



S9925: FAST AI DATA PRE- PROCESSING WITH NVIDIA DALI

Janusz Lisiecki, Michał Zientkiewicz, 2019-03-18



S9925: FAST AI DATA PRE-PROCESSING WITH NVIDIA DALI

Janusz Lisiecki, Michał Zientkiewicz, 2019-03-18

THE PROBLEM




CPU BOTTLENECK OF DL TRAINING

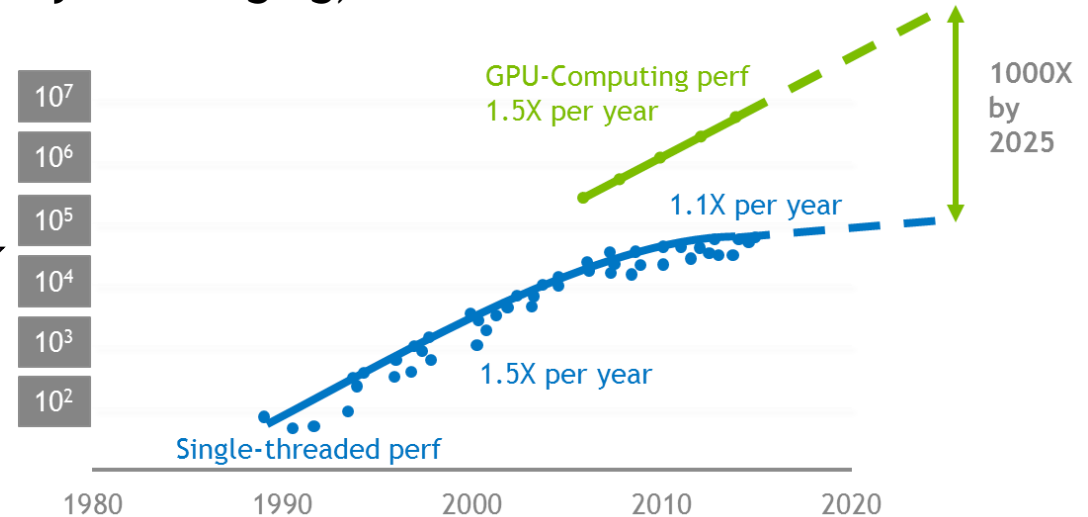
CPU : GPU ratio

Half precision arithmetic, multi-GPU, dense systems are now common (DGX1V, DGX2)

Can't easily scale CPU cores (expensive, technically challenging)

Falling CPU to GPU ratio:

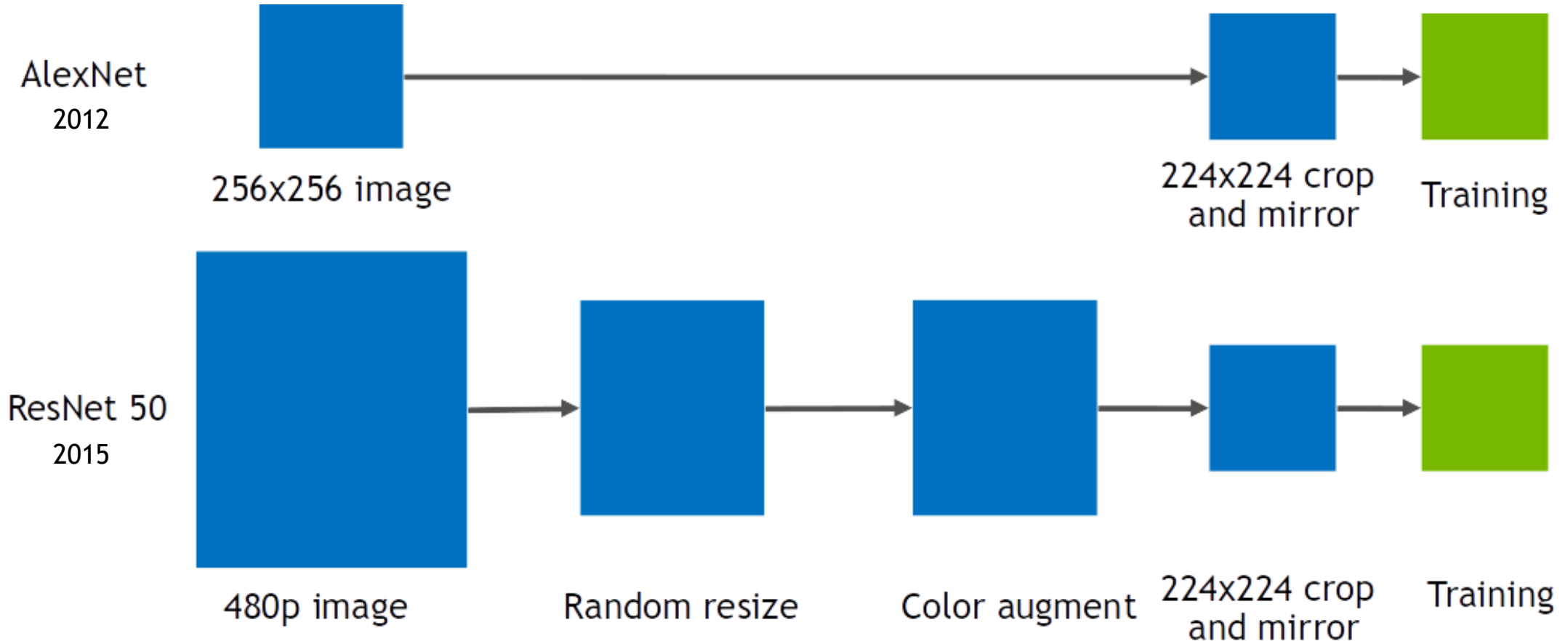
- ▶ DGX1V: 40 cores, 8 GPUs, 5 cores/ GPU
- ▶ DGX2: 48 cores , 16 GPUs , 3 cores/ GPU 



Original data up to the year 2010 collected and plotted by M. Horowitz, F. Labonte, O. Shacham, K. Olukotun, L. Hammond, and C. Batten New plot and data collected for 2010-2015 by K. Rupp

