



MXNET-AMP AND MXNET-TRT FOR FAST DEEP LEARNING TRAINING AND INFERENCE

Wenming Ye (Amazon), Przemysław Tredak (NVIDIA), 11.06.2019

DEEP LEARNING FRAMEWORKS

The landscape



DEEP LEARNING FRAMEWORKS

The landscape



DEEP LEARNING FRAMEWORKS



- Fully open source (Apache incubating project)
- Mixes imperative and declarative programming
- Combines ease of use with speed
- Ready for deployment



APACHE MXNET

Mixing programming paradigms



Static graph



Hybrid



Imperative

APACHE MXNET

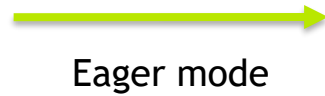
Mixing programming paradigms



GLUON



Static graph



Hybrid



Imperative

APACHE MXNET

Gluon Ecosystem

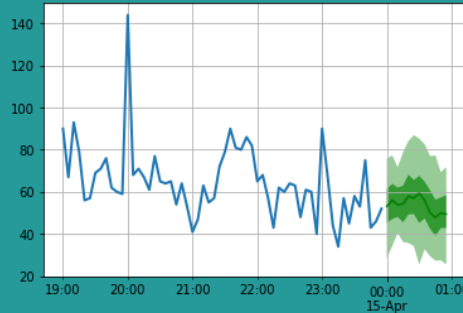


GluonCV



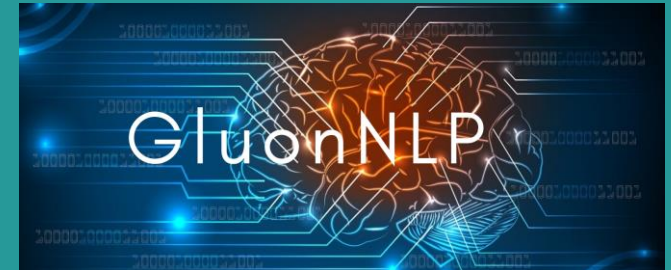
- Over 190 pretrained models with SOTA accuracy
- Training scripts
- Tutorials

GluonTS



- Time series forecasting

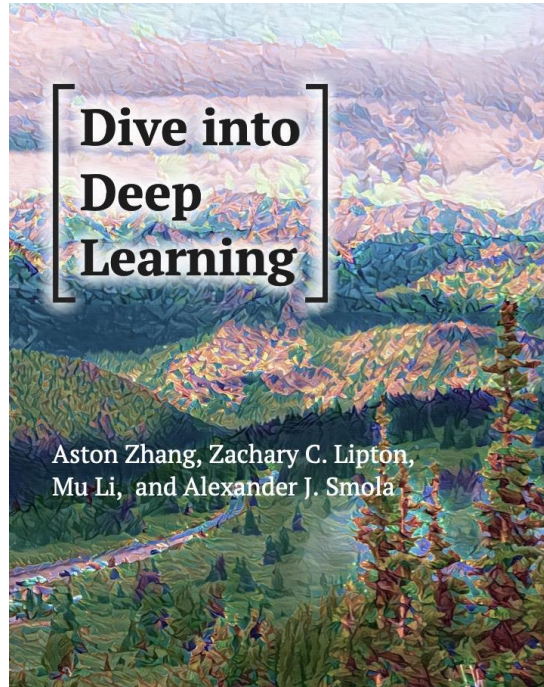
GluonNLP



- Word embeddings
- Language models
- Neural Machine Translation
- BERT, ELMO, XLNet, ERNIE, GPT-2...
- Reproducible scripts

APACHE MXNET

Gluon Ecosystem



<http://d2l.ai>

MULTI-LANGUAGE SUPPORT

Java

Perl

Julia

Clojure

Python

Scala

C++

R

Frontend

Backend

C++

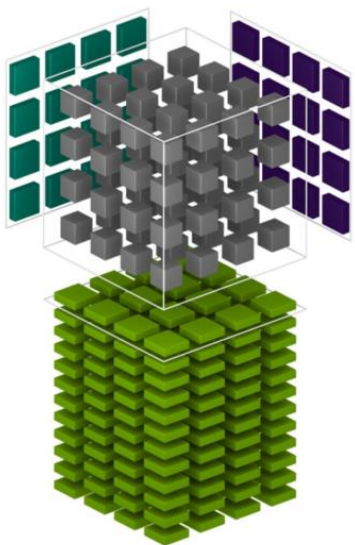
An abstract background image featuring a network of glowing green nodes connected by thin, intersecting lines. The nodes are scattered across the frame, with some appearing as bright green dots and others as slightly larger, blurred circles. The lines are thin and green, creating a complex web of connections. The overall color palette is dark, with the green elements standing out prominently.

AUTOMATIC MIXED PRECISION IN MXNET

TENSORCORES AND MIXED PRECISION

Starting with Volta, NVIDIA GPUs feature
TensorCores

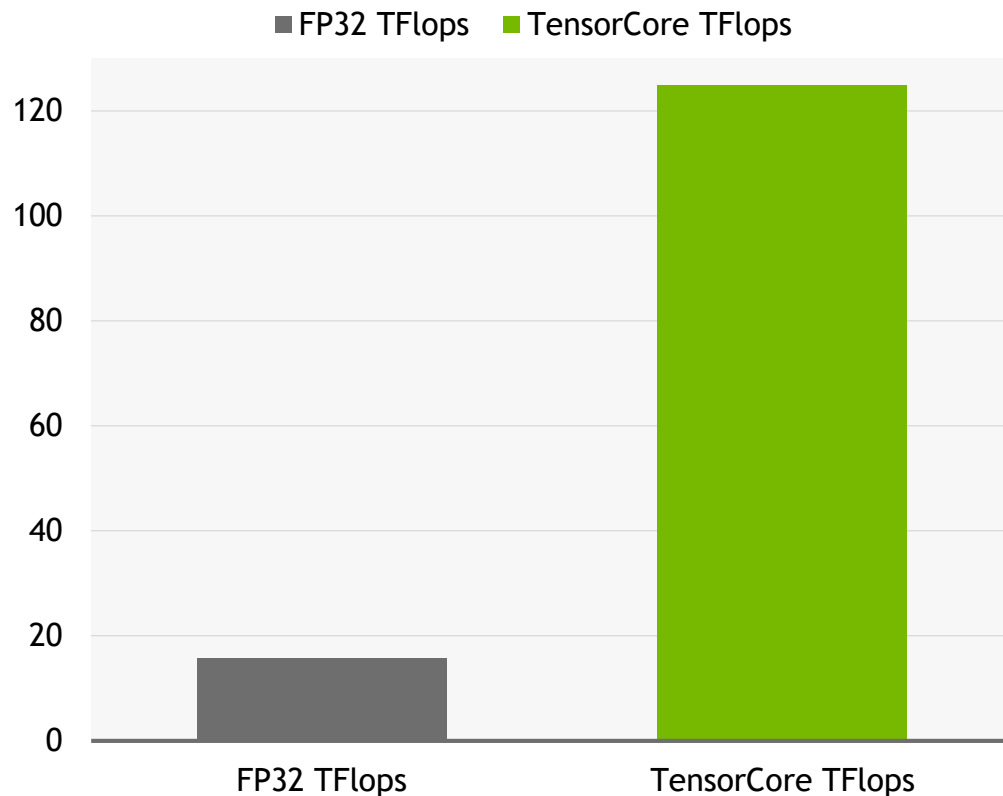
They greatly speed up matrix multiplication



