Learning Systems 2018:
Lecture **10** – Performance and Benchmarking

Crescat scientia; vita excolatur

Ian Foster and Rick Stevens
Argonne National Laboratory
The University of Chicago

Topics for Today

- Announcements
- Assignment #2
- Reprise on Compilers
- PlaidML
- Estimating Performance
- Profiling
- Benchmarking (time to accuracy)

Announcements

- Project final proposals due 11:59 pm Friday Nov 2nd
- Assignment #2 two due 11:59 pm Friday Nov 16th
- Monday – Guest Lecture from Charlie Catlett PI for Array of Things project (AI at the Edge, Sensors, Urban Science)
- Weds – Guest Lecture Venkat Vishwanath – Data Science and Deep Learning at Scale (supercomputing meets deep learning)
- Sam is holding office hours for anyone needing help with project proposalsl

Assignment #2

Assignment #2

Goal: to gain experience evaluating the performance of models, frameworks and hardware architectures

- Pick two frameworks (e.g., TensorFlow, Pytorch, Mxnet, PlaidML, CNTK, Keras/X, Keras/Y)
- Pick two different models (e.g., VGG, ResNet, MobileNet) for which pretrained version of imagenet are available
- Build a custom version of the Kaggle “cats and dogs” classification problem <https://www.kaggle.com/c/dogs-vs-cats> (which re-trains known image models) on some classification task of your choice
- You have to choose some image topics to gather for classification using the retuned model(s)
- Measure the performance of training *and* inference of the models (for two different models (e.g. vgg vs. resnet) and two different frameworks (e.g. TF vs Pytorch) on CPU, GPU and TPU or something else (e.g., your laptop; goal is to compare two different platforms to CPUs for a total of three platforms)
- Write up: Explain and account for the performance differences in terms of architecture elements, software stack, framework implementation strategy, and model structure

Most frameworks have examples of the “dogs and cats” Kaggle challenge

- You need to gather a few hundred images for the topic(s) of your choice
- (e.g. Flowers, Cars, Airplanes, Fish, etc.)

- The labels are typically derived from the directory structure in some cases you might have to partition between training and test sets, use imagemagick to produce things at similar sizes and resolutions for training

- -- dogsandcats
- -------dogs
- ---------- (1000 pictures of dogs)
- -------cats
- ---------- (1000 pictures of cats)

Transfer Learning

- Reusing a model trained on one task to do another task by tuning or retraining an existing model on the target task
- Freezing parts of the network and updating a subset of the weights
  - reuse lower levels trained on large datasets to generate high-level representation and retrain to classify that representation
  - Encoding Layers + Classification Layers $\Longrightarrow$ Prediction

CPUs, GPUs and TPUs

- Google Colaboratory can be configured to run on CPU, GPU or TPU
  - Models can be trained on one platform and run in inference mode on a different platform
  - One could also try to export/import models from one framework to another using things like ONNX
- Not all frameworks have been optimized to run on TPUs
  - Your choice of frameworks to evaluate might depend on which hardware you want to focus on
- Some laptops have GPUs (NVIDIA under Cuda, others under the PlaidML drivers)
- Other devices are interesting as well especially for inferencing
  - Raspberry PI
  - Movidius Neural Compute Stick
  - ARM processors
  - Etc.

How to Measure Performance?

- Wall clock time to complete a task?
- Just time for training? (many runs and average?)
- Just time for inferencing? One or many samples?
- Train a fixed network and fixed datasets for fixed number of epochs?
  - Vary Hardware and Vary Framework
- Train a fixed network and fixed datasets to given accuracy/loss value?
  - Vary Hardware and Vary Framework
- Inference latency? (7 ms rule)
- What about hyper parameters? What about precision? What about I/O rates?, memory bandwidth?, memory capacity/Caching?, versions of frameworks?, libraries?, python?

Model One

Platform	F1-Training	F1-Inference	F2-Training	F2-Inference
P1 (CPU)				
P2 (GPU)				
P3 (TPU)				

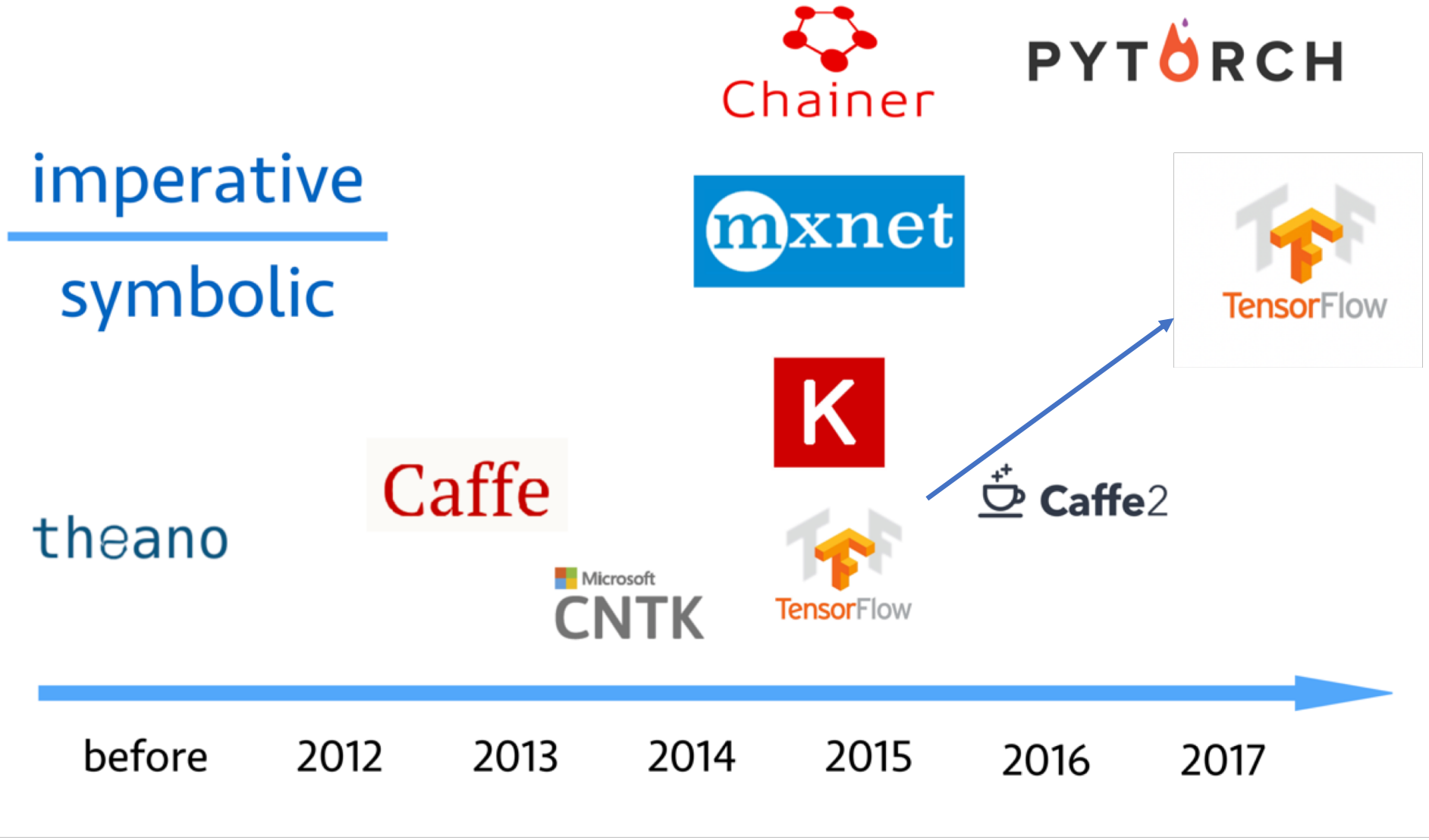
Model Two

Platform	F1-Training	F1-Inference	F2-Training	F2-Inference
P1 (CPU)				
P2 (GPU)				
P3 (TPU)				

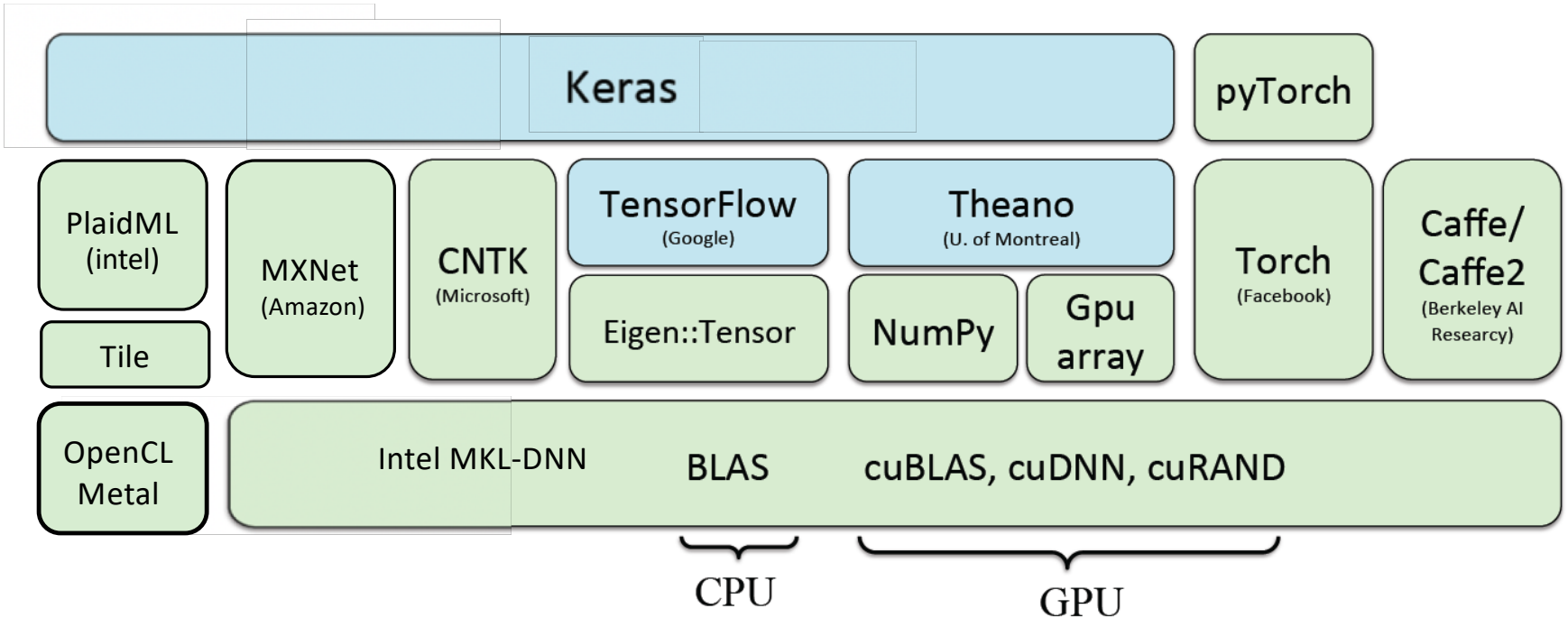
Understanding Performance

- Baseline vs test cases (baseline could be CPU)
- Sources of variability in performance
  - Hardware structure and rates of execution (memory BW and operations)
  - Utilization of hardware (occupancy, or efficiency)
  - Effective compilers or code generators
  - Instruction level parallelism
  - Optimization of low level libraries
  - Size of work units (matrix and vectors)
  - Reuse of work (back prop has opportunities)
  - Expression of the network and operations on the network
  - I/O rates and buffering

Reprise on Compilers


Chainer
PYTORCH
imperative
symbolic
mxnet
TensorFlow
K
Caffe
Caffe2
theano
Microsoft
CNTK
TensorFlow
before
2012
2013
2014
2015
2016
2017

Deep Learning Software Stacks



How to get high-performance on many platforms?

- If you are a platform vendor you want to provide high-performance libraries for your hardware
  - CUDA → cuDNN etc.
  - z86 → MKL-DNN
  - AMD/GPU

A platform for making deep learning work everywhere.

Vertex.AI (the creators of PlaidML) is excited to join Intel's Artificial Intelligence Products Group. PlaidML will soon be re-licensed under Apache 2. Read the announcement here!

PlaidML is the *easiest, fastest* way to learn and deploy deep learning on any device, especially those running macOS or Windows:

- **Fastest:** PlaidML is often 10x faster (or more) than popular platforms (like TensorFlow CPU) because it supports all GPUs, *independent of make and model*.
  - PlaidML accelerates deep learning on AMD, Intel, NVIDIA, ARM, and embedded GPUs.
- **Easiest:** PlaidML is simple to install and supports multiple frontends (Keras and ONNX currently)
- **Free:** PlaidML is completely open source and doesn't rely on any vendor libraries with proprietary and restrictive licenses.

For most platforms, getting started with accelerated deep learning is as easy as running a few commands (assuming you have Python (v2 or v3) installed (if this doesn't work, see the installation instructions):

```
virtualenv plaidml
source plaidml/bin/activate
pip install plaidml-keras plaidbench
```

Automatic Kernel Generation in PlaidML

May 19, 2018 | By: Jeremy Bruestle

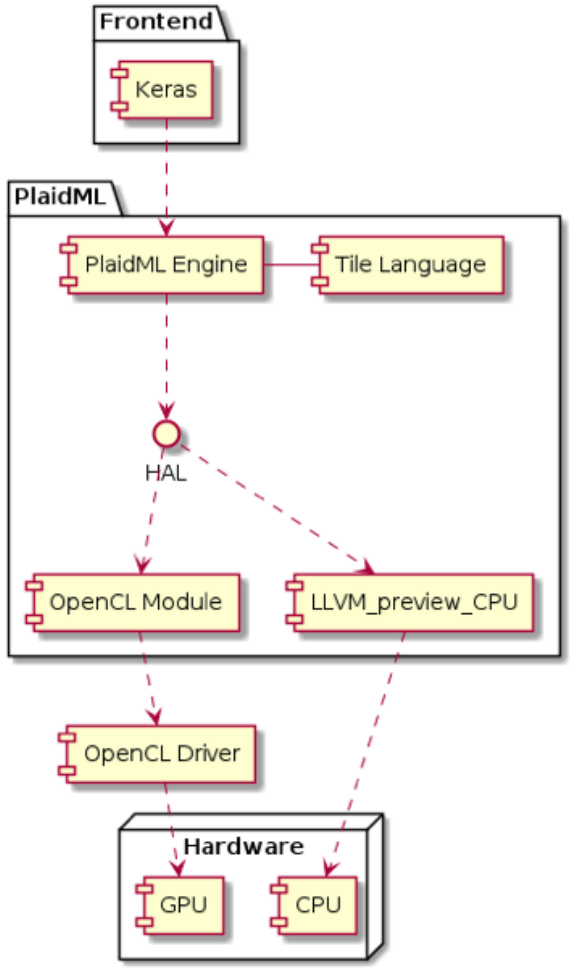
Historically, an engineer-intensive aspect of developing a machine learning backend was producing efficient device kernels from mathematical expressions of tensor operations. Practitioners wishing to utilize cutting edge research kernels in their networks needed to either wait for this development cycle to complete or rely on order-of-magnitude slower CPU implementations. Now, PlaidML, NNVM/TVM, and Tensor Comprehensions generate efficient kernels automatically from tensor expressions, bypassing this costly bottleneck. Below we'll give an overview of how PlaidML transforms an operation requested by a graph in a frontend (such as Keras or ONNX) into an optimized OpenCL kernel.

We'll also take a detailed look at the unique ways that PlaidML automatically performs several key aspects of generating efficiently parallelized kernels, including streamlining cache performance and minimizing edge-case conditionals.

Comments on Hacker News

Several core steps of our kernel generation algorithms are to optimize caching. We *flatten* Tile code, converting multidimensional tensor indices into pointer offsets. We split large tensors into *tiles* that can be cached in their entirety, and pick *tile sizes* that optimize these tiles for the specific hardware being used. We *layout* these tiles within the cache to minimize bank conflicts and maximize the provision of data to multiply accumulators. We *thread* the execution of the kernel to take advantage of the GPU's SIMD processor. And finally in *codegen* we construct a Semtree representation of the kernel we will generate.

Throughout this article we will follow a real example: A batched 3×3 convolution with "same" padding followed by a ReLU. This operation is simple enough to provide a clear concrete example, while complex enough to demonstrate both how PlaidML addresses the core challenges of kernel generation and also how PlaidML approaches common complications, including padding and the fusion of multiple operations into a single kernel. In addition, this case is a straightforward operation whose performance is nonetheless frequently critical to overall network performance.



Tile Language

$$O[n] = \sum_m I[m, n]$$

```
O[n: N] = +(I[m, n]);
```

$$O[n] = \max_m (I[m, n])$$

```
O[n: N] = >(I[m, n]);
```

Matrix Multiplication

Next we'll consider matrix multiplication. Let's look at the mathematical expression for the matrix multiplication `C = AB` written out in element-level detail:

$$C[i, j] = \sum_k \left(A[i, k] \cdot B[k, j]\right)$$

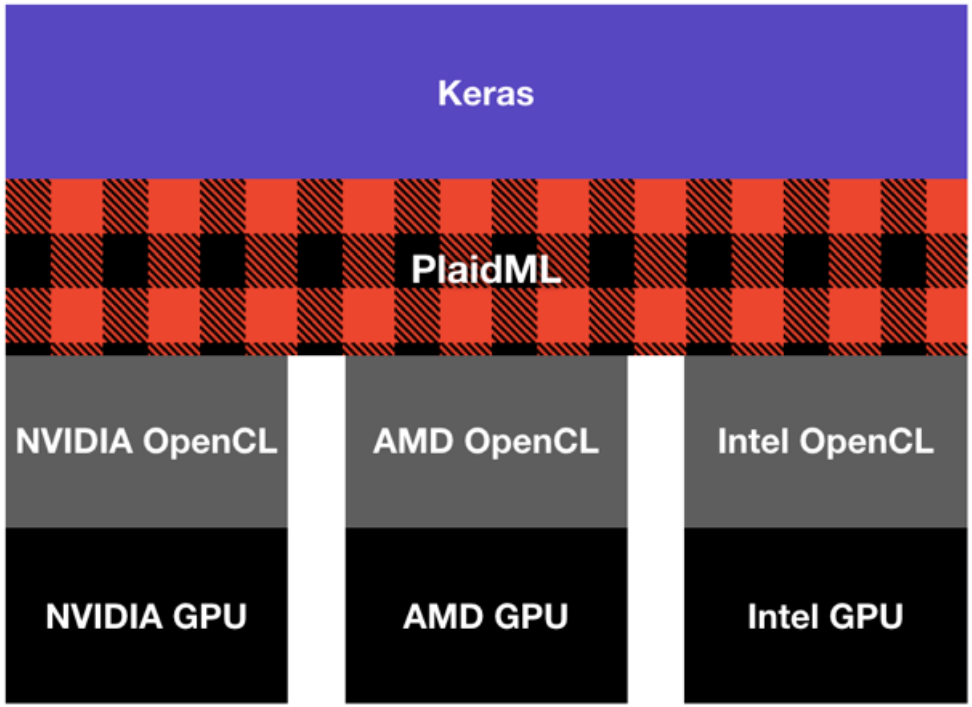
We can convert this to Tile code using the same correspondence as the previous example: The summation sign becomes plus, the summation index is omitted, dimensions are given for the output tensor, and the statement ends in a semicolon. Here's the result:

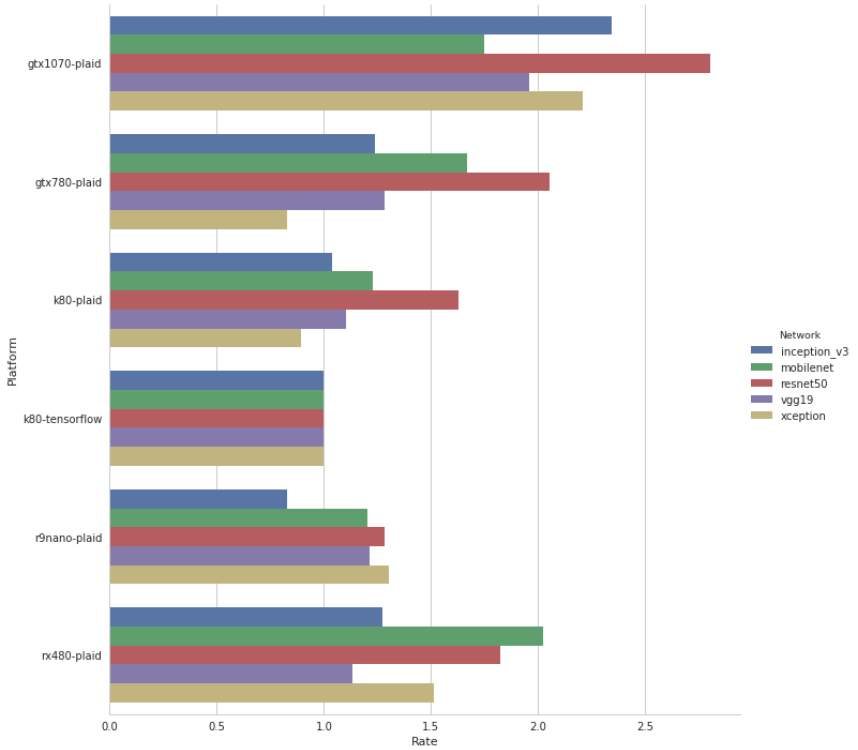
```
C[i, j: M, N] = +(A[i, k] * B[k, j]);
```

To have the dimensions correct, we need `M` to be the first dimension of `A` and `N` the last dimension of `B`. Here's how this looks as part of a full Tile function:

```
function (A[M, L], B[L, N]) -> (C) {
    C[i, j: M, N] = +(A[i, k] * B[k, j]);
}
```

PlaidML – targeting all platforms





PlaidML

Validated Hardware

Vertex.AI runs a comprehensive set of tests for each release against these hardware targets:

- AMD
  - R9 Nano
  - RX 480
  - Vega 10
- NVIDIA
  - K80, GTX 780, GT 640M
  - GTX 1070, 1050
- Intel
  - HD4000
  - HD Graphics 505

PlaidML

Validated Networks

We support all of the Keras application networks from current versions of 2.x. Validated networks are tested for performance and correctness as part of our continuous integration system.

- CNNs
  - Inception v3
  - ResNet50
  - VGG19
  - Xception
  - MobileNet
  - DenseNet
  - ShuffleNet
- LSTM
  - examples/imdb_lstm.py (from keras)

Hello VGG

One of the great things about Keras is how easy it is to play with state of the art networks. Here's all the code you need to run VGG-19:

```
#!/usr/bin/env python
import numpy as np
import os
import time

os.environ["KERAS_BACKEND"] = "plaidml.keras.backend"

import keras
import keras.applications as kapp
from keras.datasets import cifar10

(x_train, y_train_cats), (x_test, y_test_cats) = cifar10.load_data()
batch_size = 8
x_train = x_train[:batch_size]
x_train = np.repeat(np.repeat(x_train, 7, axis=1), 7, axis=2)
model = kapp.VGG19()
model.compile(optimizer='sgd', loss='categorical_crossentropy',
              metrics=['accuracy'])

print("Running initial batch (compiling tile program)")
y = model.predict(x=x_train, batch_size=batch_size)

# Now start the clock and run 10 batches
print("Timing inference...")
start = time.time()
for i in range(10):
    y = model.predict(x=x_train, batch_size=batch_size)
print("Ran in {} seconds".format(time.time() - start))
```

Installation

To get the basic framework and command-line interface:

```
pip install plaidbench
```

If you know which ML frontends you'll want to use, you can install their pre-requisites ahead of time:

```
pip install plaidbench[keras]
pip install plaidbench[onnx]
```

You can also install various ML backends -- for example,

```
pip install plaidml-keras
pip install tensorflow
pip install caffe2
pip install onnx-plaidml
pip install onnx-tf
```

If you don't have a particular package installed, and you run benchmarks that require the package, Plaidbench will try to determine what needs to be installed and tell you how to install it.

If you're using PlaidML as a backend, you'll want to run `plaidml-setup` to configure it correctly for your hardware.

Usage

Plaidbench provides a simple command-line interface; global flags are provided immediately, and subcommands are used to select the frontend framework and to provide framework-specific options.

For example, to benchmark ShuffleNet on ONNX using PlaidML, writing results to the directory `~/shuffle_results`, you can run:

```
plaidbench --result ~/shuffle_results onnx --plaid shufflenet
```

For a complete overview of the supported global flags, use `plaidbench --help`; for the individual subcommand flags, specify `--help` with the subcommand (e.g. `plaidbench keras --help`).

