional, statically typed intermediate representation (IR) unifies and generalizes existing DL IRs to express state-of-the-art models"
- "With its *extensible design* and *expressive language*, Relay serves as a **foundation for future work** in applying compiler techniques to the domain of deep learning systems"

Ansor: Generating High-Performance Tensor Programs for Deep Learning

Nov 2020



Ansor: Generating High-Performance Tensor Programs for Deep Learning

Lianmin Zheng [1], Chengfan Jia [2], Minmin Sun [2], Zhao Wu [2], Cody Hao Yu [3], Ameer Haj-Ali [1], Yida Wang [3], Jun Yang [2], Danyang Zhuo [1,4], Koushik Sen [1], Joseph E. Gonzalez [1], Ion Stoica [1]

[1] *UC Berkeley,* [2] *Alibaba Group,* [3] *Amazon Web Services,* [4] *Duke University*

Abstract

High-performance tensor programs are crucial to guarantee efficient execution of deep neural networks. However, obtaining performant tensor programs for different operators on various hardware platforms is notoriously challenging. Currently, deep learning systems rely on vendor-provided kernel libraries or various search strategies to get performant tensor programs. These approaches either require significant engineering effort to develop platform-specific optimization code or fall short of finding high-performance programs due to restricted search space and ineffective exploration strategy.

We present Ansor, a tensor program generation framework for deep learning applications. Compared with existing search strategies, Ansor explores many more optimization combinations by sampling programs from a hierarchical representation of the search space. Ansor then fine-tunes the sampled programs with evolutionary search and a learned cost model to identify the best programs. Ansor can find high-performance programs that are outside the search space of existing state-of-the-art approaches. In addition, Ansor utilizes a task scheduler to simultaneously optimize multiple subgraphs in deep neural networks. We show that Ansor improves the execution performance of deep neural networks relative to the state-of-the-art on the Intel CPU, ARM CPU, and NVIDIA GPU by up to 3.8×, 2.6×, and 1.7×, respectively.

1 Introduction

Low-latency execution of deep neural networks (DNN) plays a critical role in autonomous driving [14], augmented reality [3], language translation [15], and other applications of AI. DNNs can be expressed as a directed acyclic computational graph (DAG), in which nodes represent the operators (*e.g.*, convolution, matrix multiplication) and directed edges represent the dependencies between operators. Existing deep learning frameworks (*e.g.*, Tensorflow [1], PyTorch [39], MXNet [10]) map the operators in DNNs to vendor-provided kernel libraries (*e.g.*, cuDNN [13], MKL-DNN [27]) to achieve high performance. However, these kernel libraries require significant engineering effort to manually tune for each hardware platform and operator. The significant manual effort required to produce efficient operator implementations for each target accelerator limits the development and innovation of new operators [7] and specialized accelerators [35].

Given the importance of DNNs' performance, researchers and industry practitioners have turned to search-based compilation [2, 11, 32, 49, 59] for automated generation of *tensor programs*, *i.e.*, low-level implementations of tensor operators. For an operator or a (sub-)graph of multiple operators, users define the computation in a high-level declarative language (§2), and the compiler then searches for programs tailored towards different hardware platforms.

To find performant tensor programs, it is necessary for a search-based approach to explore a large enough search space to cover all the useful tensor program optimizations. However, existing approaches fail to capture many effective optimization combinations, because they rely on either predefined manually-written templates (*e.g.*, TVM [12], FlexTensor [59]) or aggressive pruning by evaluating incomplete programs (*e.g.*, Halide auto-scheduler [2]), which prevents them from covering a comprehensive search space (§2). The rules they use to construct the search space are also limited.

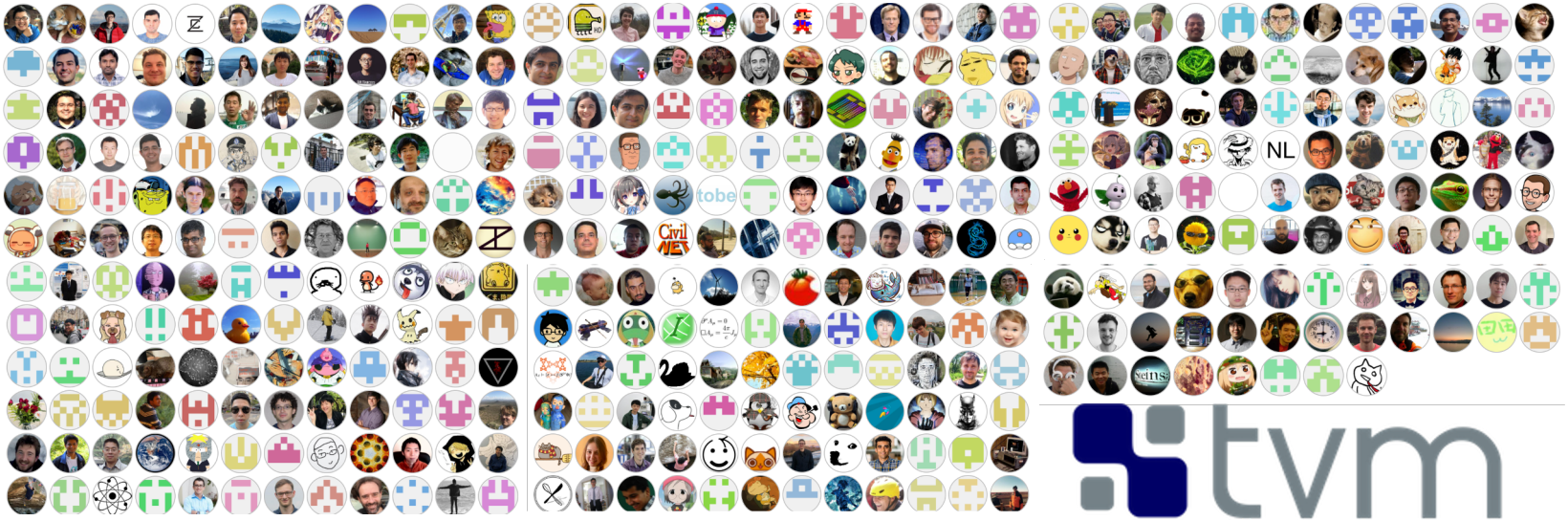
In this paper, we explore a novel search strategy for generating high-performance tensor programs. It can automatically generate a large search space with comprehensive coverage of optimizations and gives every tensor program in the space a chance to be chosen. It thus enables to find high-performance programs that existing approaches miss.

Realizing this goal faces multiple challenges. First, it requires automatically constructing a large search space to cover as many tensor programs as possible for a given computation definition. Second, we need to search efficiently without comparing incomplete programs in the large search space that can be orders of magnitude larger than what existing templates can cover. Finally, when optimizing an entire DNN with many subgraphs, we should recognize and prioritize the subgraphs that are critical to the end-to-end performance.

Read the paper

- "...obtaining performant tensor programs for different operators on various hardware platforms is notoriously challenging"
- Ansor is "a tensor program generation framework for deep learning applications"
- "Ansor can find high-performance programs that are outside the search space of existing state-of-the-art approaches"
- "We show that Ansor improves the execution performance of deep neural networks relative to the state-of-the-art on the Intel CPU, ARM CPU, and NVIDIA GPU by up to **3.8×, 2.6×, and 1.7×**, respectively"

Thank you Apache TVM contributors! 500+!



Who is using TVM?