Using them requires **mixed precision**

$$D = \begin{pmatrix} A_{0,0} & A_{0,1} & A_{0,2} & A_{0,3} \\ A_{1,0} & A_{1,1} & A_{1,2} & A_{1,3} \\ A_{2,0} & A_{2,1} & A_{2,2} & A_{2,3} \\ A_{3,0} & A_{3,1} & A_{3,2} & A_{3,3} \end{pmatrix} \begin{pmatrix} B_{0,0} & B_{0,1} & B_{0,2} & B_{0,3} \\ B_{1,0} & B_{1,1} & B_{1,2} & B_{1,3} \\ B_{2,0} & B_{2,1} & B_{2,2} & B_{2,3} \\ B_{3,0} & B_{3,1} & B_{3,2} & B_{3,3} \end{pmatrix} + \begin{pmatrix} C_{0,0} & C_{0,1} & C_{0,2} & C_{0,3} \\ C_{1,0} & C_{1,1} & C_{1,2} & C_{1,3} \\ C_{2,0} & C_{2,1} & C_{2,2} & C_{2,3} \\ C_{3,0} & C_{3,1} & C_{3,2} & C_{3,3} \end{pmatrix}$$

FP16 or FP32 FP16 FP16 FP16 or FP32



MIXED PRECISION RECIPE

Theory

- ▶ Cast input to FP16, cast back to FP32 before the softmax
- ▶ Keep “master copy” of the weights in FP32
- ▶ Scale the loss to keep gradients in FP16 dynamic range

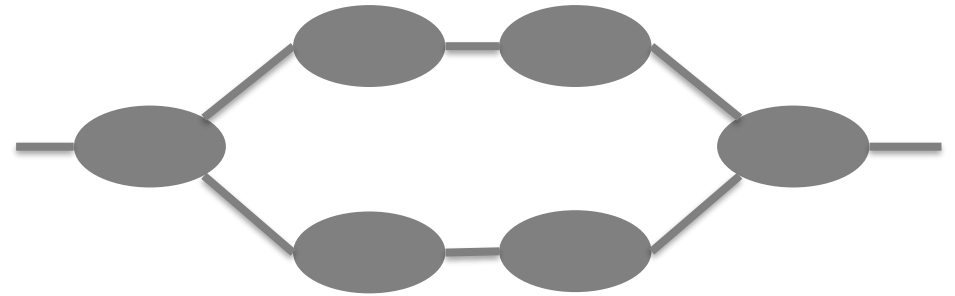
MIXED PRECISION RECIPE

In practice

- ▶ Cast input to FP16, cast back to FP32 before the softmax
 - ▶ What about Norm, Mean, etc.?
- ▶ Keep “master copy” of the weights in FP32
 - ▶ `optimizer.multi_precision=True`
- ▶ Scale the loss to keep gradients in FP16 dynamic range
 - ▶ What should the loss scale be? How to make it dynamic?

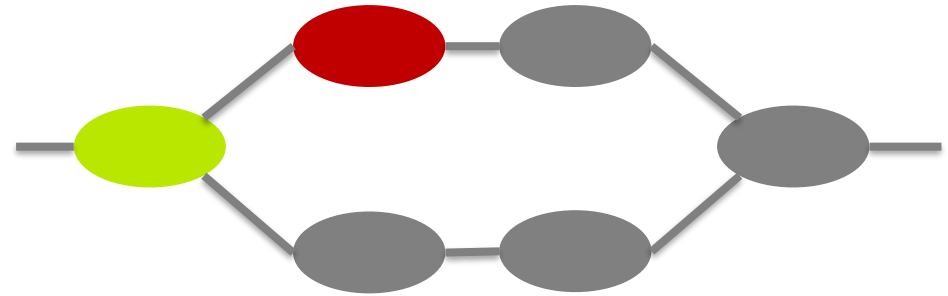
AMP: AUTOMATIC MIXED PRECISION

- ▶ Automatic casting of the model
 - ▶ Convolution, FullyConnected -> FP16
 - ▶ Norm, Mean, SoftMax, etc. -> FP32
 - ▶ Add, Mul etc. -> Cast to widest type
- ▶ Utilities for dynamic loss scaling



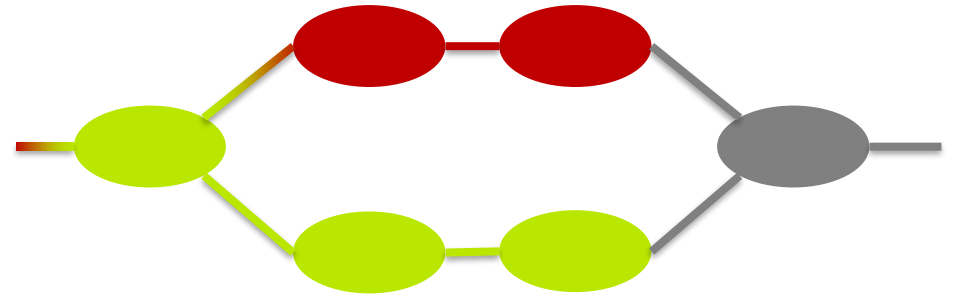
AMP: AUTOMATIC MIXED PRECISION

- ▶ Automatic casting of the model
 - ▶ Convolution, FullyConnected -> **FP16**
 - ▶ Norm, Mean, SoftMax, etc. -> **FP32**
 - ▶ Add, Mul etc. -> Cast to widest type
- ▶ Utilities for dynamic loss scaling



