ACCELERATING ML DEVELOPMENT WITH PYTORCH

SOUMITH CHINTALA
FACEBOOK AI

PYTORCH OVERVIEW

NVIDIA SUPPORT & COLLABORATION

SOFTWARE COLLABORATION

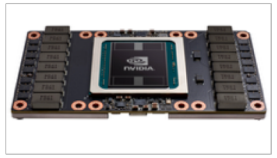
PyTorch

ONNX

NVIDIA APEX

HARDWARE SUPPORT

NVIDIA JETSON TX1



CORE LIBRARY INTEGRATION

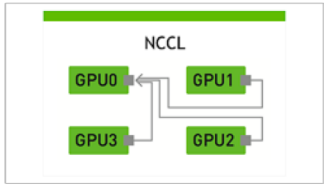
cuDNN

DALI nvJPEG

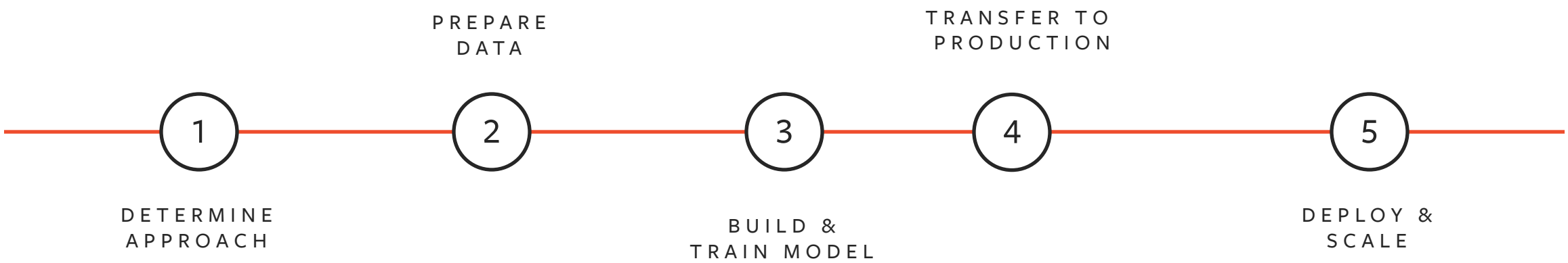
cuBLAS

SCALABILITY & DEPLOYMENT

TensorRT



GOING FROM RESEARCH TO PRODUCTION


1
DETERMINE APPROACH
2
PREPARE DATA
3
BUILD & TRAIN MODEL
4
TRANSFER TO PRODUCTION
5
DEPLOY & SCALE

`torch.jit`

Code == Model

Code == Model == Data

1.0.0

torch.jit

Code == Model

```
def myfun(x):
    y = x * x
    z = y.tanh()
    return z
```

Code == Model == Data

```
@torch.jit.script
def myfun(x):
    y = x * x
    z = y.tanh()
    return z
```

torch.jit

Code == Model

```
def myfun(x):
    y = x * x
    z = y.tanh()
    return z
```

Code == Model == Data

```
graph(%x : Dynamic) {
 %y : Dynamic = aten::mul(%x,
%x)
%z : Dynamic = aten::tanh(%y)
return (%z)
}
```

torch.jit

Export and run anywhere

Code == Model

```
def myfun(x):
    y = x * x
    z = y.tanh()
    return z
```

Code == Model == Data

```
graph(%x : Dynamic) {
 %y : Dynamic = aten::mul(%x,
%x)
%z : Dynamic = aten::tanh(%y)
return (%z)
}
```

torch.jit

Eager execution

```
def myfun(x):
    y = x * x
    z = y.tanh()
    return z
```

Execution with ahead-of-time analysis

```
@torch.jit.script
def myfun(x):
    y = x * x
    z = y.tanh()
    return z
```

torch.jit

Eager execution

```
def myfun(x):
    y = x * x
    z = y.tanh()
    return z
```