Every Alexa wake-up today across all devices uses a TVM-optimized model

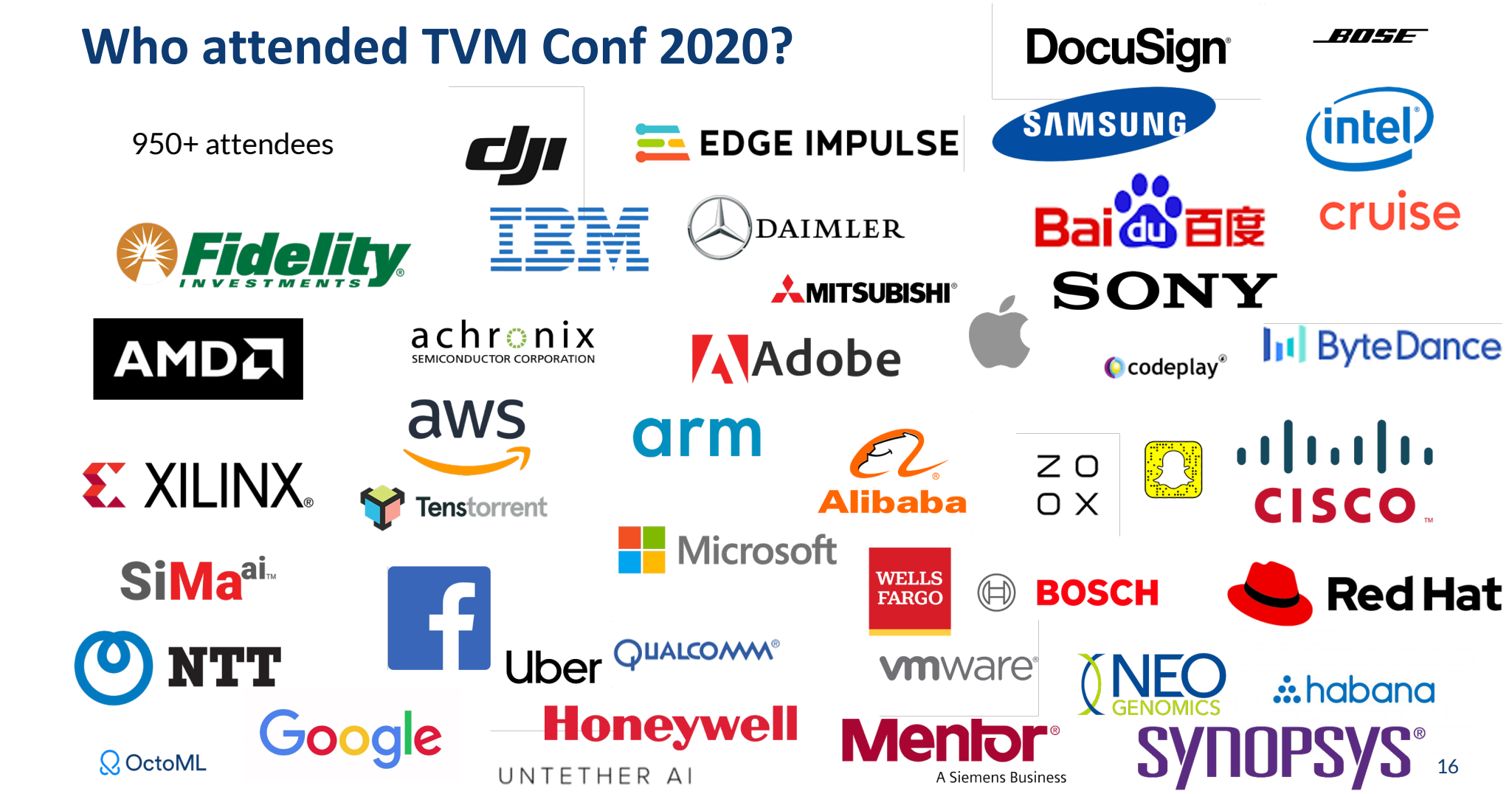
"At Facebook, we've been contributing to TVM for the past year and a half or so, and it's been a really awesome experience"

"We're really excited about the performance of TVM." - Andrew Tulloch, AI Researcher

Bing query understanding: 3x faster on CPU
QnA bot: 2.6 faster on CPU, 1.8x faster on GPU

Who attended TVM Conf 2020?

950+ attendees



Deep Learning Systems Landscape (open source)

Orchestrators	 
Frameworks	   
Accelerators	   
Vendor Libraries	NVIDIA cuDNN Intel oneDNN Arm Compute Library
Hardware	CPUs GPUs Accelerators

How does TVM work?

Graph Level Optimizations

Rewrites dataflow graphs (nodes and edges) to simplify the graph and reduce device peak memory usage

Operator Level Optimizations

Hardware target-specific low-level optimizations for individual operators/nodes in the graph.

Efficient Runtime

TVM optimized models run in the lightweight TVM Runtime System, providing a minimal API for loading and executing the model in Python, C++, Rust, Go, Java or Javascript

Deep Learning Operators

- Deep Neural Networks look like Directed Acyclic Graphs (DAGs)
- Operators are the building blocks (nodes) of neural network models
- Network edges represent data flowing between operators

Matrix Multiplication

Broadcast Add

RNN

ArgMin/ArgMax

LSTM

Convolution

OneHotEncoder

Gemm

Sigmoid

Batch Normalization

Pooling

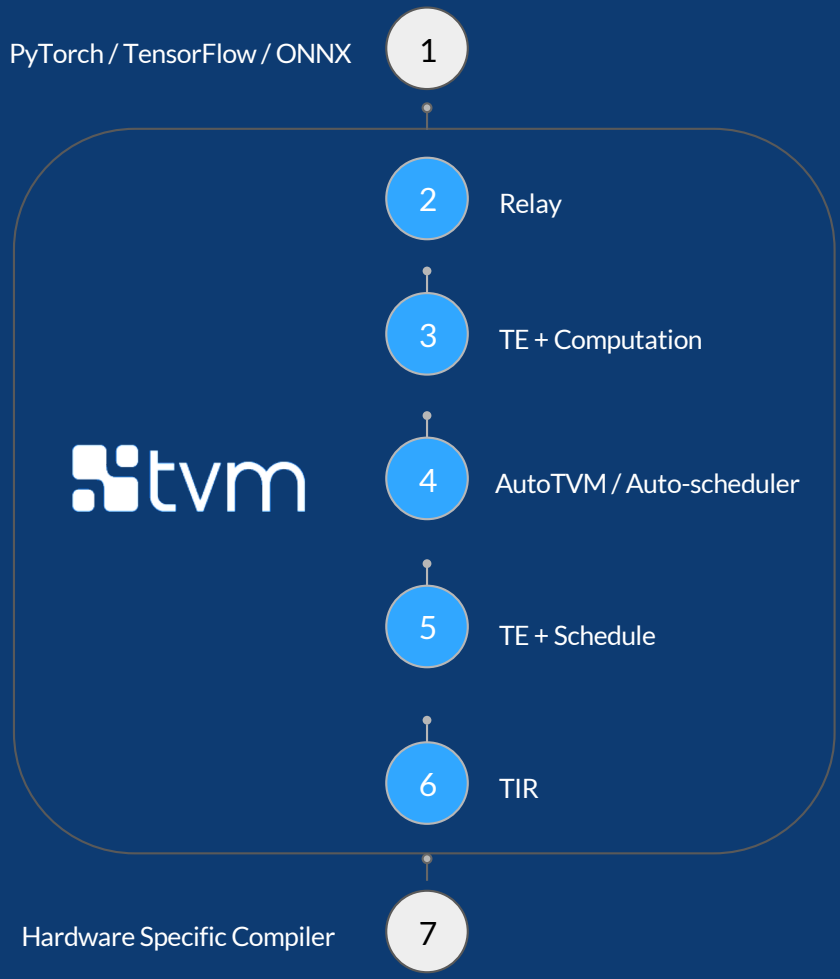
LeakyRelu

DynamicQuantizeLinear

Softmax

Dropout

TVM Internals



Relay

- Relay has a functional, statically typed intermediate representation (IR)

Auto-scheduler (a.k.a. Ansor)

Collaborators: OctoML Berkeley UNIVERSITY OF CALIFORNIA aws Alibaba Group 阿里巴巴集团

- **Goal:** Automatically turn tensor operations (like matmul or conv2d) into efficient code implementation
- AutoTVM *(1st gen)*: template-based search algorithm to find efficient implementation for tensor operations.
    - required domain experts to write a manual template for every operator on every platform, > 15k loc in TVM
- Auto-scheduler *(2nd gen)* replaces AutoTVM
- Auto-scheduler/Ansor aims to a fully automated scheduler for generating high-performance code for tensor computations, without manual templates
- Auto-scheduler can achieve better performance with faster search time in a more automated way b/c of innovations in search space construction and search algorithm

AutoTVM vs Auto-scheduler

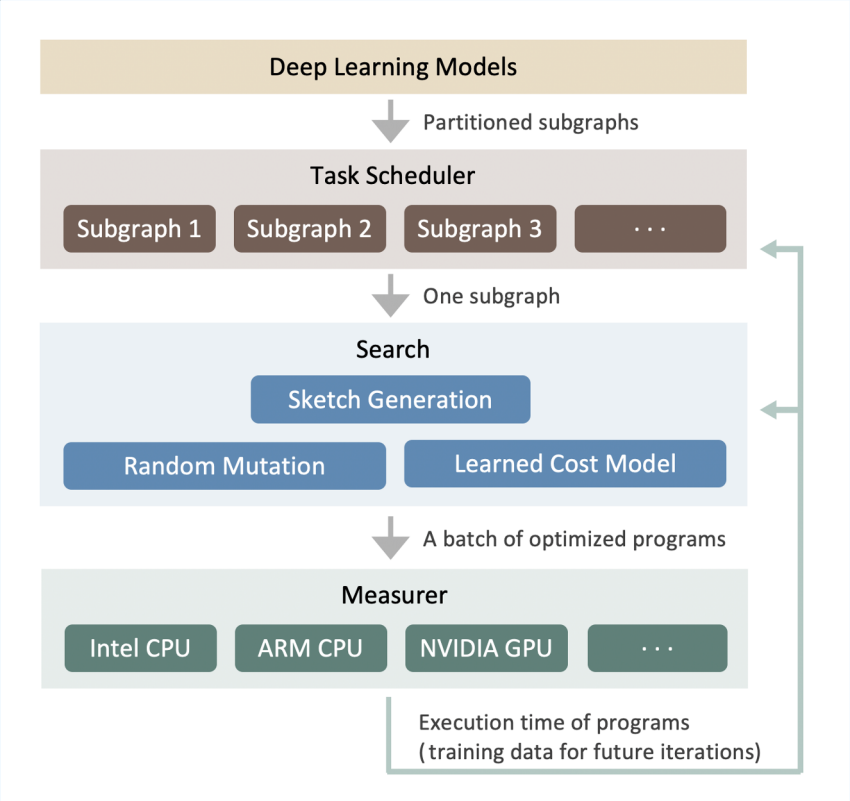
	AutoTVM Workflow	Auto-scheduler Workflow