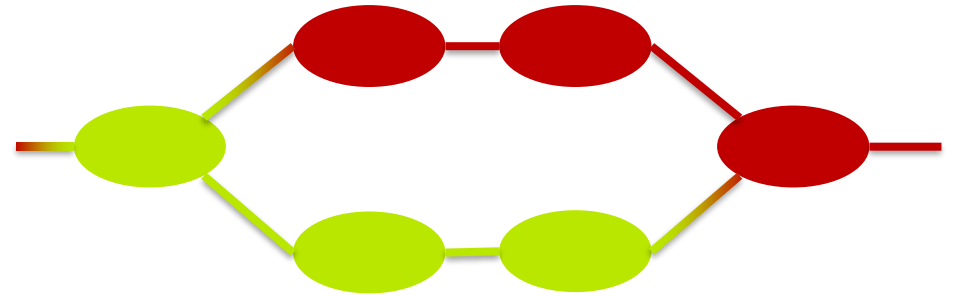
AMP: AUTOMATIC MIXED PRECISION

- ▶ Automatic casting of the model
 - ▶ Convolution, FullyConnected -> **FP16**
 - ▶ Norm, Mean, SoftMax, etc. -> **FP32**
 - ▶ Add, Mul etc. -> Cast to widest type
- ▶ Utilities for dynamic loss scaling



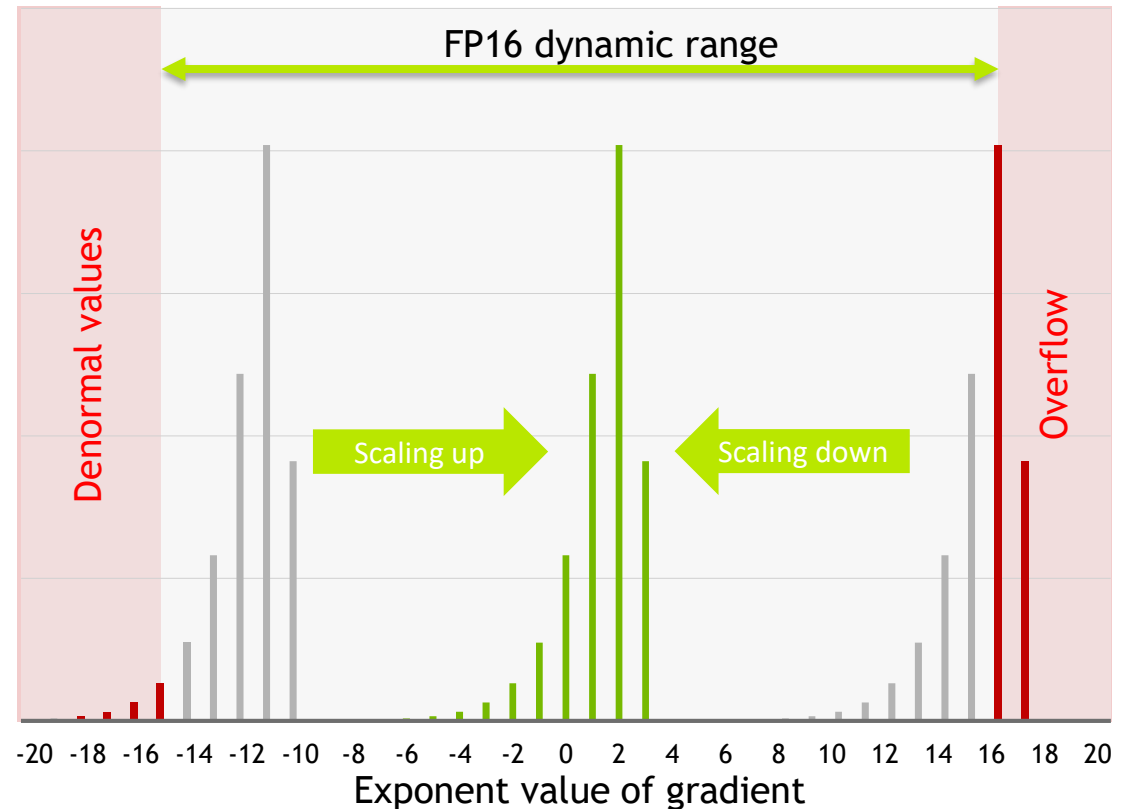
AMP: AUTOMATIC MIXED PRECISION

- ▶ Automatic casting of the model
 - ▶ Convolution, FullyConnected -> **FP16**
 - ▶ Norm, Mean, SoftMax, etc. -> **FP32**
 - ▶ Add, Mul etc. -> Cast to widest type
- ▶ Utilities for dynamic loss scaling



AMP: AUTOMATIC MIXED PRECISION

- ▶ Automatic casting of the model
 - ▶ Convolution, FullyConnected -> **FP16**
 - ▶ Norm, Mean, SoftMax, etc. -> **FP32**
 - ▶ Add, Mul etc. -> Cast to widest type
- ▶ Utilities for dynamic loss scaling



MIXED PRECISION RECIPE

AMP

```
net = get_network()  
trainer = mx.gluon.Trainer(...)  
  
with autograd.record(True):  
    out = net(data)  
    l = loss(out, label)  
  
autograd.backward(loss)
```

