NVIDIA

S9243

Fast and **Accurate** Object Detection

with PyTorch and TensorRT

Floris Chabert, Solutions Architect
Prethvi Kashinkunti, Solutions Architect

March 19 2019

OVERVIEW

Topics

What & Why?

- Problem
- Our solution

How?

- Architecture
- Performance
- Optimizations

Future

PROBLEM

Performance and Workflow

Lack of object detection codebase with **high accuracy** and **high performance**

- Single stage detectors (YOLO, SSD) - fast but low accuracy
- Region based models (faster, mask-RCNN) - high accuracy, low inference performance

No **end-to-end GPU** processing

- **Data loading** and pre-processing on CPU can be slow
- **Post-processing** on CPU is a performance bottleneck
- **Large tensors copy** between host and GPU memory is expensive

No **full detection workflow** integrating **NVIDIA optimized libraries** all together

- Using **DALI**, **Apex** and **TensorRT**

SOLUTION

End-to-End Object Detection

Fast and **accurate**

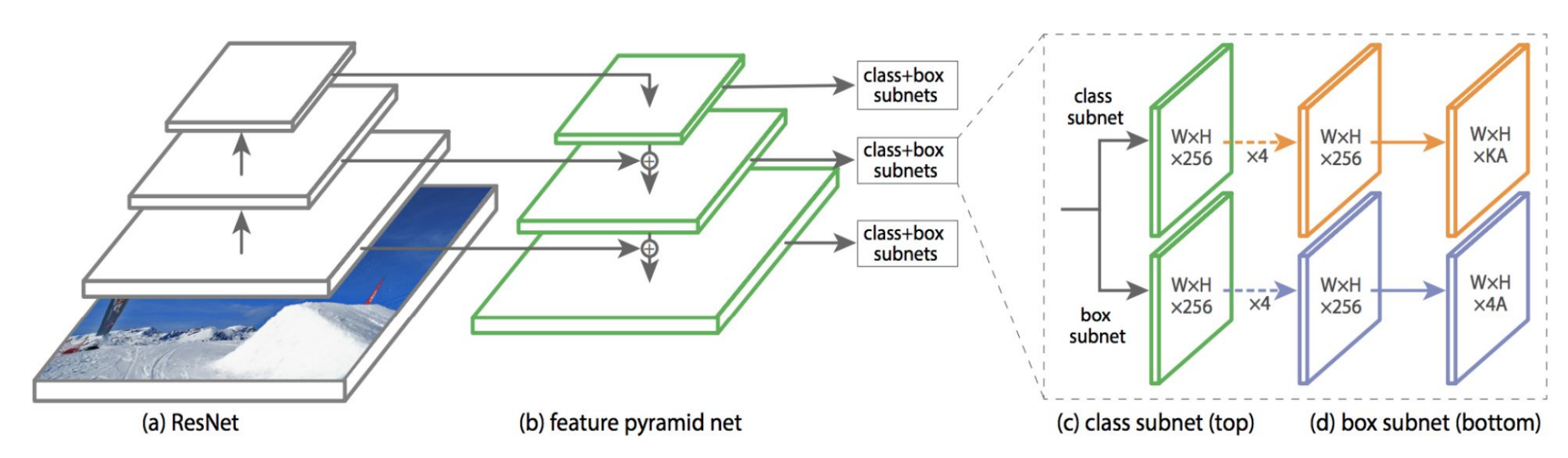
- **Single shot** object detector based on RetinaNet
- **Accuracy** similar to two-stages object detectors
- **End-to-end** optimized for GPU
- **Distributed** and **mixed precision** training and inference

Codebase

- **Open source**, easily **customizable** tools
- Written in **PyTorch/Apex** with **CUDA** extensions
- Production ready inference through **TensorRT**

ARCHITECTURE

RetinaNet



The one-stage RetinaNet network architecture [1] with FPN [2]