Step 1: Write a compute definition
 	
(relatively easy part)
	`# Matrix multiply`
	
	`C = te.compute((M, N), lambda x, y:`
	`  te.sum(A[x, k] * B[k, y], axis=k))`
		`# The same`

Step 2: Write a schedule template
 	
(difficult part)
	`# 20-100 lines of tricky DSL code`
	
	`# Define search space`
	`cfg.define_split("tile_x", batch, num_outputs=4)`
	`cfg.define_split("tile_y", out_dim, num_outputs=4)`
	`…`
	
	`# Apply config into the template`
	`bx, txz, tx, xi = cfg["tile_x"].apply(s, C, C.op.axis[0])`
	`by, tyz, ty, yi = cfg["tile_y"].apply(s, C, C.op.axis[1])`
	`s[C].reorder(by, bx, tyz, txz, ty, tx, yi, xi)`
	`s[CC].compute_at(s[C], tx)`
	`…`
		`# Not required`

Step 3: Run auto-tuning (automatic search)	`tuner.tune(…)`	`task.tune(…)`

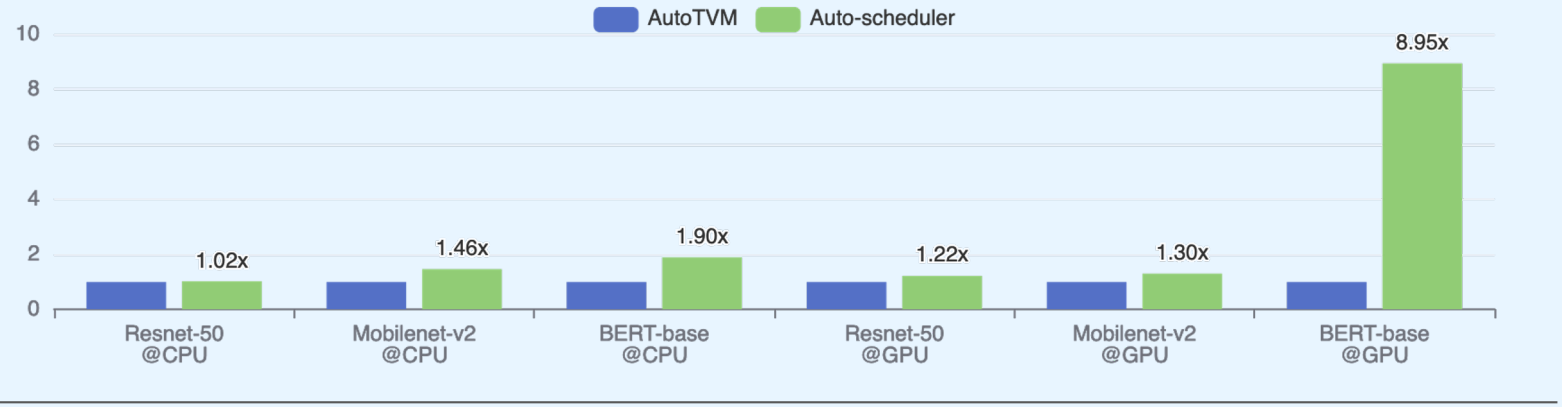
Auto-scheduler's Search Process



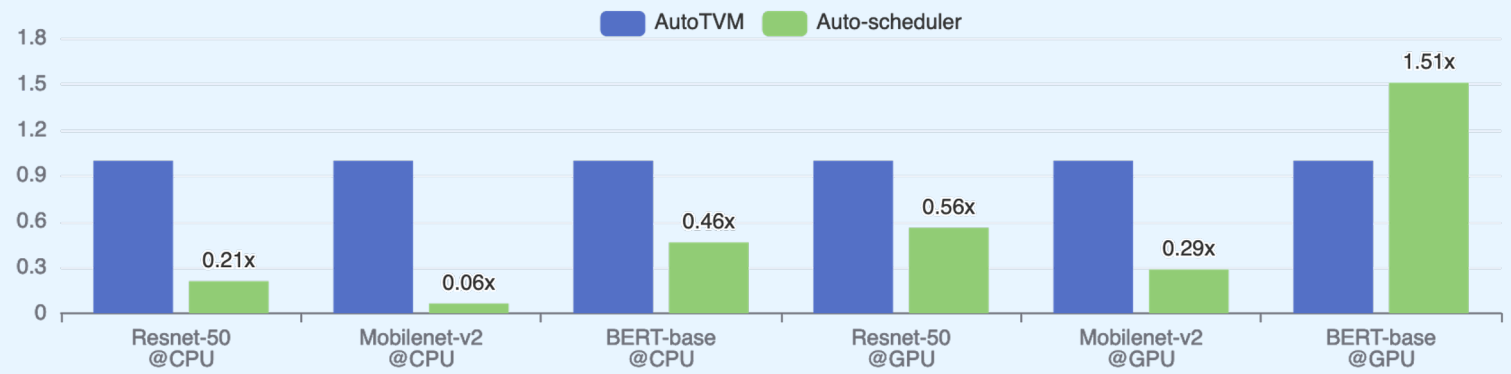
Source: Apache TVM Blog: Introducing Auto-scheduler

Benchmarks: AutoTVM vs Auto-scheduler

Code Performance Comparison
(higher is better)



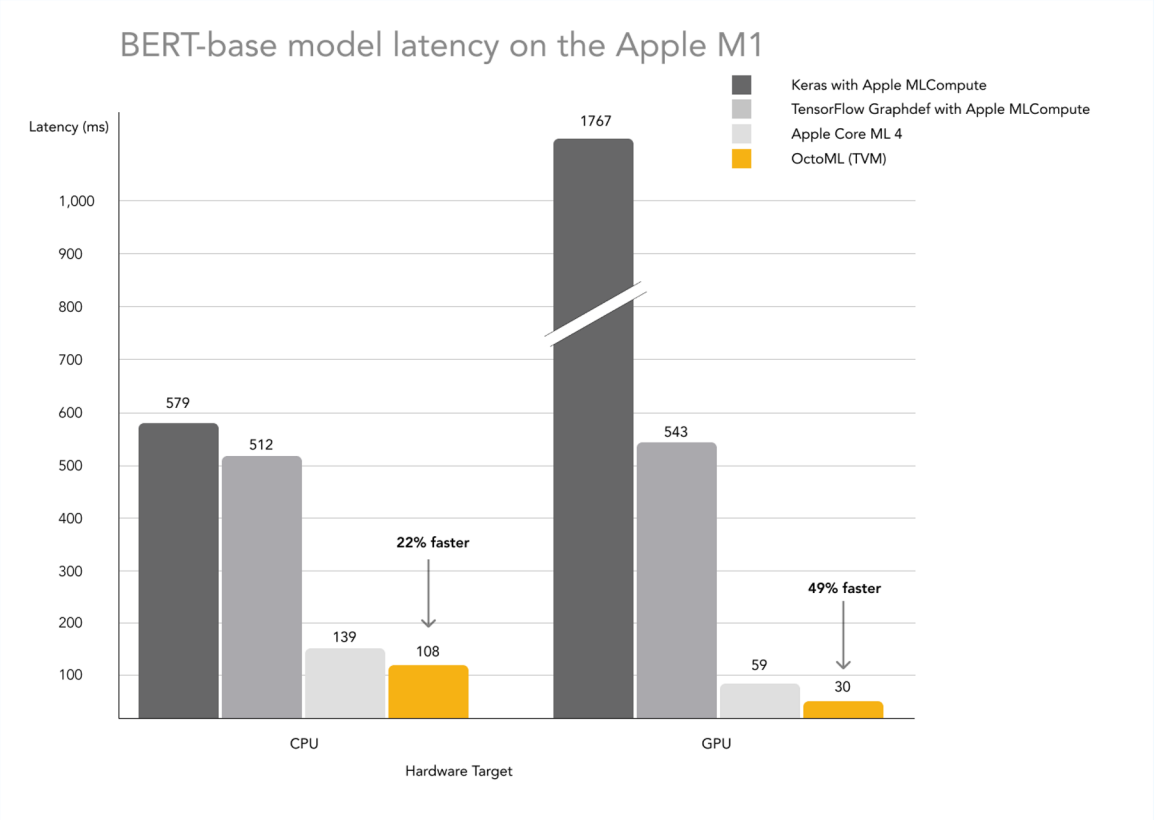
Search Time Comparison
(lower is better)



Source: Apache TVM Blog: Introducing Auto-scheduler

Auto-scheduling on Apple M1

(lower is better)