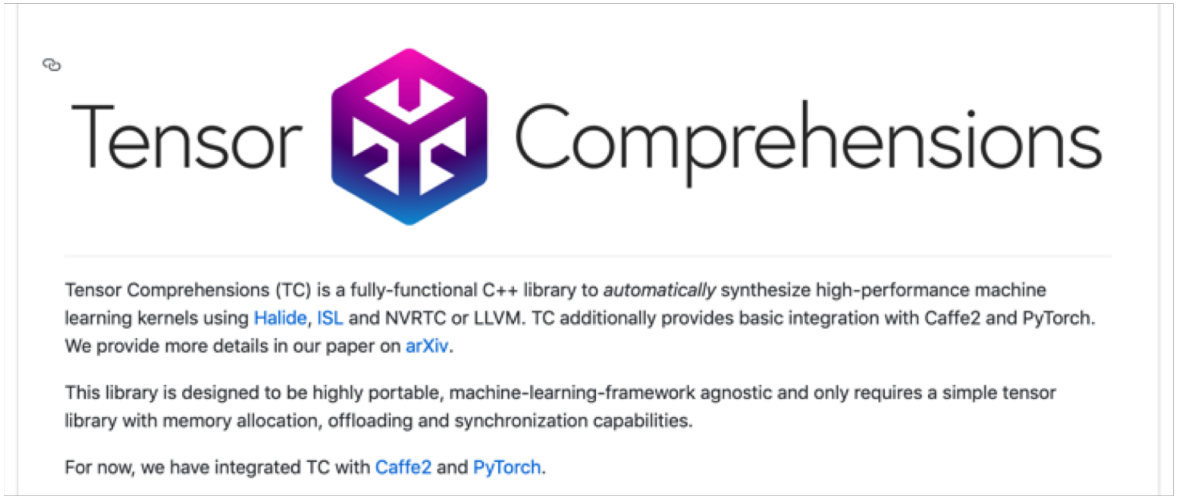
Supported Configurations

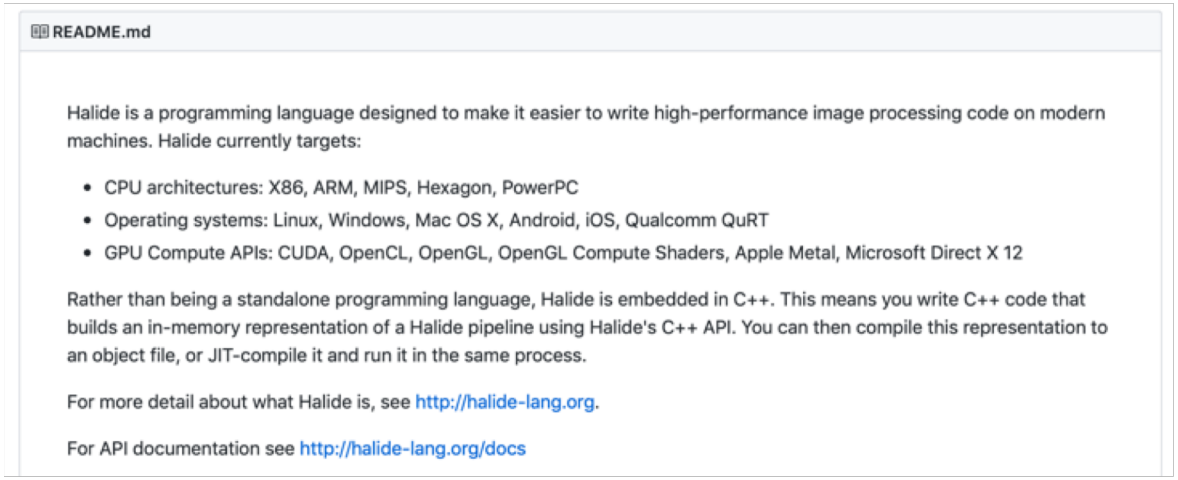
Plaidbench supports:

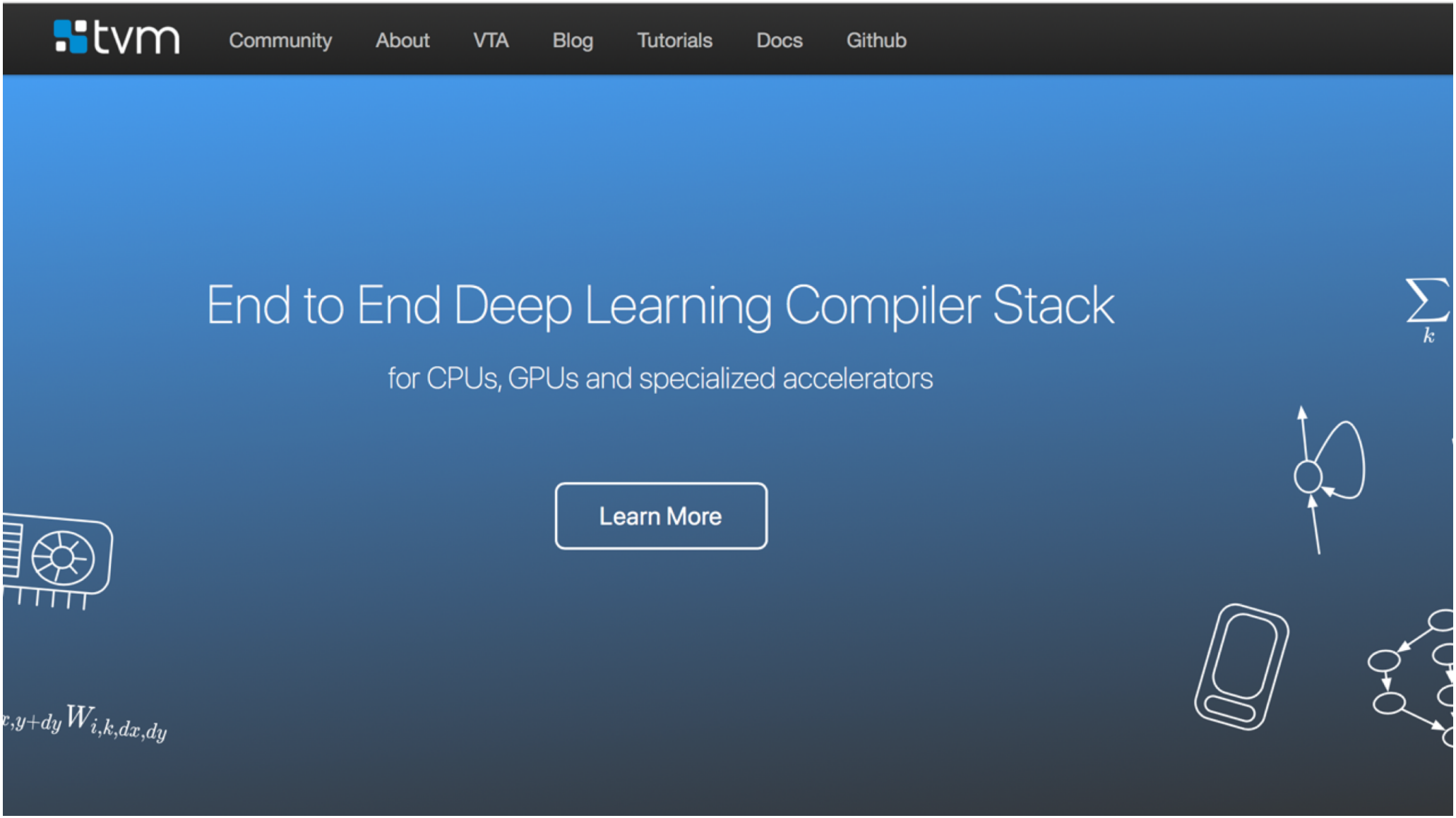
- Keras
  - Backends: PlaidML and Tensorflow
  - Networks: Inception-V3, ResNet50, Vgg16, Vgg19, Xception, and (in Keras 2.0.6 and later) MobileNet.
  - Training vs. Inference performance, and fp16 vs. fp32 performance.
- ONNX
  - Backends: PlaidML, Caffe2, and Tensorflow
  - Networks: AlexNet, DenseNet, Inception-V1, Inception-V2, Resnet50, ShuffleNet, SqueezeNet, Vgg16, and Vgg19.

Comparisons

- TVM
- Tensor Comprehensions
- PlaidML

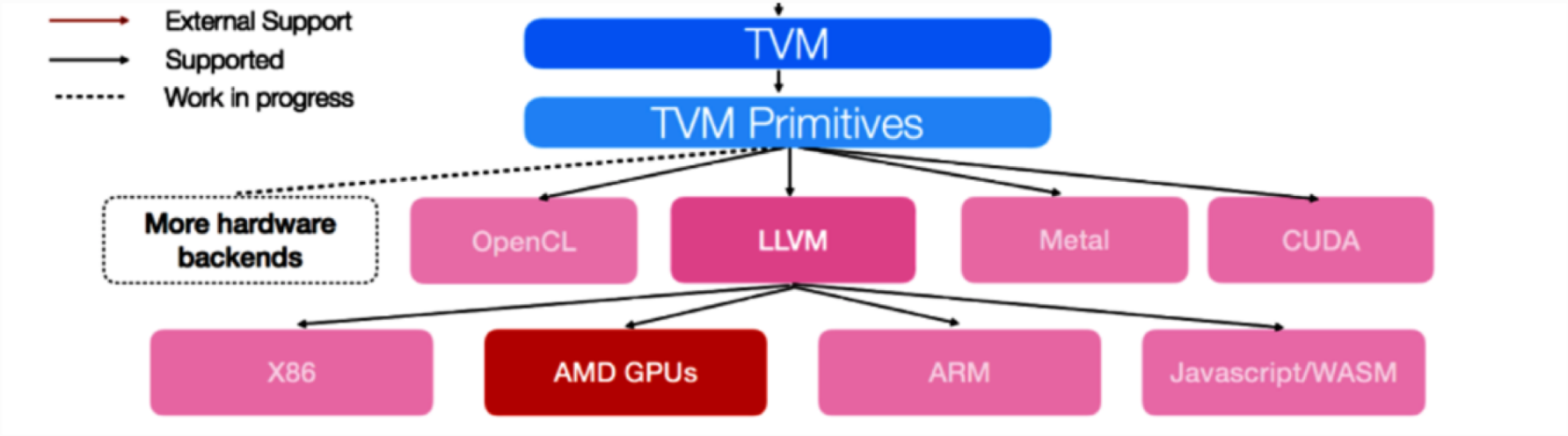




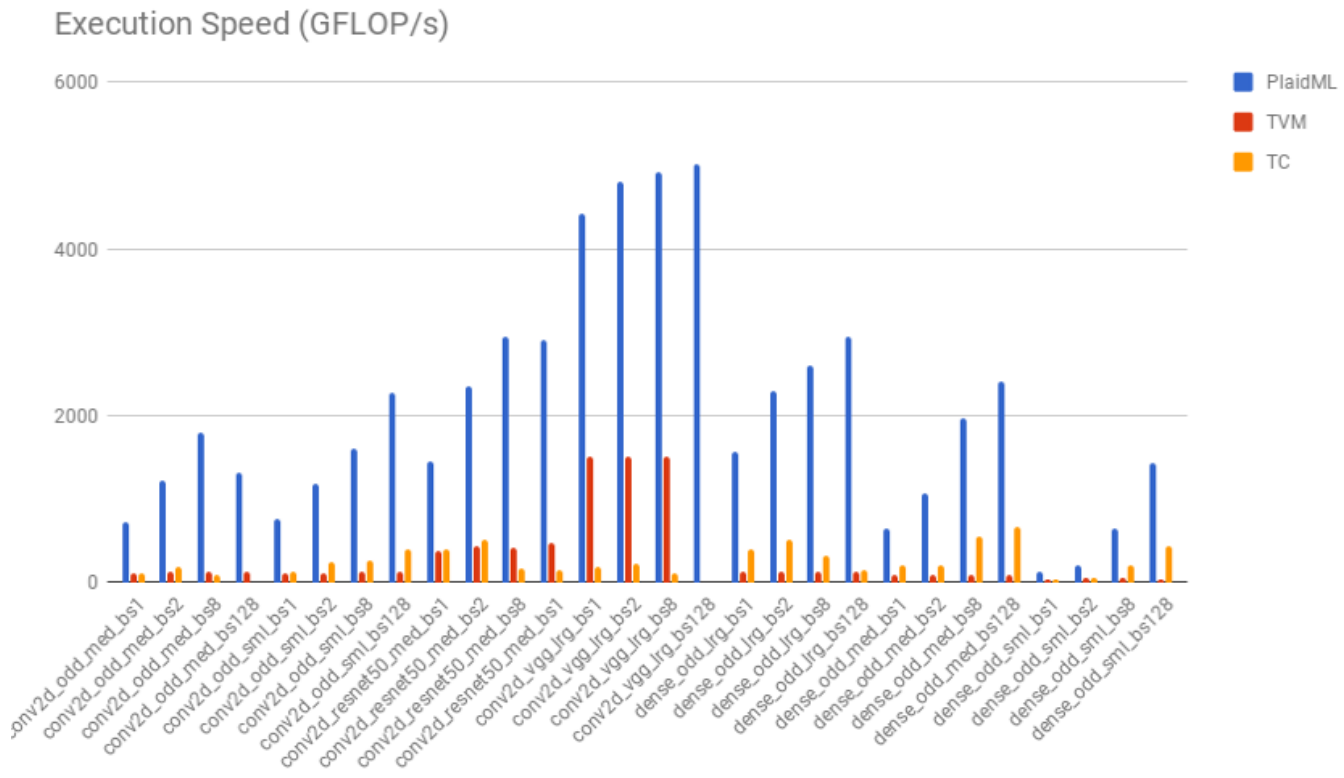

tvm
Community
About
VTA
Blog
Tutorials
Docs
Github
End to End Deep Learning Compiler Stack
for CPUs, GPUs and specialized accelerators
Learn More

Overview for Supported Hardware Backend of TVM

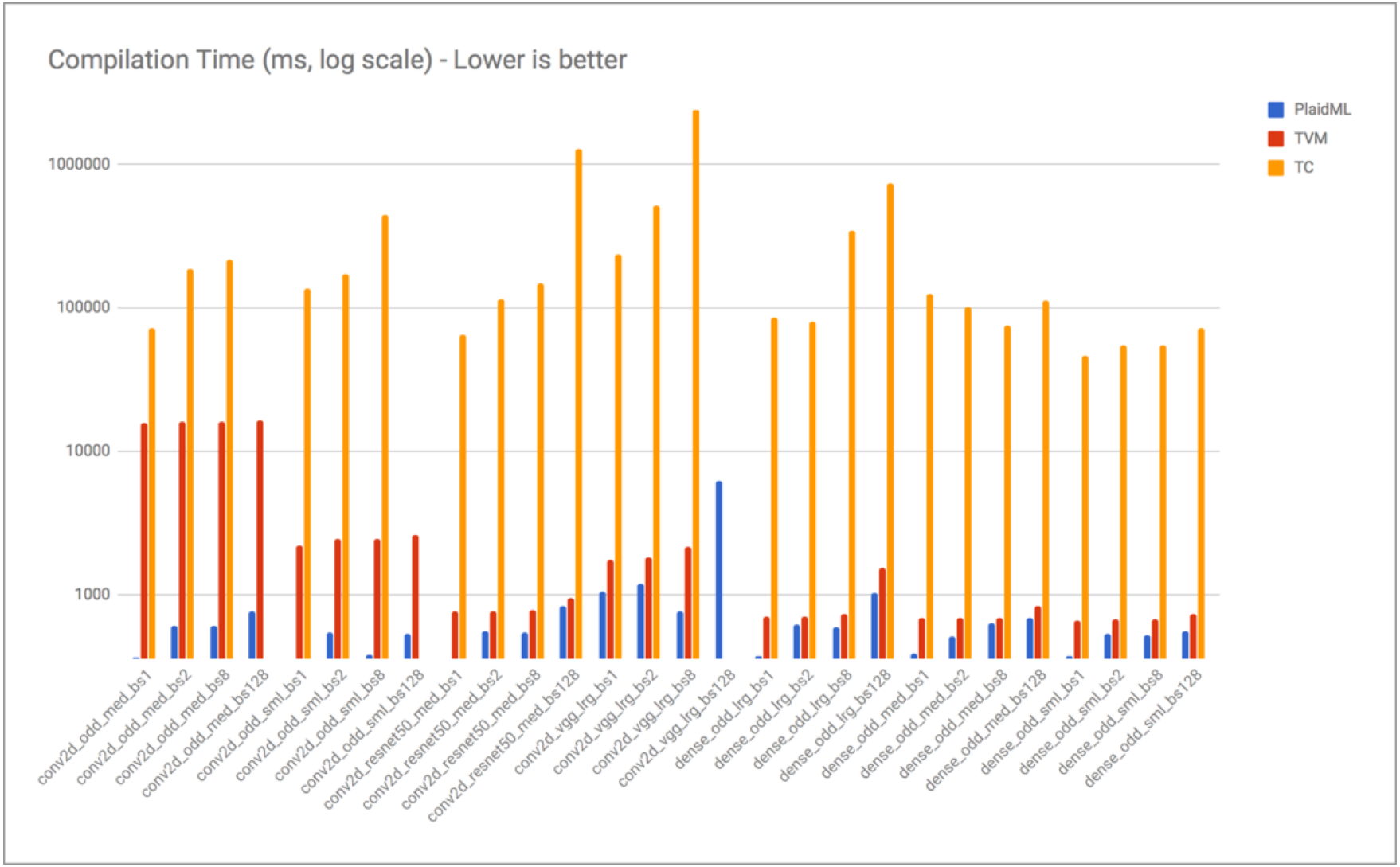
The image below shows hardware backend currently supported by TVM:



PlaidML compiler comparison



http://vertex.ai/blog/compiler-comparison

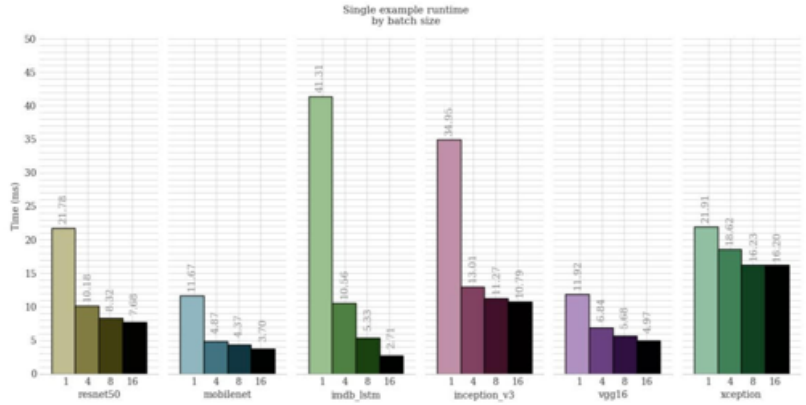

Compilation Time (ms, log scale) - Lower is better
PlaidML
TVM
TC
1000000
100000
10000
1000
conv2d_odd_med_bs1
conv2d_odd_med_bs2
conv2d_odd_med_bs8
conv2d_odd_med_bs128
conv2d_odd_sml_bs1
conv2d_odd_sml_bs2
conv2d_odd_sml_bs8
conv2d_odd_sml_bs128
conv2d_resnet50_med_bs1
conv2d_resnet50_med_bs2
conv2d_resnet50_med_bs8
conv2d_resnet50_med_bs128
conv2d_vgg_lrg_bs1
conv2d_vgg_lrg_bs2
conv2d_vgg_lrg_bs8
conv2d_vgg_lrg_bs128
dense_odd_lrg_bs1
dense_odd_lrg_bs2
dense_odd_lrg_bs8
dense_odd_lrg_bs128
dense_odd_med_bs1
dense_odd_med_bs2
dense_odd_med_bs8
dense_odd_med_bs128
dense_odd_sml_bs1
dense_odd_sml_bs2
dense_odd_sml_bs8
dense_odd_sml_bs128

Compiler Comparisons

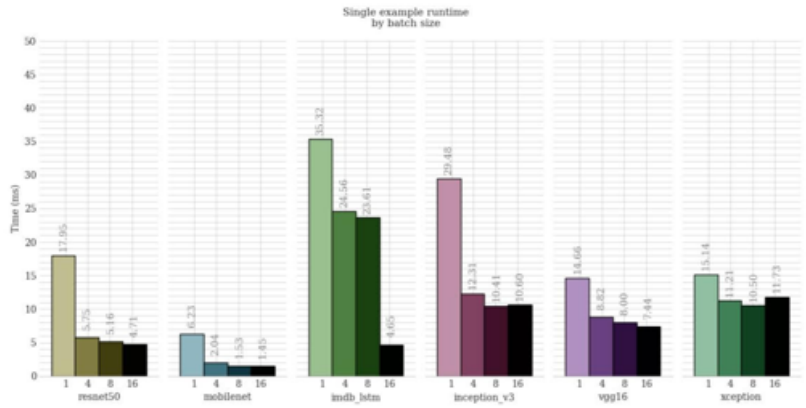
	PlaidML	TVM	TC
Autotuner Speed	< 1 sec / kernel	No	~30 min
Automatic Operation Gradients	Yes	No	No
Supported Networks	All Major Nets	Inference only	Single Block
Dilation	Yes	Yes	Rescales Filters
Padding	Yes	Yes	No
Striding	Yes	Yes	Inference only
Complex LHS / Reduction	Yes	No	No
Drivers Supported	CUDA, OpenCL, OpenGL, Metal, LLVM	CUDA, OpenCL, Metal, ROCm, WebGL	CUDA

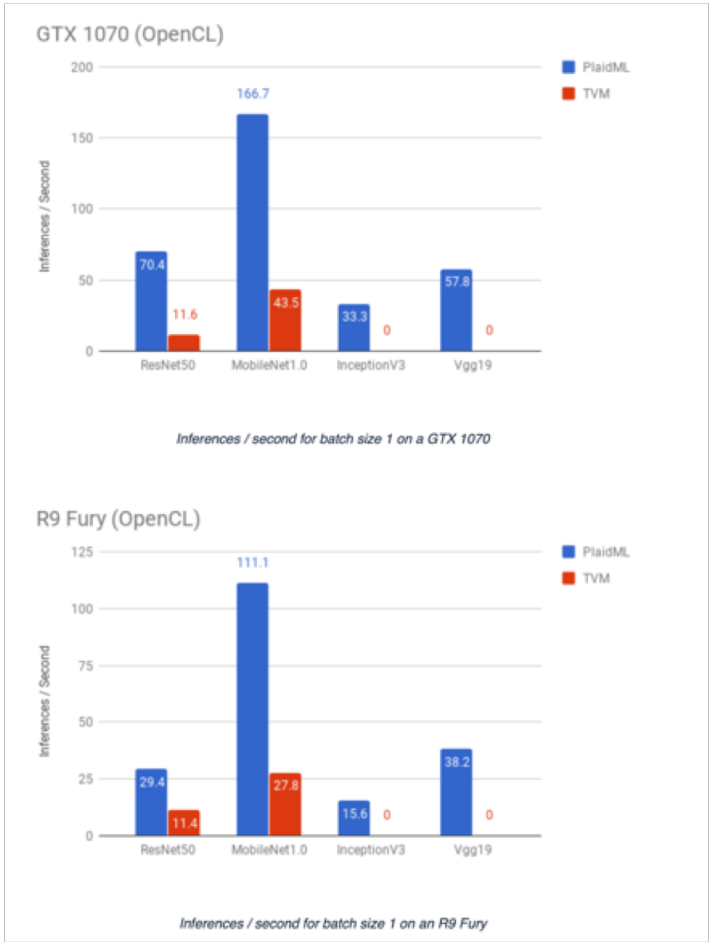
NVIDIA GTX 1070

TensorFlow 1.8



PlaidML 0.3.2



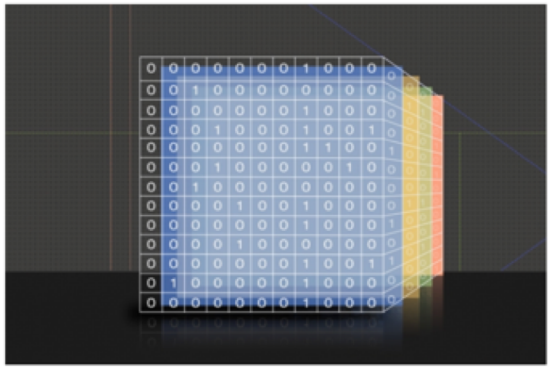


The Tensor Algebra Compiler (taco)