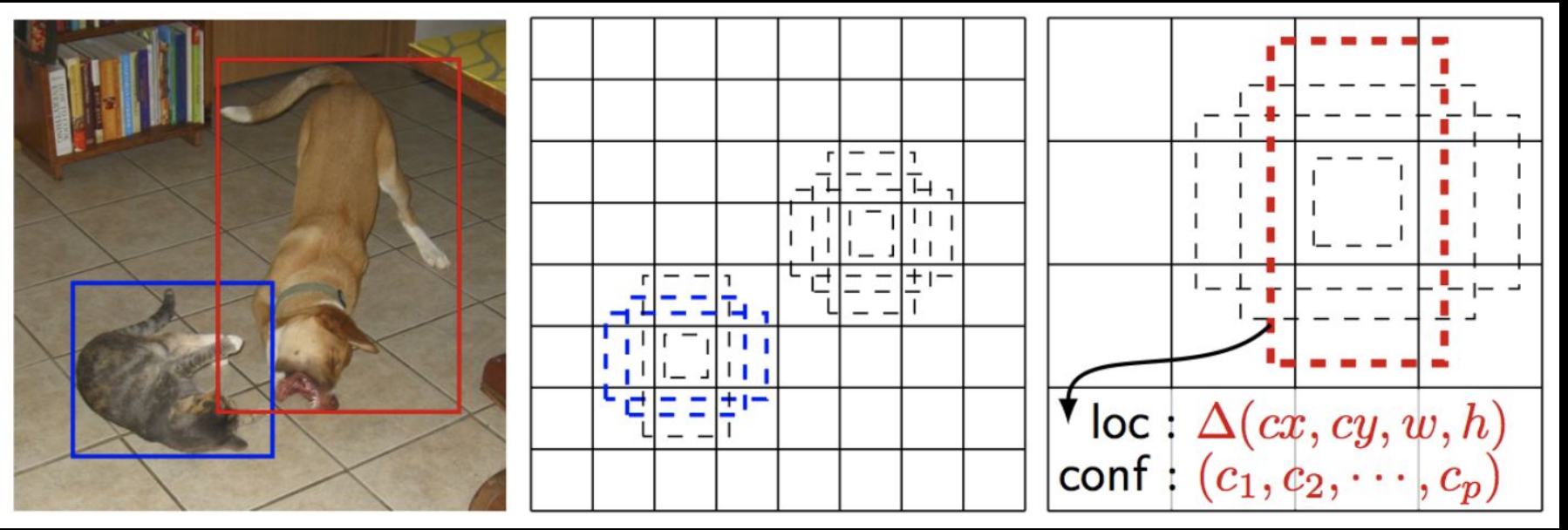
ARCHITECTURE

Single Shot Detection

YOLO detection model [3]

ARCHITECTURE

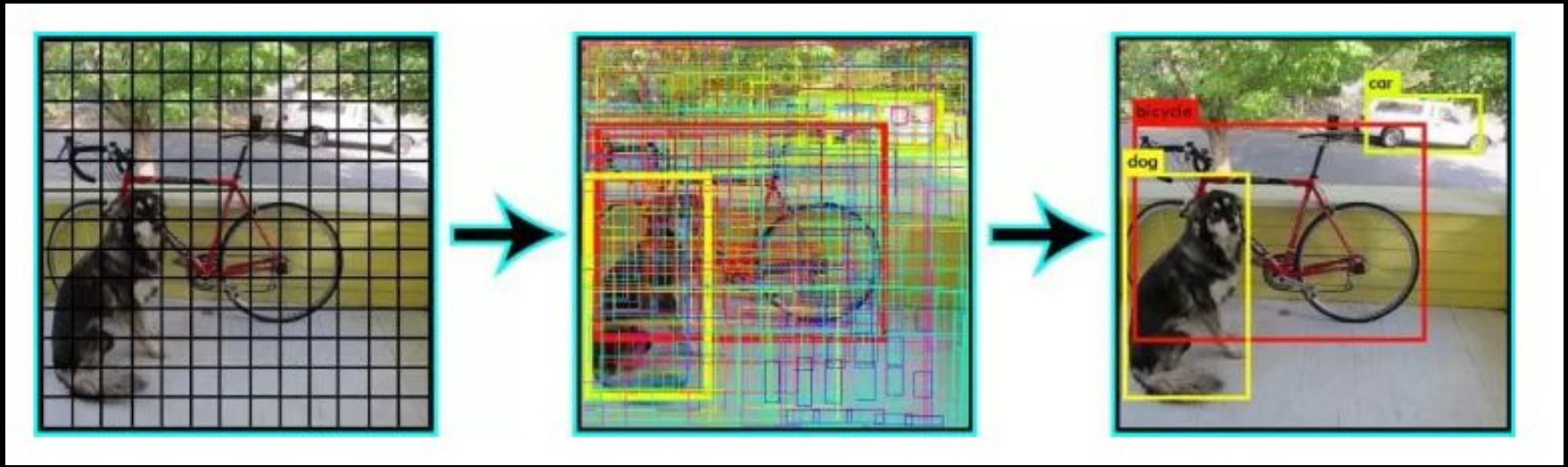
Bounding Boxes and Anchors



Single Shot MultiBox Detector framework [4]