MIXED PRECISION RECIPE

AMP

```
amp.init()

net = get_network()

trainer = mx.gluon.Trainer(...)

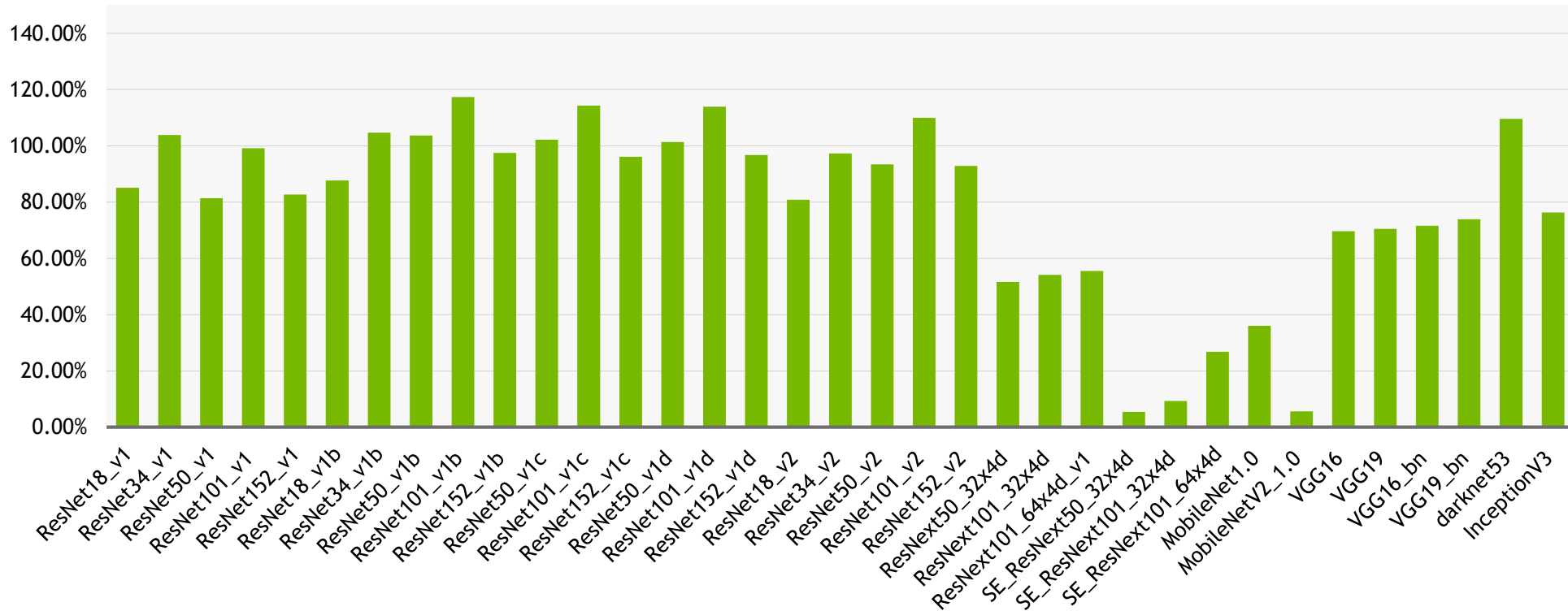
amp.init_trainer(trainer)

with autograd.record(True):
    out = net(data)
    l = loss(out, label)

    with amp.loss_scale(loss, trainer) as scaled_loss:
        autograd.backward(scaled_loss)
```

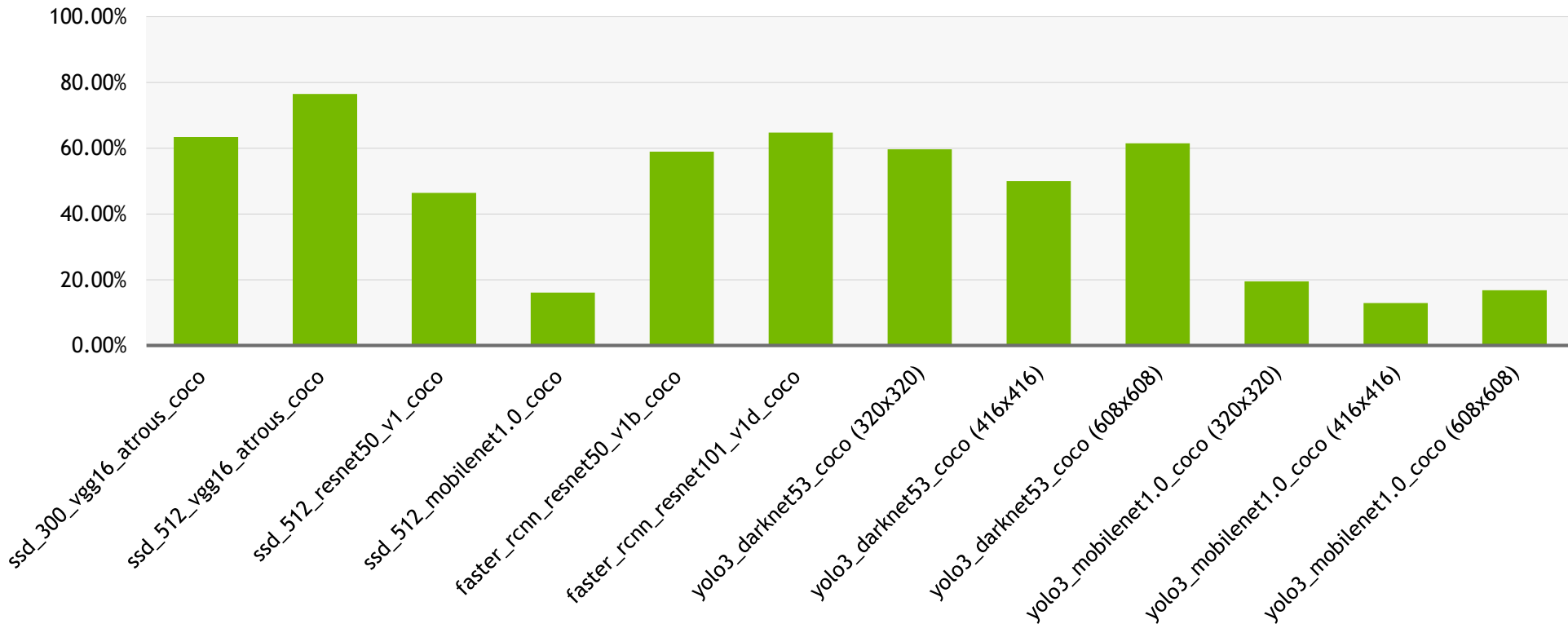

PERFORMANCE - CLASSIFICATION

Speedup when using AMP (single GPU, same batch size)