CPU BOTTLENECK OF DL TRAINING

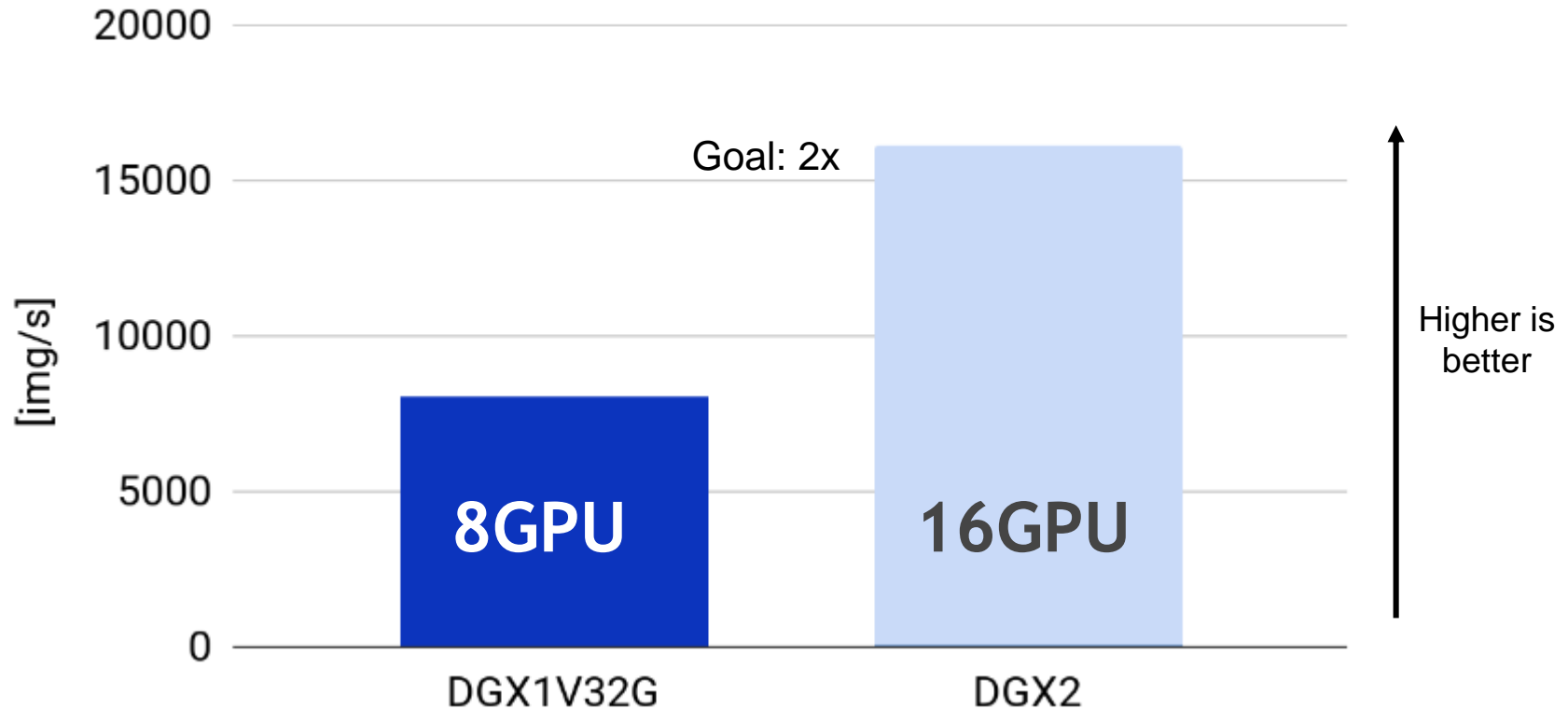
Complexity of I/O pipeline



CPU BOTTLENECK OF DL TRAINING

In practice

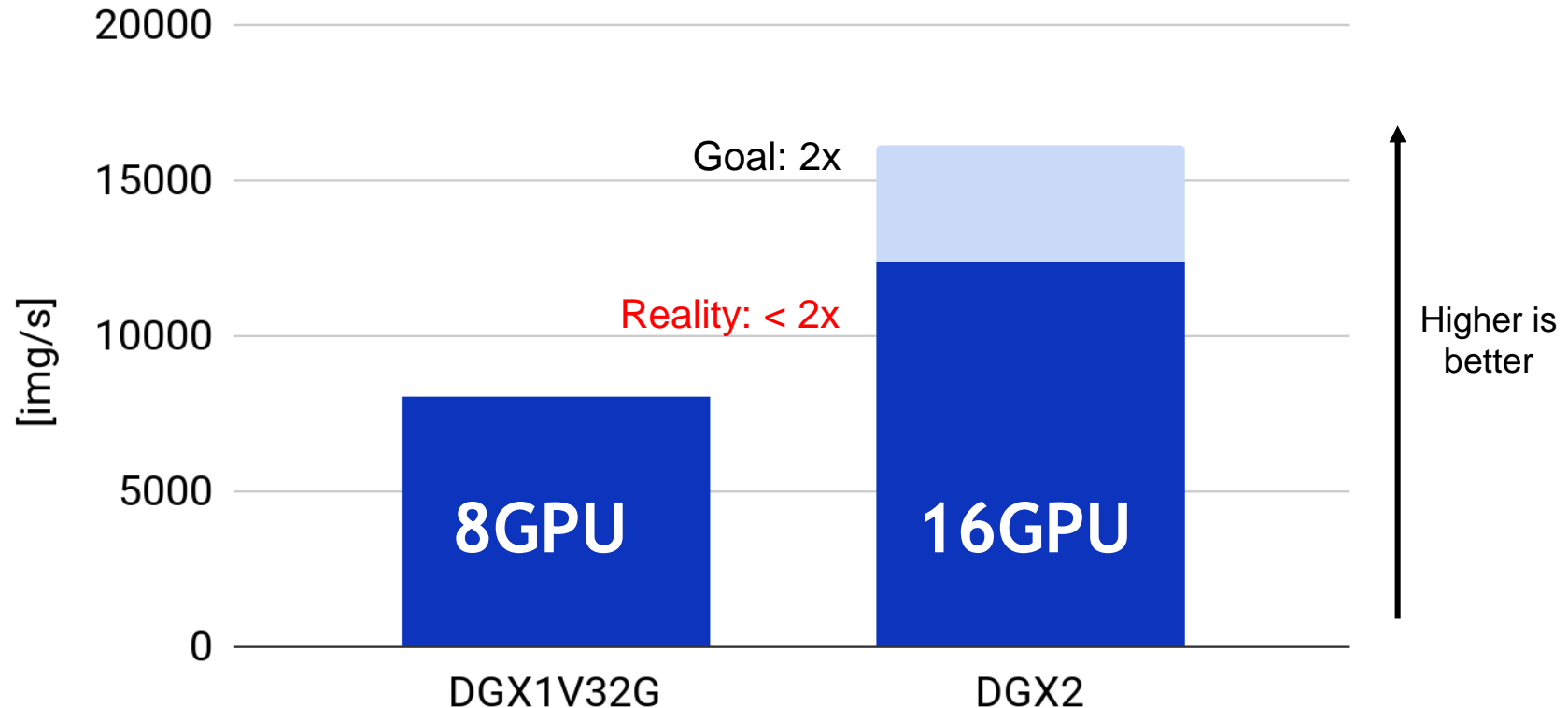
When we put 2x GPU we don't get adequate perf improvement
Training speed, ResNet50, MXNet container 18.09, b=256



CPU BOTTLENECK OF DL TRAINING

In practice

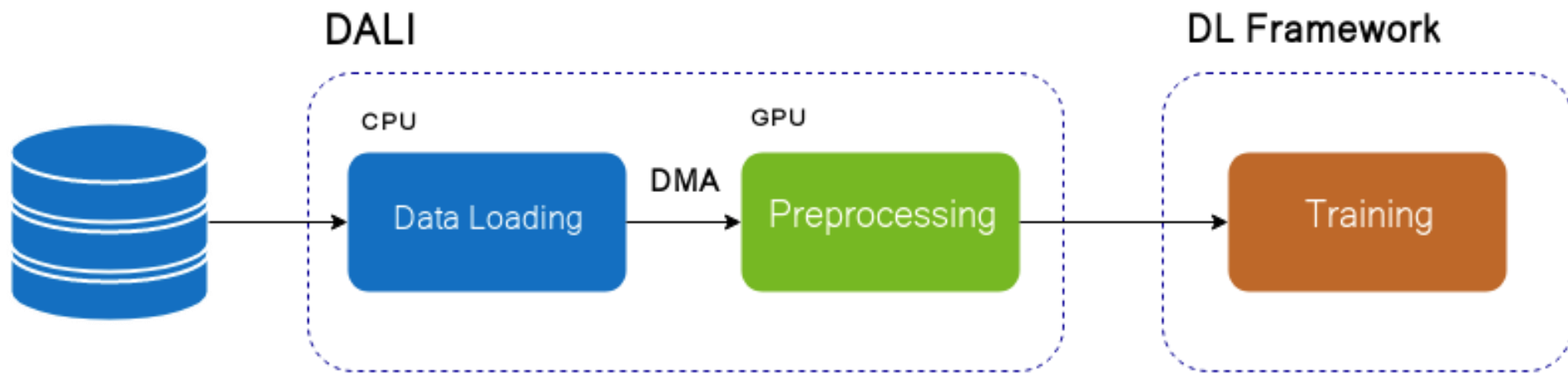
When we put 2x GPU we don't get adequate perf improvement
Training speed, ResNet50, MXNet container 18.09, b=256



DALI TO THE RESCUE

WHAT IS DALI?

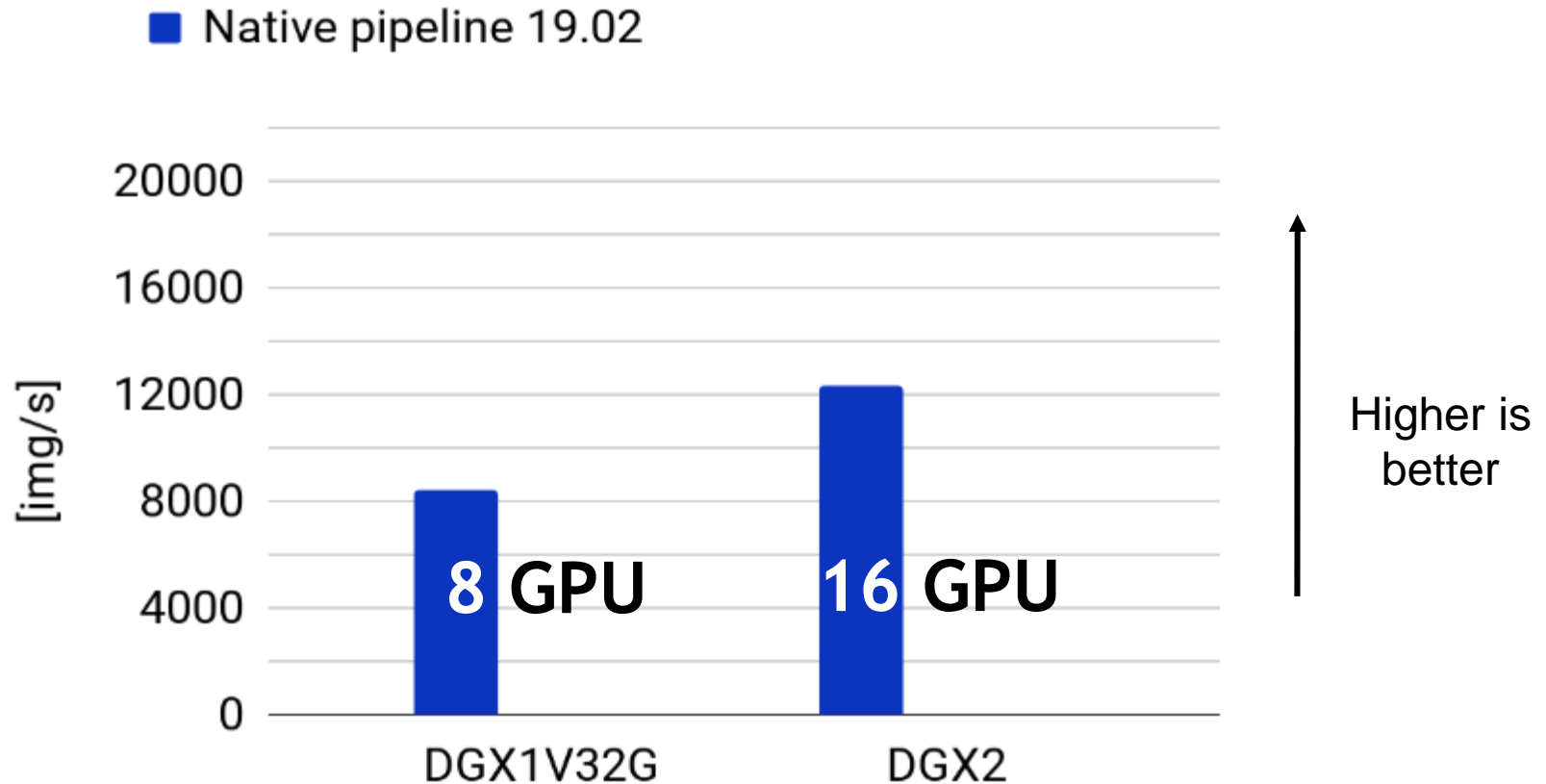
High Performance Data Processing Library