ARCHITECTURE

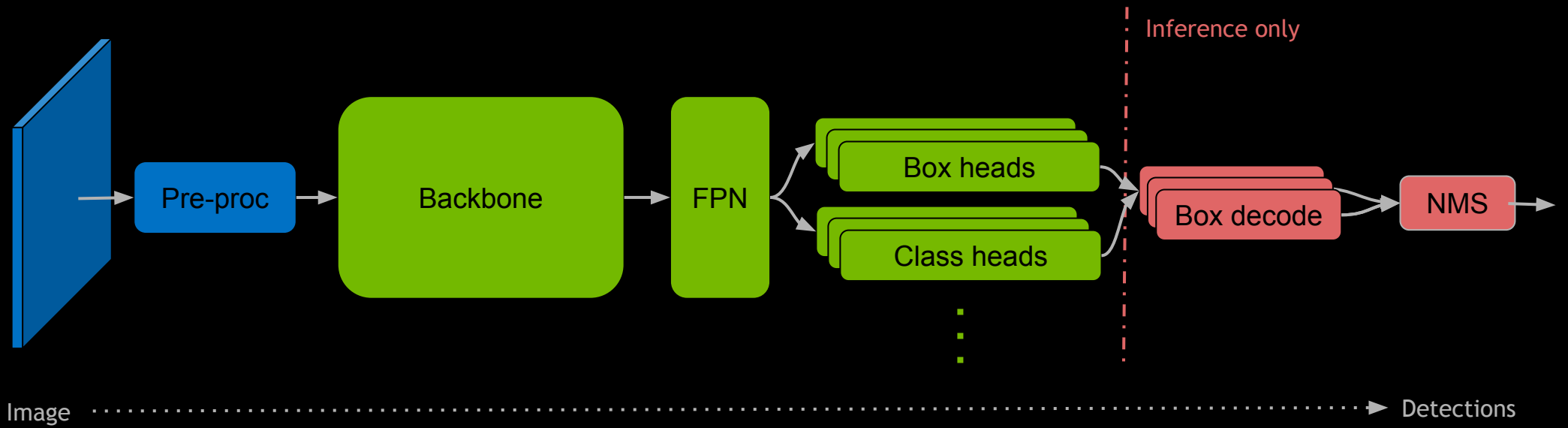
Non Maximum Suppression



YOLO detection model [3]

ARCHITECTURE

End-to-end GPU processing



ARCHITECTURE

PyTorch Forward Pass

```
def forward(self, x):
     if self.training: x, targets = x

     # Backbone and class/box heads
     features = self.backbone(x)
     cls_heads = [self.cls_head(t) for t in features]
     box_heads = [self.box_head(t) for t in features]

     if self.training:
          return self._compute_loss(x, cls_heads, box_heads, targets)

     # Decode and filter boxes
     decoded = []
     for cls_head, box_head in zip(cls_heads, box_heads):
          decoded.append(decode(cls_head.sigmoid(), box_head, stride,
               self.threshold, self.top_n, self.anchors[stride]))

     # Perform non-maximum suppression
     decoded = [torch.cat(tensors, 1) for tensors in zip(*decoded)]
     return nms(*decoded, self.nms, self.detections)
```

ARCHITECTURE

Features

Customizable backbone - easy accuracy vs performance trade-offs
- Supports variable feature maps and ensembles

End-to-end processing on the **GPU**

High performance through NVIDIA libraries/tools integration
- Optimized **pre-processing** with **DALI**
- **Mixed precision**, **distributed** training with **Apex**
- Easy model export to **TensorRT** for **inference** with **optimized post-processing**

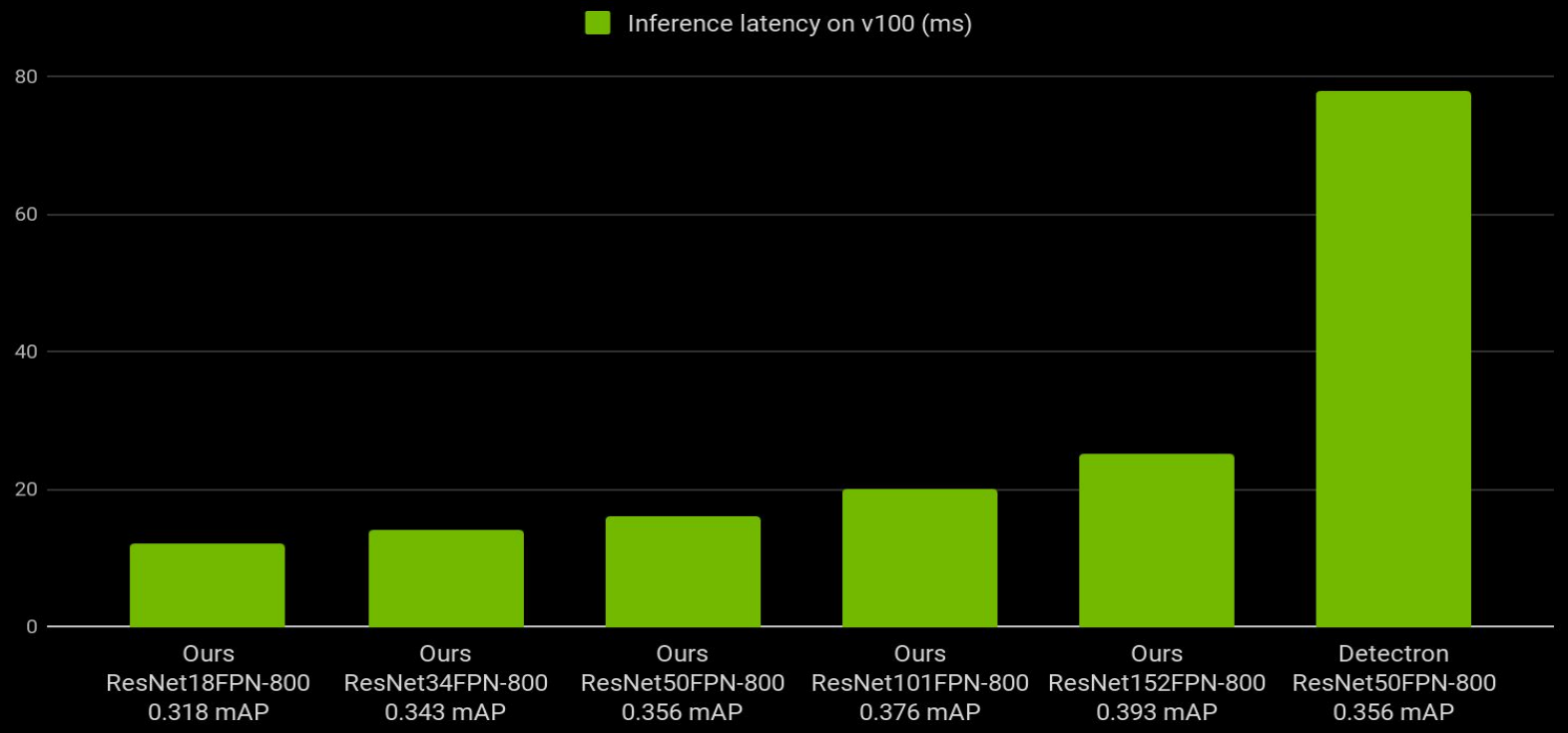
Light PyTorch codebase for research and customization
- With optimized **CUDA extensions** and plugins