- **22% faster on CPU**
- **49% faster on GPU**

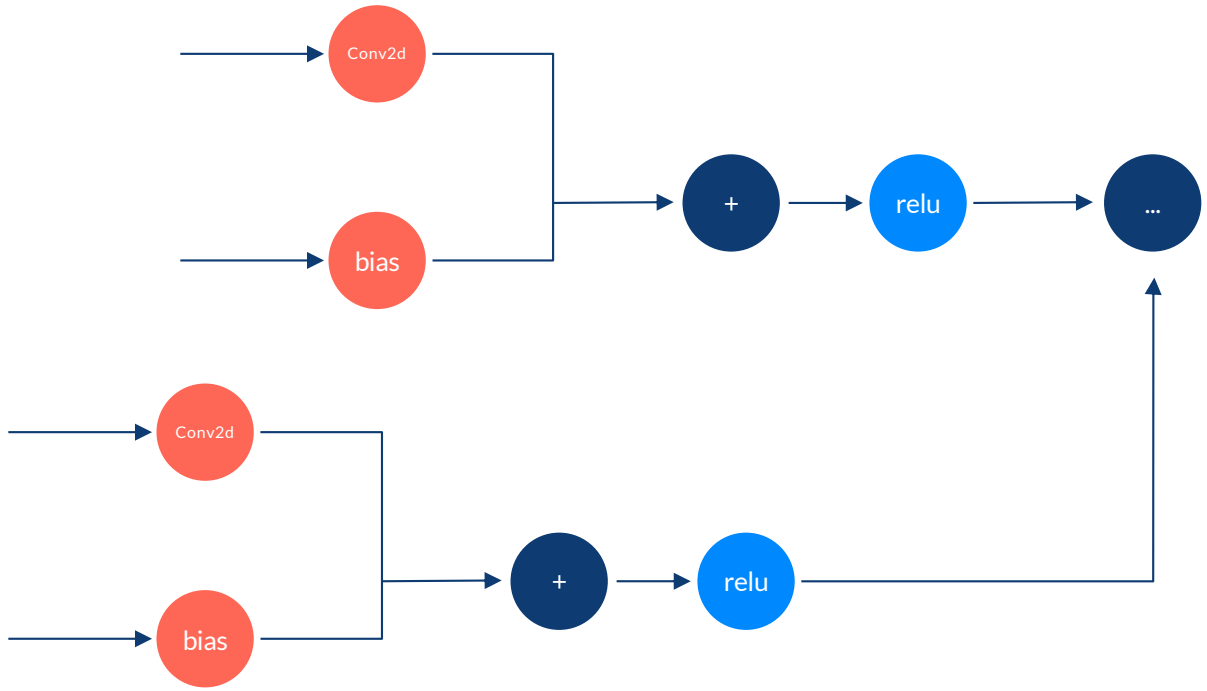
How?

- Effective Auto-scheduler searching
- Fuse qualified subgraphs

OctoML

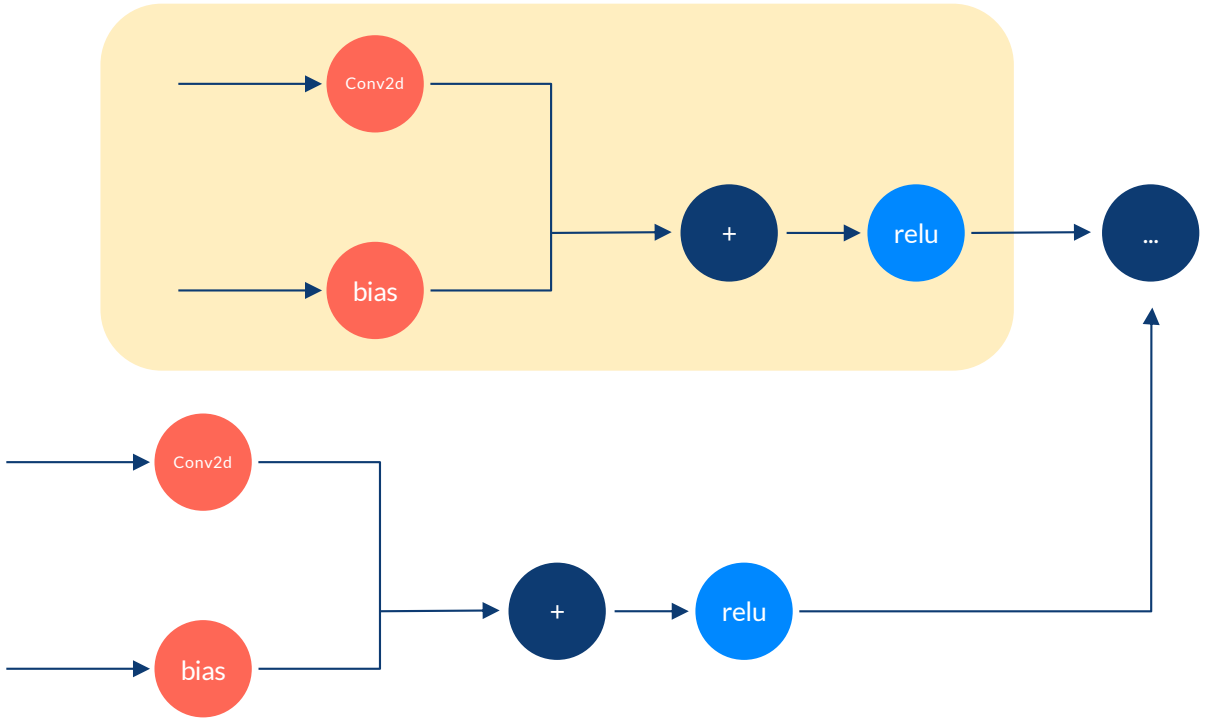
Source: OctoML Blog: Beating Apple's CoreML4

Relay



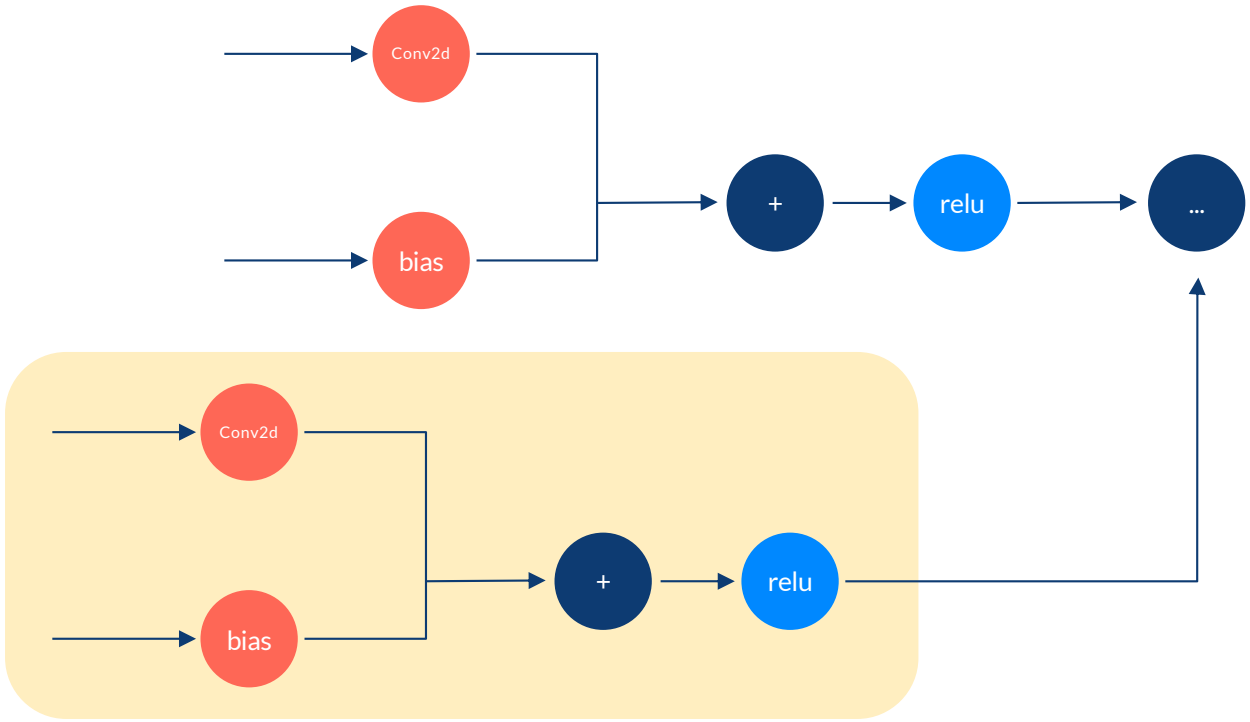
Relay: Fusion

Combine into a single fused operation which can then be optimized specifically for your target.