DALI RESULTS

RN50 MXNet

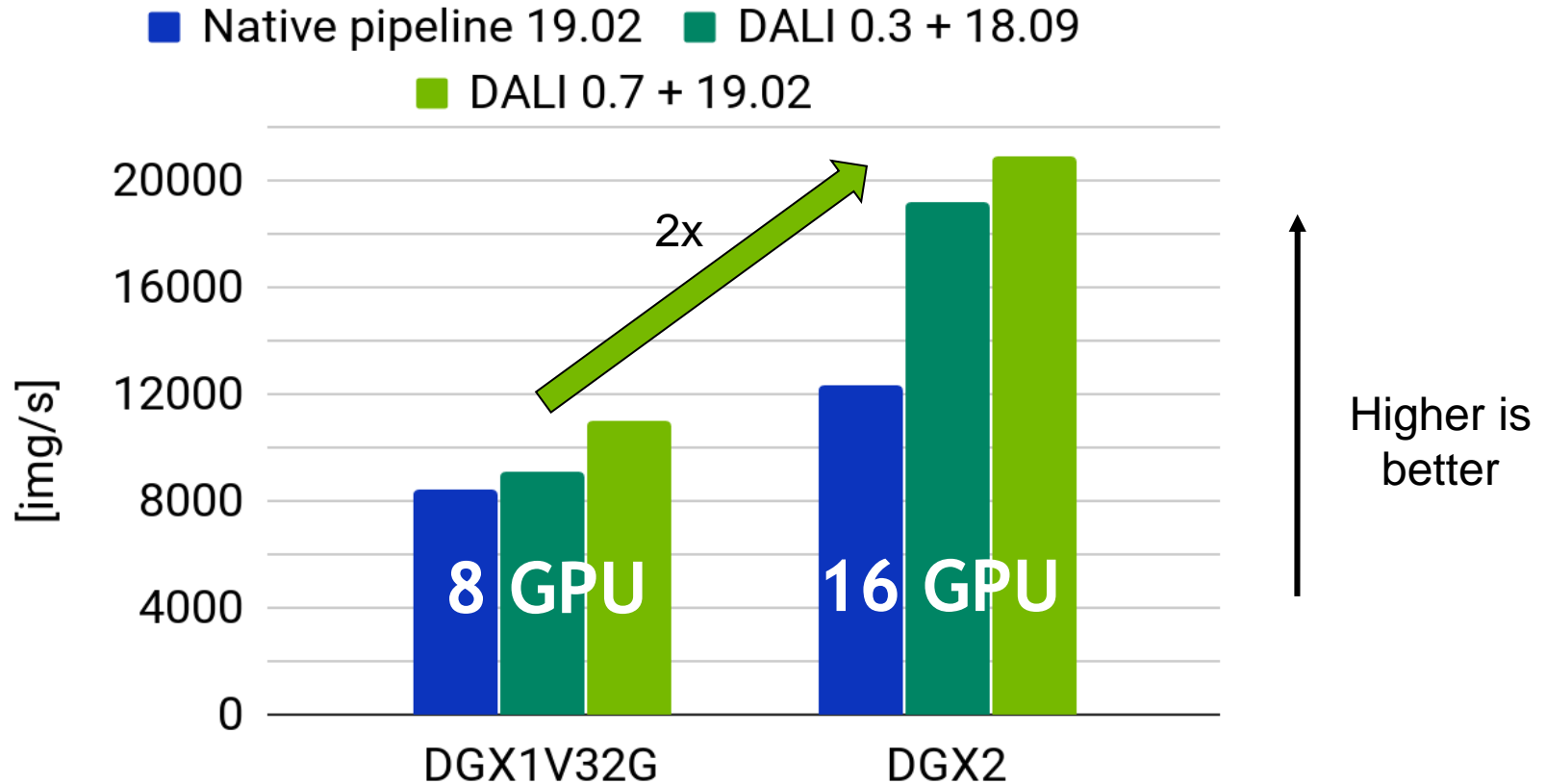
Training speed, ResNet50, MXNet container, b=256



DALI RESULTS

RN50 MXNet

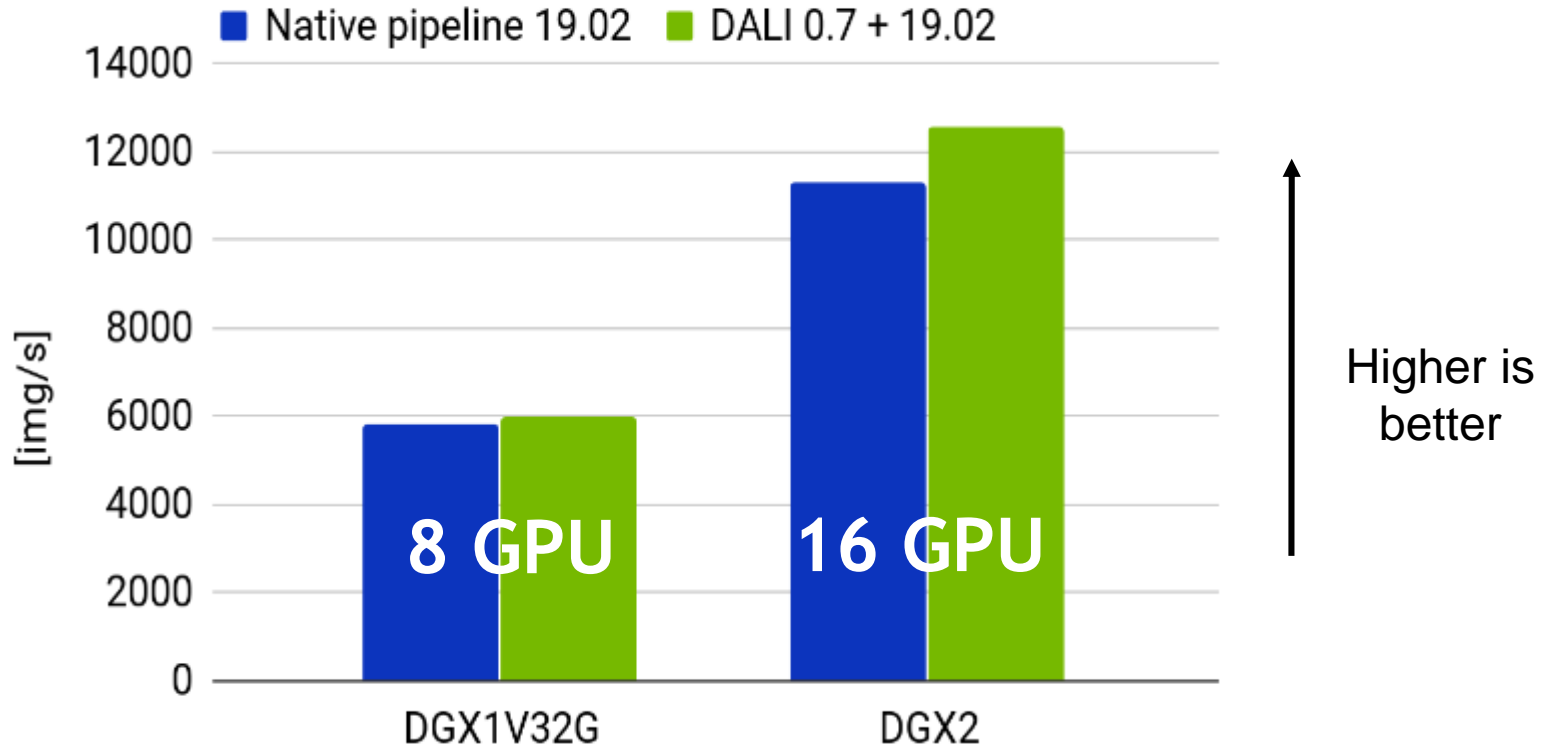
Training speed, ResNet50, MXNet container, b=256