PERFORMANCE

Training Time (lower is better)

PERFORMANCE

Inference Latency (lower is better)



WORKFLOW

Command Line Utility

- Training and evaluation

```
> retinanet train model.pth --images images_train/ --annotations annotations_train.json
> retinanet infer model.pth --images images_val/ --annotations annotations_val.json
```

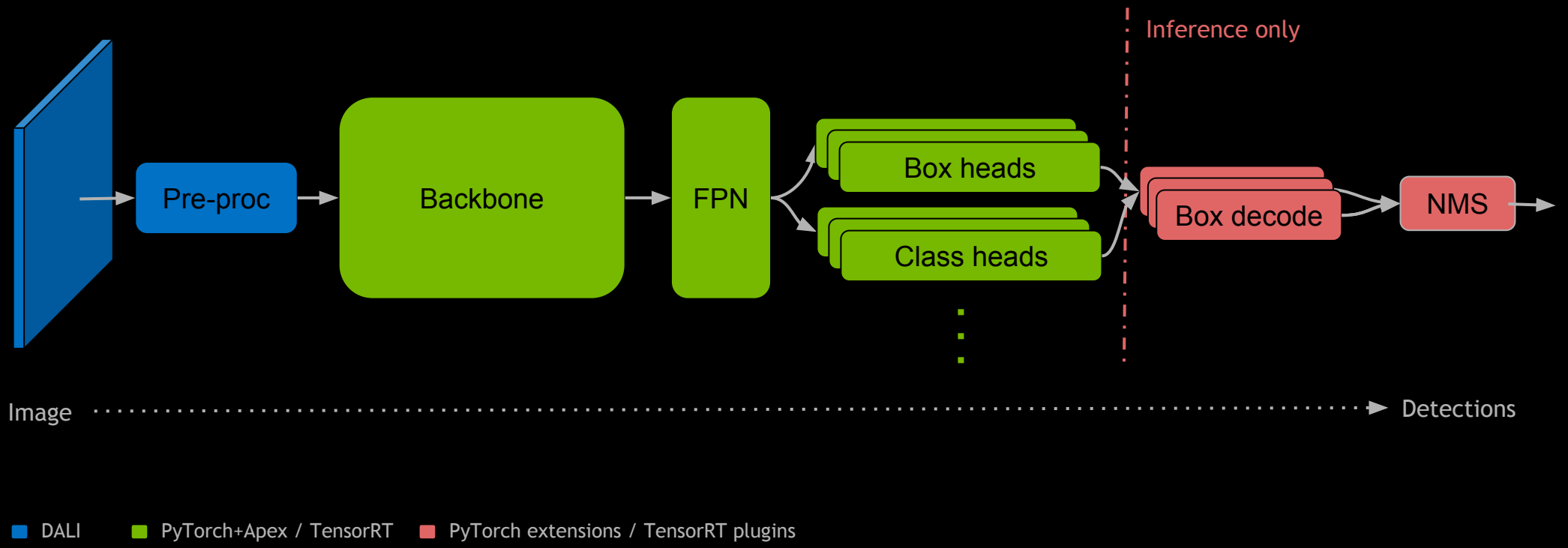
- Export to TensorRT and inference

```
> retinanet export model.pth engine.plan
> retinanet infer engine.plan --images images_prod/
```

- Production-ready inference engine

OPTIMIZATION

DALI, PyTorch+Apex, and TensorRT



DALI

Highly **optimized** open source library for **data preprocessing**

- **Execution engine** for fast preprocessing pipeline
- **Accelerated blocks** for image loading and augmentation
- **GPU support** for JPEG decoding and image manipulation