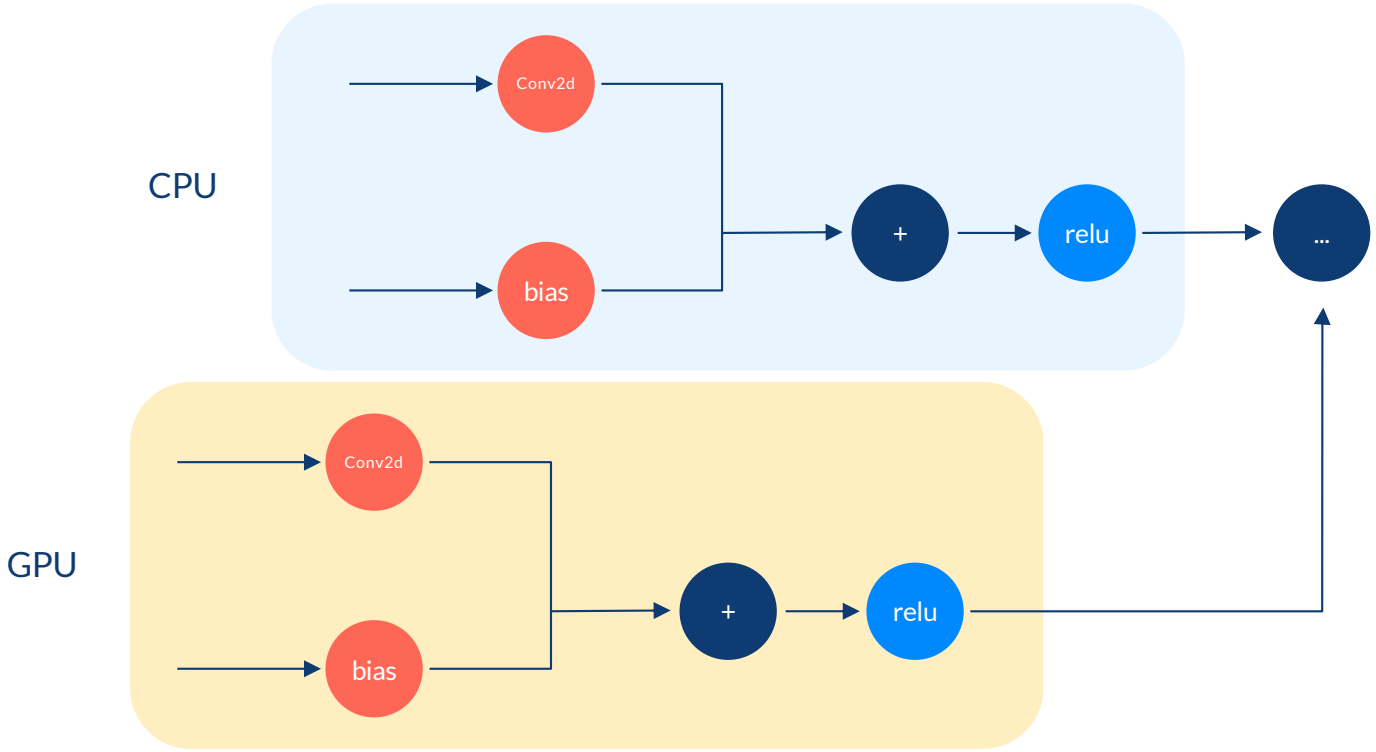
Relay: Fusion

Combine into a single fused operation which can then be optimized specifically for your target.



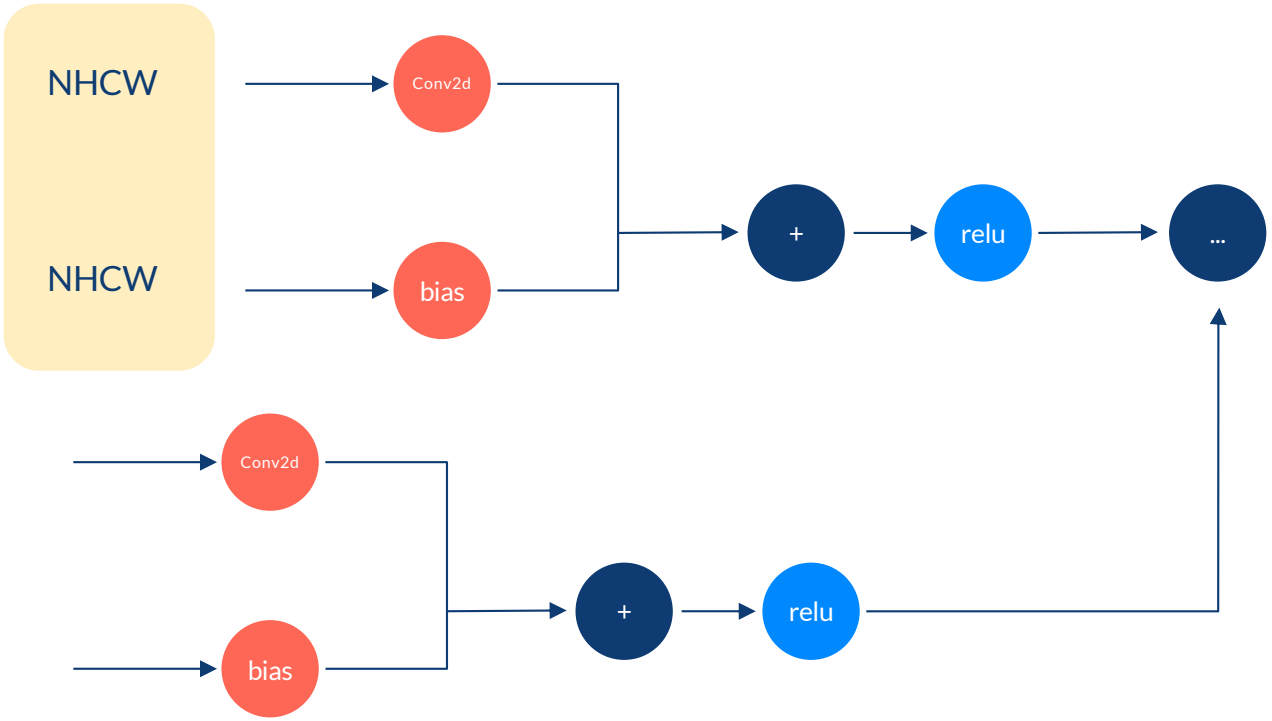
Relay: Device Placement

Partition your network to run on multiple devices.



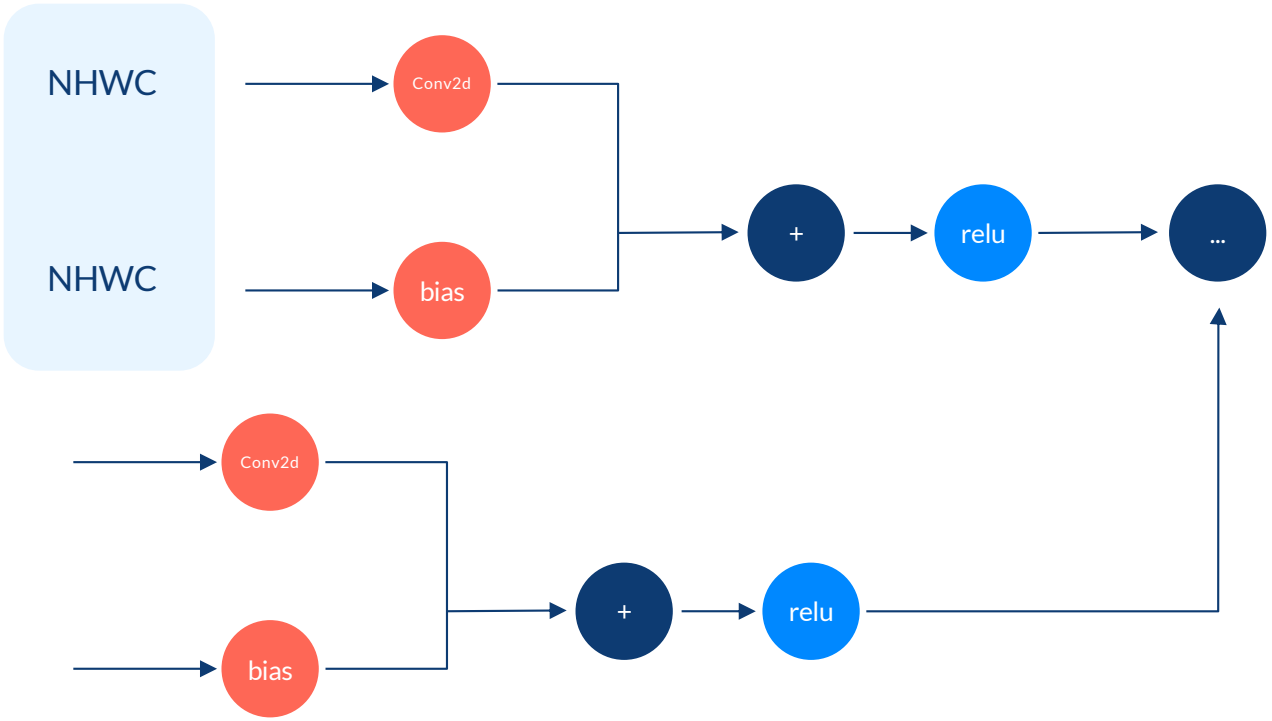
Relay: Layout Transformation

Generate efficient code for different data layouts.



Relay: Layout Transformation

Generate efficient code for different data layouts.