DALI RESULTS

RN50 PyTorch

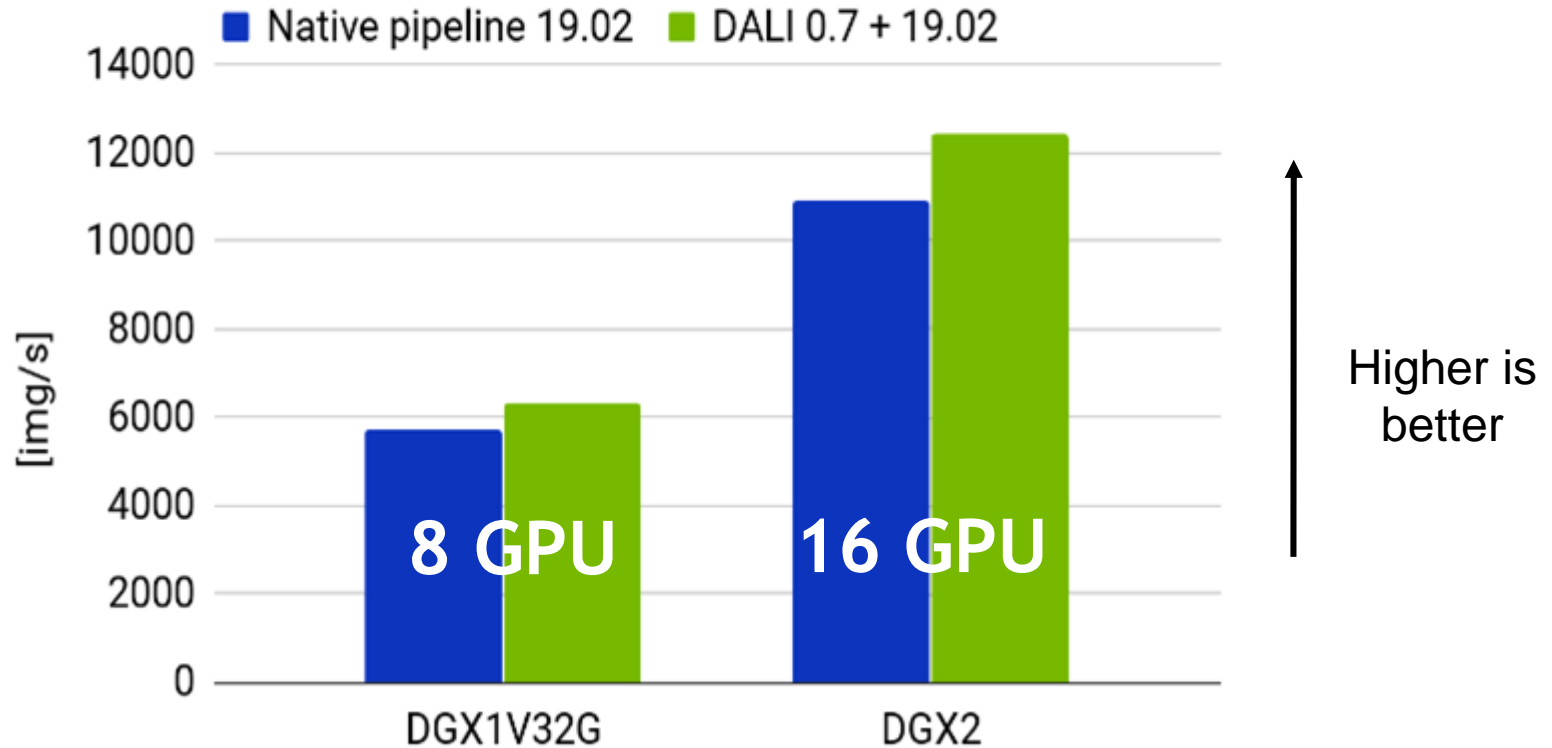
Training speed, ResNet50, PyTorch container, b=256



DALI RESULTS

RN50 TensorFlow

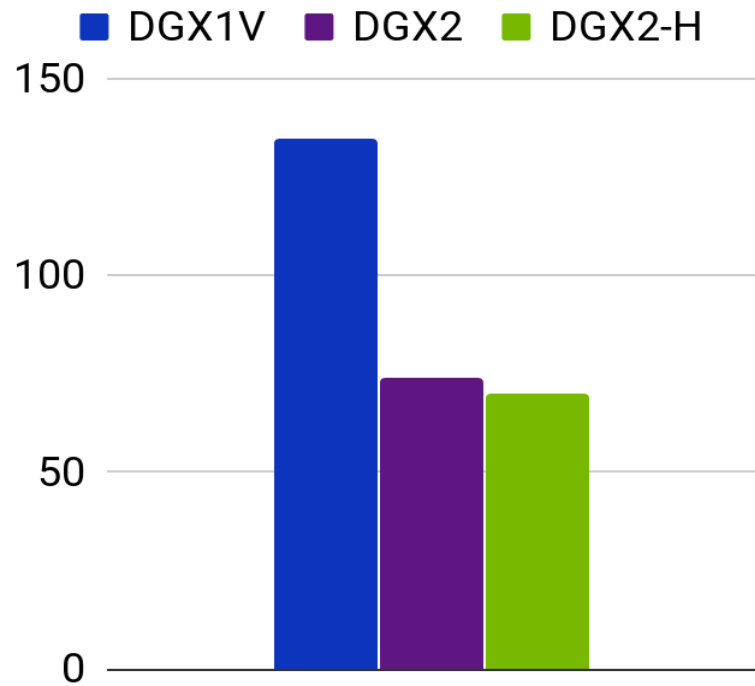
Training speed, ResNet50, TensorFlow container, b=256



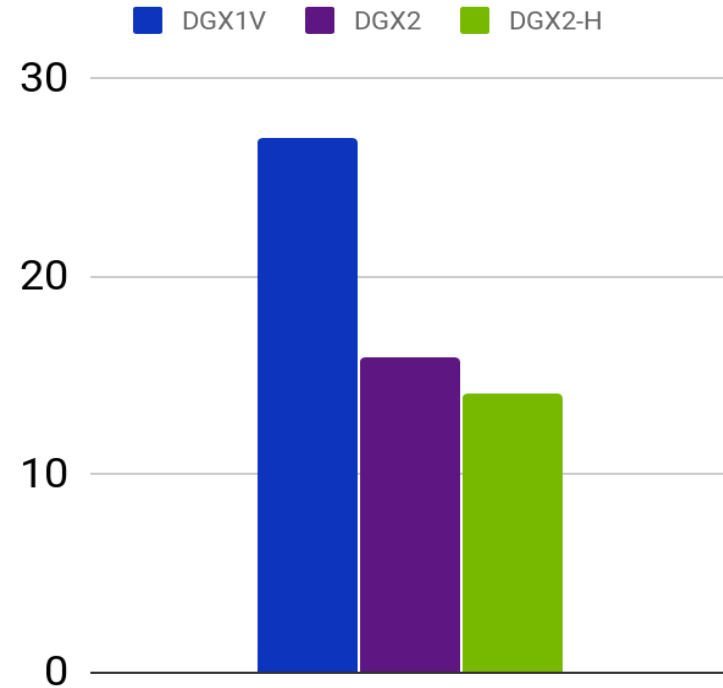
DALI RESULTS - MLPERF

Perfect scaling

Training time - ResNet50 [minutes]



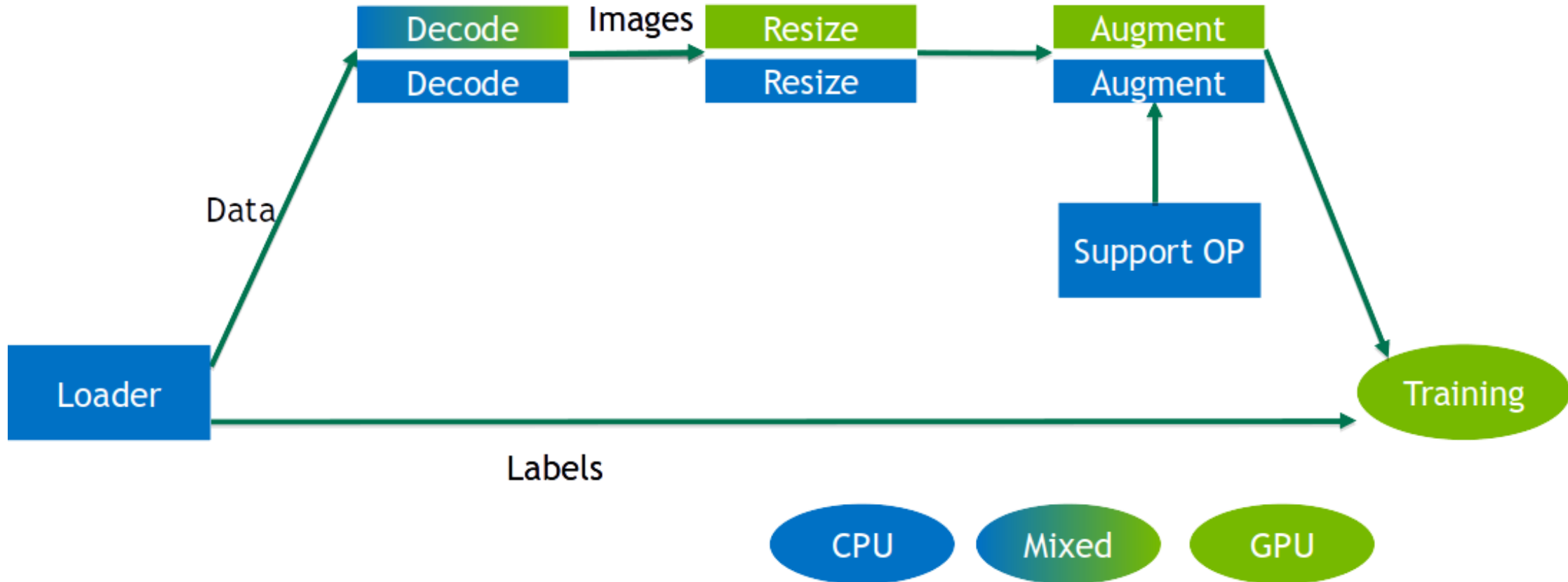
Training time - SSD [minutes]