DALI

Pipeline Operators Definition

```
def __init__(self, batch_size, num_threads, device_id, training, *args):
     …
     self.decode = ops.nvJPEGDecoder(device="mixed", output_type=types.RGB)
     self.resize = ops.Resize(device="gpu", image_type=types.RGB, resize_longer=size)
     self.pad = ops.Paste(device="gpu", paste_x=0, paste_y=0, min_canvas_size=size)
     self.crop_norm = ops.CropMirrorNormalize(device="gpu", mean=mean, std=std,
          crop=size, image_type=types.RGB, output_dtype= types.FLOAT)

     if training:
          self.coin_flip = ops.CoinFlip(probability=0.5)
          self.horizontal_flip = ops.Flip(device="gpu")
          self.box_flip = ops.BbFlip(device="cpu)
```

DALI

Data Loading Graph

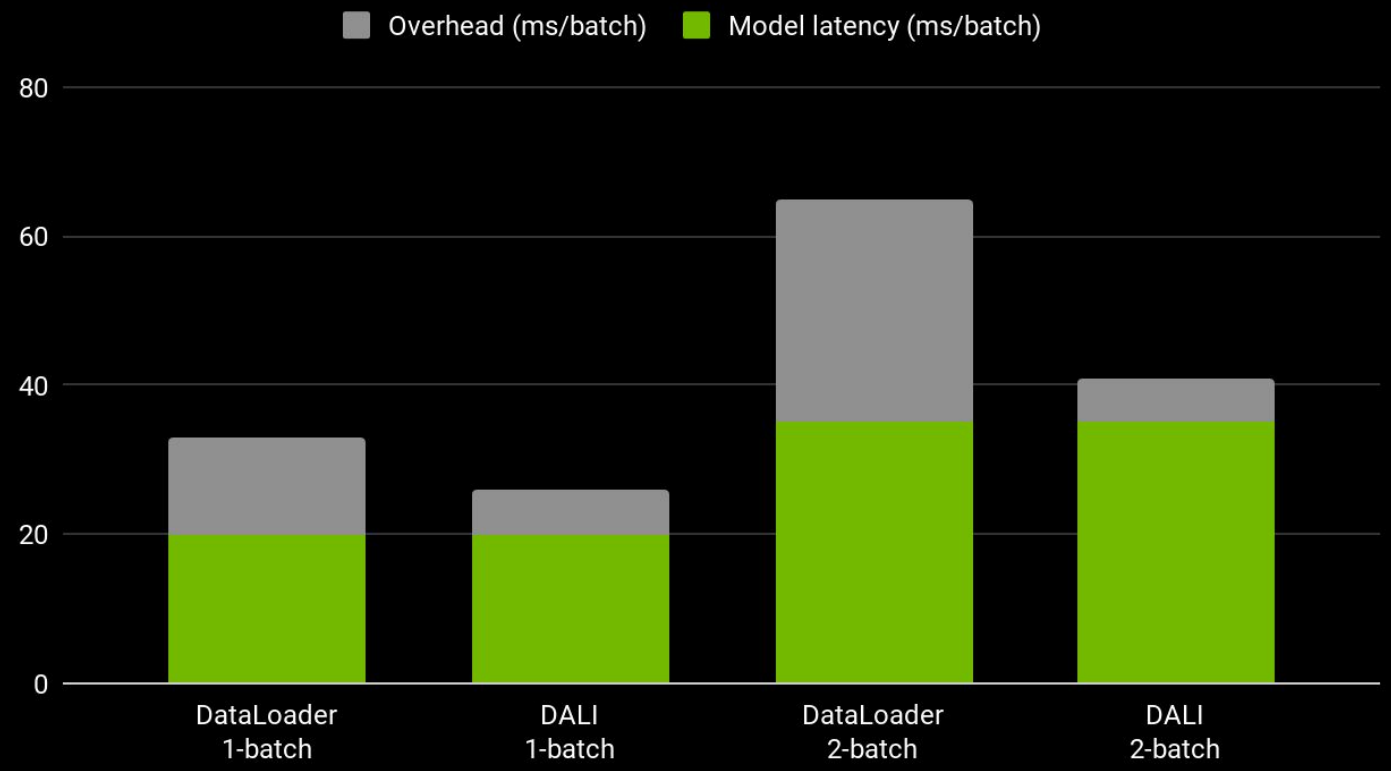
```
def define_graph(self):
     inputs, bboxes, labels, ids = self.input()
     images = self.decode(images)
     images = self.resize(images)

     if self.training:
          do_flip = self.coin_flip()
          images = self.image_flip(images, horizontal=do_flip)
          boxes = self.box_flip(boxes, horizontal=do_flip)

     images = self.pad(images)
     images = self.crop_norm(images)
     return images, boxes, labels, ids
```

DALI

Inference Latency (lower is better)



APEX

Library of utilities for PyTorch

- Optimized multi-process **distributed** training
- Streamlined **mixed precision** training
- And more...