Docs Publications Demo GitHub

A fast and versatile library for linear and tensor algebra

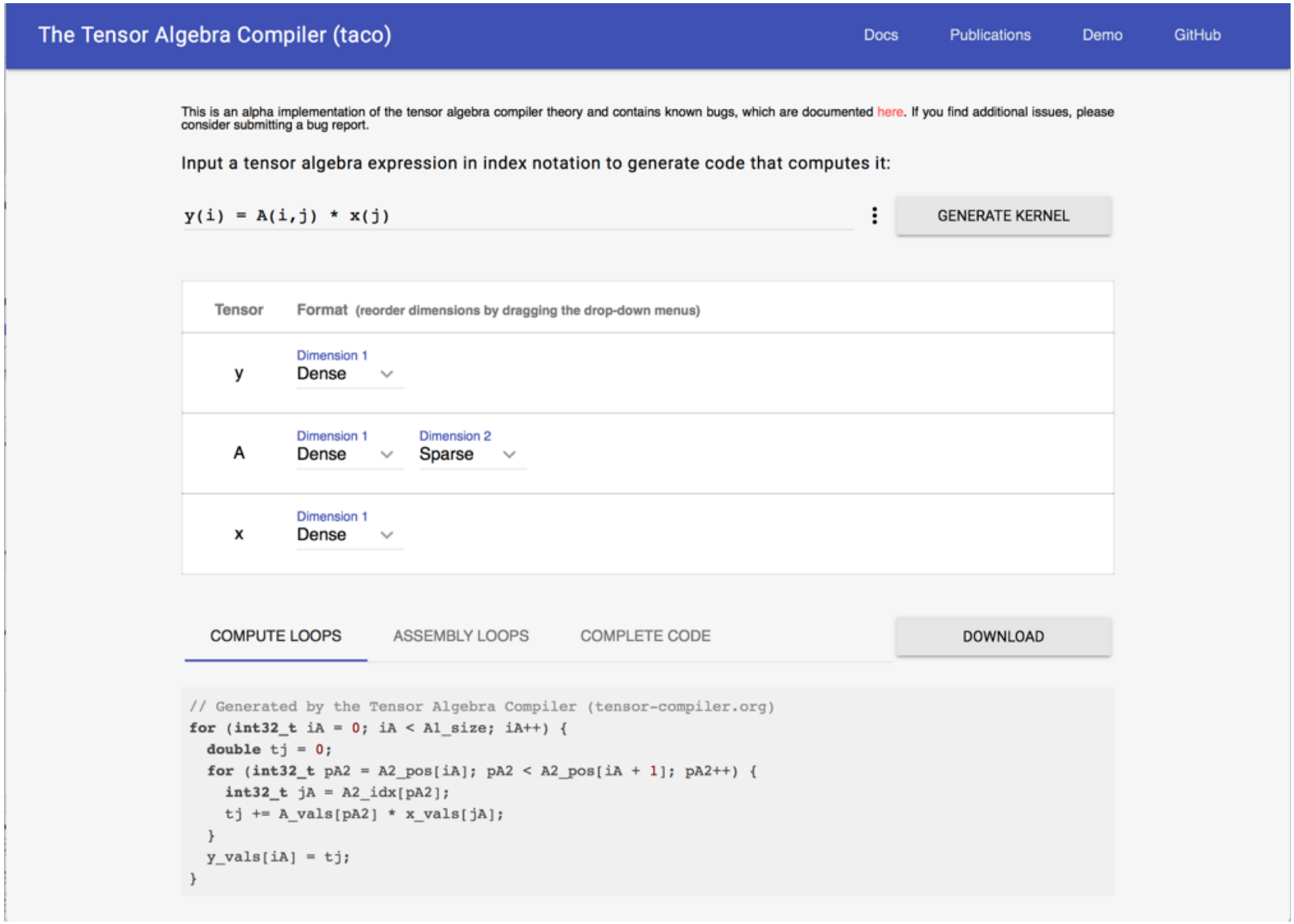
TRY ONLINE GITHUB



We're in the news!

READ THE MIT NEWS ARTICLE

WATCH OUR TALK AT OOPSLA 2017


The Tensor Algebra Compiler (taco)
Docs
Publications
Demo
GitHub
This is an alpha implementation of the tensor algebra compiler theory and contains known bugs, which are documented here. If you find additional issues, please consider submitting a bug report.
Input a tensor algebra expression in index notation to generate code that computes it:
y(i) = A(i,j) * x(j)
GENERATE KERNEL
Tensor
Format (reorder dimensions by dragging the drop-down menus)
y
Dimension 1
Dense
A
Dimension 1
Dense
Dimension 2
Sparse
x
Dimension 1
Dense
COMPUTE LOOPS
ASSEMBLY LOOPS
COMPLETE CODE
DOWNLOAD
// Generated by the Tensor Algebra Compiler (tensor-compiler.org)
for (int32_t iA = 0; iA < A1_size; iA++) {
double tj = 0;
for (int32_t pA2 = A2_pos[iA]; pA2 < A2_pos[iA + 1]; pA2++) {
int32_t jA = A2_idx[pA2];
tj += A_vals[pA2] * x_vals[jA];
}
y_vals[iA] = tj;
}

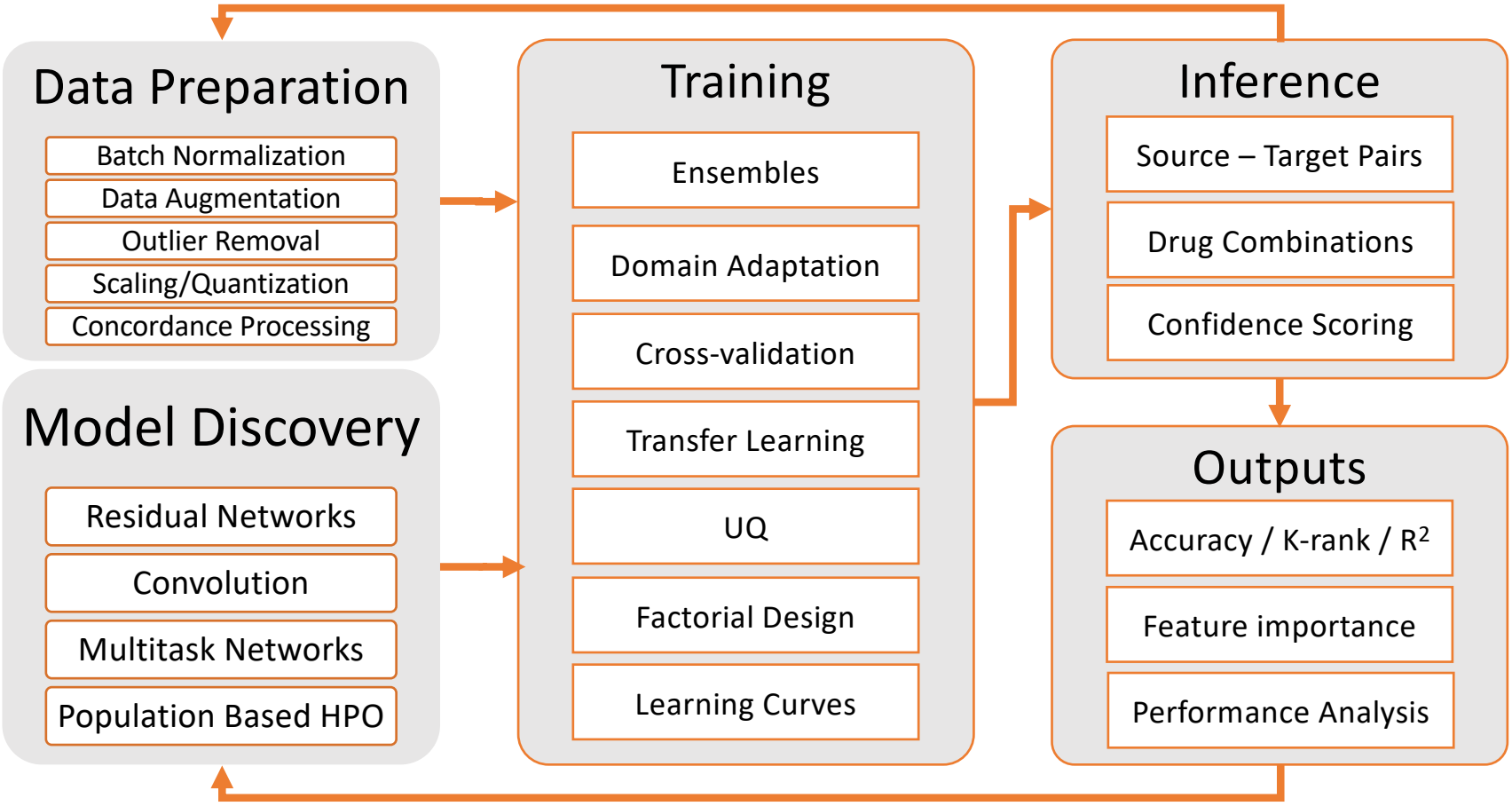
Estimating Performance

CANDLE Challenge Problem Workflow Structure

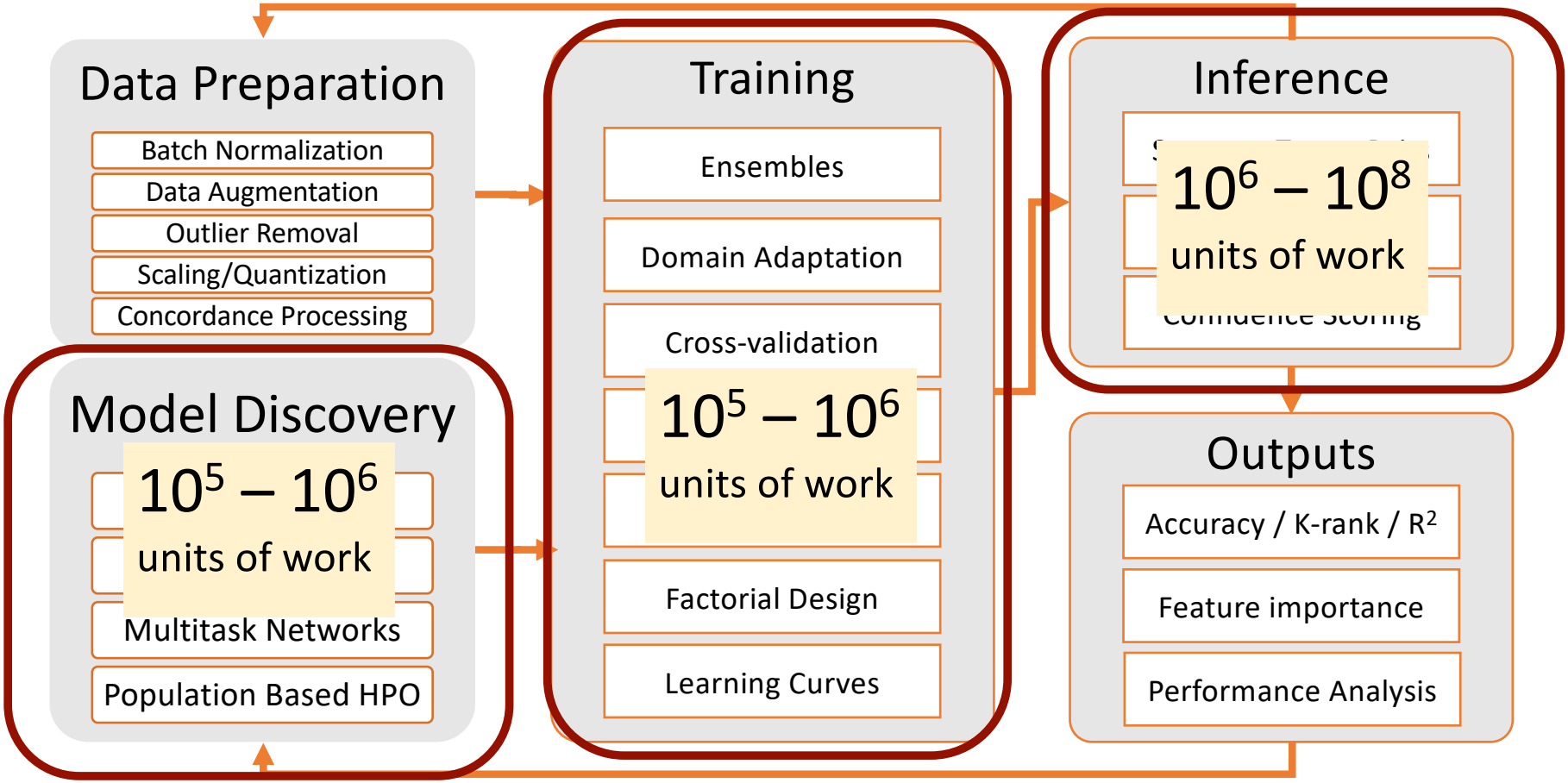
Each CP has three compute intensive phases and two data processing phases

1. Prepare input datasets, including normalization, outlier removal, joins and merges, etc.
2. Broad search of model space (AutoML+HPO) to find good performing model structures and hyper-parameter settings $\Longrightarrow$ running $O(10^4)$ – $O(10^6)$ model instances (HPO model selection $\Longrightarrow$ small number of model templates < 10)
3. Training a large-scale ensemble of the best model types on relevant training data for a "factorial study" $\Longrightarrow$ $O(10^6)$ models on $O(10^4)$ datasets (5x cross validation and optional bootstrap UQ) (CV model selection $\Longrightarrow$ small number of models < 10 per Source-Target pair)
4. Inferencing with selected models with UQ on new samples ($O(10^7)$ in (and out) of each Source-Target Pair distributions (UQ implies sampling model space $O(10^2)$ times)
5. Post-process inference output for actionable decisions and to capture performance, confidence and scoring (for validation)

P1 Challenge Problem Workflow(s) Specification



P1 Challenge Problem Workflow(s) Specification



Has the project defined a quantitative figure of merit (FOM) and established a "reasonable" baseline (i.e. non-trivial, representative of the final challenge problem) on leadership class systems like Mira or Titan?

- Yes. We have established a FOM based on throughput of units of work in our Challenge Problem workflows
- Each Pilot has constructed a number of “benchmark” models that represent typical work in the challenge problem
- These benchmarks are being measured on pre-exascale platforms and are being used in discussions with vendors on performance estimation
- The benchmarks are close approximations to the key models that are at the heart of the challenge problems

CANDLE Figure of Merit (FOM)

- The figure of merit for CANDLE is the average “rate” we can complete units of work per day for the challenge problem workflows.
- Each Pilot has defined a representative computation (typically training a model) that represents a reasonable “unit” of work in the CP workflow
- CANDLE FOM is the average rate at which we can complete these units of work on Exascale systems divided by the average rate we can complete these same tasks on Titan.
- Since our FOM is a rate ratio we use Harmonic Mean to compute average rates between the three Challenge Problems:

$$H = 3*X_1X_2X_3/(X_1X_2 + X_1X_3 + X_2X_3)$$

Titan Nodes ~ 18,688 A Nodelets ~ 94,000

Example FOM Calculation

- P1 unit 60 hrs to train on Titan, 24/60 * 18,688 = 7,475 u/day
- P1 unit 4 hrs to train on A “nodelet”, 24/4 * 94,000 = 564,000 u/day
- P1 Ratio = 564,000 / 7,475 = 75.45x

- P2 unit 6 hrs to train on Titan, 24/6 * 18,688 = 74,752 u/day
- P2 unit 0.5 hrs to train on A “nodelet”, 24/0.5 * 94,000 = *4,512,000 u*/day
- P2 Ratio = 4,512,000/ 74,752 = 60.36x

- P3 unit 12 hrs to train on Titan, 24/12 * 18,688 = 37,376 u/day
- P3 unit .25 hrs to train on A “nodelet”, 24/0.25 * 94,000 = 9,024,000 u/day
- P3 Ratio = 9,024,000 / 37,376 = 241.43x

- $H_{titan} = \frac{1\,676\,313\,600}{97\,177} \approx 17\,250.1$ $H_A = \frac{27\,072\,000}{19} \approx 1.42484 \times 10^6$

- H_A / H_{titan} = FOM = $\frac{68\,509\,785}{829\,426} \approx 82.599$ vs Mean[75.45, 60.36, 241.43] = 125.747

Method for Estimating Performance Ratio

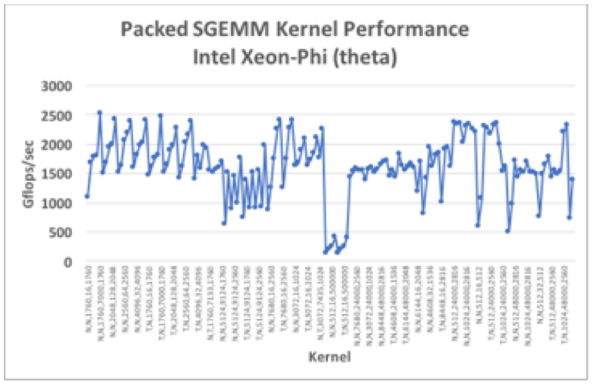
- Run Training for N Epochs and record training time
- Profile using TF.profiler the Keras baseline code
- Extract from profile top operations (95+% time)
- Extract from profile “Matrix” Sizes and times for each operation
- Run DeepBench on target platform (or compute matrix operation time estimates from target simulators)
- Extrapolate Matrix operation time from DeepBench examples
- Roll up time estimate per “pass”
- Multiply time of a pass x number of minibatches $\Longrightarrow$ run time for Epoch
- Multiply run time of a epoch x number of epochs $\Longrightarrow$ time per run
- Divide time in a day by time per run
- Multiply number of runs per day per node x number of nodes $\Longrightarrow$ units per day for Exascale
- Normalize Exscale rate by Titan throughput per day $\Longrightarrow$ ratio

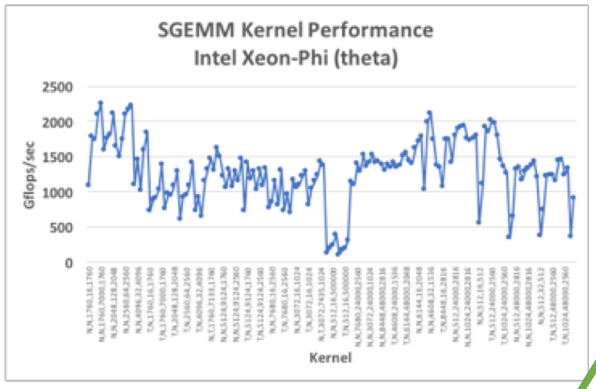
P1B3 Operation Profile

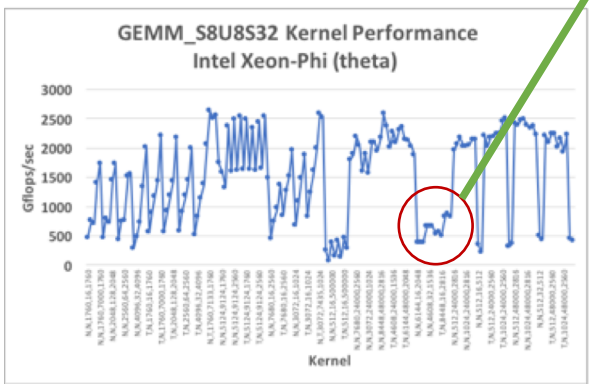
Operation	Size	Sum	Fraction	Time	Sum-Time	Fract-Time
MatMul	121.67MB	(100.00%,	33.33%),	115.52ms	(100.00%,	40.21%),
Assign	0B	(0.00%,	0.00%),	73.65ms	(59.79%,	25.64%),
Mul	121.02MB	(66.67%,	33.15%),	61.46ms	(34.15%,	21.39%),
Add	120.35MB	(33.52%,	32.97%),	22.31ms	(12.75%,	7.77%),
Sub	0B	(0.00%,	0.00%),	2.22ms	(4.98%,	0.77%),
RandomUniform	660.00KB	(0.54%,	0.18%),	1.75ms	(4.21%,	0.61%),
RealDiv	660.40KB	(0.36%,	0.18%),	1.75ms	(3.60%,	0.61%),
BiasAdd	0B	(0.00%,	0.00%),	1.19ms	(3.00%,	0.41%),
ReluGrad	0B	(0.00%,	0.00%),	1.11ms	(2.58%,	0.39%),
BiasAddGrad	6.60KB	(0.18%,	0.00%),	1.00ms	(2.20%,	0.35%),
VariableV2	0B	(0.00%,	0.00%),	868us	(1.85%,	0.30%),
Fill	660.00KB	(0.18%,	0.18%),	849us	(1.55%,	0.30%),
Floor	0B	(0.00%,	0.00%),	848us	(1.25%,	0.30%),
Placeholder	0B	(0.00%,	0.00%),	0us	(0.00%,	0.00%),

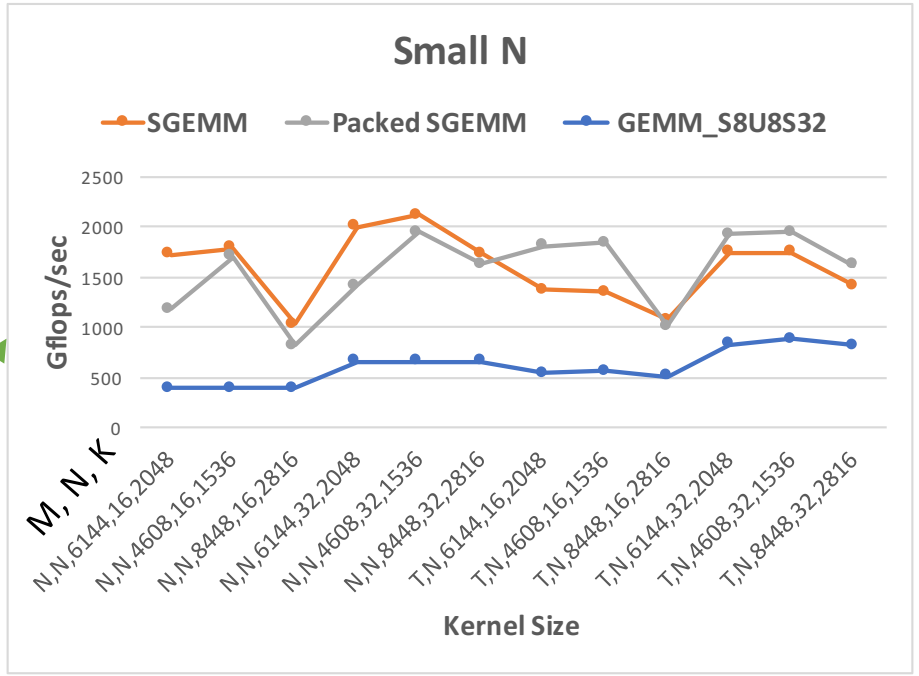
P1B3 Matrix Sizes and Times for One Pass on x86

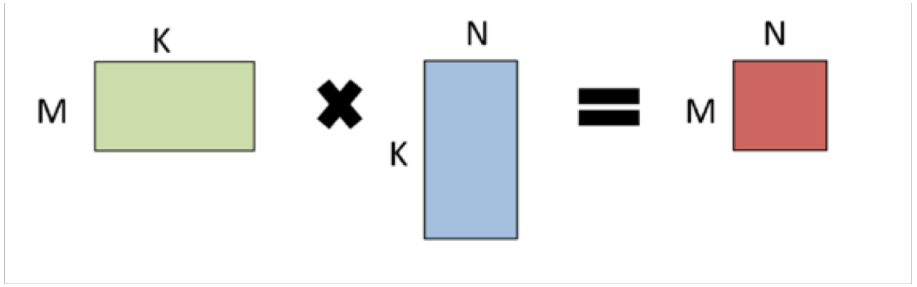
0:100x100,	1:100x50	(run*2|defined*2)	exec_time:	549us
0:-1x29532,	1:100x1000	(run*1|defined*1)	exec_time:	42.27ms
0:-1x29532,	1:29532x1000	(run*1|defined*1)	exec_time:	65.31ms
0:100x1,	1:50x1	(run*1|defined*1)	exec_time:	42us
0:100x100,	1:500x100	(run*1|defined*1)	exec_time:	609us
0:100x1000,	1:1000x500	(run*1|defined*1)	exec_time:	1.21ms
0:100x1000,	1:100x500	(run*1|defined*1)	exec_time:	1.98ms
0:100x50,	1:100x1	(run*1|defined*1)	exec_time:	26us
0:100x50,	1:100x50	(run*1|defined*1)	exec_time:	293us
0:100x50,	1:50x1	(run*1|defined*1)	exec_time:	22us
0:100x500,	1:1000x500	(run*1|defined*1)	exec_time:	2.11ms
0:100x500,	1:100x100	(run*1|defined*1)	exec_time:	592us
0:100x500,	1:500x100	(run*1|defined*1)	exec_time:	515us
0:100x1000,	1:29532x1000	(run*0|defined*1)	exec_time:	0us


Packed SGEMM Kernel Performance
Intel Xeon-Phi (theta)
Gflops/sec
3000
2500
2000
1500
1000
500
0
Kernel


SGEMM Kernel Performance
Intel Xeon-Phi (theta)
Gflops/sec
2500
2000
1500
1000
500
0
Kernel


GEMM_S8U8S32 Kernel Performance
Intel Xeon-Phi (theta)
Gflops/sec
3000
2500
2000
1500
1000
500
0
Kernel


Small N
SGEMM
Packed SGEMM
GEMM_S8U8S32
Gflops/sec
2500
2000
1500
1000
500
0
M, N, K
N,N,6144,16,2048
N,N,4608,16,1536
N,N,8448,16,2816
N,N,6144,32,2048
N,N,4608,32,1536
N,N,8448,32,2816
T,N,6144,16,2048
T,N,4608,16,1536
T,N,8448,16,2816
T,N,6144,32,2048
T,N,4608,32,1536
T,N,8448,32,2816
Kernel Size


K
N
N
M
K
M

What is the highest FOM achieved thus far? Is the project currently on track to hit the threshold value in KPP-1?