Execution with ahead-of-time analysis

```
@torch.jit.script
def myfun(x):
    y = x * x
    z = y.tanh()
    return z
```

torch.jit

Eager execution

```
def myfun(x):
    y = x * x
    z = y.tanh()
    return z
```

Execution with ahead-of-time analysis

```
@torch.jit.script
def myfun(x):
    y = x * x
    z = y.tanh()
    return z
```

torch.jit

Eager execution

```
def myfun(x):
    y = x * x
    z = y.tanh()
    return z
```

Execution with ahead-of-time analysis

```
@torch.jit.script
def myfun(x):
    y = x * x
    z = y.tanh()
    return z
```

torch.jit

Eager execution

```
def myfun(x):
    y = x * x
    z = y.tanh()
    return z
```

Execution with ahead-of-time analysis

```
@torch.jit.script
def myfun(x):
    y = x * x
    z = y.tanh()
    return z
```

torch.jit

Eager execution

```
def myfun(x):
    y = x * x
    z = y.tanh()
    return z
```

Execution with ahead-of-time analysis

```
@torch.jit.script
def myfun(x):
    return tanh_mul(x)
```

torch.jit

Eager execution

```
def myfun(x):
    y = x * x
    z = y.tanh()
    return z
```

Execution with ahead-of-time analysis

```
@torch.jit.script
def myfun(x):
    return tanh_mul(x)
```

saturate faster hardware

whole program optimizations

PyTorch

Models are Python programs

- Simple
- Debuggable — `print` and `pdb`
- Hackable — use any Python library

- Needs Python to run
- Difficult to optimize and parallelize

PyTorch *Eager Mode*

Models are Python programs

- Simple
- Debuggable — `print` and `pdb`
- Hackable — use any Python library

- Needs Python to run
- Difficult to optimize and parallelize

PyTorch *Eager Mode*

Models are Python programs

- Simple
- Debuggable — `print` and `pdb`
- Hackable — use any Python library

- Needs Python to run
- Difficult to optimize and parallelize

PyTorch *Script Mode*

Models are programs written in an optimizable subset of Python

- Production deployment
- No Python dependency
- Optimizable

PYTORCH JIT

Tools to transition eager code into script mode

EAGER MODE

For prototyping, training, and experiments

`@torch.jit.script`

`torch.jit.trace`

SCRIPT MODE

For use at scale in production

Transitioning a model with `torch.jit.trace`

Take an existing eager model, and provide example inputs.

The tracer runs the function, recording the tensor operations performed.

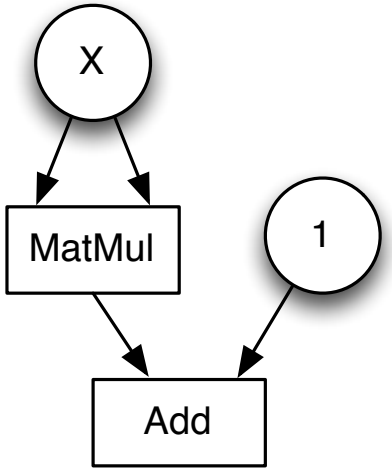
We turn the recording into a Torch Script module.

- Can reuse existing eager model code
- ⚠ Control-flow is ignored

```
import torch
import torchvision


def foo(x, y):
  return 2*x + y


# trace a model by providing example inputs
traced_foo = torch.jit.trace(foo,
                             (torch.rand(3), torch.rand(3)))


traced_resnet = torch.jit.trace(torchvision.models.resnet18(),
                                torch.rand(1, 3, 224, 224))
```