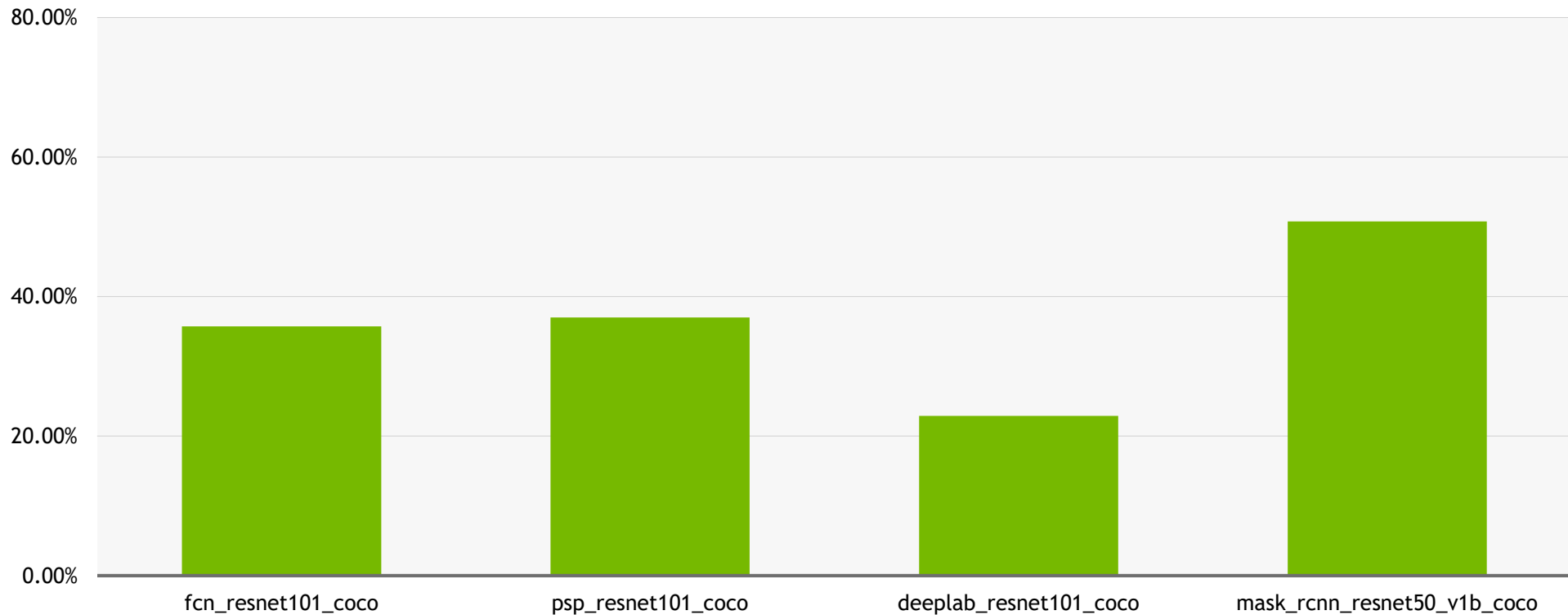
PERFORMANCE - DETECTION

Speedup when using AMP (single GPU, same batch size)



PERFORMANCE - SEGMENTATION

Speedup when using AMP (single GPU, same batch size)



An abstract network diagram with green nodes and lines on a dark background. The nodes are represented by small green circles, some of which are larger and more prominent. They are connected by thin, light green lines that crisscross the frame, creating a complex web of connections. The background is a deep black, with some faint, larger green circles that appear to be out of focus or part of the network's structure.