- Four of our benchmark codes are running at >7x Titan on single nodes and scaled to full Summit are at >10x Titan $\Longrightarrow$ estimated KPP-1 of >70x for Exascale
- One benchmark code is at 13.8x single node and estimated at 20.4x scaled to full Summit $\Longrightarrow$ estimated KPP-1 of 142x for Exascale
- Vendor estimates for CANDLE benchmarks range from 68x to 200x
- We believe a combination of increasing accelerator utilization by 30% relative to Titan or exploiting BFP16 or some combination of the two is what is required to hit or exceed 50x KPP-1 target
- On average we see increased utilization by a factor of ~2 in our measurements
- We believe CANDLE is on track to exceed the KPP-1 threshold perhaps by a significant amount

CANDLE Benchmark Performance

Benchmark	Batch Size	Parameters	Time per Epoch Single Node (sec)							
			P100 (1.19x)	V100 (3.97x)	KNL(1.5x)	Titan (1x)	A (7.6x)	A Ratio	B	B Ratio
P1B1	100	244,400,085	8 [4.8x]	7 [5.4x]	33 [1.15x]	[38]				
P1B2	60	29,540,618	2 (2x) (100%)	2 (2x)	6 (0.66)	4 /5 (64%)	0.056	71x		
P1B3	100	30,088,701	3060 (1.2x)(5%)	5987 (0.60)	7309 (0.49)	3612 (29%)	53.1	68x	18	~200x
TC1	20	154,923,632	76 (4.4x) (88%)	44 (7.7x)	1934 (0.17)*	338				
NT3	20	154,922,918	29 (2.9x)	11 (7.8x)	510 (0.16)*	86				
UNO	32	21,293,001	480 [8.4x]	610 [6.6x]	3774 [1.07]	[4041]				
COMBO	32	5,823,000	80 [14.7x]	73 [16.1]	1377 [0.85x]	[1177]				
P2B1	64	18,867,360	420 [9.0x]	300 [12.6x]	4250 [0.89x]	[3783]				
P3B1	10	33,164,419	2.037 (12.8x)	1.996 (13x)	15 (1.7x)	26	0.217	120x	0.13	~200x
P3B2	100	48,419	660 (3.8x)	1,210 (2.1x)	1708 (1.5x)	2,493				
P3B3	10	1,763,724	14 (6.2x)	10 (8.8x)	1149 (0.07)	88				

(2.9x) [6.4x] (6x) [7.5x] (0.67) [0.79x] Means (ops) [est]

Profiling

Profiling

- Where are we spending the time?
- What is the structure of the model as seen by the hardware
- tf.profile.profiling
- Graphical timelines and “(g)prof” level analysis
- Execution profile (by sampling and code analysis)
- Number of times each element of a graph is executed
- Type and Size of data types (scalars, vectors, matrices, etc.)
- First order estimate of performance
- Can help place you on the roofline

TensorFlow Profiler and Advisor

Features

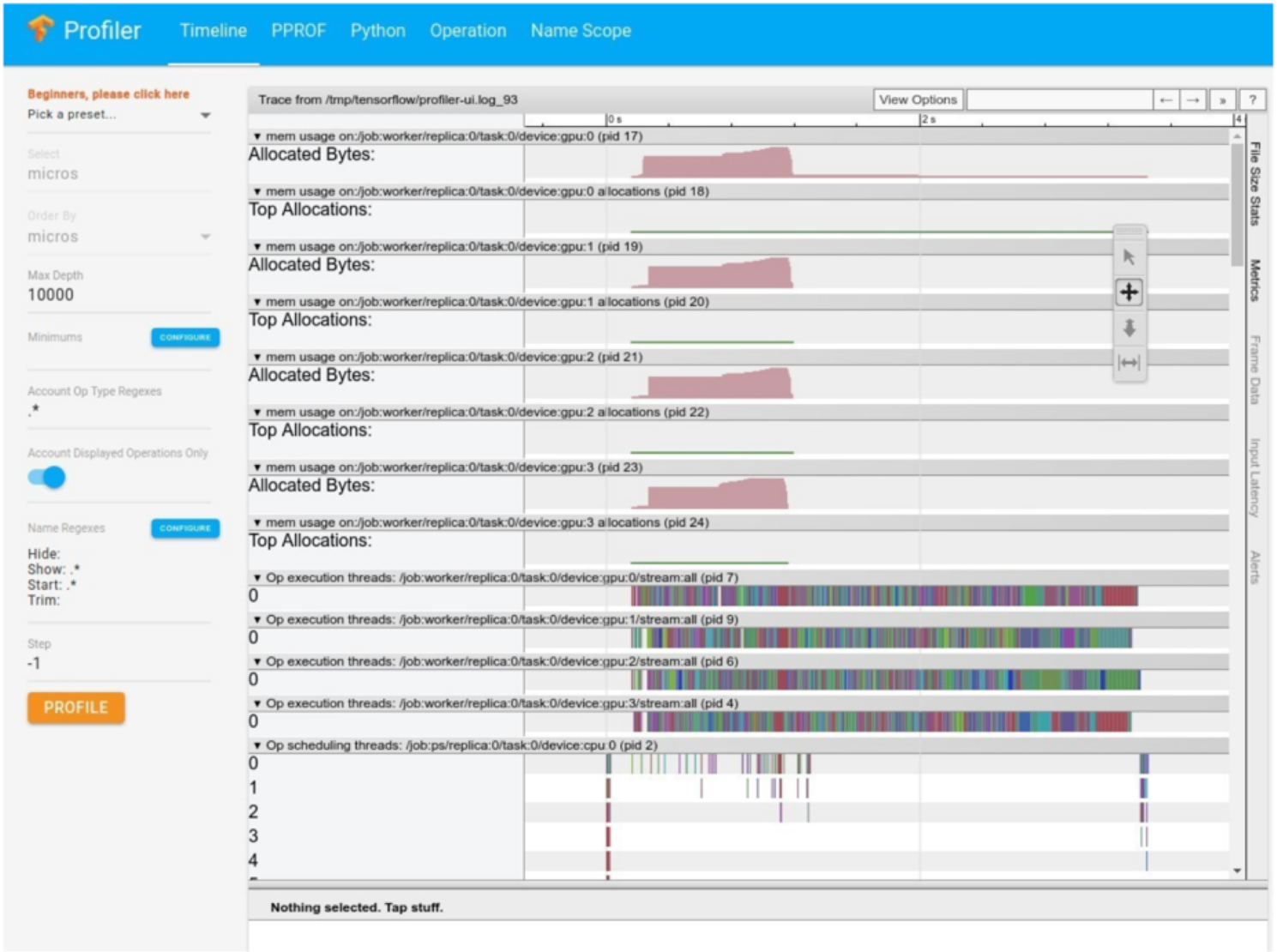
- Profile model architectures.
  - parameters, tensor shapes, float operations, device placement, etc.
- Profile multiple-steps model performance.
  - execution time, memory consumption.
- Auto profile and advise.
  - accelerator utilization check
  - expensive operation check
  - operation configuration check
  - distributed runtime check (Not OSS)

Quick Start

```
# When using high-level API, session is usually hidden.
#
# Under the default ProfileContext, run a few hundred steps.
# The ProfileContext will sample some steps and dump the profiles
# to files. Users can then use command line tool or Web UI for
# interactive profiling.
with tf.contrib.tfprof.ProfileContext('/tmp/train_dir') as pctx:
  # High level API, such as slim, Estimator, etc.
  train_loop()

bazel-bin/tensorflow/core/profiler/profiler \
    --profile_path=/tmp/train_dir/profile_xx
tfprof> op -select micros,bytes,occurrence -order_by micros

# To be open sourced...
bazel-bin/tensorflow/python/profiler/profiler_ui \
    --profile_path=/tmp/profiles/profile_1
```

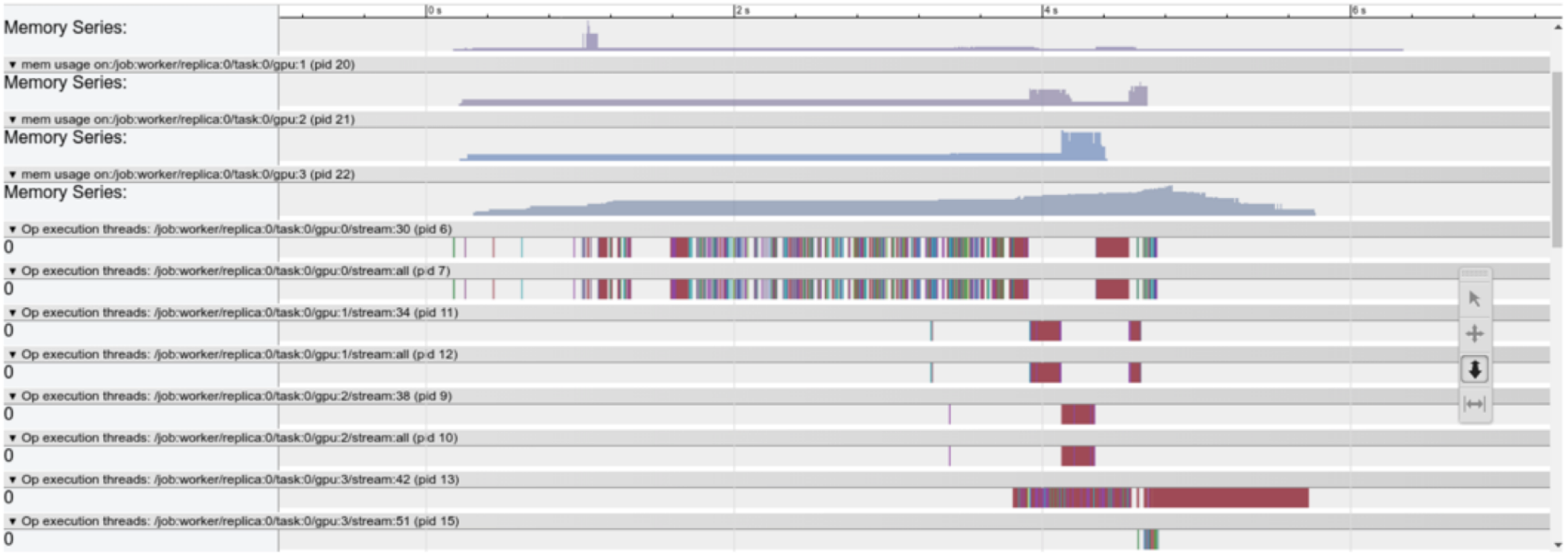

Profiler
Timeline
PPROF
Python
Operation
Name Scope
Beginners, please click here
Pick a preset...
Select
micros
Order By
micros
Max Depth
10000
Minimums
CONFIGURE
Account Op Type Regexes
.*
Account Displayed Operations Only
Name Regexes
CONFIGURE
Hide:
Show: .*
Start: .*
Trim:
Step
-1
PROFILE
Trace from /tmp/tensorflow/profiler-ui.log_93
View Options
0 s
2 s
4
mem usage on:/job:worker/replica:0/task:0/device:gpu:0 (pid 17)
Allocated Bytes:
mem usage on:/job:worker/replica:0/task:0/device:gpu:0 allocations (pid 18)
Top Allocations:
mem usage on:/job:worker/replica:0/task:0/device:gpu:1 (pid 19)
Allocated Bytes:
mem usage on:/job:worker/replica:0/task:0/device:gpu:1 allocations (pid 20)
Top Allocations:
mem usage on:/job:worker/replica:0/task:0/device:gpu:2 (pid 21)
Allocated Bytes:
mem usage on:/job:worker/replica:0/task:0/device:gpu:2 allocations (pid 22)
Top Allocations:
mem usage on:/job:worker/replica:0/task:0/device:gpu:3 (pid 23)
Allocated Bytes:
mem usage on:/job:worker/replica:0/task:0/device:gpu:3 allocations (pid 24)
Top Allocations:
Op execution threads: /job:worker/replica:0/task:0/device:gpu:0/stream:all (pid 7)
0
Op execution threads: /job:worker/replica:0/task:0/device:gpu:1/stream:all (pid 9)
0
Op execution threads: /job:worker/replica:0/task:0/device:gpu:2/stream:all (pid 6)
0
Op execution threads: /job:worker/replica:0/task:0/device:gpu:3/stream:all (pid 4)
0
Op scheduling threads: /job:ps/replica:0/task:0/device:cpu 0 (pid 2)
0
1
2
3
4
File Size Stats
Metrics
Frame Data
Input Latency
Alerts
Nothing selected. Tap stuff.

Visualize time and memory

```
# The following example generates a timeline.
tfprof> graph -step -1 -max_depth 100000 -output timeline:outfile=<filename>

generating trace file.

******************************************************
Timeline file is written to <filename>.
Open a Chrome browser, enter URL chrome://tracing and load the timeline file.
******************************************************
```



Attribute TensorFlow graph running time to your Python codes.

```
tfprof> code -max_depth 1000 -show_name_regexes .*model_analyzer.*py.* -select micros -account_type_rege
_TFProfRoot (0us/22.44ms)
  model_analyzer_test.py:149:run_filename_as_m...:none (0us/22.44ms)
    model_analyzer_test.py:33:_run_code_in_main:none (0us/22.44ms)
      model_analyzer_test.py:208:<module>:test.main() (0us/22.44ms)
        model_analyzer_test.py:132:testComplexCodeView:x = lib.BuildFull... (0us/22.44ms)
          model_analyzer_testlib.py:63:BuildFullModel:return sgd_op.min... (0us/21.83ms)
          model_analyzer_testlib.py:58:BuildFullModel:cell, array_ops.c... (0us/333us)
          model_analyzer_testlib.py:54:BuildFullModel:seq.append(array_... (0us/254us)
            model_analyzer_testlib.py:42:BuildSmallModel:x = nn_ops.conv2d... (0us/134us)
            model_analyzer_testlib.py:46:BuildSmallModel:initializer=init_... (0us/40us)
            ...
          model_analyzer_testlib.py:61:BuildFullModel:loss = nn_ops.l2_... (0us/28us)
          model_analyzer_testlib.py:60:BuildFullModel:target = array_op... (0us/0us)
        model_analyzer_test.py:134:testComplexCodeView:sess.run(variable... (0us/0us)
```

Show your model variables and the number of parameters.

```
tfprof> scope -account_type_regexes VariableV2 -max_depth 4 -select params
_TFProfRoot (--/930.58k params)
  global_step (1/1 params)
  init/init_conv/DW (3x3x3x16, 432/864 params)
  pool_logit/DW (64x10, 640/1.28k params)
    pool_logit/DW/Momentum (64x10, 640/640 params)
  pool_logit/biases (10, 10/20 params)
    pool_logit/biases/Momentum (10, 10/10 params)
  unit_last/final_bn/beta (64, 64/128 params)
  unit_last/final_bn/gamma (64, 64/128 params)
  unit_last/final_bn/moving_mean (64, 64/64 params)
  unit_last/final_bn/moving_variance (64, 64/64 params)
```

Show the most expensive operation types.

```
tfprof> op -select micros,bytes,occurrence -order_by micros
node name | requested bytes | total execution time | accelerator execution time | cpu execution time | c
SoftmaxCrossEntropyWithLogits      36.58MB (100.00%, 0.05%),      1.37sec (100.00%, 26.68%),           0
MatMul                          2720.57MB (99.95%, 3.66%),      708.14ms (73.32%, 13.83%),     280.76ms (1
ConcatV2                         741.37MB (96.29%, 1.00%),       389.63ms (59.49%, 7.61%),        31.80ms
Mul                             3957.24MB (95.29%, 5.33%),       338.02ms (51.88%, 6.60%),       80.88ms (
Add                              740.05MB (89.96%, 1.00%),       321.76ms (45.28%, 6.28%),        13.50ms
Sub                               32.46MB (88.97%, 0.04%),       216.20ms (39.00%, 4.22%),          241us
Slice                            708.07MB (88.92%, 0.95%),       179.88ms (34.78%, 3.51%),        25.38ms
AddN                             733.21MB (87.97%, 0.99%),       158.36ms (31.26%, 3.09%),        50.10ms
Fill                             954.27MB (86.98%, 1.28%),       138.29ms (28.17%, 2.70%),        16.21ms
Select                           312.33MB (85.70%, 0.42%),       104.75ms (25.47%, 2.05%),        18.30ms
ApplyAdam                        231.65MB (85.28%, 0.31%),        92.66ms (23.43%, 1.81%),            0us
```

Benchmarking

Table 1: CNN models

Layer	AlexNet	VGGNet	GoogLeNet	ResNet
CONV	5	13	57	53
POOL	3	5	14	2
NORM	2		2	53
ReLU	7	15	57	49
FC	3	3	1	1
Concat			9	
Scale				53
Eltwise				16
Total	20	36	140	227

Table 2: Timing benchmarks on AlexNet

Platform		Layerwise Pass (ms) CONV	POOL	LRN	ReLU	FC	Total (ms)	Forward Pass (ms)
TK1	CPU	318.7±0.2	6.1±0.1	103.8±0.0	4.6±0.0	186.3±0.1	619.8±0.2	619.5±0.2
		51.42%	0.99%	16.74%	0.75%	30.05%		
	GPU	24.6±3.5	2.3±0.6	2.4±0.5	5.2±1.2	35.1±5.9	73.3±10.7	54.7±2.4
		33.53%	3.15%	3.22%	7.11%	**47.95%**		
TX1	CPU	66.9±5.3	7.6±0.0	172.4±0.3	2.4±0.0	644.7±5.3	894.3±4.8	892.7±2.3
		7.48%	0.85%	19.28%	0.27%	**72.09%**		
	GPU	24.2±8.3	1.3±2.6	2.7±3.0	5.9±5.9	15.2±4.7	52.8±15.7	29.3±6.5
		45.79%	2.51%	5.12%	11.23%	28.76%		
FLOPs		666M	1M	2M	0.7M	59M	729M	
		91.36%	0.14%	0.27%	0.10%	8.09%		

Table 3: Timing benchmarks on VGGNet

Platform		Layerwise Pass (ms) CONV	POOL	ReLU	FC	Total (ms)	Forward Pass (ms)
TK1	CPU	7160.5±0.7	60.1±0.1	95.6±0.1	381.6±0.2	7697.9±0.6	7697.8±0.5
		93.02%	0.78%	1.24%	4.96%		
	GPU	263.1±19.3	7.2±0.5	17.5±1.2	57.6±0.5	347.6±20.1	326.7±2.1
		75.68%	2.06%	5.03%	16.58%		
TX1	CPU	1952.9±12.2	71.3±1.5	52.5±1.9	747.7±24.9	2824.6±23.2	2809.1±10.6
		69.14%	2.52%	1.86%	26.47%		
	GPU	136.3±5.4	3.4±1.6	9.9±4.9	32.8±1.3	184.2±7.4	175.3±2.0
		73.98%	1.84%	5.35%	17.82%		
FLOPs		15360M	6M	14M	124M	15503M	
		99.08%	0.04%	0.09%	0.79%		

Table 4: Memory of CNN models on platforms (MB)

Type/Platform		AlexNet	VGGNet	GoogleNet	ResNet
Weights & Biases		233	528	26	97
Data		8	110	53	221
Workspace		11	168	46	79
TK1	CPU	324	972	161	409
	GPU	560	1508	196	533
TX1	CPU	362	1013	200	453
	GPU	589	1537	226	562

DAWNBench

An End-to-End Deep Learning Benchmark and Competition

Cody Coleman, Deepak Narayanan, Daniel Kang, Tian Zhao, Jian Zhang,
Luigi Nardi, Peter Bailis, Kunle Olukotun, Chris Ré, Matei Zaharia
Stanford University

dawn.cs.stanford.edu/benchmark

Many existing deep learning benchmarks

Accuracy

- ImageNet
- CIFAR10
- MS COCO
- SQuAD
- WMT Machine Translation

Throughput (examples/second)

- Baidu DeepBench
- TensorFlow Benchmarks
- "Benchmarking state-of-the-art Deep Learning Software Tools"
- jcjohnson/cnn-benchmarks
- soumith/convnet-benchmarks

Many existing deep learning benchmarks

Accuracy

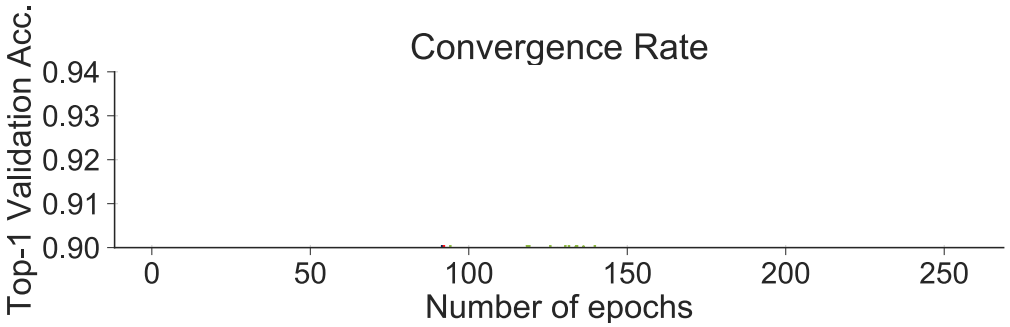
- ImageNet
- CIFAR10
- MS COCO
- SQuAD
- WMT Machine Translation

Throughput (examples/second)

- Baidu DeepBench
- TensorFlow Benchmarks
- "Benchmarking state-of-the-art Deep Learning Software Tools"
- jcjohnson/cnn-benchmarks
- soumith/convnet-benchmarks

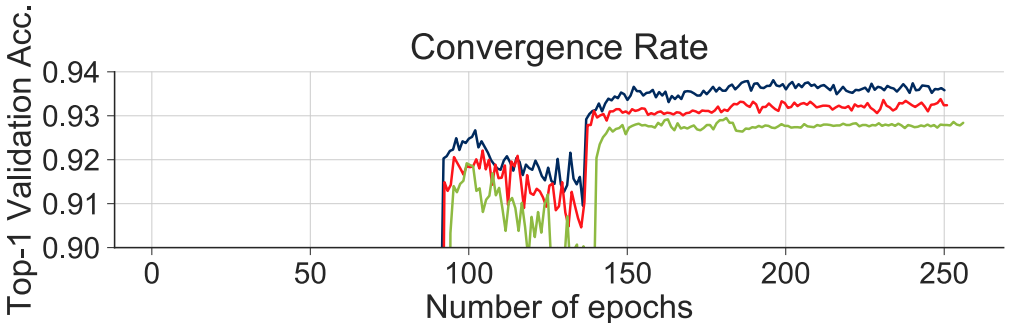
Not time to accuracy

Example: batch size affects accuracy



End-to-end training of a ResNet56 CIFAR10 model on a Nvidia P100 machine with 512 GB of memory and 28 CPU cores, using TensorFlow 1.2 compiled from source with CUDA 8.0 and CuDNN 5.1.