<https://mlperf.org/results>

INSIDE DALI

DALI: CURRENT ARCHITECTURE



HOW TO USE DALI

Define Graph

Instantiate operators

```
def __init__(self, batch_size, num_threads, device_id):
    super(SimplePipeline, self).__init__(batch_size, num_threads, device_id)
    self.input = ops.FileReader(file_root = image_dir)
    self.decode = ops.nvJPEGDecoder(device = "mixed", output_type = types.RGB)
    self.resize = ops.Resize(device = "gpu", resize_x = 224, resize_y = 224)
```

Load

Decode

Resize

Define graph in imperative way

```
def define_graph(self):
    jpegs, labels = self.input()
    images = self.decode(jpegs)
    images = self.resize(images)
    return (images, labels)
```

Use it

```
pipe.build()
images, labels = pipe.run()
```

HOW TO USE DALI

Define Graph

Instantiate operators

```
def __init__(self, batch_size, num_threads, device_id):  
    super(SimplePipeline, self).__init__(batch_size, num_threads, device_id)  
    self.input = ops.FileReader(file_root = image_dir)  
    self.decode = ops.nvJPEGDecoder(device = "mixed", output_type = types.RGB)  
    self.resize = ops.Resize(device = "gpu", resize_x = 224, resize_y = 224)
```

Define graph in imperative way

```
def define_graph(self):  
    jpegs, labels = self.input()  
    images = self.decode(jpegs)  
    images = self.resize(images)  
    return (images, labels)
```

Use it

```
pipe.build()  
images, labels = pipe.run()
```



HOW TO USE DALI

Define Graph

Instantiate operators

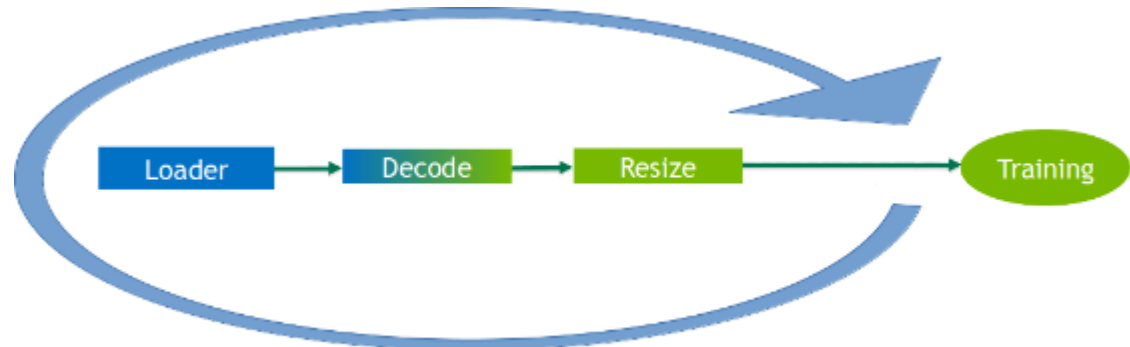
```
def __init__(self, batch_size, num_threads, device_id):
    super(SimplePipeline, self).__init__(batch_size, num_threads, device_id)
    self.input = ops.FileReader(file_root = image_dir)
    self.decode = ops.nvJPEGDecoder(device = "mixed", output_type = types.RGB)
    self.resize = ops.Resize(device = "gpu", resize_x = 224, resize_y = 224)
```

Define graph in imperative way

```
def define_graph(self):
    jpegs, labels = self.input()
    images = self.decode(jpegs)
    images = self.resize(images)
    return (images, labels)
```

Use it

```
pipe.build()
images, labels = pipe.run()
```



HOW TO USE DALI

Define Graph

Instantiate operators

```
def __init__(self, batch_size, num_threads, device_id):  
    super(SimplePipeline, self).__init__(batch_size, num_threads, device_id)  
    self.input = ops.FileReader(file_root = image_dir)  
    self.decode = ops.nvJPEGDecoder(device = "mixed", output_type = types.RGB)  
    self.resize = ops.Resize(device = "gpu", resize_x = 224, resize_y = 224)
```

Define graph in imperative way

```
def define_graph(self):  
    jpegs, labels = self.input()  
    images = self.decode(jpegs)  
    images = self.resize(images)  
    return (images, labels)
```

Use it

```
pipe.build()  
images, labels = pipe.run()
```

HOW TO USE DALI

Use in PyTorch

PyTorch DataLoader

```
train_loader = torch.utils.data.DataLoader(...)
prefetcher = data_prefetcher(train_loader)
input, target = prefetcher.next()
i = -1
while input is not None:
    i += 1
    (...)
    input, target = prefetcher.next()
```

DALI iterator

```
dali_pipe = TrainPipe(...)
train_loader = DALIClassificationIterator(dali_pipe)

for i, data in enumerate(train_loader):
    input = data[0]["data"]
    target = data[0]["label"].squeeze()
    (...)
```

HOW TO USE DALI

Use in MXNet

MXNet Dataloader and DataBatch

```
train_data = SyntheticDataIter(...)

for i, batches in enumerate(train_data):
    data = [b.data[0] for b in batches]
    label = [b.label[0].as_in_context(b.data[0].context)
for b in batches]
    (...)
```

DALI iterator

```
dali_pipes = [TrainPipe(...) for gpu_id in gpus]
train_data = DALIClassificationIterator(dali_pipe)

for i, batches in enumerate(train_data):
    data = [b.data[0] for b in batches]
    label = [b.label[0].as_in_context(b.data[0].context) for
b in batches]
    (...)
```

HOW TO USE DALI

Use in TensorFlow

TensorFlow Dataset

```
def get_data():  
    ds = tf.data.Dataset.from_tensor_slices(files)  
    ds.define_operations(...)  
    return ds
```

```
classifier.train(input_fn=get_data,...)
```

DALI TensorFlow operator

```
def get_data():  
    dali_pipe = TrainPipe(...)  
    daliop = dali_tf.DALIIterator()  
    with tf.device("/gpu:0"):  
        img, labels = daliop(pipeline=dali_pipe, ...)  
    return img, labels
```

```
classifier.train(input_fn=get_data,...)
```


NEW USE CASES

OBJECT DETECTION

Single Shot Multibox Detector Model (SSD)

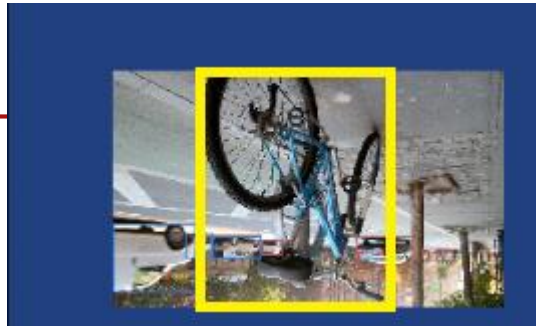
Use operators in the DALI graph:

```
images = self.paste(images, paste_x = px, paste_y = py, ratio = ratio)
bboxes = self.bbpaste(bboxes, paste_x = px, paste_y = py, ratio = ratio)

crop_begin, crop_size, bboxes, labels = self.prospective_crop(bboxes, labels)
images = self.slice(images, crop_begin, crop_size)

images = self.flip(images, horizontal = rng, vertical = rng2)
bboxes = self.bbflip(bboxes, horizontal = rng, vertical = rng2)

return (images, bboxes, labels)
```



VIDEO

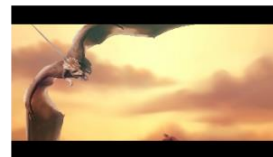
Video Pipeline Example

Instantiate operator:

```
self.input = ops.VideoReader(device="gpu", filenames=data, sequence_length=len)
```

Use it in the DALI graph:

```
frames = self.input(name="Reader")  
output_frames = self.Crop(frames)  
return output_frames
```



VIDEO

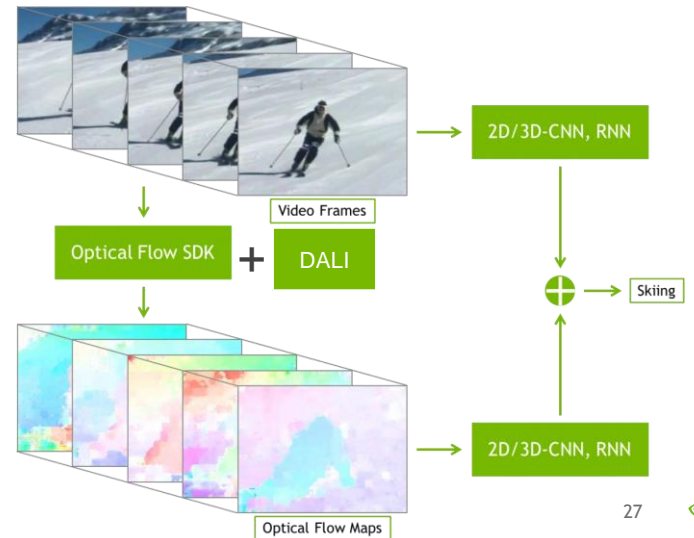
Optical Flow Example

Instantiate operator:

```
self.input = ops.VideoReader(file_root = video_files, sequence_length = len, step = step)
self.opticalFlow = ops.OpticalFlow()
self.takeFirst = ops.ElementExtract(element_map = [0])
```

Use it in the DALI graph:

```
frames = self.input()
flow = self.opticalFlow(frames)
first = self.takeFirst(frames)
return first, flow
```



MAKING LIFE EASIER

MORE EXAMPLES

Help you get started

ResNet50 for PyTorch, MXNet, TensorFlow

How to read data in various frameworks

How to create custom operators

Pipeline for the detection

Video pipeline

More to come...