APEX

Distributed Training

DistributedDataParallel wrapper

- Easy **multiprocess** distributed training
- Optimized for **NCCL**

```
def worker(rank, args, world, model, state):
   if torch.cuda.is_available():
       torch.cuda.set_device(rank)
       torch.distributed.init_process_group(backend='nccl', init_method='env://')

torch.multiprocessing.spawn(worker, args=(args, world, model, state), nprocs=world)
```

APEX

Mixed Precision

Safe and optimized mixed precision

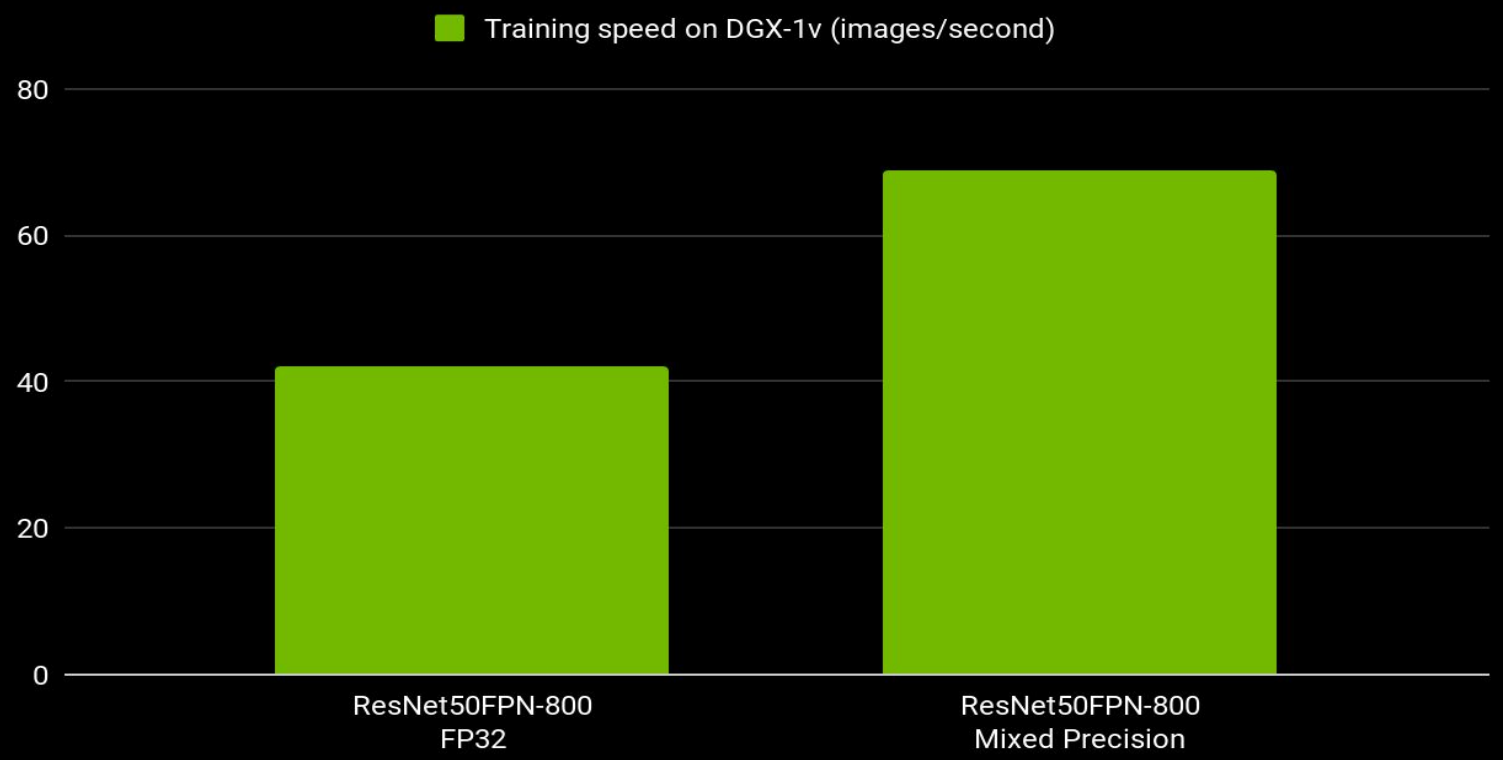
- **Convert** ops to Tensor Core-friendly FP16, keep unsafe ops on FP32
- **Optimizer** wrapper with loss scaling under the hood

```
# Initialize Amp
model, optimizer = amp.initialize(model, optimizer, opt_level='O2')

# Backward pass with scaled loss
with amp.scale_loss(loss, optimizer) as scaled_loss:
     scaled_loss.backward()
```

APEX

Training Throughput (higher is better)