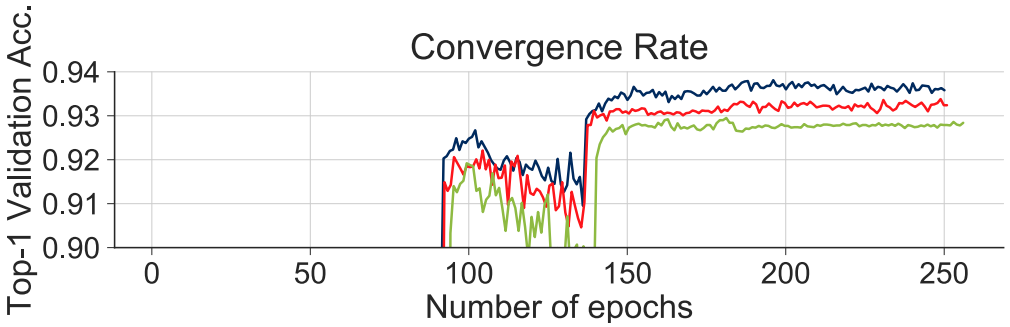
Example: batch size affects accuracy



Batch size = 32 Batch size = 256 Batch size = 2048

End-to-end training of a ResNet56 CIFAR10 model on a Nvidia P100 machine with 512 GB of memory and 28 CPU cores, using TensorFlow 1.2 compiled from source with CUDA 8.0 and CuDNN 5.1.

Example: batch size affects accuracy

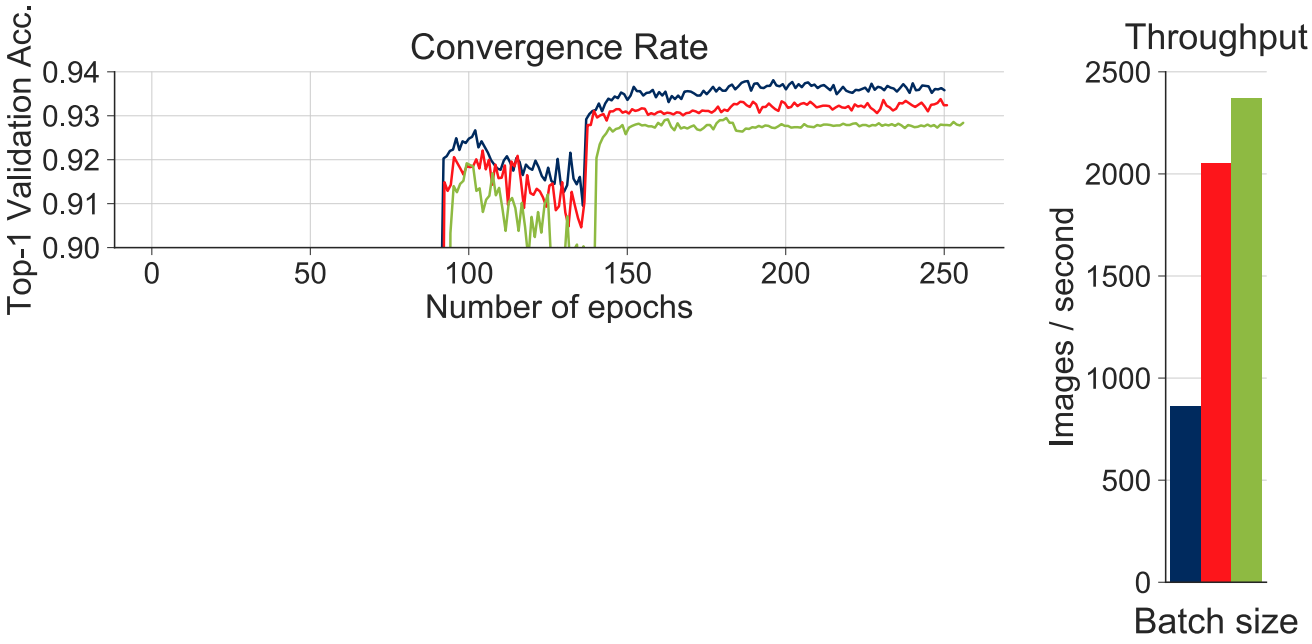


A batch size of 32 achieves the highest accuracy

Batch size = 32 Batch size = 256 Batch size = 2048

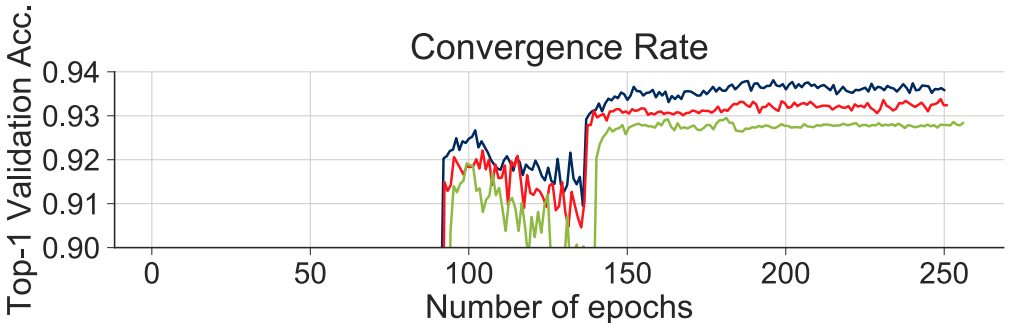
End-to-end training of a ResNet56 CIFAR10 model on a Nvidia P100 machine with 512 GB of memory and 28 CPU cores, using TensorFlow 1.2 compiled from source with CUDA 8.0 and CuDNN 5.1.

Example: batch size affects accuracy and throughput

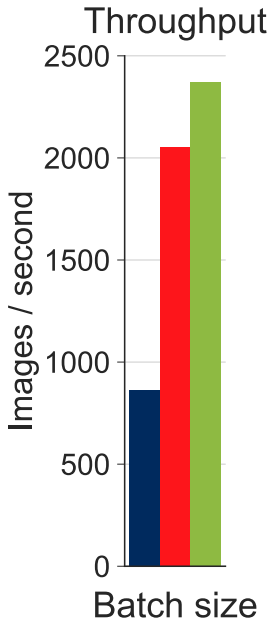


End-to-end training of a ResNet56 CIFAR10 model on a Nvidia P100 machine with 512 GB of memory and 28 CPU cores, using TensorFlow 1.2 compiled from source with CUDA 8.0 and CuDNN 5.1.

Example: batch size affects accuracy and throughput



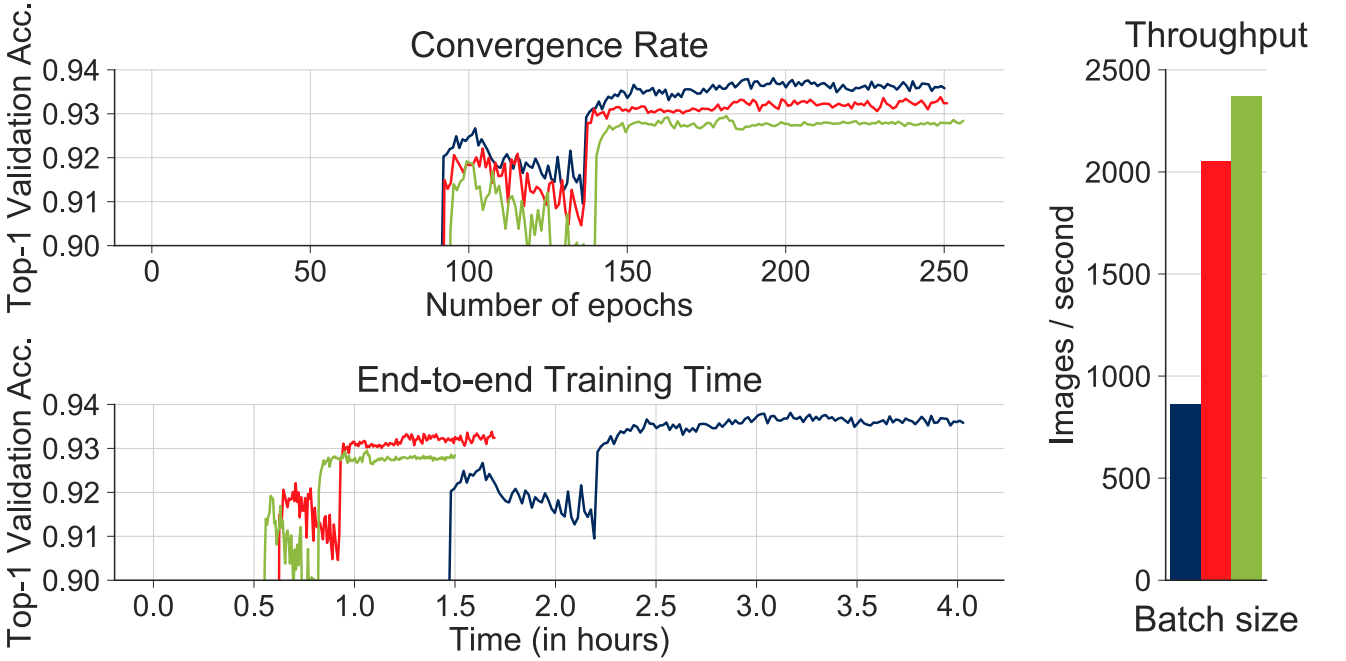
A batch size of 2048 achieves the highest throughput



Batch size = 32 Batch size = 256 Batch size = 2048

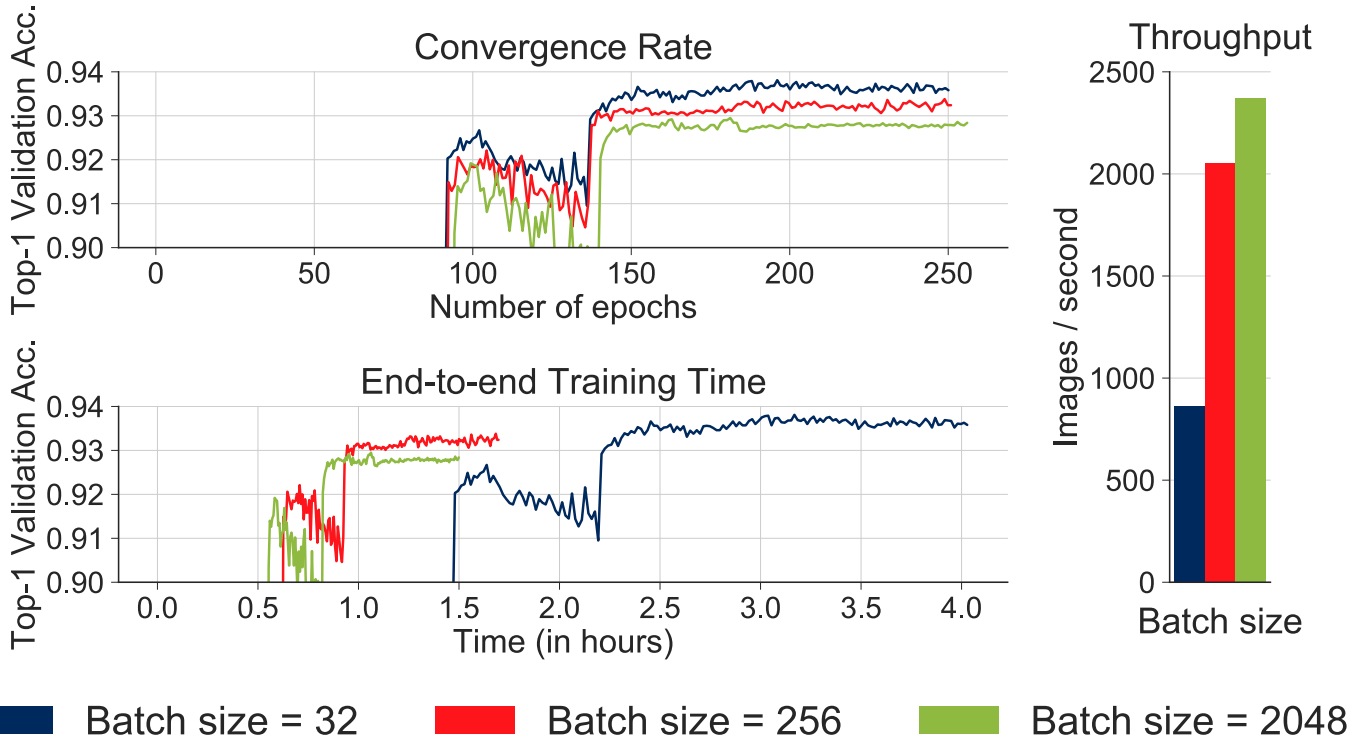
End-to-end training of a ResNet56 CIFAR10 model on a Nvidia P100 machine with 512 GB of memory and 28 CPU cores, using TensorFlow 1.2 compiled from source with CUDA 8.0 and CuDNN 5.1.

Example: batch size affects accuracy and throughput



End-to-end training of a ResNet56 CIFAR10 model on a Nvidia P100 machine with 512 GB of memory and 28 CPU cores, using TensorFlow 1.2 compiled from source with CUDA 8.0 and CuDNN 5.1.

A batch size of 256 represents a reasonable trade-off between convergence rate and throughput



End-to-end training of a ResNet56 CIFAR10 model on a Nvidia P100 machine with 512 GB of memory and 28 CPU cores, using TensorFlow 1.2 compiled from source with CUDA 8.0 and CuDNN 5.1.

What if we combine optimizations?

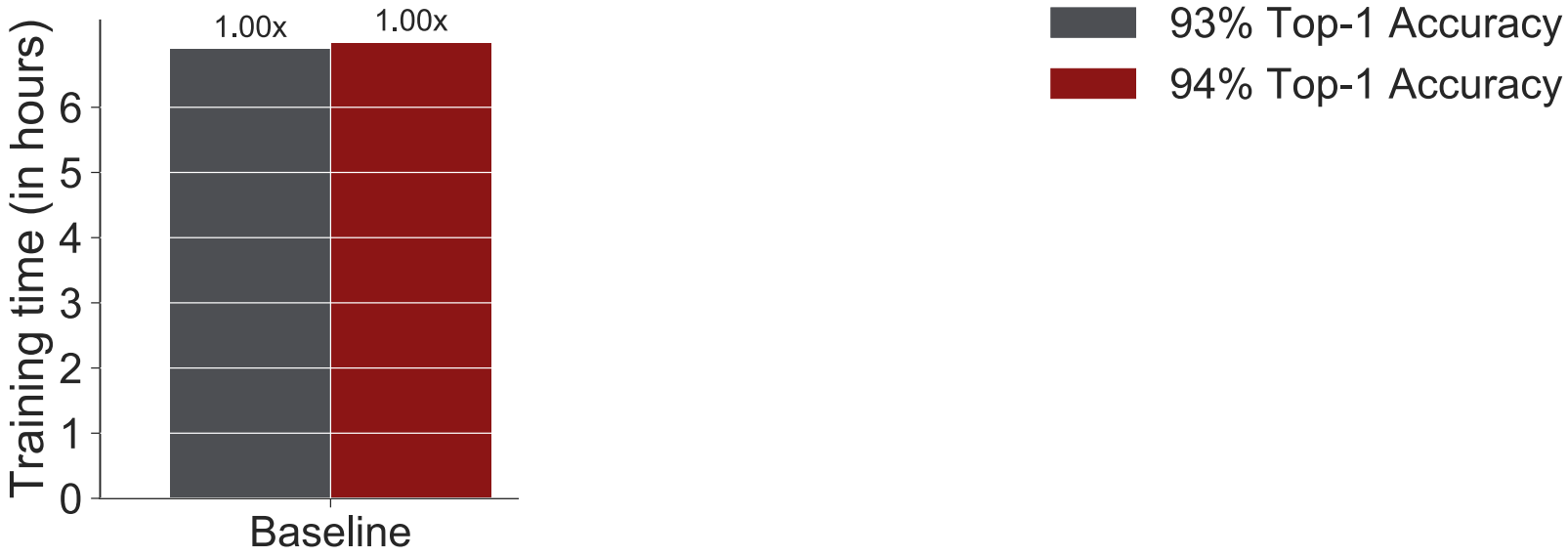
1.25x Stochastic depth
3.1x Minimal effort backpropagation
3x Reduced precision
29x Accurate, large minibatch SGD
3x Nvidia V100 vs Nvidia P100

What if we combine optimizations?

1.25x Stochastic depth
3.1x Minimal effort backpropagation
3x Reduced precision
29x Accurate, large minibatch SGD
3x Nvidia V100 vs Nvidia P100

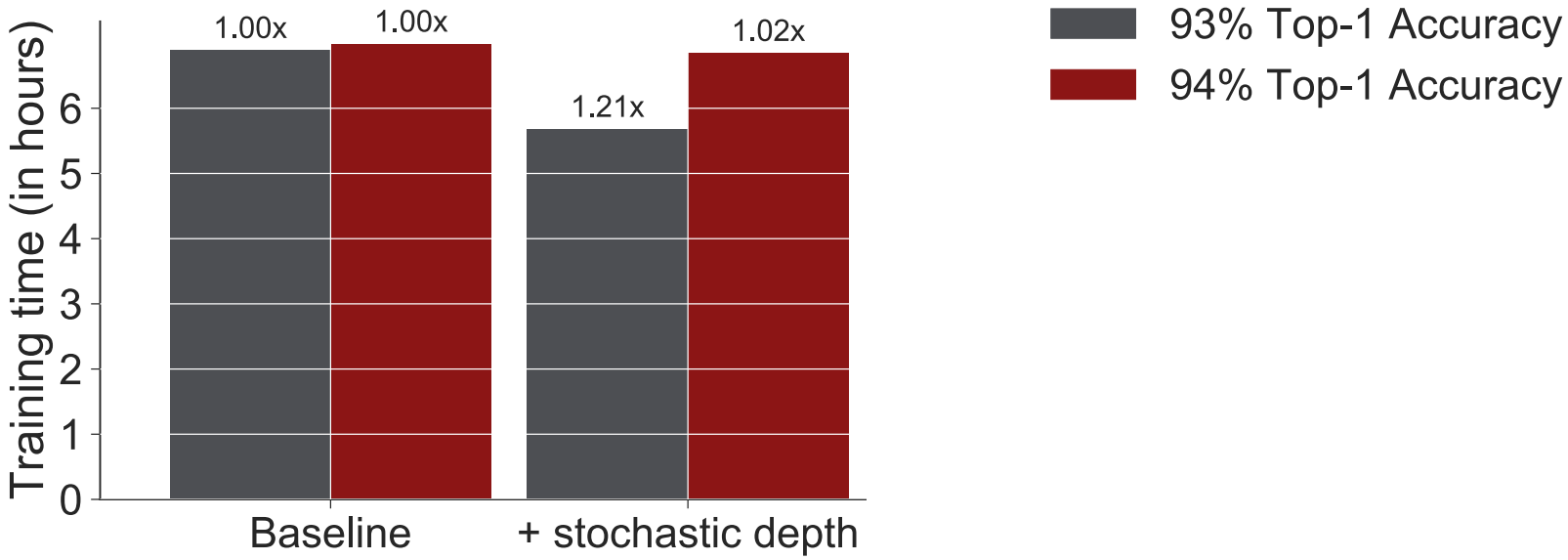
Does that give us a combined speed-up of **1011x**?

What if we combine optimizations?



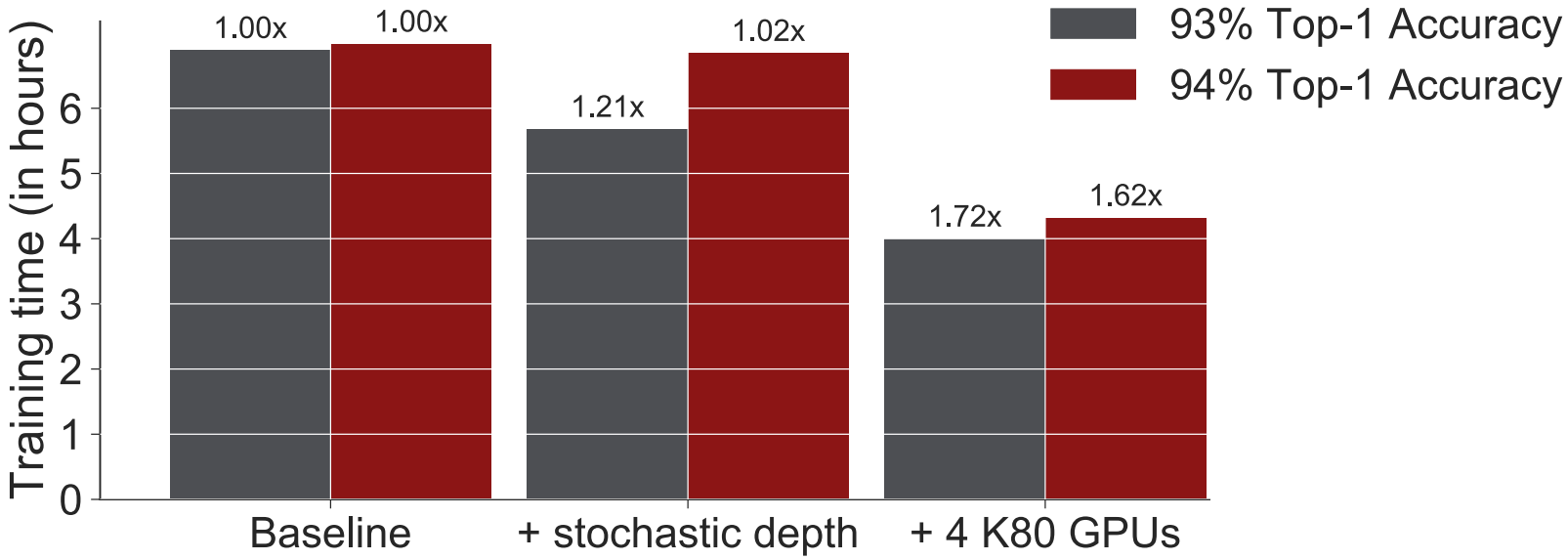
End-to-end training of ResNet110 on CIFAR10 in PyTorch, where the baseline is on machine with a single K80 and a batch size of 128.

What if we combine optimizations?



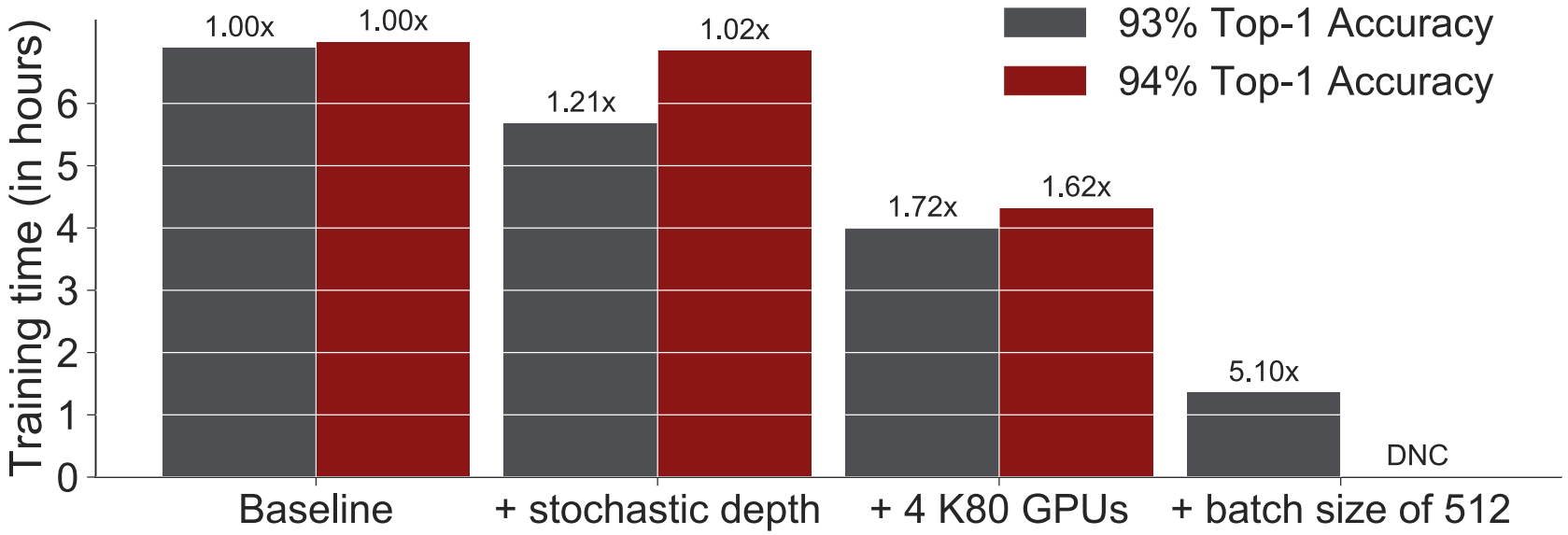
End-to-end training of ResNet110 on CIFAR10 in PyTorch, where the baseline is on machine with a single K80 and a batch size of 128.

What if we combine optimizations?