Documentation available online:

<https://docs.nvidia.com/deeplearning/sdk/dali-developer-guide/docs/index.html>

The screenshot displays the NVIDIA DALI documentation website. The main navigation menu includes: Documentation home, USER GUIDE, Installation, Getting started, and Tutorials. A sidebar menu lists: Framework Integration, MXNet, pyTorch, TensorFlow, ResNet-N with TensorFlow and DALI, Requirements, Using TensorFlow DALI plugin: simple example, Using TensorFlow DALI plugin: using various readers, TensorFlow Plugin API reference, Python API, and Supported operations. The main content area features several article previews:

- MXNet with DALI - ResNet 50 example**: Overview. This example shows, how to use DALI pipelines with Apache MXNet.
- ImageNet training in PyTorch**: This implements training of popular model architectures, such as ResNet, AlexNet, and VGG on the ImageNet dataset. This version has been modified to use DALI. It assumes that the dataset is raw JPEGS from the ImageNet dataset.
- ResNet-N with TensorFlow and DALI**: This demo implements residual networks model and use DALI for the data augmentation pipeline from the original paper. Common utilities for defining the network and performing basic training are located in the nvutils directory. Use of nvutils is demonstrated in the model scripts.

For parallelization, we use the Horovod distribution framework, which works in concert with MPI. To train ResNet-50 (--layers=50) using 8 V100 GPUs, for example on DGX-1, use the following command (--dali_cpu indicates to the script to use CPU backend for DALI):

```
$ mpiexec --allow-run-as-root --bind-to socket -np 8 python resnet.py \
  --layers=50 \
  --data_dir=/data/imagenet \
  --data_loader_dir=/data/imagenet-loader \
  --precision=fp16 \
  --log_dir=/output/resnet50 \
  --dali_cpu
```

Here we have assumed that imagenet is stored in tfrecord format in the directory /data/imagenet. After training completes, evaluation is performed using the validation dataset.

Some common training parameters can be tweaked from the command line. Others must be configured within the network scripts themselves.

Original scripts modified from [nvidia-examples](#) scripts in [NGC TensorFlow Container](#)

Requirements

TensorFlow

```
pip install tensorflow-gpu==1.15.0
```

OpenMPI

```
wget -q -O - https://www.open-mpi.org/software/ompi/v3.0/downloads/ompi-3.0.0.tar.gz | tar -xzf -
cd ompi-3.0.0
```

PLUGIN MANAGER

Adds Extensibility

Create operator

```
template<>
void Dummy<GPUBackend>::RunImpl(DeviceWorkspace *ws, const int idx) {
    (...)
}
DALI_REGISTER_OPERATOR(CustomDummy, Dummy<GPUBackend>, GPU);
```

Load Plugin from python

```
import nvidia.dali.plugin_manager as plugin_manager
plugin_manager.load_library('./customdummy/build/libcustomdummy.so')
ops.CustomDummy(...)
```

DALI



```
graph RL
    p1[plugin1.so] --> DALI[DALI]
    p2[plugin2.so] --> DALI
    p3[plugin3.so] --> DALI
```

plugin1.so

plugin2.so

plugin3.so

CHALLENGES

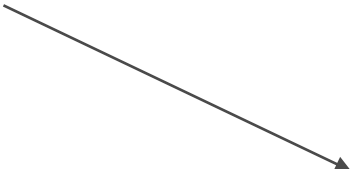
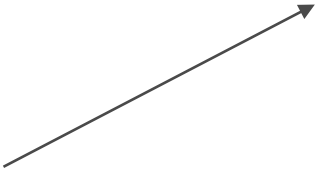
CHALLENGES

Object Detection

Data-dependent random transformation



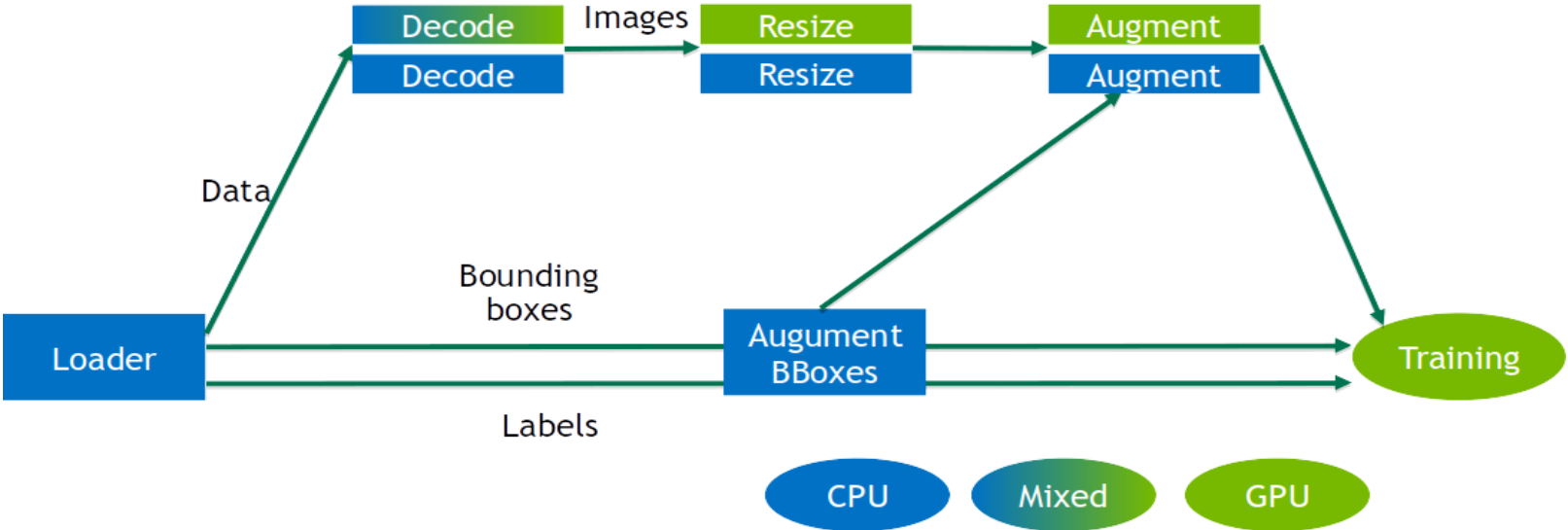
Random
crop



CHALLENGES

Object Detection

More types of data, not only images and labels - bounding boxes as well
Previously only images were processed
Now processing of bounding boxes drives image processing

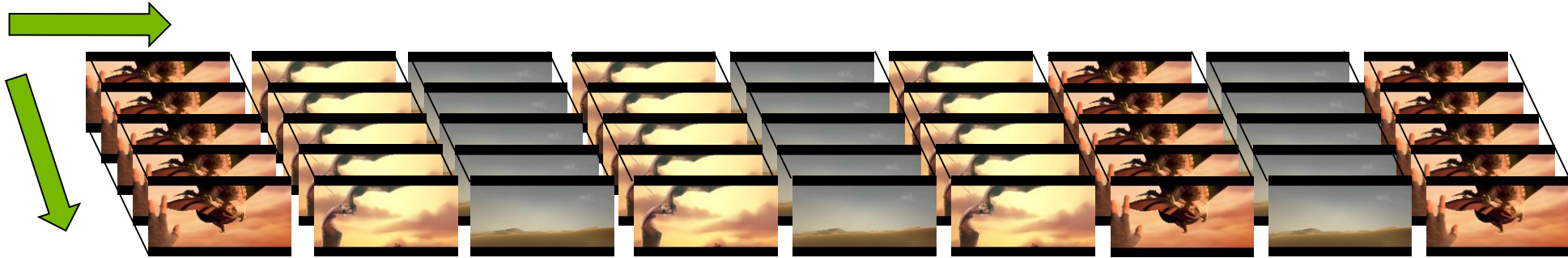


CHALLENGES

Video

Integrated NVDEC to utilize H.264 and HEVC

Samples are no longer single image - sequence (NFHWC \leftrightarrow NCFHW)