TIR Script

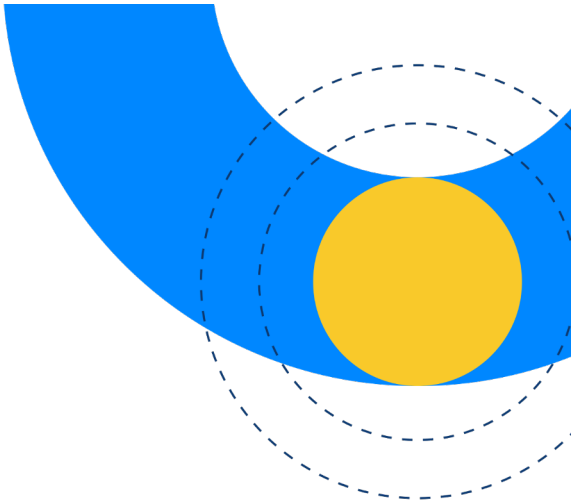
- TIR provides more flexibility than high level tensor expressions.
- Not everything is expressible in TE and auto-scheduling is not always perfect.
  - AutoScheduling 3.0 (code-named AutoTIR coming later this year)
  - We can also directly write TIR directly using TIRScript.

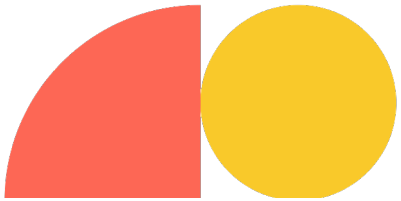
```
@tvm.script.tir
def fuse_add_exp(a: ty.handle, c: ty.handle) -> None:
    A = tir.match_buffer(a, (64,))
    C = tir.match_buffer(c, (64,))
    B = tir.alloc_buffer((64,))

    with tir.block([64], "B") as [vi]:
        B[vi] = A[vi] + 1

    with tir.block([64], "C") as [vi]:
        C[vi] = exp(B[vi])
```

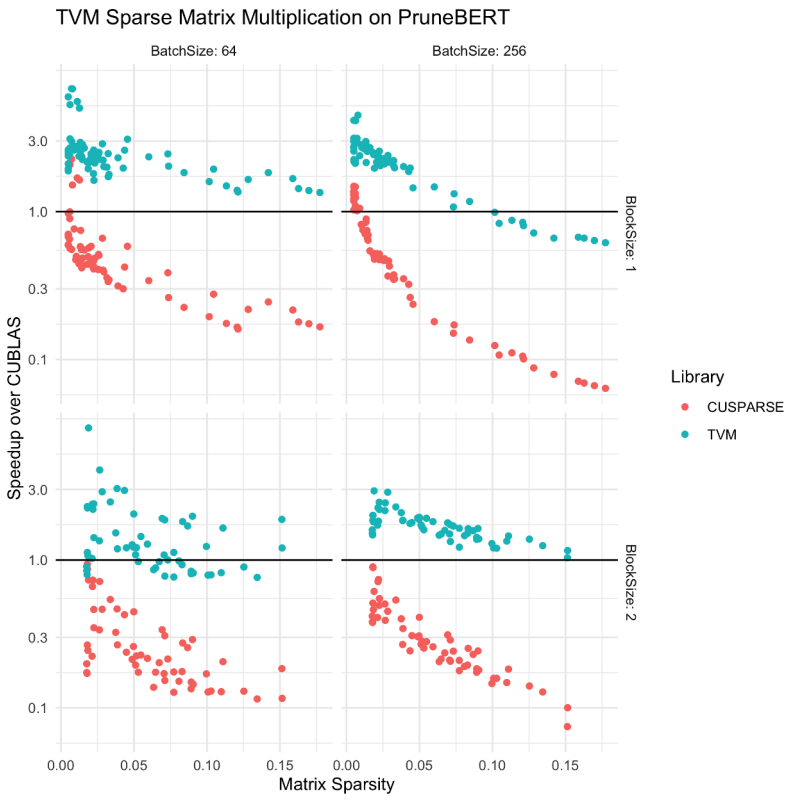
Select Performance Results





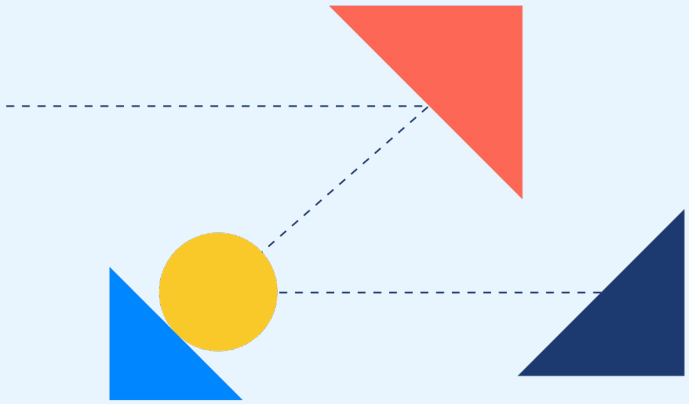
Faster Kernels for Dense-Sparse Multiplication

- Performance comparison on PruneBERT
- 3-10x faster than cuBLAS and cuSPARSE.
- 1 engineer writing TensorIR kernels



Performance at OctoML in 2020

TVM $\log_2$ fold improvement over baseline



Over 60 model x hardware benchmarking studies

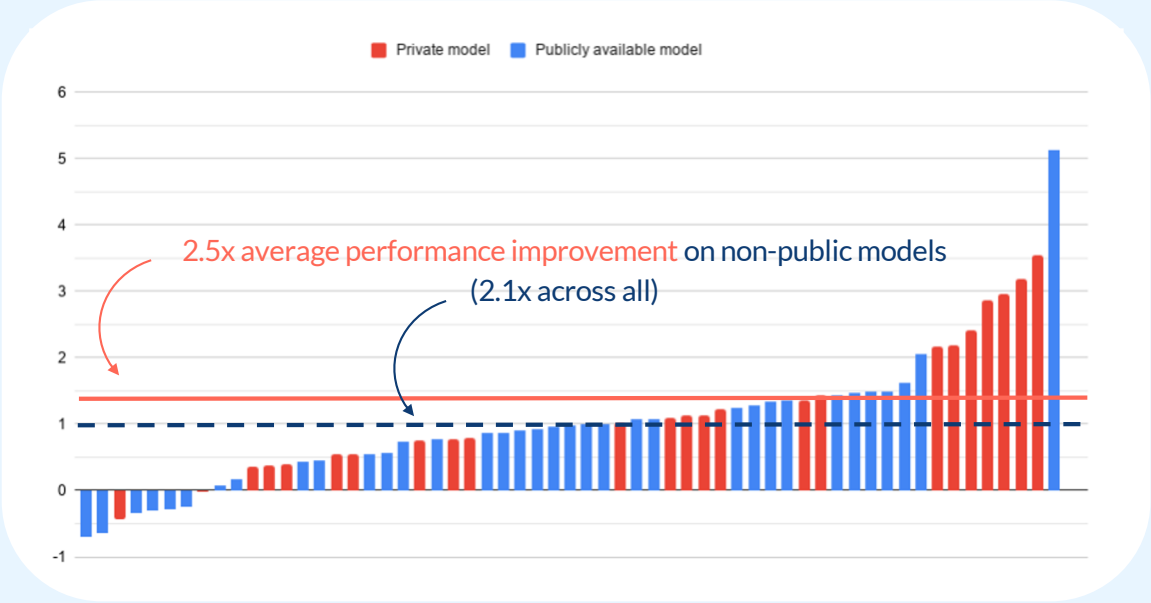
Each study compared TVM against best* baseline on the target

Sorted by ascending $\log_2$ gain over baseline

Model x hardware comparison points

Across a broad variety of models and platforms

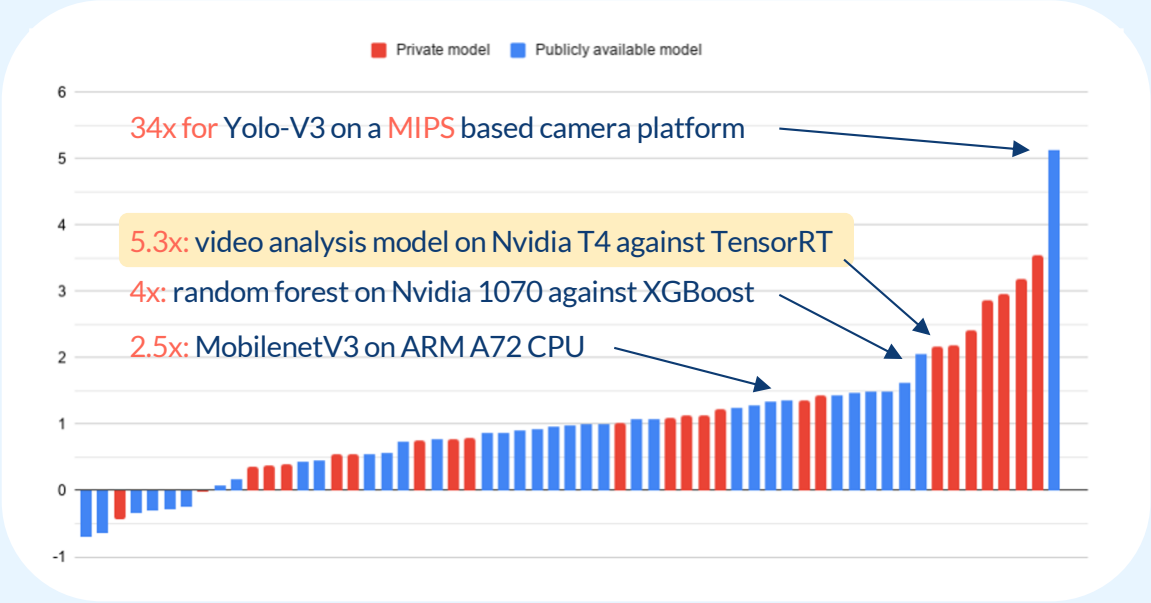
TVM $\log_2$ fold improvement over baseline