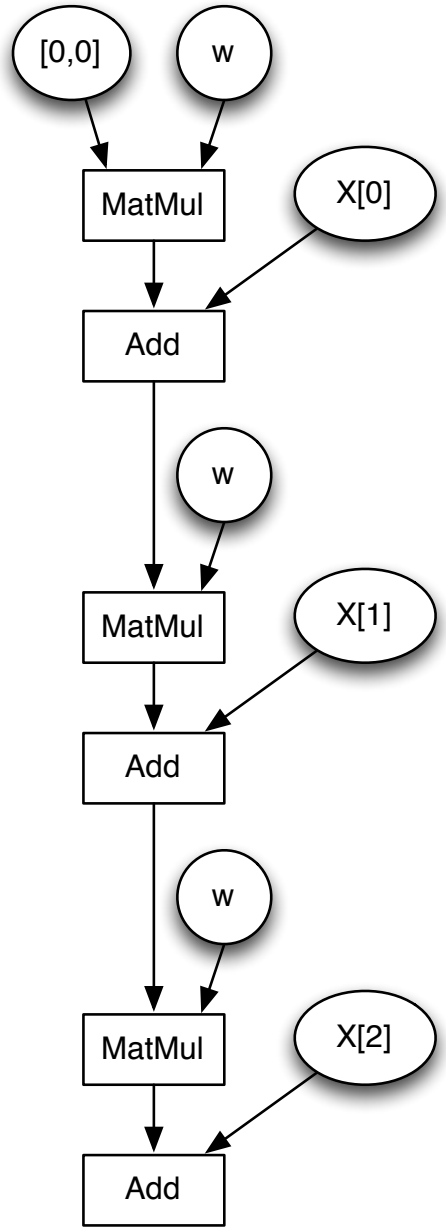
Tracing

```
def foo(x, t):
  y = x.mm(x)
  print(y) # still works!
  return y + t

x = torch.Tensor([[1,2],[3,4]])
foo(x, 1)

trace = torch.jit.trace(foo, (x, 1))
trace.save("serialized.pt")
```



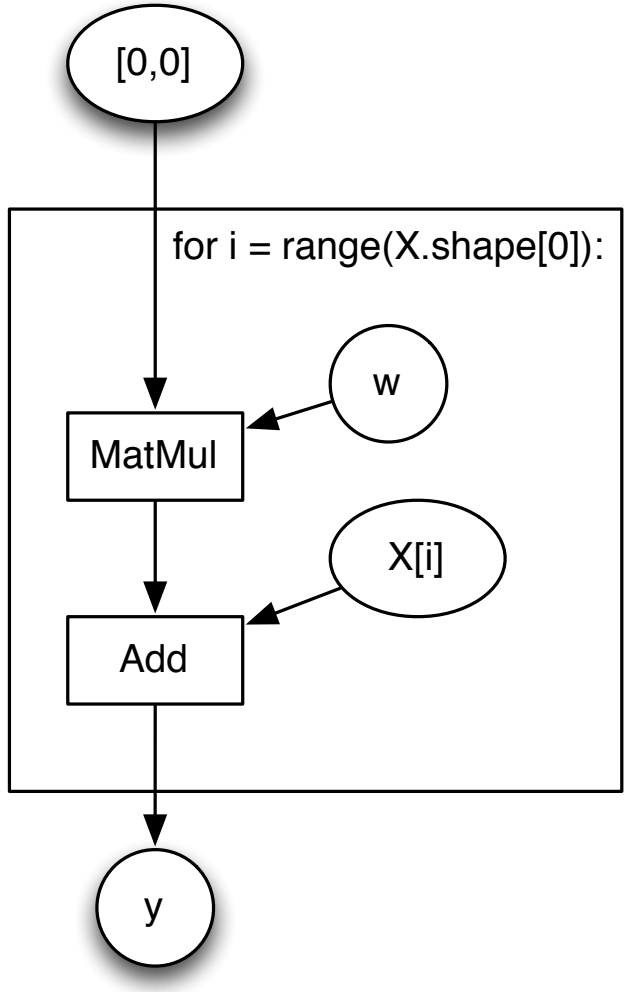
Tracing

```
def foo(x, t):
  y = x.mm(x)
  print(y) # still works!
  return y + t

def bar(x, w):
    y = torch.zeros(1, 2)
    for t in x:
        y = foo(y, w, t)
    return y

trace = torch.jit.trace(foo, (x, 1))
trace.save("serialized.pt")
```