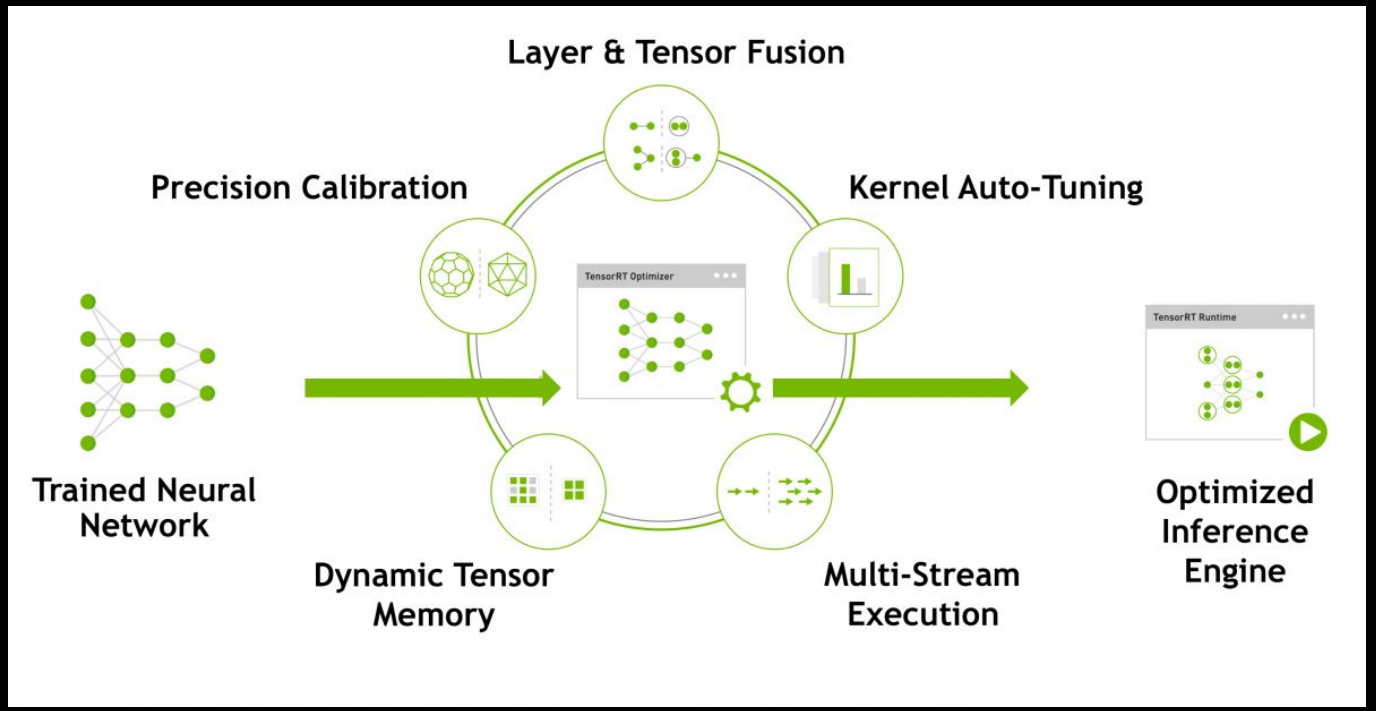
TENSORRT

Platform for high-performance deep learning inference deployment

- **Optimizes** network performance for inference on a target GPU
- **Lower precision** conversion with minimal accuracy loss
- **Production** ready for datacenter, embedded, and automotive applications

TENSORRT

Optimization Workflow



TENSORRT

Workflow

PyTorch -> ONNX -> TensorRT engine

- **Export** PyTorch backbone, FPN, and {cls, bbox} heads to ONNX model
- **Parse** converted **ONNX** file into TensorRT optimizable network
- **Add** custom C++ TensorRT plugins for bbox decode and NMS

TensorRT automatically applies:

- **Graph optimizations** (layer fusion, remove unnecessary layers)
- **Layer by layer kernel autotuning** for target GPU
- **Conversion to reduced precision** if desired (FP16, INT8)

TENSORRT

Inference Model Export

```
// Parse ONNX FCN
auto parser = createParser(*network, gLogger);
parser->parse(onnx_model, onnx_size);
…
// Add decode plugins
for (int i = 0; i < nbBoxOutputs; i++) {
      auto decodePlugin = DecodePlugin(score_thresh, top_n, anchors[i], scale);
      auto layer = network->addPluginV2(inputs.data(), inputs.size(), decodePlugin);
}
…
// Add NMS plugin
auto nmsPlugin = NMSPlugin(nms_thresh, detections_per_im);
auto layer = network->addPluginV2(concat.data(), concat.size(), nmsPlugin);
// Build CUDA inference engine
auto engine = builder->buildCudaEngine(*network);
```

TENSORRT

Plugins

Custom C++ plugins for **bounding box decoding** and **non-maximum suppression**

- Leverage **CUDA** for optimized decoding and NMS

- Enables **full detection workflow** on the GPU
    - No need to copy large feature maps back to host for post-processing