Model x hardware comparison points

Across a broad variety of models and platforms

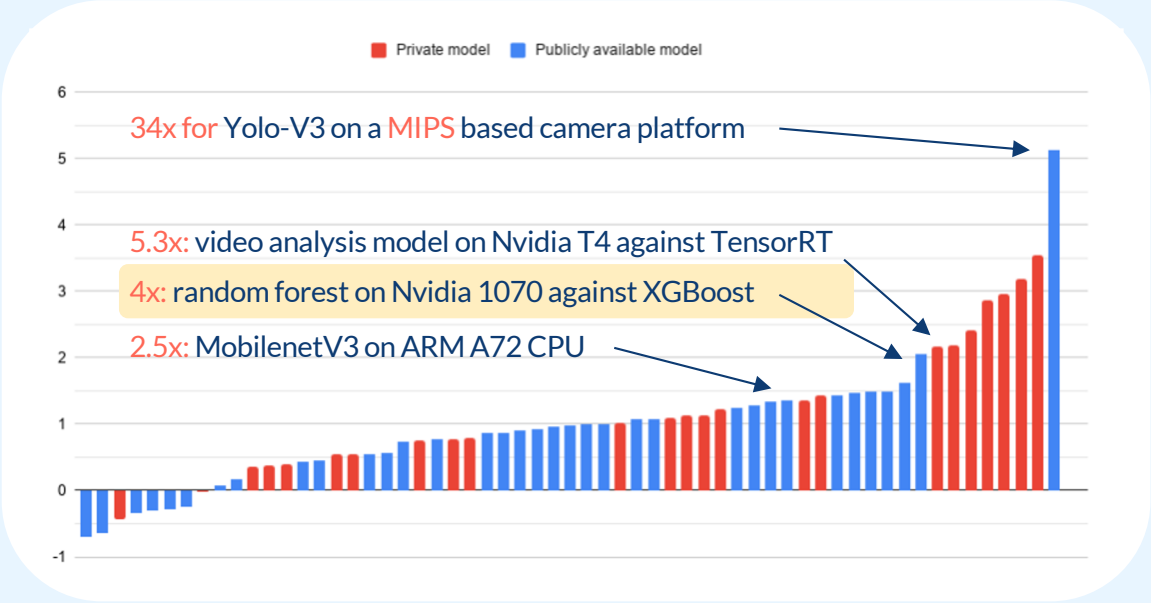
TVM $\log_2$ fold improvement over baseline



Model x hardware comparison points

Across a broad variety of models and platforms

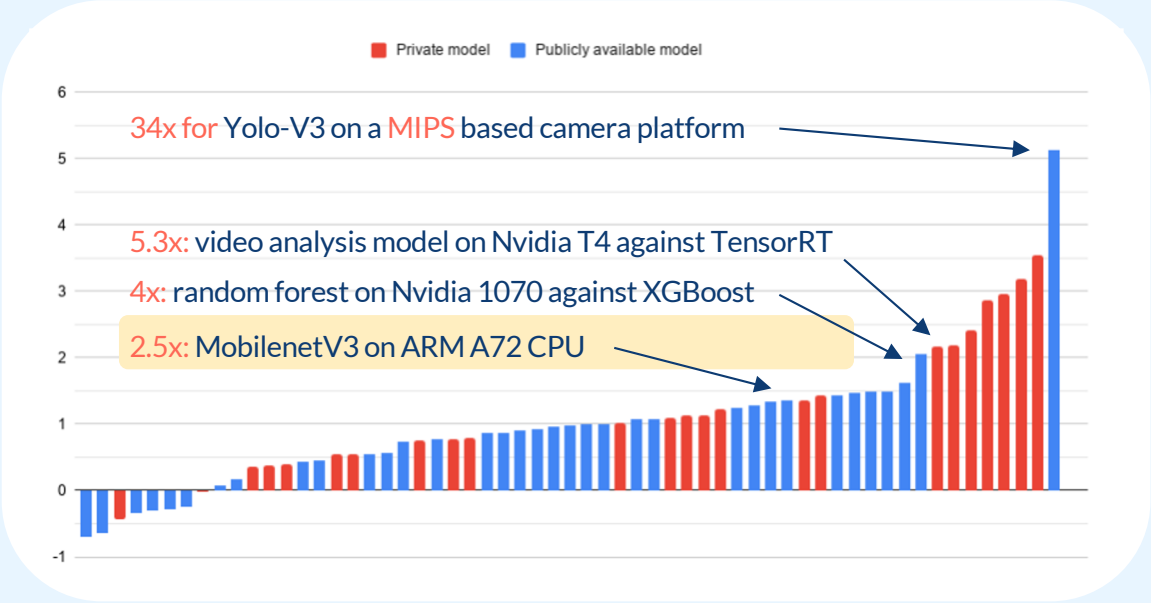
TVM $\log_2$ fold improvement over baseline



Model x hardware comparison points

Across a broad variety of models and platforms

TVM $\log_2$ fold improvement over baseline



Model x hardware comparison points

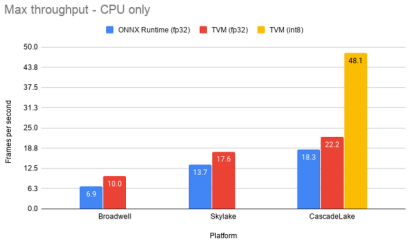
Case Study: 90% cloud inference cost reduction

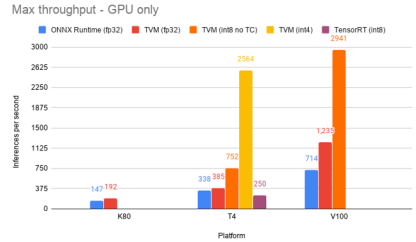
Background

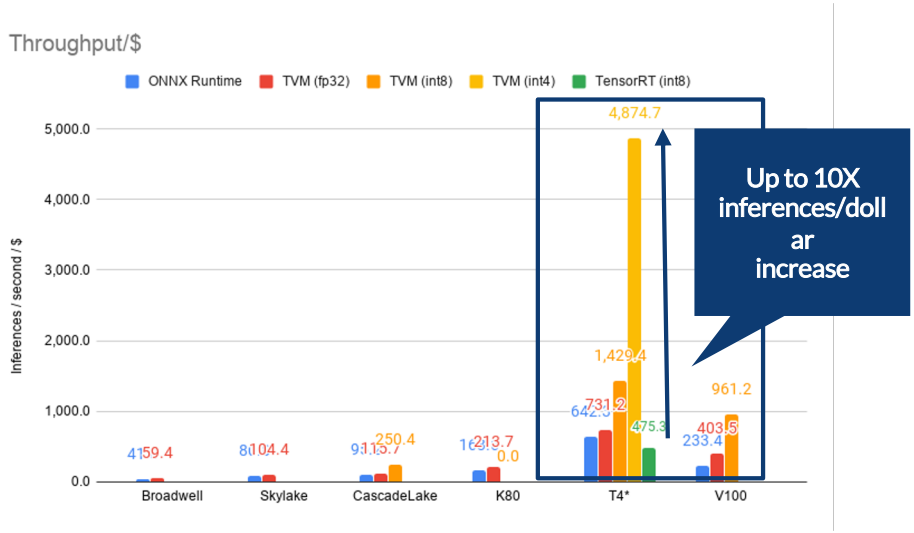
- Top 10 Tech Company running multiple variations of customized CV models
- Model in batch processing /offline mode using standard HW targets of a major public cloud.
- Billions of inferences per month
- Benchmarking on CPU and GPU

Results

- 3.8x - TensorRT 8bit to TVM 8bit
- 10x - TensorRT 8bit to TVM 4bit
- **Potential to reduce hourly costs by 90%**