MXNET-AMP: LOOKING FORWARD

TRAINING OPTIMIZATION

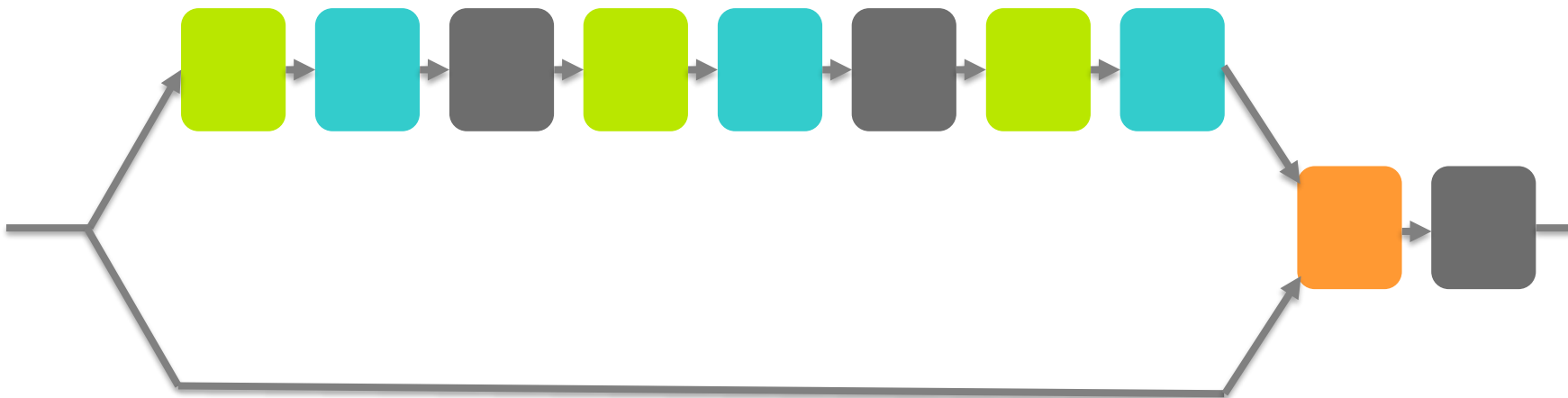
MLPerf



MLPerf

TRAINING OPTIMIZATION

MLPerf ResNet 50 under the hood



Convolution

BatchNorm

ReLU

Add

TRAINING OPTIMIZATION

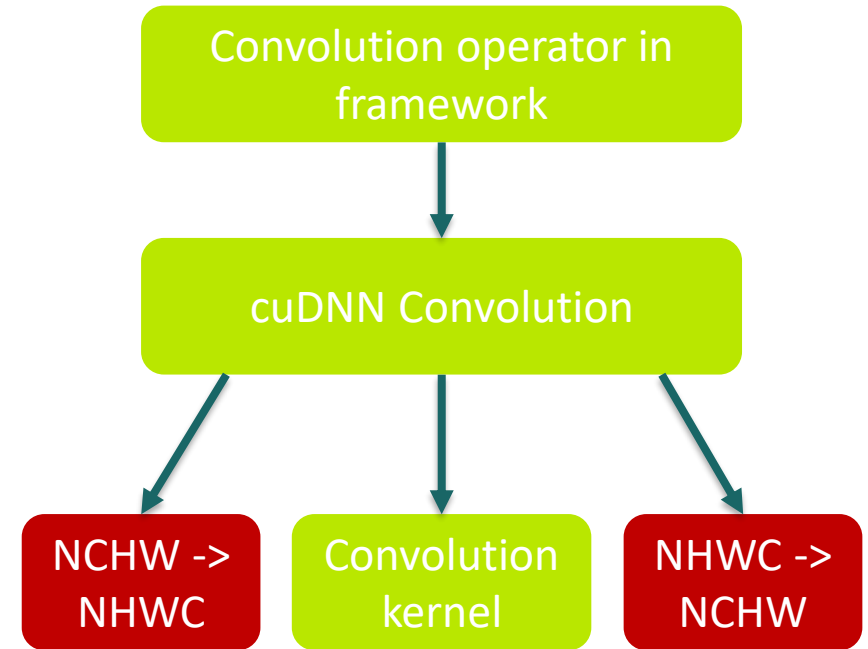
Anatomy of TensorCore convolution



TRAINING OPTIMIZATION

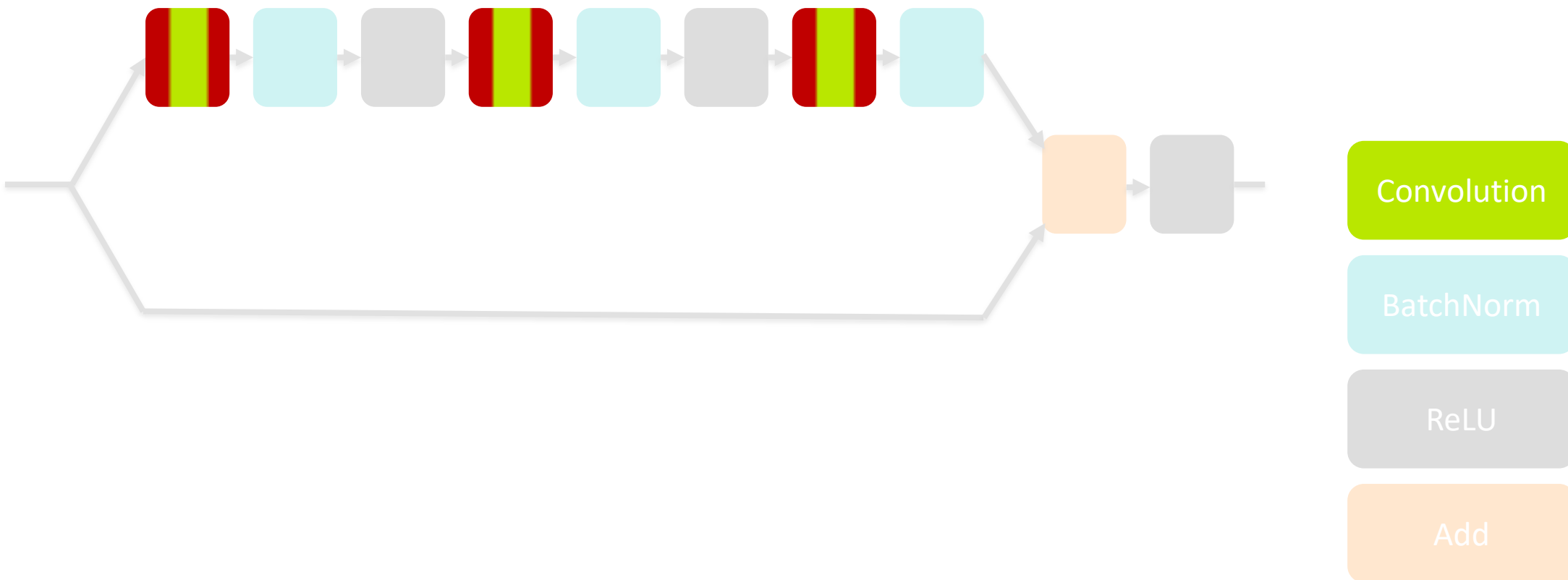
Anatomy of TensorCore convolution

- TensorCore convolution requires special data layout (NHWC) for efficient loading of inputs and weights
- DL frameworks use NCHW data layout, since it is the best choice for FP32
- cuDNN internally transposes data before calling convolution kernel



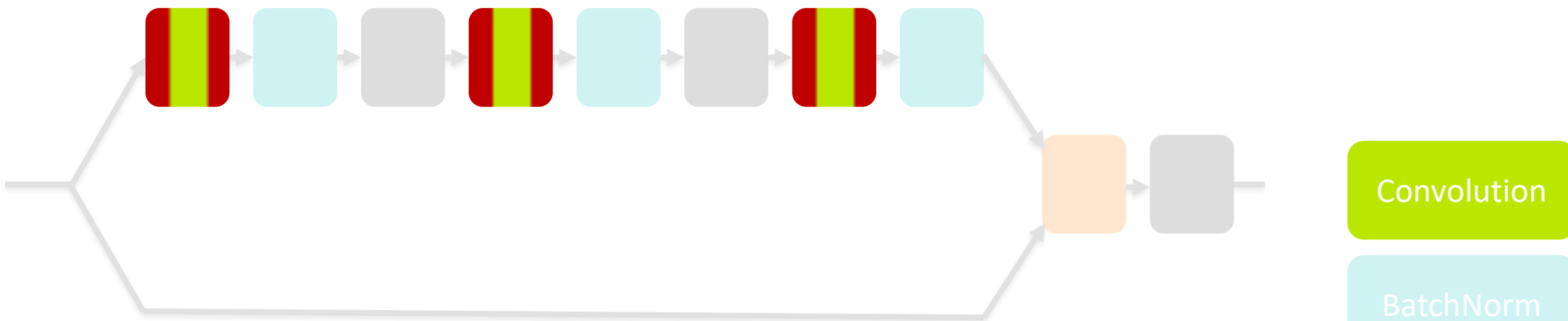
TRAINING OPTIMIZATION

Anatomy of TensorCore convolution



TRAINING OPTIMIZATION

Anatomy of TensorCore convolution



AMP should use NHWC!