Reuse operators - flatten the sequence



CHALLENGES

CPU based pipeline

CPU/GPU high or network traffic consumes GPU cycles

- CPU operators coverage

Sweet spot for SSD mixed pipeline - part CPU, part GPU

- Test what works best for you

Operator name	CPU	GPU
BlurFast	✓	
Blur	✓	✓
BlendMode	✓	✓
Brightness	✓	✓
ColorBayer		
ColorBayer2	✓	
ColorBayer3	✓	
ColorBayer4	✓	
ColorBayer5	✓	✓
ColorBayer6	✓	✓
ColorBayer7	✓	✓
ColorBayer8	✓	✓
ColorBayer9	✓	✓
ColorBayer10	✓	✓
ColorBayer11	✓	✓
ColorBayer12	✓	✓
ColorBayer13	✓	✓
ColorBayer14	✓	✓
ColorBayer15	✓	✓
ColorBayer16	✓	✓
ColorBayer17	✓	✓
ColorBayer18	✓	✓
ColorBayer19	✓	✓
ColorBayer20	✓	✓
ColorBayer21	✓	✓
ColorBayer22	✓	✓
ColorBayer23	✓	✓
ColorBayer24	✓	✓
ColorBayer25	✓	✓
ColorBayer26	✓	✓
ColorBayer27	✓	✓
ColorBayer28	✓	✓
ColorBayer29	✓	✓
ColorBayer30	✓	✓
ColorBayer31	✓	✓
ColorBayer32	✓	✓
ColorBayer33	✓	✓
ColorBayer34	✓	✓
ColorBayer35	✓	✓
ColorBayer36	✓	✓
ColorBayer37	✓	✓
ColorBayer38	✓	✓
ColorBayer39	✓	✓
ColorBayer40	✓	✓
ColorBayer41	✓	✓
ColorBayer42	✓	✓
ColorBayer43	✓	✓
ColorBayer44	✓	✓
ColorBayer45	✓	✓
ColorBayer46	✓	✓
ColorBayer47	✓	✓
ColorBayer48	✓	✓
ColorBayer49	✓	✓
ColorBayer50	✓	✓
ColorBayer51	✓	✓
ColorBayer52	✓	✓
ColorBayer53	✓	✓
ColorBayer54	✓	✓
ColorBayer55	✓	✓
ColorBayer56	✓	✓
ColorBayer57	✓	✓
ColorBayer58	✓	✓
ColorBayer59	✓	✓
ColorBayer60	✓	✓
ColorBayer61	✓	✓
ColorBayer62	✓	✓
ColorBayer63	✓	✓
ColorBayer64	✓	✓
ColorBayer65	✓	✓
ColorBayer66	✓	✓
ColorBayer67	✓	✓
ColorBayer68	✓	✓
ColorBayer69	✓	✓
ColorBayer70	✓	✓
ColorBayer71	✓	✓
ColorBayer72	✓	✓
ColorBayer73	✓	✓
ColorBayer74	✓	✓
ColorBayer75	✓	✓
ColorBayer76	✓	✓
ColorBayer77	✓	✓
ColorBayer78	✓	✓
ColorBayer79	✓	✓
ColorBayer80	✓	✓
ColorBayer81	✓	✓
ColorBayer82	✓	✓
ColorBayer83	✓	✓
ColorBayer84	✓	✓
ColorBayer85	✓	✓
ColorBayer86	✓	✓
ColorBayer87	✓	✓
ColorBayer88	✓	✓
ColorBayer89	✓	✓
ColorBayer90	✓	✓
ColorBayer91	✓	✓
ColorBayer92	✓	✓
ColorBayer93	✓	✓
ColorBayer94	✓	✓
ColorBayer95	✓	✓
ColorBayer96	✓	✓
ColorBayer97	✓	✓
ColorBayer98	✓	✓
ColorBayer99	✓	✓
ColorBayer100	✓	✓

Operator name	CPU	GPU
ColorSpaceConversion	✓	✓
ColorTwist	✓	✓
Contrast	✓	✓
Copy	✓	✓
Crop	✓	✓

Operator name	CPU	GPU
ImageResize	✓	
ImageResizeFast	✓	
Blur	✓	✓
BlurFast	✓	✓
BlurFast2	✓	✓
BlurFast3	✓	✓
BlurFast4	✓	✓
BlurFast5	✓	✓
BlurFast6	✓	✓
BlurFast7	✓	✓
BlurFast8	✓	✓
BlurFast9	✓	✓
BlurFast10	✓	✓
BlurFast11	✓	✓
BlurFast12	✓	✓
BlurFast13	✓	✓
BlurFast14	✓	✓
BlurFast15	✓	✓
BlurFast16	✓	✓
BlurFast17	✓	✓
BlurFast18	✓	✓
BlurFast19	✓	✓
BlurFast20	✓	✓
BlurFast21	✓	✓
BlurFast22	✓	✓
BlurFast23	✓	✓
BlurFast24	✓	✓
BlurFast25	✓	✓
BlurFast26	✓	✓
BlurFast27	✓	✓
BlurFast28	✓	✓
BlurFast29	✓	✓
BlurFast30	✓	✓
BlurFast31	✓	✓
BlurFast32	✓	✓
BlurFast33	✓	✓
BlurFast34	✓	✓
BlurFast35	✓	✓
BlurFast36	✓	✓
BlurFast37	✓	✓
BlurFast38	✓	✓
BlurFast39	✓	✓
BlurFast40	✓	✓
BlurFast41	✓	✓
BlurFast42	✓	✓
BlurFast43	✓	✓
BlurFast44	✓	✓
BlurFast45	✓	✓
BlurFast46	✓	✓
BlurFast47	✓	✓
BlurFast48	✓	✓
BlurFast49	✓	✓
BlurFast50	✓	✓
BlurFast51	✓	✓
BlurFast52	✓	✓
BlurFast53	✓	✓
BlurFast54	✓	✓
BlurFast55	✓	✓
BlurFast56	✓	✓
BlurFast57	✓	✓
BlurFast58	✓	✓
BlurFast59	✓	✓
BlurFast60	✓	✓
BlurFast61	✓	✓
BlurFast62	✓	✓
BlurFast63	✓	✓
BlurFast64	✓	✓
BlurFast65	✓	✓
BlurFast66	✓	✓
BlurFast67	✓	✓
BlurFast68	✓	✓
BlurFast69	✓	✓
BlurFast70	✓	✓
BlurFast71	✓	✓
BlurFast72	✓	✓
BlurFast73	✓	✓
BlurFast74	✓	✓
BlurFast75	✓	✓
BlurFast76	✓	✓
BlurFast77	✓	✓
BlurFast78	✓	✓
BlurFast79	✓	✓
BlurFast80	✓	✓
BlurFast81	✓	✓
BlurFast82	✓	✓
BlurFast83	✓	✓
BlurFast84	✓	✓
BlurFast85	✓	✓
BlurFast86	✓	✓
BlurFast87	✓	✓
BlurFast88	✓	✓
BlurFast89	✓	✓
BlurFast90	✓	✓
BlurFast91	✓	✓
BlurFast92	✓	✓
BlurFast93	✓	✓
BlurFast94	✓	✓
BlurFast95	✓	✓
BlurFast96	✓	✓
BlurFast97	✓	✓
BlurFast98	✓	✓
BlurFast99	✓	✓
BlurFast100	✓	✓

CHALLENGES

Memory Consumption

DGX - “works for me”

A lot of non-DGX users started using DALI

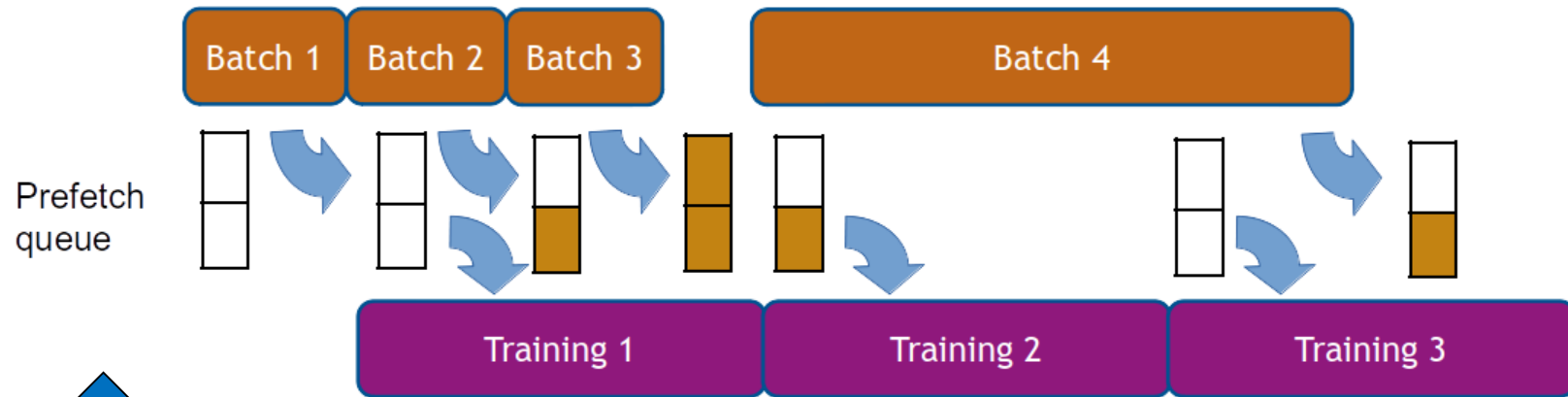
- Want to use CPU operators
- Memory consumption on the CPU side matters
- Usability more important than speed



CHALLENGES

Memory Consumption

Multiple buffering



...but memory consumption ↑

- Caching allocators?
- Subbatches?