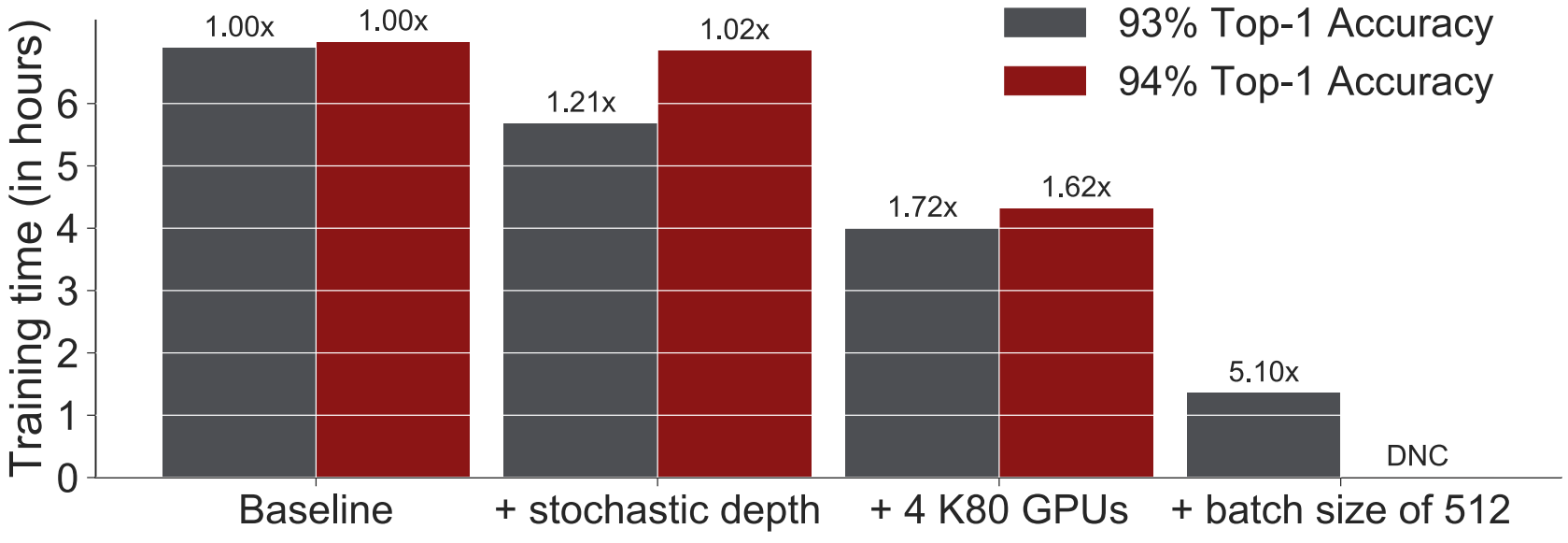
End-to-end training of ResNet110 on CIFAR10 in PyTorch, where the baseline is on machine with a single K80 and a batch size of 128.

What if we combine optimizations?

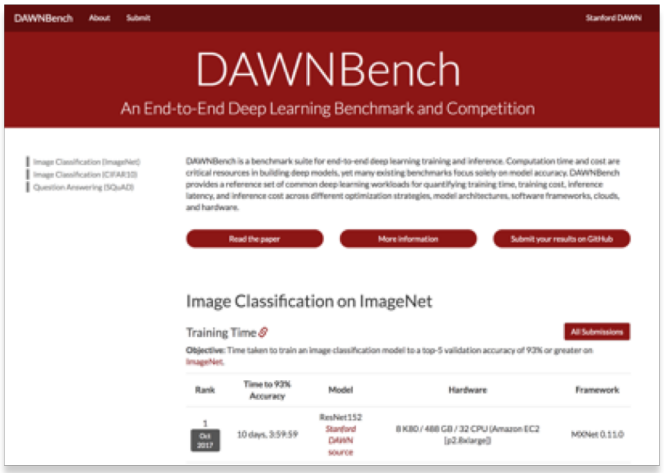


End-to-end training of ResNet110 on CIFAR10 in PyTorch, where the baseline is on machine with a single K80 and a batch size of 128.

What if we combine optimizations?



Optimizations interact in non-trivial ways



- First benchmark to measure time and cost to get a state-of-the-art accuracy
- Our goal: measure end-to-end throughput subject to accuracy

As an initial release

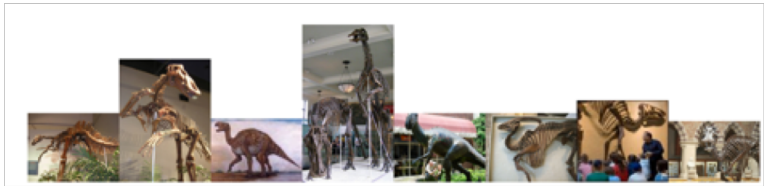
Tasks

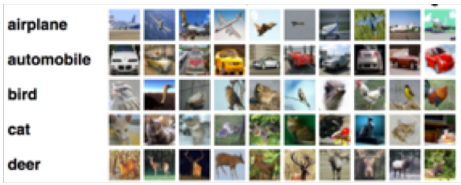
Image classification

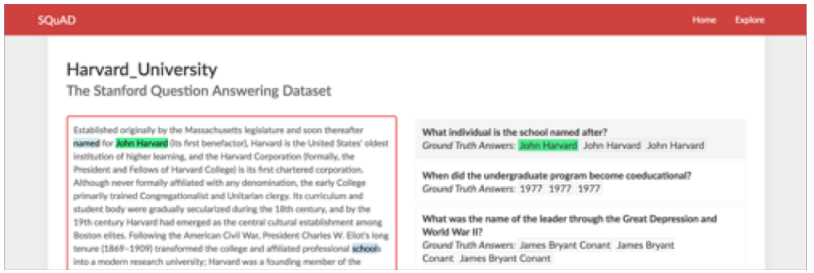
- ImageNet
- CIFAR10

Question answering

- SQuAD







For each task

Accuracy threshold close to the state-of-the-art

Metrics

Training time

Training cost (USD)

Inference latency

Inference cost (USD)

Table 1: CNN models

Layer	AlexNet	VGGNet	GoogLeNet	ResNet
CONV	5	13	57	53
POOL	3	5	14	2
NORM	2		2	53
ReLU	7	15	57	49
FC	3	3	1	1
Concat			9	
Scale				53
Eltwise				16
Total	20	36	140	227

Table 2: Timing benchmarks on AlexNet

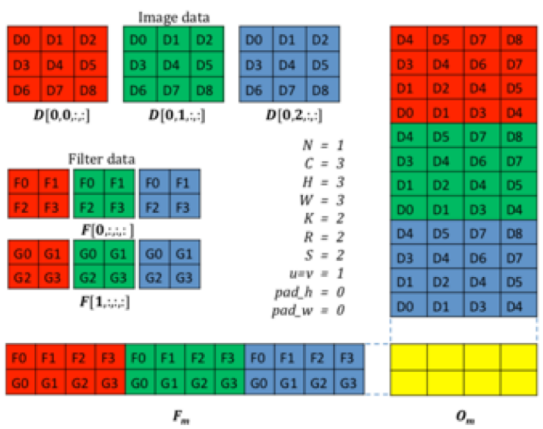
Platform		Layerwise Pass (ms) CONV	POOL	LRN	ReLU	FC	Total (ms)	Forward Pass (ms)
TK1	CPU	318.7±0.2	6.1±0.1	103.8±0.0	4.6±0.0	186.3±0.1	619.8±0.2	619.5±0.2
		51.42%	0.99%	16.74%	0.75%	30.05%		
	GPU	24.6±3.5	2.3±0.6	2.4±0.5	5.2±1.2	35.1±5.9	73.3±10.7	54.7±2.4
		33.53%	3.15%	3.22%	7.11%	**47.95%**		
TX1	CPU	66.9±5.3	7.6±0.0	172.4±0.3	2.4±0.0	644.7±5.3	894.3±4.8	892.7±2.3
		7.48%	0.85%	19.28%	0.27%	**72.09%**		
	GPU	24.2±8.3	1.3±2.6	2.7±3.0	5.9±5.9	15.2±4.7	52.8±15.7	29.3±6.5
		45.79%	2.51%	5.12%	11.23%	28.76%		
FLOPs		666M	1M	2M	0.7M	59M	729M	
		91.36%	0.14%	0.27%	0.10%	8.09%		

Table 3: Timing benchmarks on VGGNet

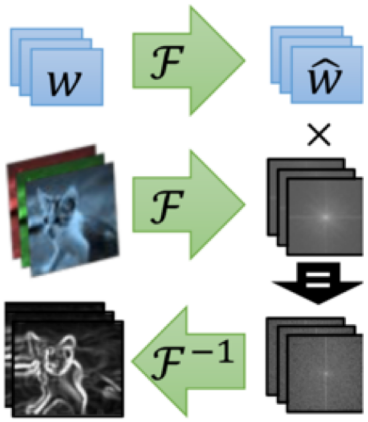
Platform		Layerwise Pass (ms) CONV	POOL	ReLU	FC	Total (ms)	Forward Pass (ms)
TK1	CPU	7160.5±0.7	60.1±0.1	95.6±0.1	381.6±0.2	7697.9±0.6	7697.8±0.5
		93.02%	0.78%	1.24%	4.96%		
	GPU	263.1±19.3	7.2±0.5	17.5±1.2	57.6±0.5	347.6±20.1	326.7±2.1
		75.68%	2.06%	5.03%	16.58%		
TX1	CPU	1952.9±12.2	71.3±1.5	52.5±1.9	747.7±24.9	2824.6±23.2	2809.1±10.6
		69.14%	2.52%	1.86%	26.47%		
	GPU	136.3±5.4	3.4±1.6	9.9±4.9	32.8±1.3	184.2±7.4	175.3±2.0
		73.98%	1.84%	5.35%	17.82%		
FLOPs		15360M	6M	14M	124M	15503M	
		99.08%	0.04%	0.09%	0.79%		

Table 4: Memory of CNN models on platforms (MB)

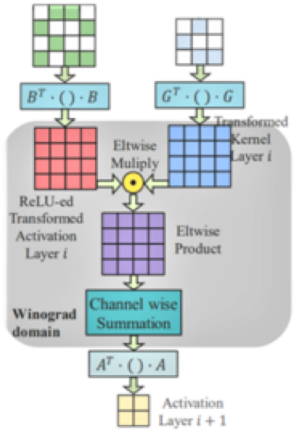
Type/Platform		AlexNet	VGGNet	GoogleNet	ResNet
Weights & Biases		233	528	26	97
Data		8	110	53	221
Workspace		11	168	46	79
TK1	CPU	324	972	161	409
	GPU	560	1508	196	533
TX1	CPU	362	1013	200	453
	GPU	589	1537	226	562



(a) `im2col` (adapted from [38])



(b) FFT



(c) Winograd (adapted from [162])

Fig. 12. Computation Methods for Convolutional Operators

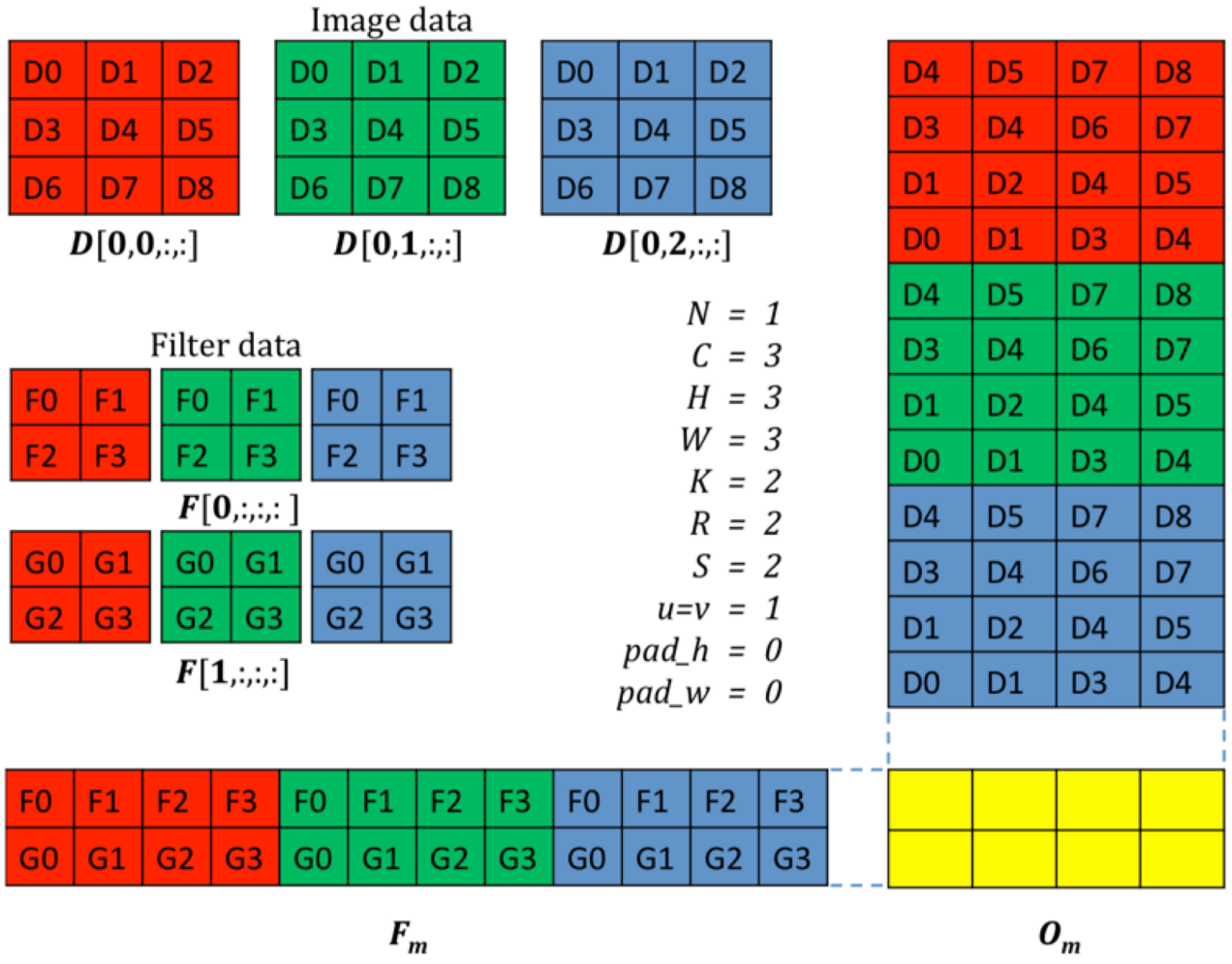


Figure 1: Convolution lowering

While processor-friendly, the GEMM method (as described above) consumes a considerable amount of memory, and thus was not scalable. Practical implementations of the GEMM method, such as in CUDNN[38], implement "implicit GEMM", in which the Toeplitz matrix is never materialized. It was also reported[49] that the Strassen matrix multiplication[223] can be used for the underlying computation, reducing the number of operations by up to 47%.

A second method to compute convolutions is to make use of the Fourier domain, in which convolution is defined as an element-wise multiplication [166,232]. In this method, both the data and the kernels are transformed using FFT, multiplied, and the inverse FFT is applied on the result:

$$y_{i,j,*,*} = \mathcal{F}^{-1}\left(\sum_{m=0}^{C_{in}} \mathcal{F}\left(x_{i,m,*,*}\right) \circ \mathcal{F}\left(w_{j,m,*,*}\right)\right)$$

where $\mathcal{F}$ denotes the Fourier Transform and $\circ$ is element-wise multiplication. Note that for a single minibatch, it is enough to transform w once and reuse the results.

Experimental results[232] have shown that the larger the convolution kernels are, the more beneficial FFT becomes, yielding up to 16× performance over the GEMM method, which has to process patches of proportional size to the kernels. Additional optimizations were made to the FFT and IFFT operations[232], using DNN-specific knowledge: (a) The process uses decimation-in-frequency for FFT and decimation-in-time for IFFT in order to mitigate bit-reversal instructions; (b) multiple FFTs with sizes ≤32 are batched together and performed at the warp-level on the GPU; and (c) pre-computation of twiddle factors.

Working with DNNs, FFT-based convolution can be optimized further. In ZNNi[267], the authors observed that due to zero-padding, the convolutional kernels, which are considerably smaller than the images, mostly consist of zeros. Thus, pruned FFT[222] can be executed for transforming the kernels, reducing the number of operations by 3×. In turn, the paper reports 5× and 10× speedups for CPUs and GPUs, respectively.

Table 6. Work-Depth Analysis of Convolution Implementations

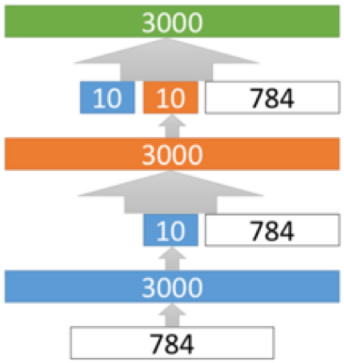
Method	**Work (W)**	**Depth (D)**
Direct	$N \cdot C_{out} \cdot H' \cdot W' \cdot C_{in} \cdot K_y \cdot K_x$	$\lceil \log_2 C_{in} \rceil + \lceil \log_2 K_y \rceil + \lceil \log_2 K_x \rceil$
im2col	$N \cdot C_{out} \cdot H' \cdot W' \cdot C_{in} \cdot K_y \cdot K_x$	$\lceil \log_2 C_{in} \rceil + \lceil \log_2 K_y \rceil + \lceil \log_2 K_x \rceil$
FFT	$c \cdot HW \log_2(HW) \cdot (C_{out} \cdot C_{in} + N \cdot C_{in} + N \cdot C_{out}) + HWN \cdot C_{in} \cdot C_{out}$	$2\lceil \log_2 HW \rceil + \lceil \log_2 C_{in} \rceil$
Winograd ($m \times m$ tiles, $r \times r$ kernels)	$\alpha(r^2 + \alpha r + 2\alpha^2 + \alpha m + m^2) + C_{out} \cdot C_{in} \cdot P$ $(\alpha \equiv m - r + 1, \quad P \equiv N \cdot \lceil H/m \rceil \cdot \lceil W/m \rceil)$	$2\lceil \log_2 r \rceil + 4\lceil \log_2 \alpha \rceil + \lceil \log_2 C_{in} \rceil$

(a) Data Parallelism (b) Model Parallelism (c) Layer Pipelining

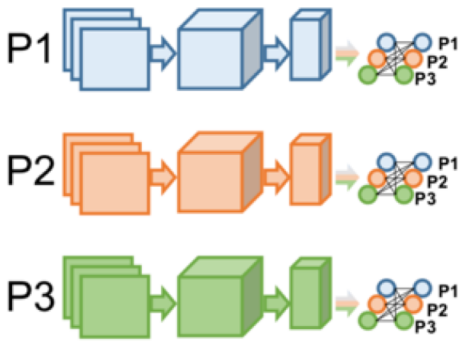
Fig. 14. Neural Network Parallelism Schemes

6.1.1 Neural Architecture Support for Large Minibatches. By applying various modifications to the training process, recent works have successfully managed to increase minibatch size to 8k samples[83], 32k samples[249], and even 64k[218] without losing considerable accuracy. While the generalization issue still exists (Section 3), it is not as severe as claimed in prior works[211]. One bottleneck that hinders scaling of data parallelism, however, is the BN operator, which requires a full synchronization point upon invocation. Since BN recurs multiple times in some DNN architectures[93], this is too costly. Thus, popular implementations of BN follow the approach driven by large-batch papers[83,105,249], in which small subsets (e.g., 32 samples) of the minibatch are normalized independently. If at least 32 samples are scheduled to each processor, then this synchronization point is local, which in turn increases scaling.

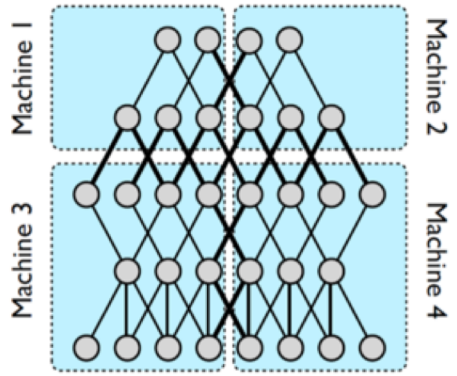
Another approach to the BN problem is to define a different operator altogether. Weight Normalization (WN)[208] proposes to separate the parameter (w) norm from its directionality by way of re-parameterization. In WN, the weights are defined as $w = \left(\frac{g}{\|v\|}\right) \cdot v$, where g represents weight magnitude and v a normalized direction (as changing the magnitude of v will not introduce changes in $\nabla \ell$). WN decreases the depth (**D**) of the operator from $O(\log N)$ to $O(1)$, removing inter-dependencies within the minibatch. According to the authors, WN reduces the need for BN, achieving comparable accuracy using a simplified version of BN (without variance correction).