- **Integrated** into TensorRT engine and used transparently during inference

TENSORRT

Plugins

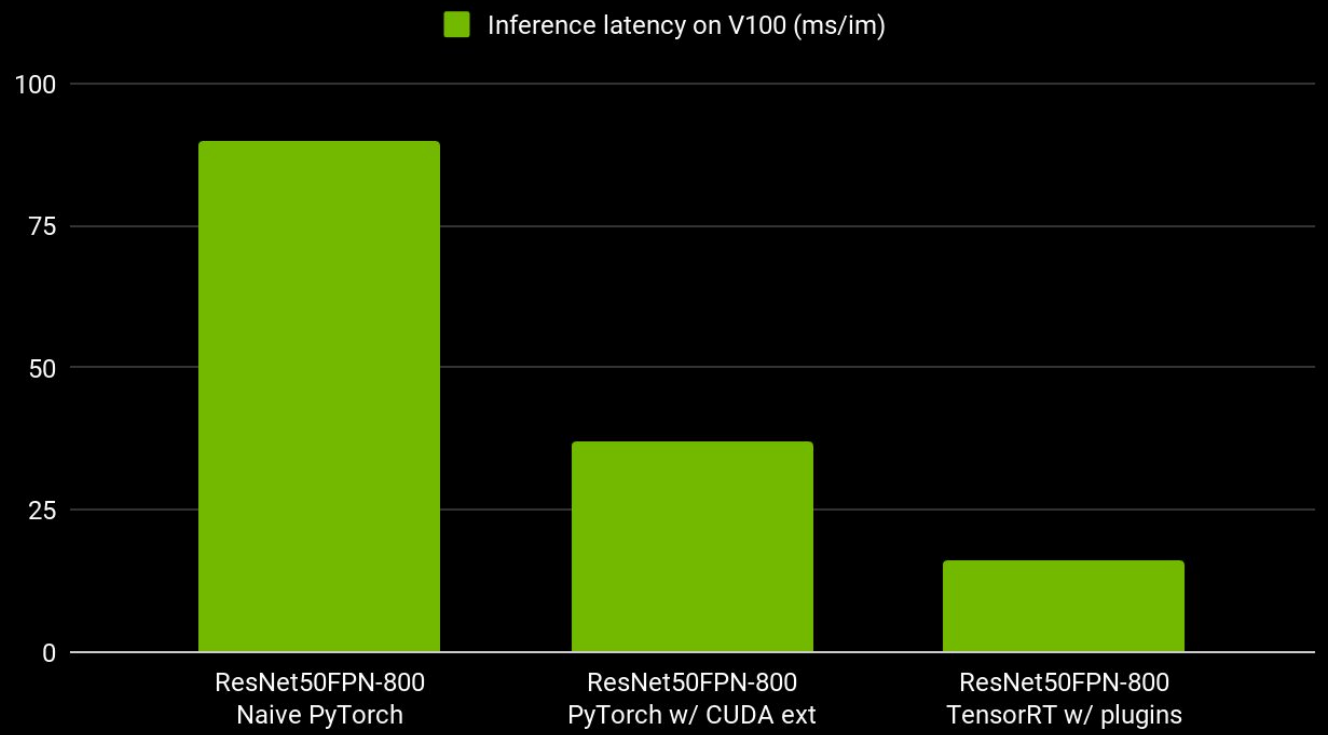
```
class DecodePlugin : public IPluginV2 {
      void configureWithFormat(const Dims* inputDims, …) override;
      int enqueue(int batchSize, const void *const *inputs, …) override;
      void serialize(void *buffer, …) const override;
      …
}

class DecodePluginCreator : public IPluginCreator {
      IPluginV2 *createPlugin (const char *name, …) override;
      IPluginV2 *deserializePlugin (const char *name, …) override;
      …
}

REGISTER_TENSORRT_PLUGIN(DecodePluginCreator);
```

TENSORRT

Inference Latency (lower is better)



FUTURE

- **TRT Inference Server** and **DeepStream** support
- Network **pruning** for faster inference
- New **SoTA backbones**
- **Dynamic depth** for inference
- New **regularization** techniques

WHAT NOW?

Go check out the code and try it!

https://github.com/NVIDIA/retinanet-examples

REFERENCES

[1] **Focal Loss for Dense Object Detection** - *Tsung-Yi Lin, Priya Goyal, Ross Girshick, Kaiming He, Piotr Dollar*

[2] **Feature Pyramid Networks for Object Detection** - *Tsung-Yi Lin, Piotr Dollár, Ross Girshick, Kaiming He, Bharath Hariharan, Serge Belongie*

[3] **You Only Look Once: Unified, Real-Time Object Detection -** *Joseph Redmon, Santosh Divvala, Ross Girshick, Ali Farhadi*

[4] **SSD: Single Shot MultiBox Detector -** *Wei Liu, Dragomir Anguelov, Dumitru Erhan, Christian Szegedy, Scott Reed, Cheng-Yang Fu, Alexander C. Berg*

[5] **Deep Residual Learning for Image Recognition** - *Kaiming He, Xiangyu Zhang, Shaoqing Ren, Jian Sun*

[6] **Learning Transferable Architectures for Scalable Image Recognition** - *arret Zoph, Vijay Vasudevan, Jonathon Shlens, Quoc V. Le*


NVIDIA