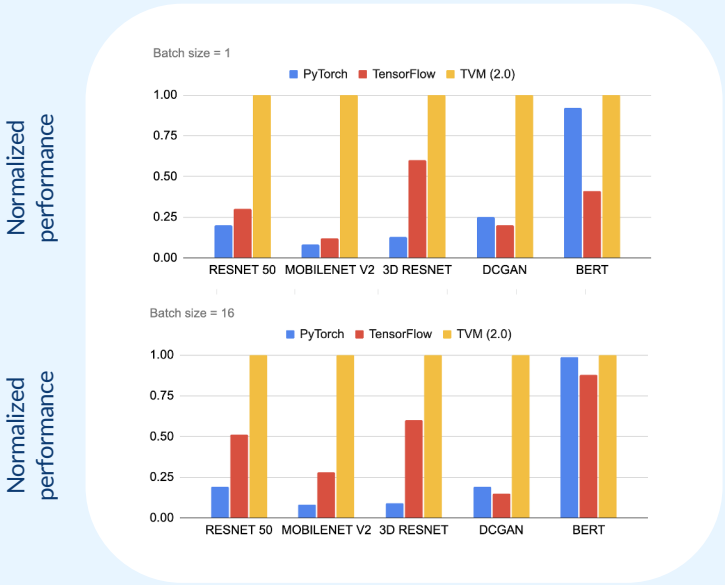




*V100 at hourly price of $3.00 per hour, T4 at $0.53

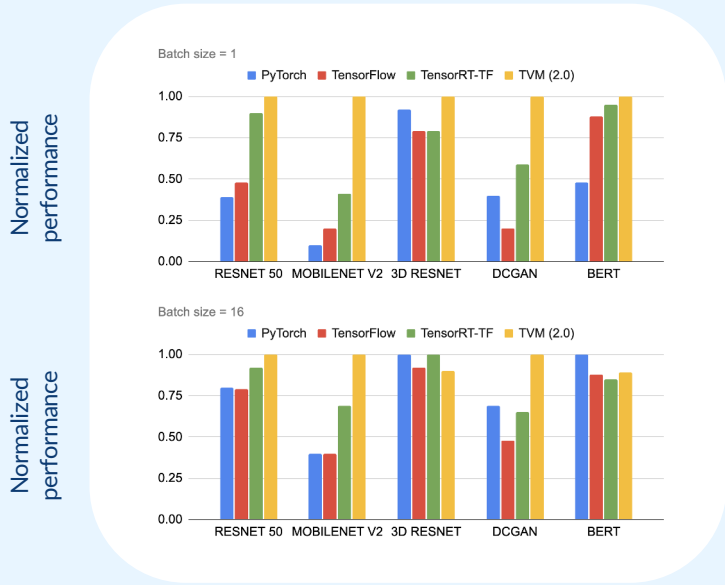
Results: TVM on CPU and GPU

Intel X86 - 2-5X Performance



20 core Intel-Platinum-8269CY fp32 performance data

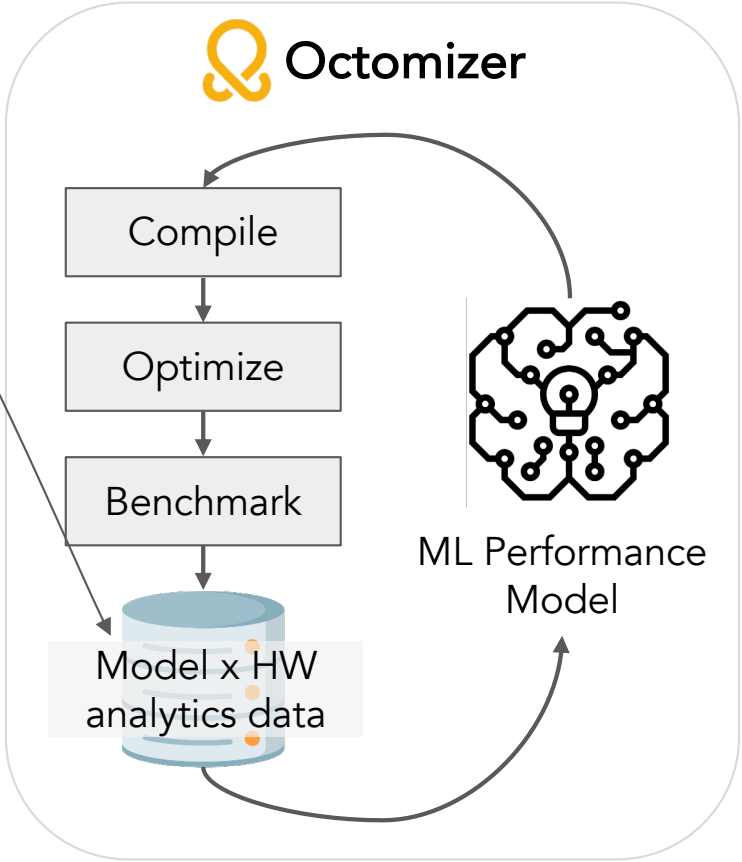
NVIDIA GPU - 20-50% versus TensorRT



V100 fp32 performance data

Why use the Octomizer vs “just” TVM OSS?

- Access to OctoML’s “cost models”
  - We aggregate Models x HW data
  - Continuous improvement
- No need to install any SW, latest TVM
- No need to set up benchmarking HW
- “Outer loop” automation
  - optimize/package multiple models against many HW targets in one go
- Access to comprehensive benchmarking data
  - E.g., for procurement, for HW vendor competitive analysis
- Access to OctoML support



Octomizer Live Demo

API access

```python
# Specify model file and input layer parameters.
model_file = "mnist.onnx"
input_shapes = {"Input3": (1, 3, 28, 28)}
input_dtypes = {"Input3": "float32"}
model = onnx_model.ONNXModel(client, input_shapes, input_dtypes)

# Upload the file to Octomizer.
modelvar = model.upload("mnist.onnx")

# Octomize it.
wrkflow = modelvar.octomize(platform="broadwell")
wrkflow.wait()
wrkflow.save_package("mnist-octomized.whl")
```

Waitlist! octoml.ai

TOYOTA Narrative rebellion AMD verizon
iRobot Raytheon XILINX CREDIT SUISSE citi AB InBev
Texas Instruments selcom BOSCH Honeywell LG SYNOPSYS

The Octonauts!

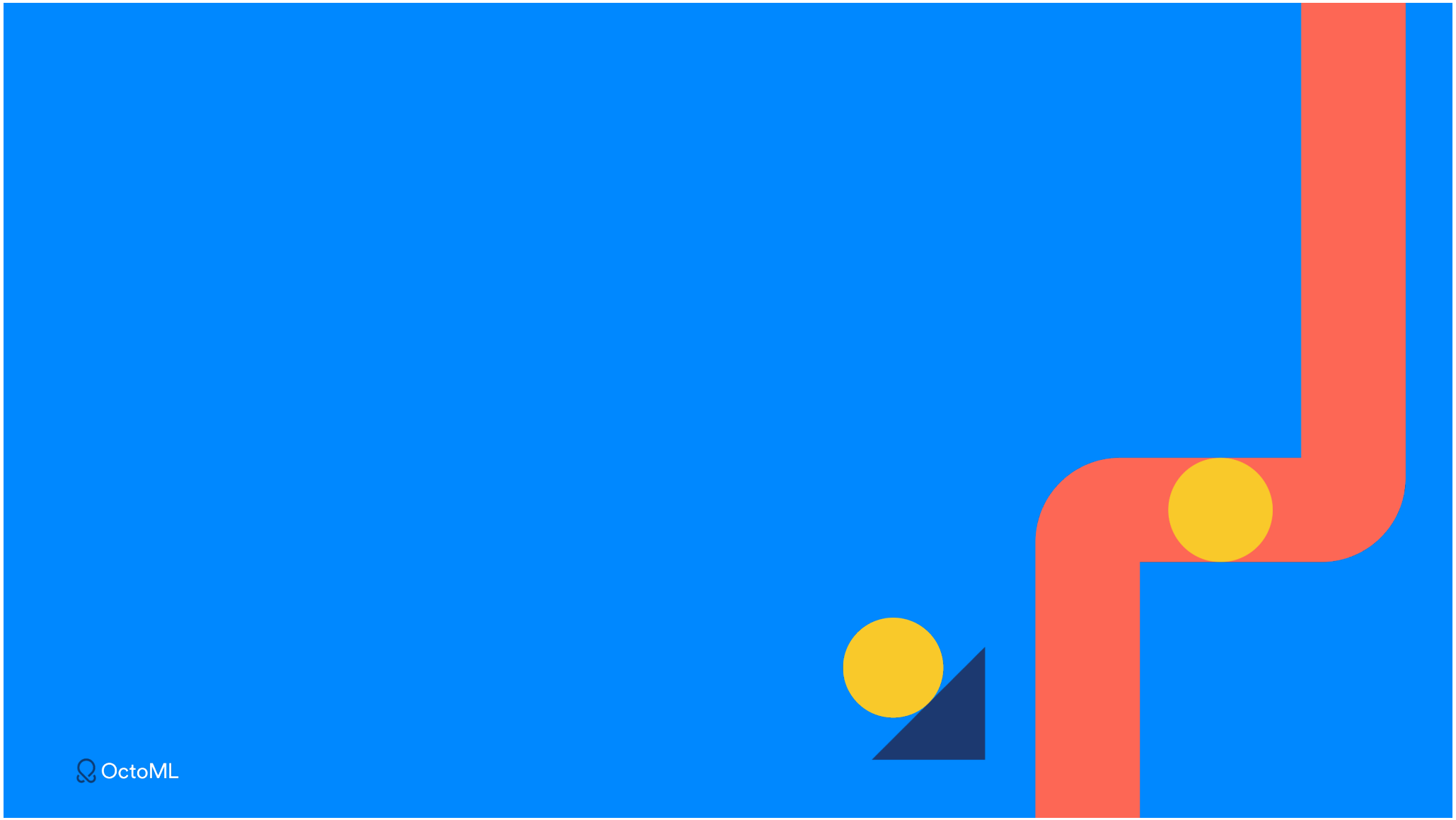
View career opportunities at
octoml.ai/careers

Thank you!

How to use Apache TVM to optimize your ML models

By Sameer Farooqui

OctoML


OctoML