(a) Deep Stacking Network [62]



(b) Hybrid Parallelism



(c) DistBelief Replica [56]

Fig. 17. Pipelining and Hybrid Parallelism Schemes

Category	Method
Model Consistency	
Synchronization	Synchronous [45,112,173,210,224,246]
	Stale-Synchronous [88,98,120,156,261]
	Asynchronous [55,56,127,182,190,205,260]
	Nondeterministic Comm. [54,121,201]
Parameter Distribution and Communication	
Centralization	Parameter Server (PS) [52,112,128,152]
	Sharded PS [39,56,120,137,142,244,254,255]
	Hierarchical PS [88,107,253]
	Decentralized [9,54,121,155]
Compression	Quantization [5,35,50,51,55,63,87,90,109,132,151,202,210,237,265]
	Sparsification [3,32,67,158,206,214,224]
	Other Methods [41,106,111,129,150,243,254]
Training Distribution	
Model Consolidation	Ensemble Learning [114,148,217]
	Knowledge Distillation [10,96]
	Model Averaging: Direct [20,33,168,266], Elastic [121,155,248,258], Natural Gradient [11,195]
Optimization Algorithms	First-Order [43,123,126,141,145,163,227]
	Second-Order [11,28,56,94,133,165,172]
	Evolutionary [169,192,234,242]
	Hyper-Parameter Search [13,89,91,118,131,163,169,219,256]
	Architecture Search: Reinforcement [12,193,264,268,269], Evolutionary [160,203,204,242,251], SMBO [27,71,159,161,176]

Fig. 18. Overview of Distributed Deep Learning Methods

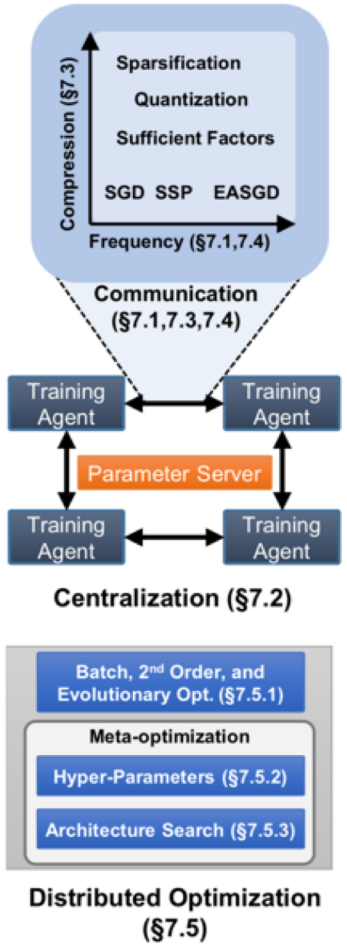
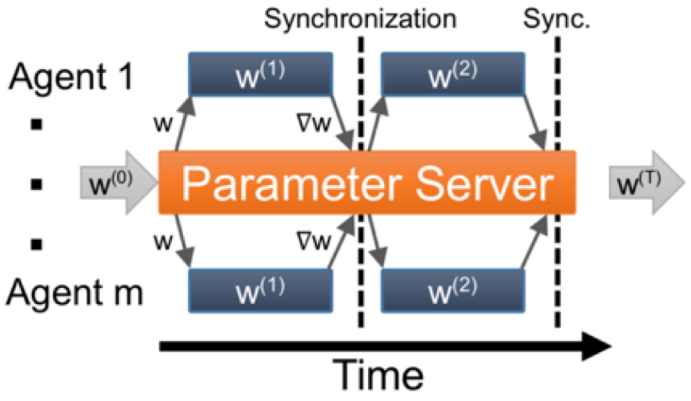
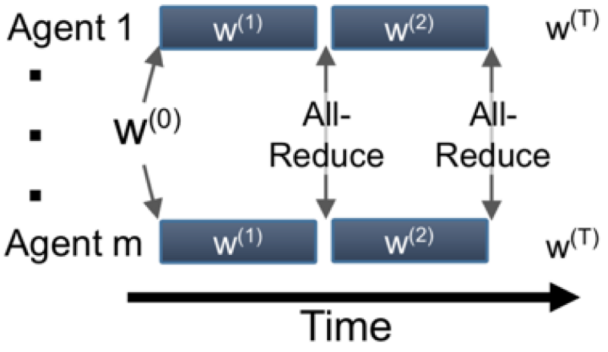


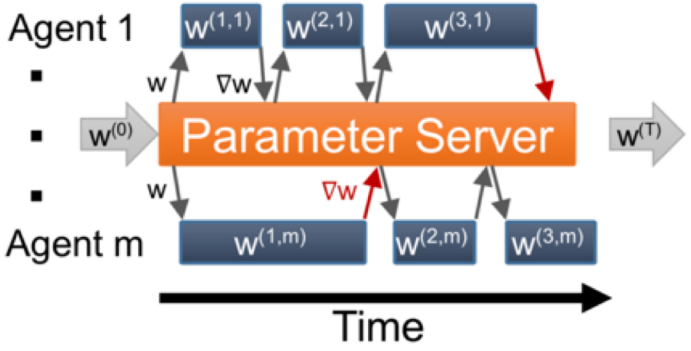
Fig. 19. Section Overview



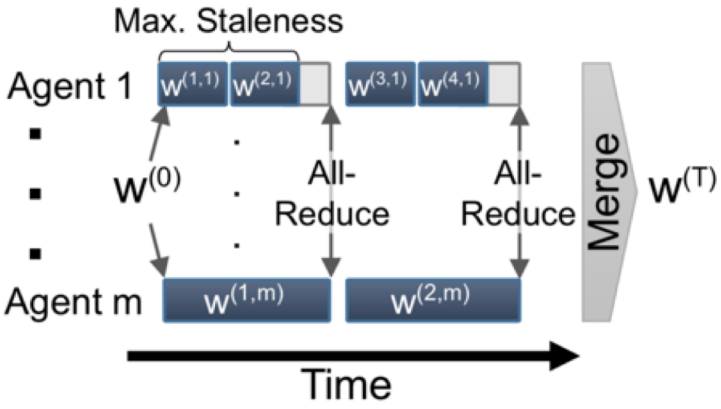
(a) Synchronous, Parameter Server



(b) Synchronous, Decentralized



(c) Asynchronous, Parameter Server

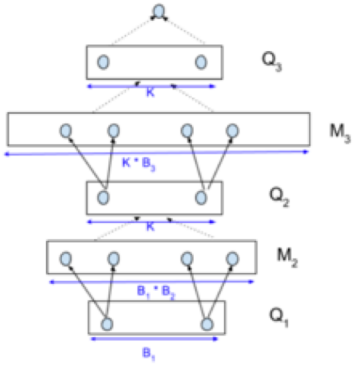


(d) Stale-Synchronous, Decentralized

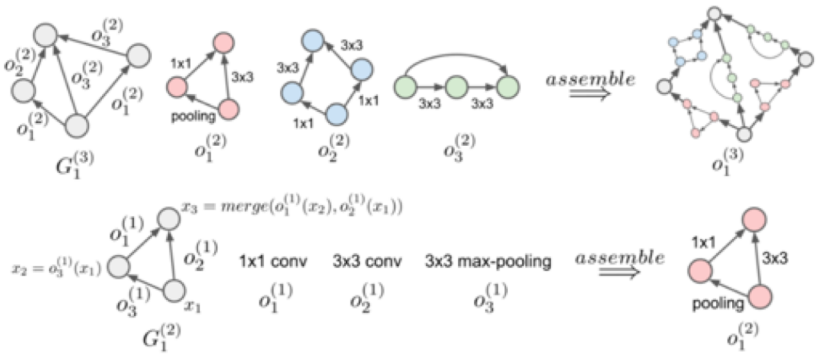
Fig. 20. Training Distribution in Deep Learning (Model Consistency, Centralization)

Recent works relax the synchronization restriction, creating an *inconsistent model* (Fig. 20c). As a result, a training agent i at time t contains a copy of the weights, denoted as $w^{(\tau,i)}$ for $\tau \leq t$, where $t - \tau$ is called the *staleness* (or lag). A well-known instance of inconsistent SGD is the HOGWILD shared-memory algorithm[205], which allows training agents to read parameters and update gradients at will, overwriting existing progress. HOGWILD has been proven to converge for sparse learning problems[205], where updates only modify small subsets of w, and generally[55]. Based on foundations of distributed asynchronous SGD[229], the proofs impose that (a) write-accesses (adding gradients) are always atomic; (b) Lipschitz continuous differentiability and strong convexity on f_w; and (c) that the staleness, i.e., the maximal number of iterations between reading w and writing ∇w, is bounded.

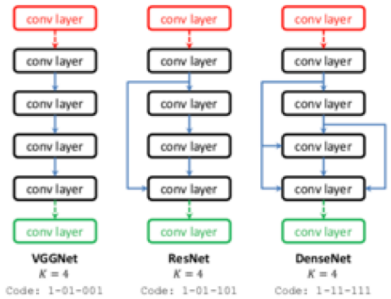
Fig. 23. Parameter Consistency Spectrum



(a) PNASNet Search-Space Traversal [159]



(b) Hierarchical Representation [160]



(c) Genetic Encoding of DNN Connections [242]

Fig. 24. Methods for Automated Architecture Search

Previously proposed 'Deep Compression' makes it possible to fit large DNNs (AlexNet and VGGNet) fully in on-chip SRAM. This compression is achieved by pruning the redundant connections and having multiple connections share the same weight. We propose an energy efficient inference engine (EIE) that performs inference on this compressed network model and accelerates the resulting sparse matrix-vector multiplication with weight sharing. Going from DRAM to SRAM gives EIE 120× energy saving; Exploiting sparsity saves 10×; Weight sharing gives 8×; Skipping zero activations from ReLU saves another 3×. Evaluated on nine DNN benchmarks, EIE is 189× and 13× faster when compared to CPU and GPU implementations of the same DNN without compression. EIE has a processing power of 102 GOPS/s working directly on a compressed network, corresponding to 3 TOPS/s on an uncompressed network, and processes FC layers of AlexNet at 1.88×10^4 frames/sec with a power dissipation of only 600mW. It is 24,000× and 3,400× more energy efficient than a CPU and GPU respectively. Compared with DaDianNao, EIE has 2.9×, 19× and 3× better throughput, energy efficiency and area efficiency.

Table I
ENERGY TABLE FOR 45NM CMOS PROCESS [9]. DRAM ACCESS USES THREE ORDERS OF MAGNITUDE MORE ENERGY THAN SIMPLE ARITHMETIC AND 128X MORE THAN SRAM.

Operation	Energy [pJ]	Relative Cost
32 bit int ADD	0.1	1
32 bit float ADD	0.9	9
32 bit int MULT	3.1	31
32 bit float MULT	3.7	37
32 bit 32KB SRAM	5	50
32 bit DRAM	**640**	**6400**

Table III
BENCHMARK FROM STATE-OF-THE-ART DNN MODELS

Layer	Size	Weight%	Act%	FLOP%	Description
Alex-6	9216, 4096	9%	35.1%	3%	Compressed AlexNet [1] for large scale image classification
Alex-7	4096, 4096	9%	35.3%	3%	
Alex-8	4096, 1000	25%	37.5%	10%	
VGG-6	25088, 4096	4%	18.3%	1%	Compressed VGG-16 [3] for large scale image classification and object detection
VGG-7	4096, 4096	4%	37.5%	2%	
VGG-8	4096, 1000	23%	41.1%	9%	
NT-We	4096, 600	10%	100%	10%	Compressed NeuralTalk [7] with RNN and LSTM for automatic image captioning
NT-Wd	600, 8791	11%	100%	11%	
NTLSTM	1201, 2400	10%	100%	11%	

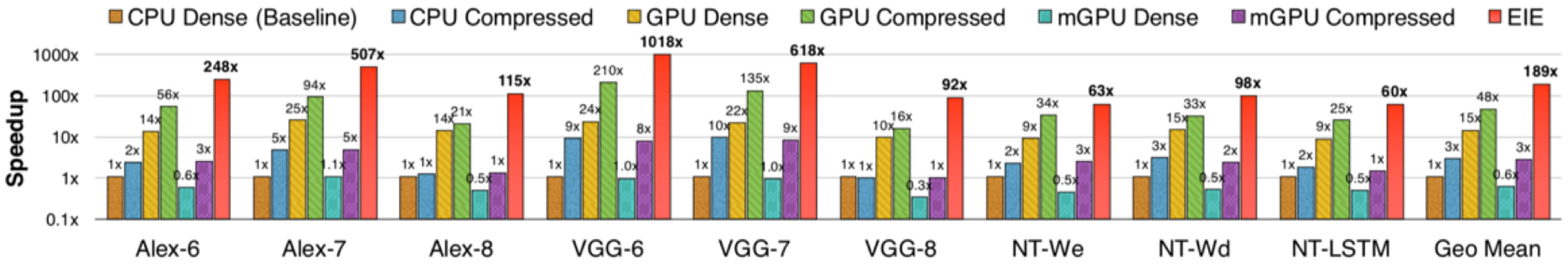


Figure 6. Speedups of GPU, mobile GPU and EIE compared with CPU running uncompressed DNN model. There is no batching in all cases.

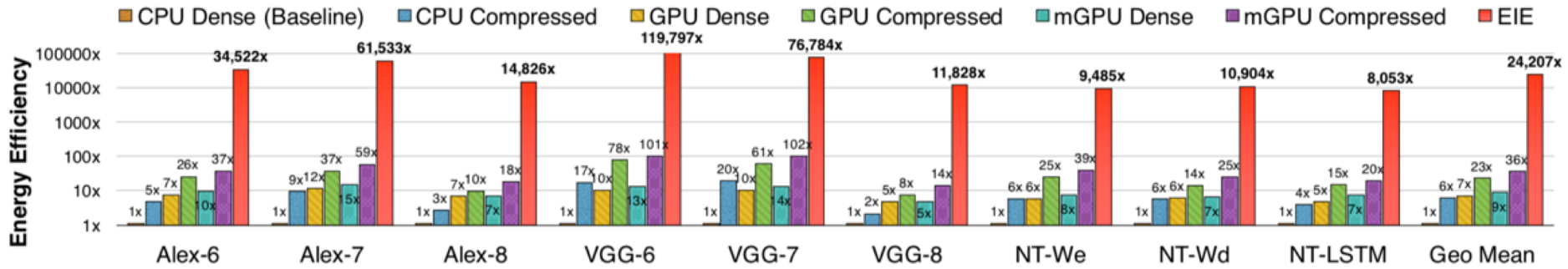


Figure 7. Energy efficiency of GPU, mobile GPU and EIE compared with CPU running uncompressed DNN model. There is no batching in all cases.

1602.01528.pdf

Table IV
WALL CLOCK TIME COMPARISON BETWEEN CPU, GPU, MOBILE GPU AND EIE. UNIT: μS

Platform	Batch Size	Matrix Type	AlexNet			VGG16			NT-		
			FC6	FC7	FC8	FC6	FC7	FC8	We	Wd	LSTM
CPU (Core i7-5930k)	1	dense	7516.2	6187.1	1134.9	35022.8	5372.8	774.2	605.0	1361.4	470.5
		sparse	3066.5	1282.1	890.5	3774.3	545.1	777.3	261.2	437.4	260.0
	64	dense	318.4	188.9	45.8	1056.0	188.3	45.7	28.7	69.0	28.8
		sparse	1417.6	682.1	407.7	1780.3	274.9	363.1	117.7	176.4	107.4
GPU (Titan X)	1	dense	541.5	243.0	80.5	1467.8	243.0	80.5	65	90.1	51.9
		sparse	134.8	65.8	54.6	167.0	39.8	48.0	17.7	41.1	18.5
	64	dense	19.8	8.9	5.9	53.6	8.9	5.9	3.2	2.3	2.5
		sparse	94.6	51.5	23.2	121.5	24.4	22.0	10.9	11.0	9.0
mGPU (Tegra K1)	1	dense	12437.2	5765.0	2252.1	35427.0	5544.3	2243.1	1316	2565.5	956.9
		sparse	2879.3	1256.5	837.0	4377.2	626.3	745.1	240.6	570.6	315
	64	dense	1663.6	2056.8	298.0	2001.4	2050.7	483.9	87.8	956.3	95.2
		sparse	4003.9	1372.8	576.7	8024.8	660.2	544.1	236.3	187.7	186.5
EIE	**Theoretical Time**		**28.1**	**11.7**	**8.9**	**28.1**	**7.9**	**7.3**	**5.2**	**13.0**	**6.5**
	Actual Time		**30.3**	**12.2**	**9.9**	**34.4**	**8.7**	**8.4**	**8.0**	**13.9**	**7.5**

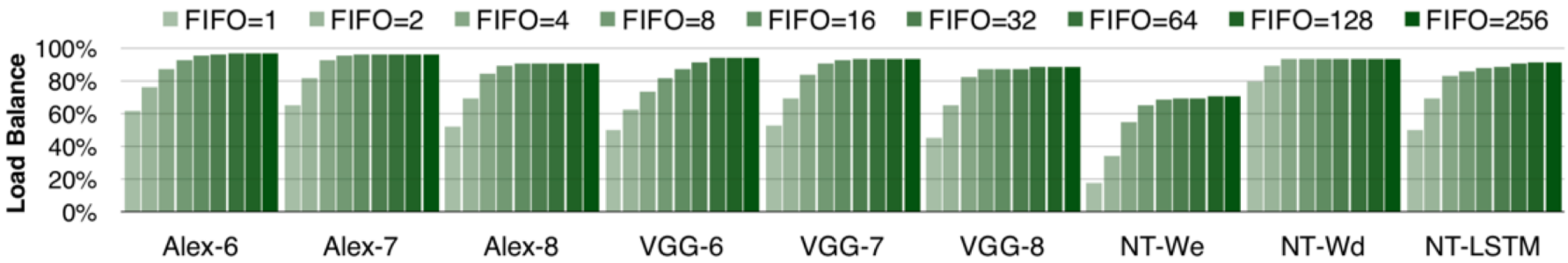


Figure 8. Load efficiency improves as FIFO size increases. When FIFO deepth>8, the marginal gain quickly diminishes. So we choose FIFO depth=8.

DAWNBench

An End-to-End Deep Learning Benchmark and Competition

Cody Coleman, Deepak Narayanan, Daniel Kang, Tian Zhao, Jian Zhang,
Luigi Nardi, Peter Bailis, Kunle Olukotun, Chris Ré, Matei Zaharia
Stanford University

dawn.cs.stanford.edu/benchmark

Many existing deep learning benchmarks

Accuracy

- ImageNet
- CIFAR10
- MS COCO
- SQuAD
- WMT Machine Translation

Throughput (examples/second)

- Baidu DeepBench
- TensorFlow Benchmarks
- "Benchmarking state-of-the-art Deep Learning Software Tools"
- jcjohnson/cnn-benchmarks
- soumith/convnet-benchmarks

Many existing deep learning benchmarks

Accuracy

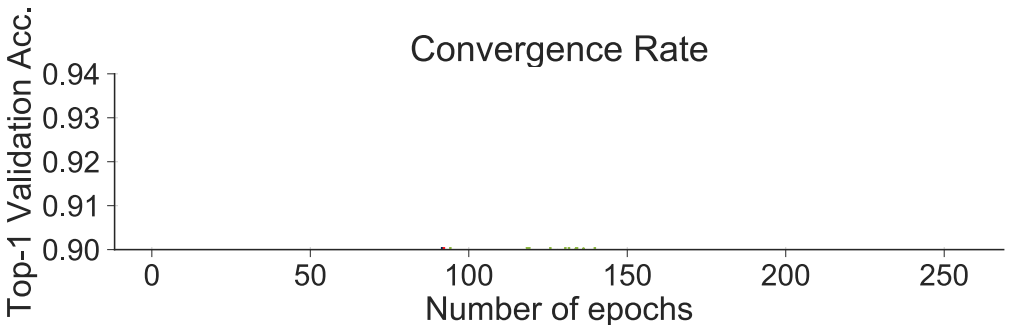
- ImageNet
- CIFAR10
- MS COCO
- SQuAD
- WMT Machine Translation

Throughput (examples/second)

- Baidu DeepBench
- TensorFlow Benchmarks
- "Benchmarking state-of-the-art Deep Learning Software Tools"
- jcjohnson/cnn-benchmarks
- soumith/convnet-benchmarks

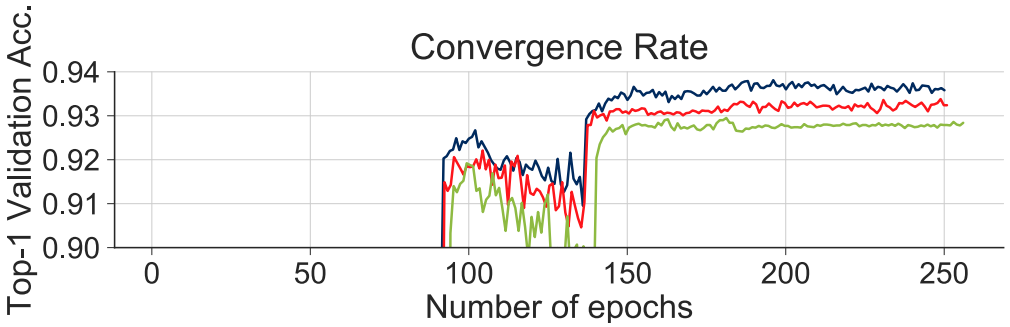
Not time to accuracy

Example: batch size affects accuracy



End-to-end training of a ResNet56 CIFAR10 model on a Nvidia P100 machine with 512 GB of memory and 28 CPU cores, using TensorFlow 1.2 compiled from source with CUDA 8.0 and CuDNN 5.1.

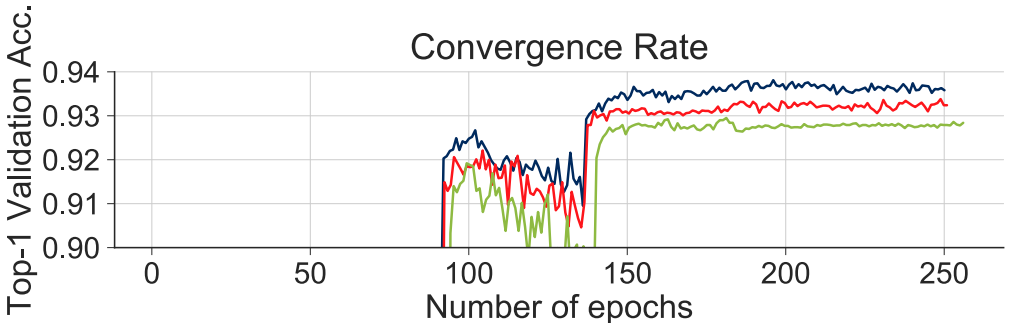
Example: batch size affects accuracy



Batch size = 32 Batch size = 256 Batch size = 2048

End-to-end training of a ResNet56 CIFAR10 model on a Nvidia P100 machine with 512 GB of memory and 28 CPU cores, using TensorFlow 1.2 compiled from source with CUDA 8.0 and CuDNN 5.1.

Example: batch size affects accuracy

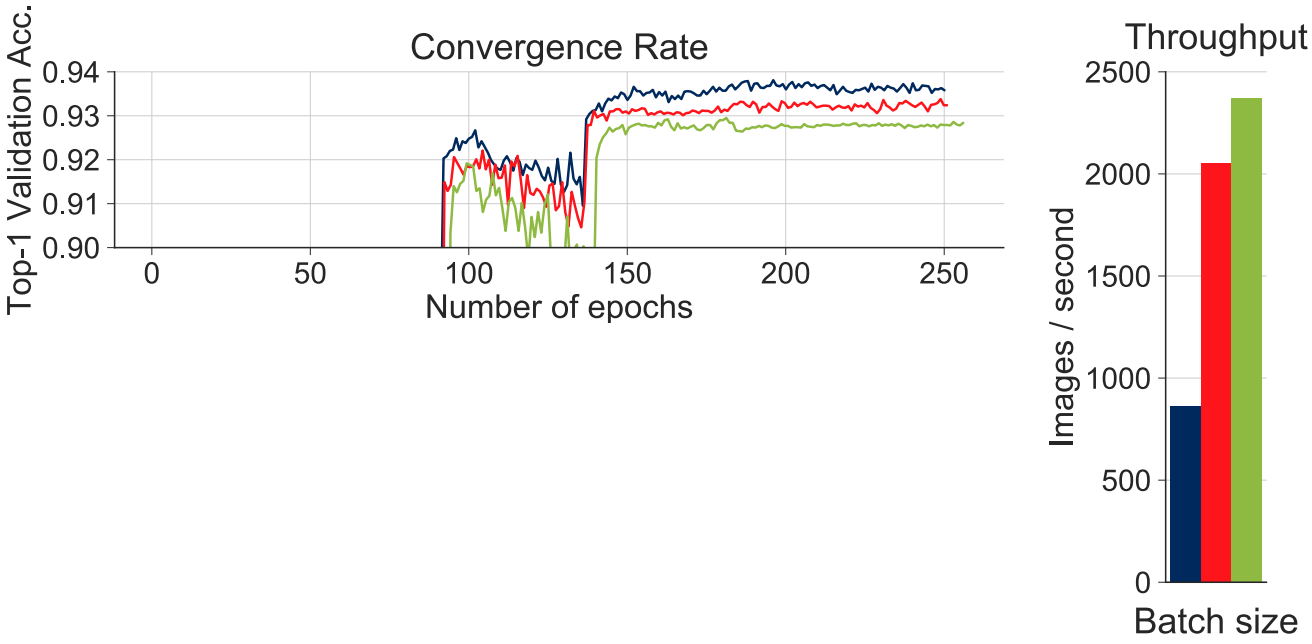


A batch size of 32 achieves the highest accuracy

Batch size = 32 Batch size = 256 Batch size = 2048

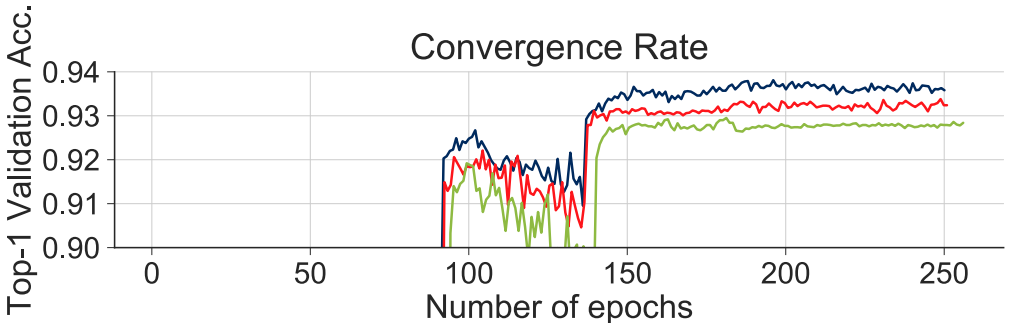
End-to-end training of a ResNet56 CIFAR10 model on a Nvidia P100 machine with 512 GB of memory and 28 CPU cores, using TensorFlow 1.2 compiled from source with CUDA 8.0 and CuDNN 5.1.

Example: batch size affects accuracy and throughput

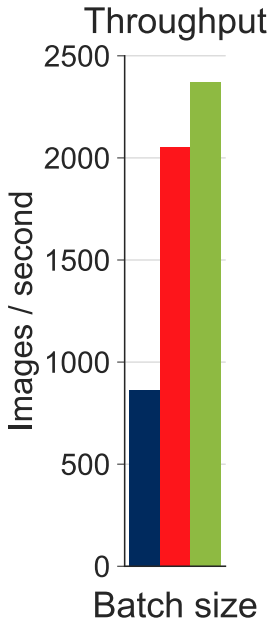


End-to-end training of a ResNet56 CIFAR10 model on a Nvidia P100 machine with 512 GB of memory and 28 CPU cores, using TensorFlow 1.2 compiled from source with CUDA 8.0 and CuDNN 5.1.