CHALLENGES

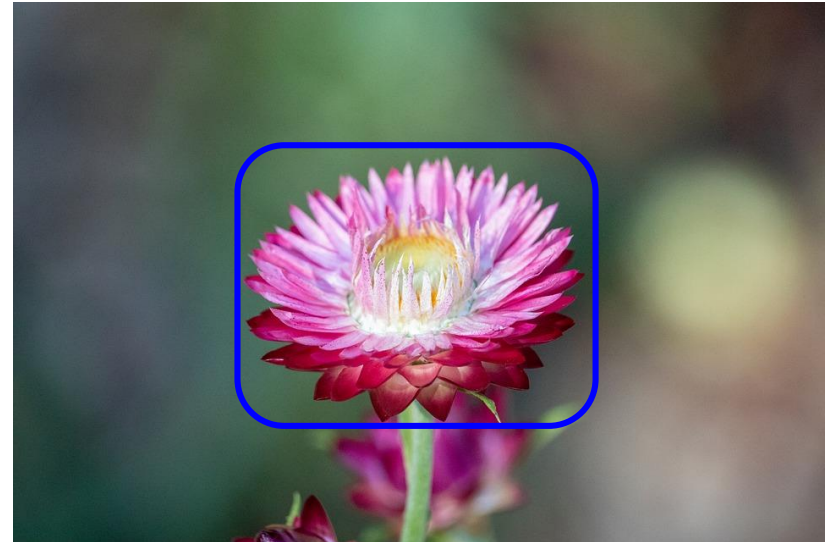
Decoding Time

Significant image decoding time

- CPU decoding already pushed to the limits

Can we do better?

- nvJPEG - huge improvement
- ROI decoding



CHALLENGES

TensorFlow Forward Compatibility

PyTorch and MXNet integration

- Python API - “easy-peasy”

TensorFlow - custom operator needed

- Frequent changes to TensorFlow C++ API
- Cannot preserve forward compatibility at the binary level
- DALI TF plug-in package is now available - compile your TensorFlow DALI op



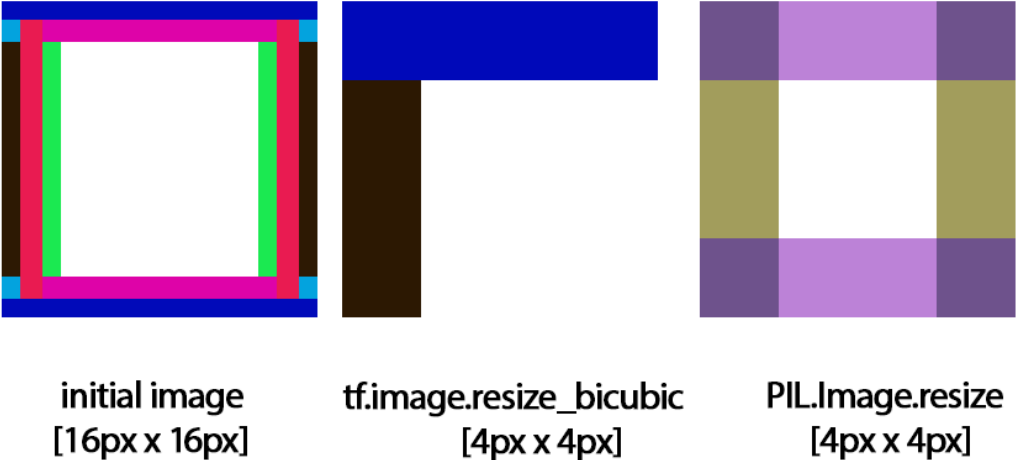
CHALLENGES

Discrepancies Between Frameworks

Bilinear filter - OpenCV vs Pillow



Bicubic filter - TensorFlow vs Pillow



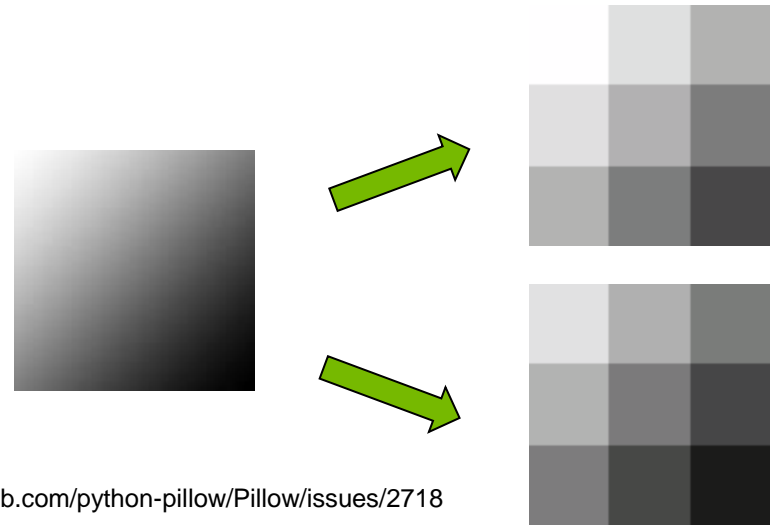
<https://hackernoon.com/how-tensorflows-tf-image-resize-stole-60-days-of-my-life-aba5eb093f35>

CHALLENGES

Discrepancies Between Frameworks

- MXNet is based on OpenCV
- PyTorch uses Pillow
- TensorFlow has its own augmentation operators

We want portability between frameworks,
but what about pre-trained models?



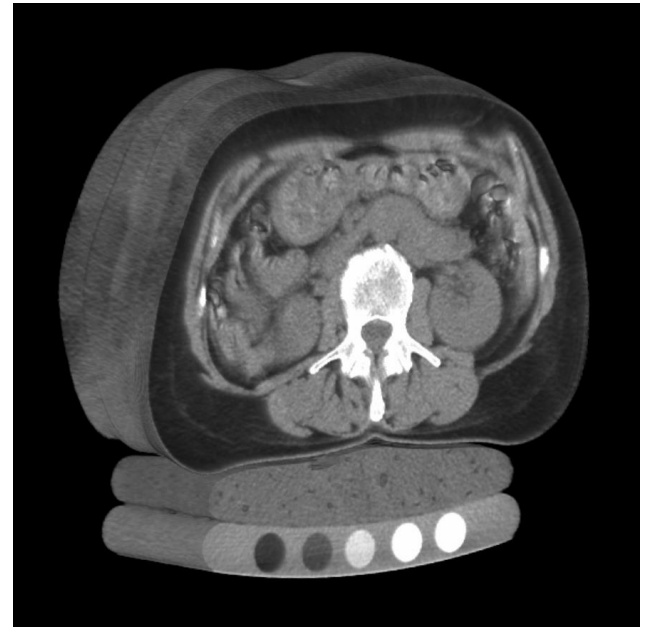
NEXT STEPS

NEW USE CASES

Medical imaging (Volumetric data)

- Performant 3D augmentations library

Segmentation?



NEW USE CASES

Extract augmentation operators in a separate library

- Inference - the same augmentation operation can be used in custom inference pipeline where full feature DALI is not required (i.e. embedded platform)
- Ability to use operator directly from Python code

```
import nvidia.dali.standaloneOps as standaloneOps
import cv2
```

```
image = cv2.imread('test.jpg',0)
standaloneOps.Rotate(image, device="gpu", angle=45, interp_type = types.INTERP_LINEAR)
cv2.imwrite("./img_tf.png", image)
```



DALI

Summary

- ▶ Open source, GPU-accelerated data augmentation and image loading library
 - ▶ Over 1100 GitHub stars
 - ▶ ¹⁾ Top 50 ML/DL Projects (out of 22,000 in 2018)
- ▶ Full pre-processing data pipeline ready for training and inference
- ▶ Easy framework integration
- ▶ Portable training workflows

¹⁾ <https://github.com/Mybridge/amazing-machine-learning-opensource-2019>



DALI

Summary

More questions? **Connect with Experts Sessions: DALI** Tue 19th, Wed 20th, 2pm (Expo Hall)

Meet us **P9291 - Fast Data Pre-processing with DALI** (Mon 18th, 6-8pm)

Attend **S9818 - TensorRT with DALI on Xavier** to learn about TensorRT inference workflow with DALI graphs and customer operators

We want to hear from you

Dali-Team@nvidia.com

<https://github.com/NVIDIA/DALI>

<https://developer.nvidia.com/dali>

ACKNOWLEDGEMENTS

Joaquin Anton
Trevor Gale
Andrei Ivanov
Simon Layton
Krzysztof Łęcki
Serge Panev
Michał Szotucha
Przemek Trędak
Albert Wolant
Pablo Ribalta
Cliff Woolley
DL Frameworks @ NVIDIA