TRAINING OPTIMIZATION

Automatic Layout Management

NCHW Conv

BN axis 1

ReLU

NCHW Conv

BN axis 1

ReLU

NCHW Conv

TRAINING OPTIMIZATION

Automatic Layout Management

NCHW->NHWC

NHWC Conv

NHWC->NCHW

BN axis 1

ReLU

NCHW->NHWC

NHWC Conv

NHWC->NCHW

BN axis 1

ReLU

NCHW->NHWC

NHWC Conv

NHWC->NCHW

TRAINING OPTIMIZATION

Automatic Layout Management

NCHW->NHWC
NHWC Conv

BN axis 3
NHWC->NCHW
NCHW->NHWC
ReLU

NHWC Conv

BN axis 3
NHWC->NCHW
NCHW->NHWC
ReLU

NHWC Conv
NHWC->NCHW

TRAINING OPTIMIZATION

Automatic Layout Management

NCHW->NHWC

NHWC Conv

BN axis 3

ReLU

NHWC Conv

BN axis 3

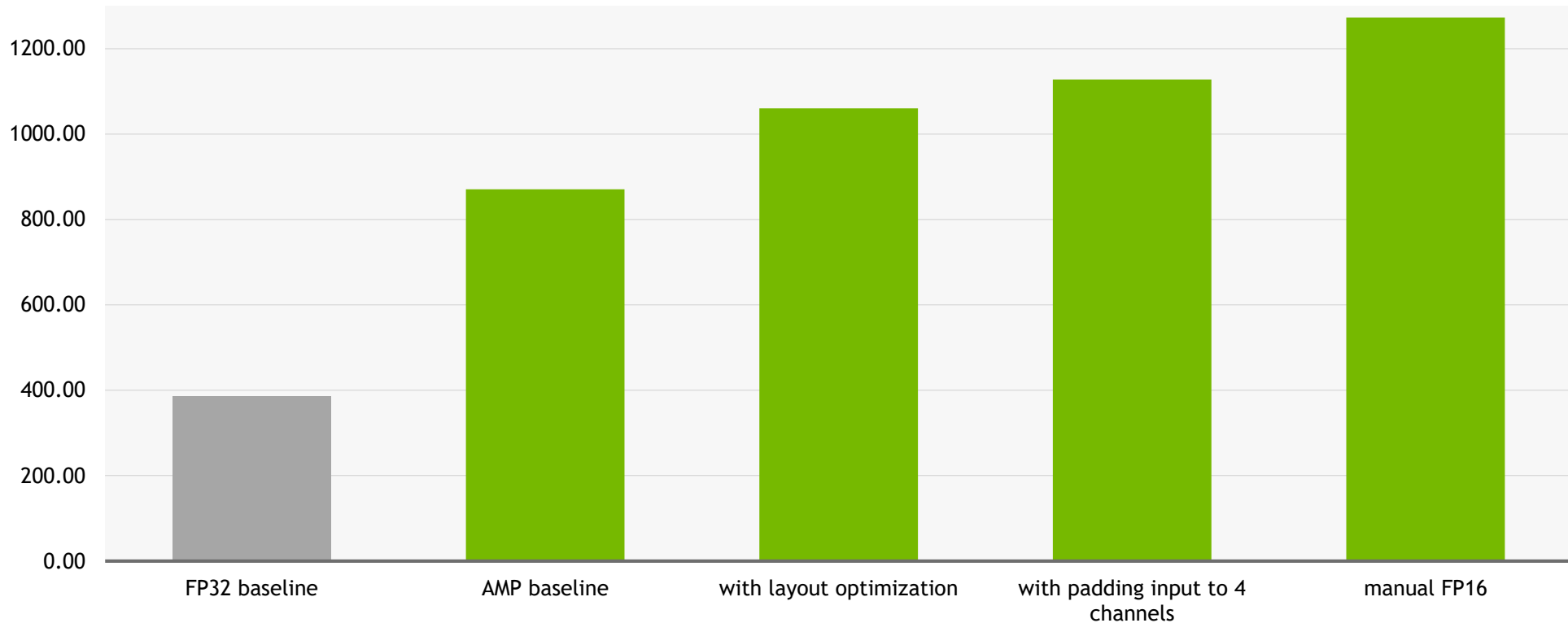
ReLU

NHWC Conv

NHWC->NCHW

PRELIMINARY PERFORMANCE RESULTS

ResNet-50, batch size 128



The background of the slide is a dark blue field with a complex network of thin, light green lines. These lines connect various points, some of which are highlighted as bright green dots. The overall effect is a sense of a dynamic, interconnected system, possibly representing a neural network or a data flow.

MXNET TENSORRT INTEGRATION

INFERENCE OPTIMIZATION

Similar to training optimizations but with a few extras

- At inference time we make use of same speedups as in training (TensorCores)

Inference time only:

Operators have fixed values after training (additional opportunity to fuse operators)

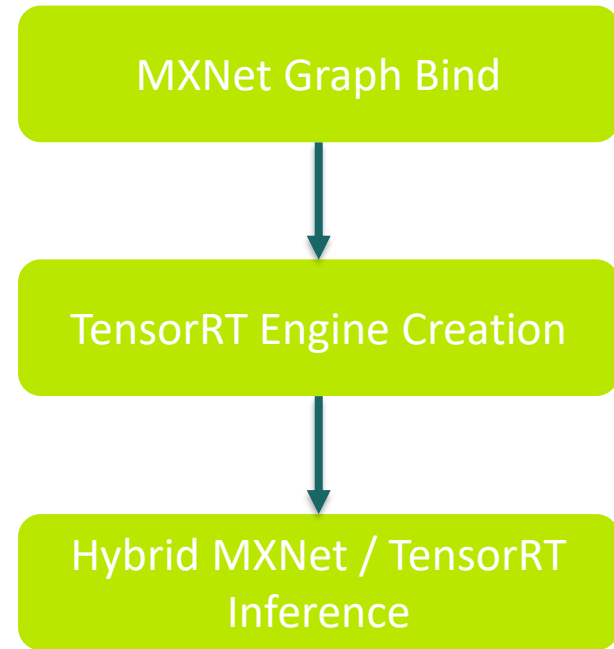
Many intermediate memory buffers needed for gradient storage can be removed (yields a storage savings, and potentially memory read / write savings)

- Our goal is to get many of these inference-only optimization for free by integrating with TensorRT (has been shown to provide dramatic inference-time speedups across a variety of workloads)
- We also want to enable these speedups with minimal developer effort or code change