Example: batch size affects accuracy and throughput



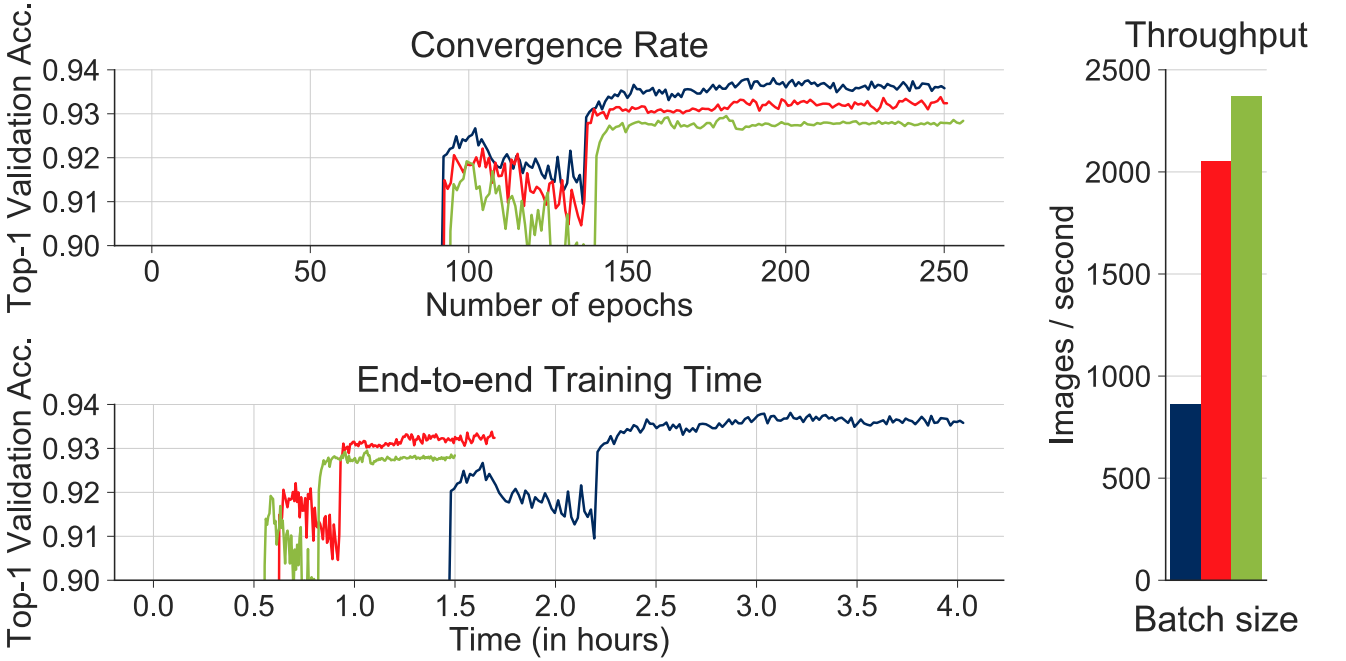
A batch size of 2048 achieves the highest throughput



Batch size = 32 Batch size = 256 Batch size = 2048

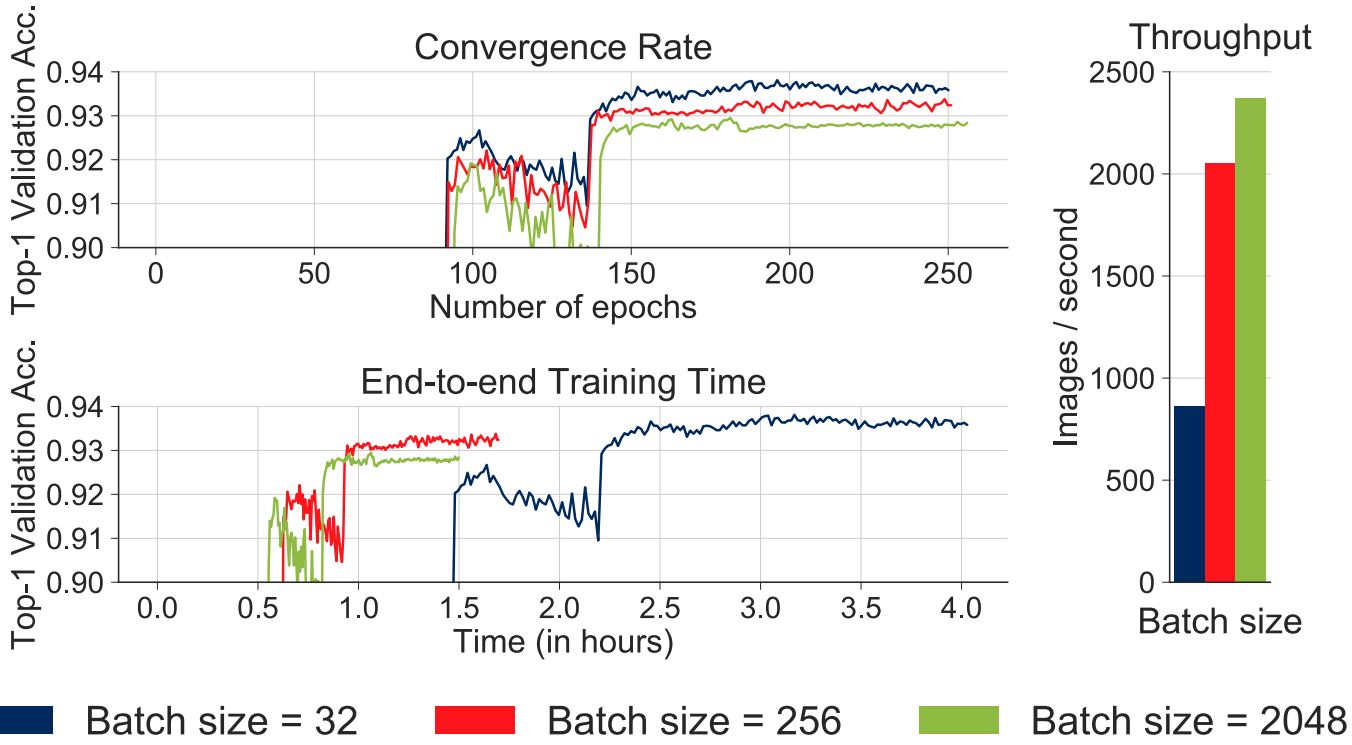
End-to-end training of a ResNet56 CIFAR10 model on a Nvidia P100 machine with 512 GB of memory and 28 CPU cores, using TensorFlow 1.2 compiled from source with CUDA 8.0 and CuDNN 5.1.

Example: batch size affects accuracy and throughput



End-to-end training of a ResNet56 CIFAR10 model on a Nvidia P100 machine with 512 GB of memory and 28 CPU cores, using TensorFlow 1.2 compiled from source with CUDA 8.0 and CuDNN 5.1.

A batch size of 256 represents a reasonable trade-off between convergence rate and throughput



End-to-end training of a ResNet56 CIFAR10 model on a Nvidia P100 machine with 512 GB of memory and 28 CPU cores, using TensorFlow 1.2 compiled from source with CUDA 8.0 and CuDNN 5.1.

What if we combine optimizations?

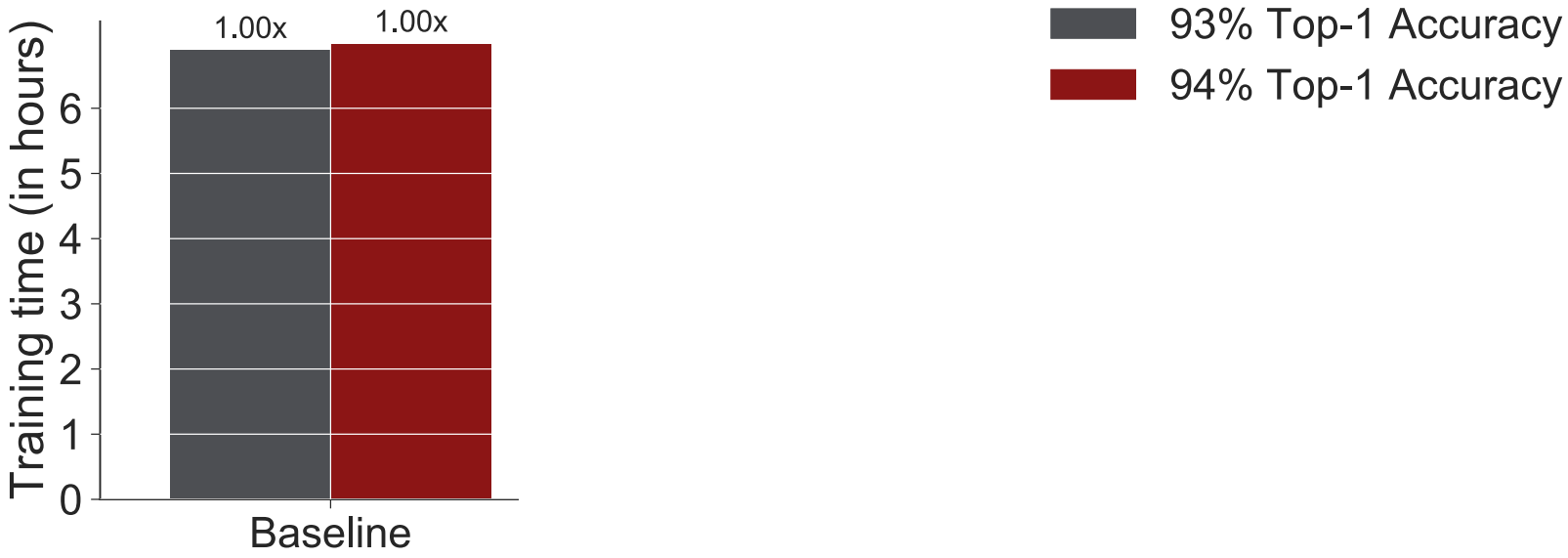
1.25x Stochastic depth
3.1x Minimal effort backpropagation
3x Reduced precision
29x Accurate, large minibatch SGD
3x Nvidia V100 vs Nvidia P100

What if we combine optimizations?

1.25x Stochastic depth
3.1x Minimal effort backpropagation
3x Reduced precision
29x Accurate, large minibatch SGD
3x Nvidia V100 vs Nvidia P100

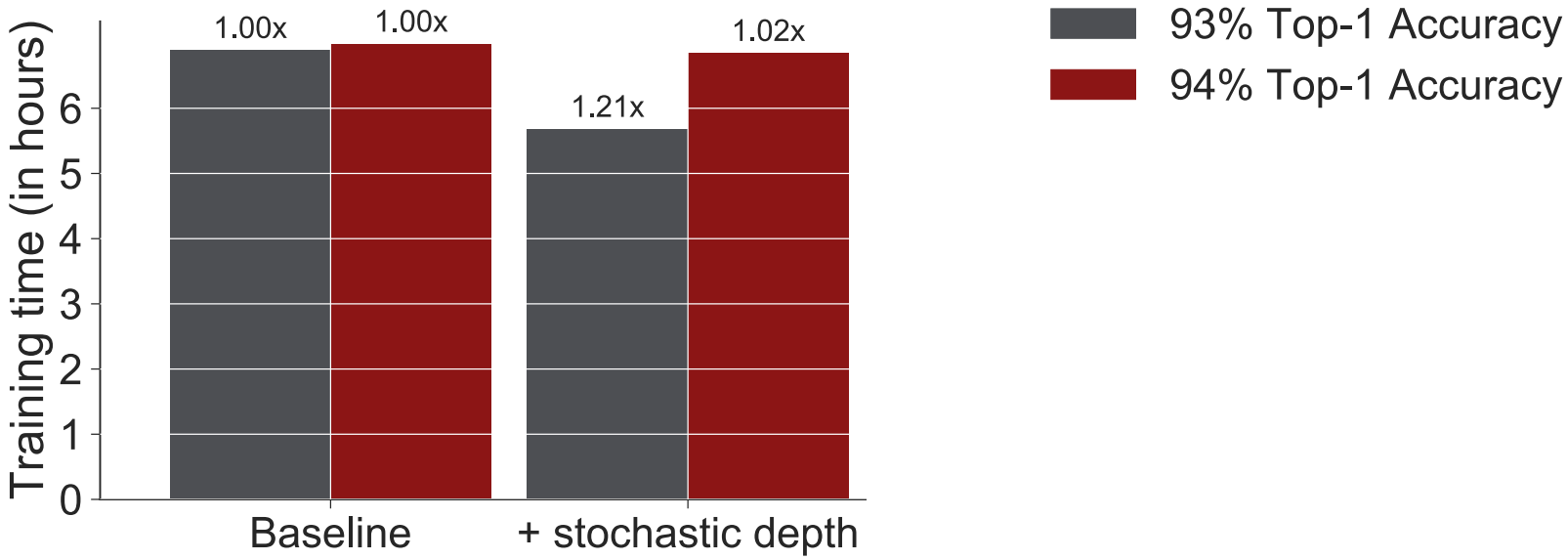
Does that give us a combined speed-up of **1011x**?

What if we combine optimizations?



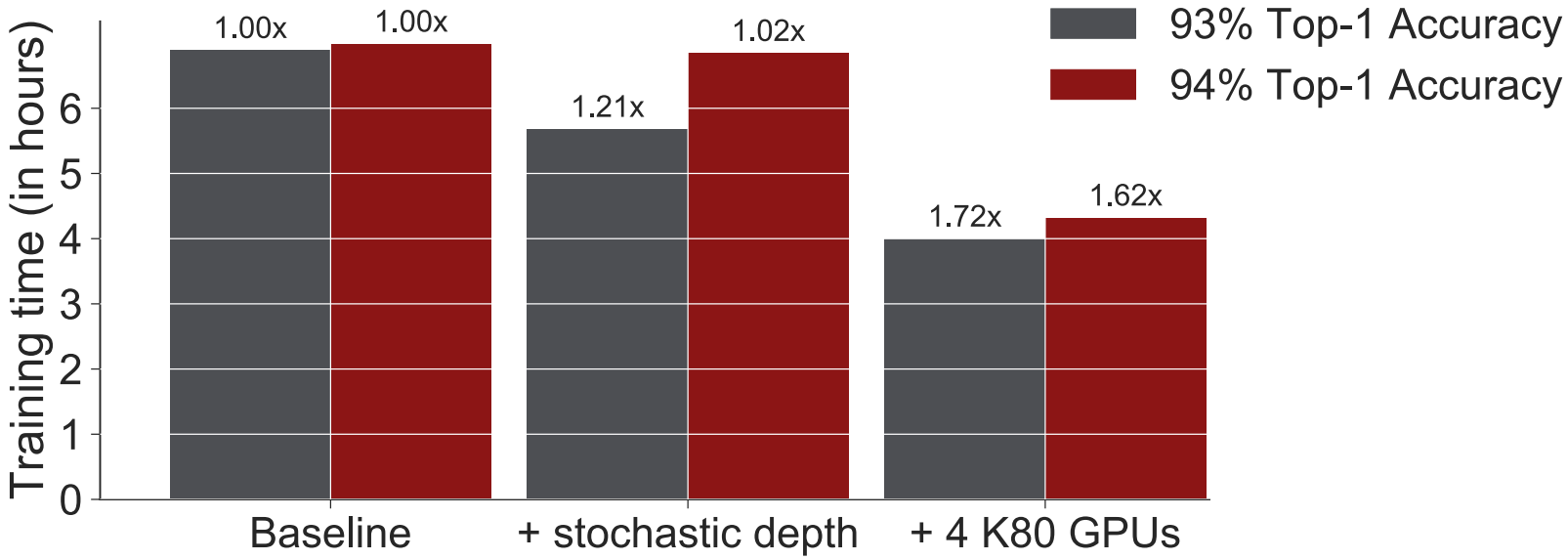
End-to-end training of ResNet110 on CIFAR10 in PyTorch, where the baseline is on machine with a single K80 and a batch size of 128.

What if we combine optimizations?



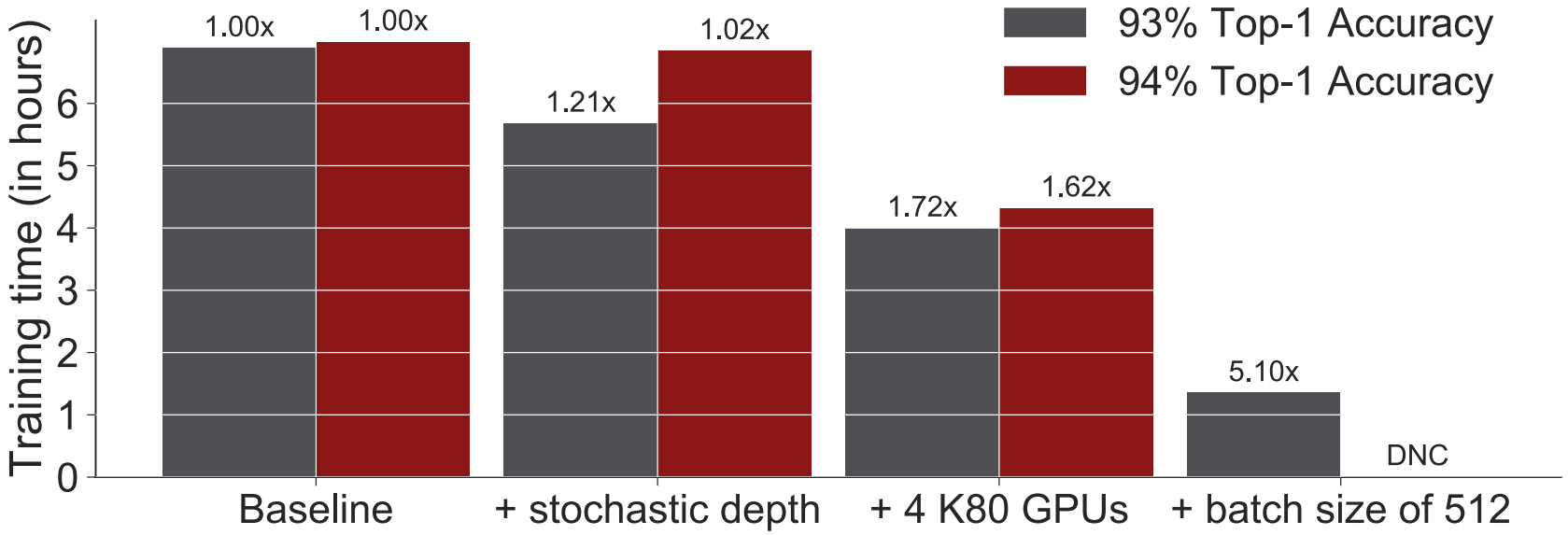
End-to-end training of ResNet110 on CIFAR10 in PyTorch, where the baseline is on machine with a single K80 and a batch size of 128.

What if we combine optimizations?



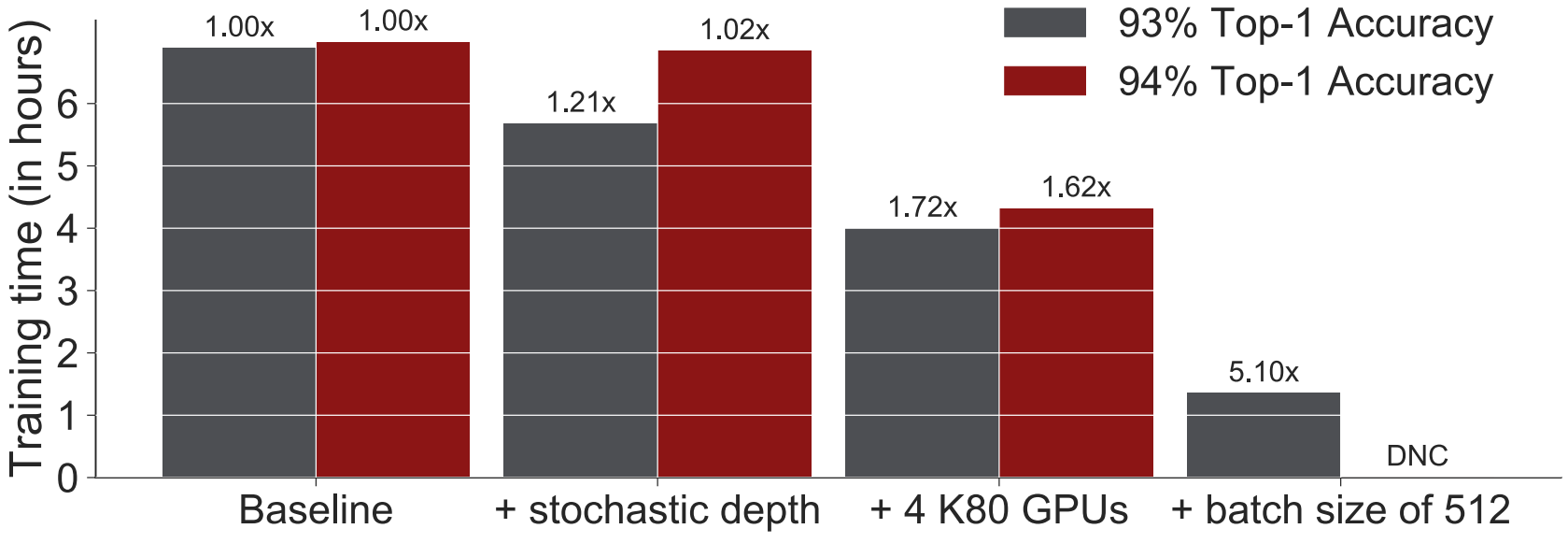
End-to-end training of ResNet110 on CIFAR10 in PyTorch, where the baseline is on machine with a single K80 and a batch size of 128.

What if we combine optimizations?

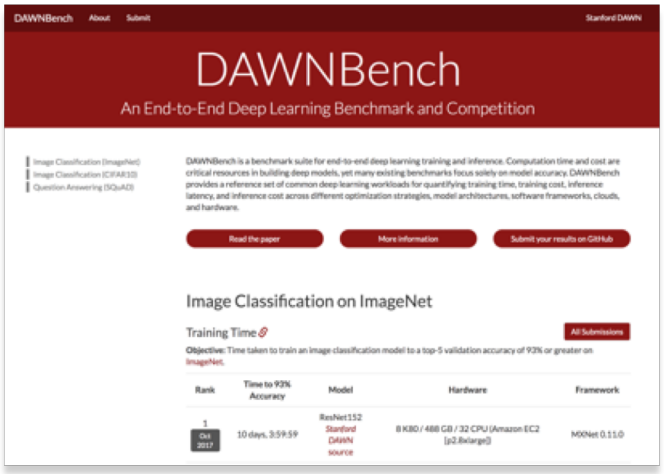


End-to-end training of ResNet110 on CIFAR10 in PyTorch, where the baseline is on machine with a single K80 and a batch size of 128.

What if we combine optimizations?



Optimizations interact in non-trivial ways



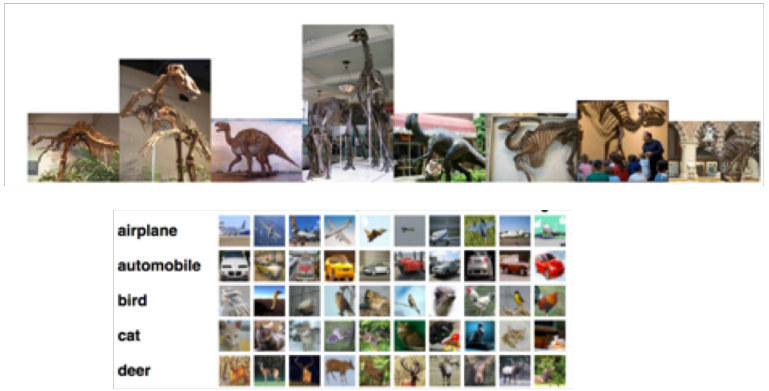
- First benchmark to measure time and cost to get a state-of-the-art accuracy
- Our goal: measure end-to-end throughput subject to accuracy

As an initial release

Tasks

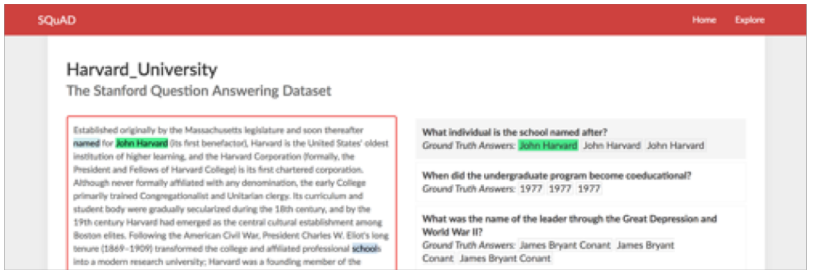
Image classification

- ImageNet
- CIFAR10



Question answering

- SQuAD



For each task

Accuracy threshold close to the state-of-the-art

Metrics

Training time

Training cost (USD)

Inference latency

Inference cost (USD)

dawn.cs.stanford.edu/benchmark