MXNET TENSORRT INTEGRATION

Subgraph compilation

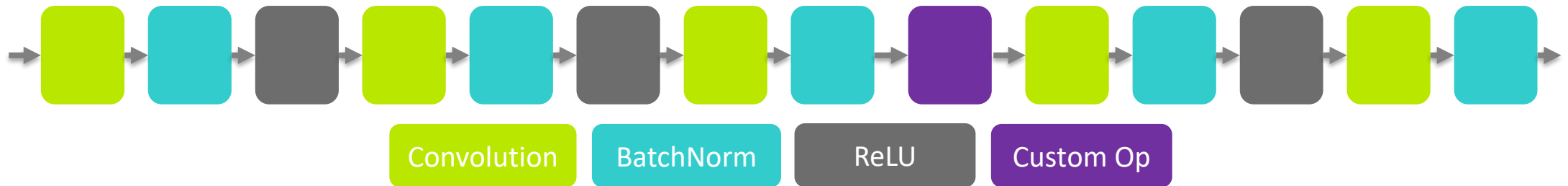
- The MXNet community developed a generic subgraph-API for MXNet and TensorRT has been integrated as a first class citizen (Supports Vendor specific implementation)
- This API lets vendor devs tell MXNet which operators are supported by their acceleration software
- While running inference MXNet will select subgraphs automatically based on this support level, and fall back to CUDA implementations for unsupported or custom operators (region of interest, tasks, etc)



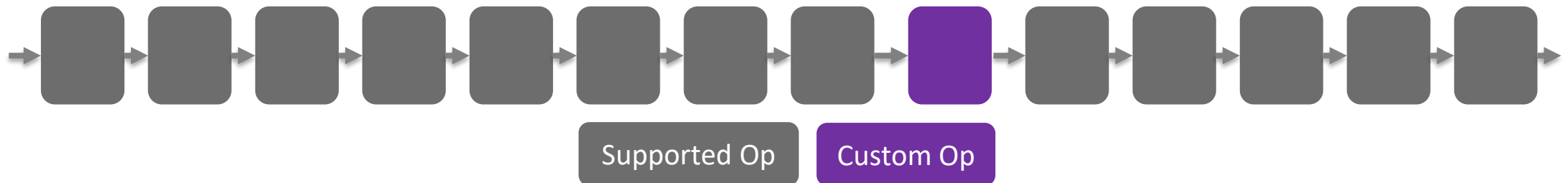
MXNET TENSORRT INTEGRATION

Subgraph compilation - under the hood

- We start with a normal computation graph for an MXNet model



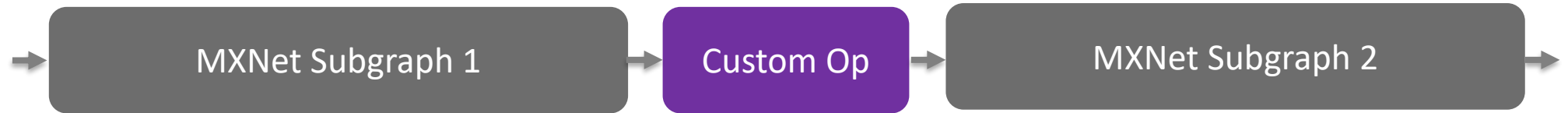
- We search for subgraphs of supported operators



MXNET TENSORRT INTEGRATION

Subgraph compilation - under the hood cont.

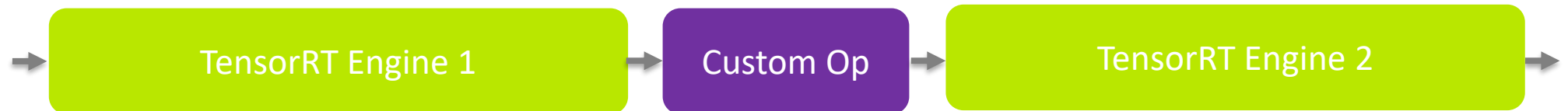
- We now have our full graph partitioned into a number of subgraphs



- We convert the operators in each subgraph to ONNX

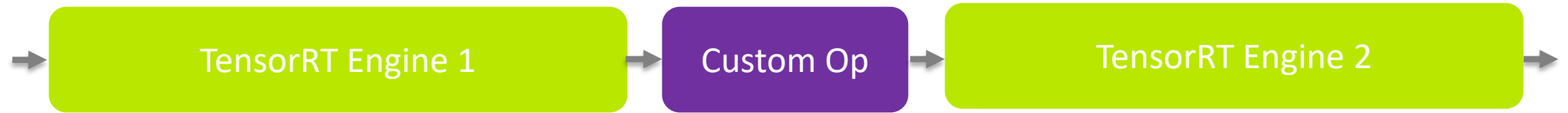


- Finally, we convert the ONNX graphs to TensorRT engines using the open source ONNX-TensorRT project



MXNET TENSORRT INTEGRATION

Subgraph compilation



- When inference is run on this graph all TensorRT engine nodes will be executed with a library call to TensorRT
- Many standard models will collapse the entire graph to a single TensorRT node
- Custom ops or unsupported ops will continue to run as normal, using their default MXNet CUDA implementations
- All of this conversion happens with little coding effort required by MXNet users
- This process is focused on high-performance inference scenarios, so the API is exposed from Python, our C API, and our C++ package

TENSORRT INTEGRATION RECIPE

Normal Inference in MXNet

```
executor = sym.simple_bind(ctx=mx.gpu(), data=batch_shape, grad_req='null',  
                           force_rebind=True)  
executor.copy_params_from(arg_params, aux_params)  
y_gen = executor.forward(is_train=False, data=input)
```

TENSORRT INTEGRATION RECIPE

TensorRT enabled Inference in MXNet

```
trt_sym = sym.get_backend_symbol('TensorRT')
arg_params, aux_params = mx.contrib.tensorrt.init_tensorrt_params(trt_sym,
                                                                    arg_params,
                                                                    aux_params)
executor = trt_sym.simple_bind(ctx=mx.gpu(), data=batch_shape, grad_req='null',
                               force_rebind=True)
executor.copy_params_from(arg_params, aux_params)
y_gen = executor.forward(is_train=False, data=input)
```

PERFORMANCE - CLASSIFICATION

Speedup when using TensorRT Integration (single GPU, same batch size)