Script

```
def foo(x, t):
  y = x.mm(x)
  print(y) # still works!
  return y + t

@script
def bar(x, w):
    y = torch.zeros(1, 2)
    for t in x:
        y = foo(y, w, t)
    return y

trace = torch.jit.trace(foo, (x, 1))
trace.save("serialized.pt")
```



Transitioning a model with `@torch.jit.script`

Write model directly in a subset of Python, annotated with `@torch.jit.script` or `@torch.jit.scri`

- Control-flow is pre
- `print` statement debugging
- Remove the annot standard Python tools.

```
class RNN(torch.jit.ScriptModule):
  def __init__(self, W_h, U_h, W_y, b_h, b_y):
    def forward(self, x, h):
      y = []
      for t in range(x.size(0)):
        h = torch.tanh(x[t] @ self.W_h + h @ self.U_h + self.b_h)
        y += [torch.tanh(h @ self.W_y + self.b_y)]
        if t % 10 == 0:
          print("stats: ", h.mean(), h.var())
      return torch.stack(y), h
```

You can mix both trace and script
in a single model.

Under the hood of `@torch.jit.script`

```
class MyModule(torch.jit.ScriptModule):
    def __init__(self, N, M):
        super(MyModule, self).__init__()
        self.weight = torch.nn.Parameter(torch.rand(N, M))

    @torch.jit.script_method
    def forward(self, input):
        if bool(input.sum() > 0):
          output = self.weight.mv(input)
        else:
          output = self.weight + input
        return output

ms = MyModule(3, 4)
```

```
ms.weight[1,2]
```

```
tensor(0.5476, grad_fn=<SelectBackward>)
```

```
ms(torch.rand(4))
```

```
tensor([1.2406, 1.4690, 1.8646], grad_fn=<MvBackward>)
```

```
ms.graph
```

```
graph(%input : Dynamic
      %5 : Dynamic) {
  %7 : int = prim::Constant[value=1]()
  %2 : int = prim::Constant[value=0]()
  %1 : Dynamic = aten::sum(%input)
  %3 : Dynamic = aten::gt(%1, %2)
  %4 : bool = prim::TensorToBool(%3)
  %output : Dynamic = prim::If(%4)
    block0() {
      %output.1 : Dynamic = aten::mv(%5, %input)
      -> (%output.1)
    }
    block1() {
      %output.2 : Dynamic = aten::add(%5, %input, %7)
      -> (%output.2)
    }
  return (%output);
}
```

```
print(ms.graph.pretty_print())
```

```
def graph(self,
    input: Tensor,
    _0: Tensor) -> Tensor:
  if bool(aten.gt(aten.sum(input), 0)):
    output = aten.mv(_0, input)
  else:
    output = aten.add(_0, input, alpha=1)
  return output
```

```
ms.save('mymodel.pt')
```

Predictable error messages `@torch.jit.script`

PARSE TIME

```
@torch.jit.script_method
def forward(self, input):
    if bool(input.sum() > 0):
      output = self.weight.mv(input_foo)
    else:
      output = self.weight + input
    return output
```

```
RuntimeError:
undefined value inpu:
@torch.jit.script_method
def forward(self, input):
    if bool(input.sum() > 0):
      output = self.weight.mv(inpu)
                              ~~~~ <--- HERE
    else:
      output = self.weight + input
    return output
```

RUNTIME

```
ms(torch.rand(111))
```

```
RuntimeError:
size mismatch, [3 x 4], [111] at caffe2/
operation failed in interpreter:
@torch.jit.script_method
def forward(self, input):
    if bool(input.sum() > 0):
      output = self.weight.mv(input)
               ~~~~~~~~~~~~~~~ <--- HERE
    else:
      output = self.weight + input
    return output
```

Loading a model without Python

Torch Script models can be saved to a model archive, and loaded in a python-free executable using a C++ API.

Our C++ Tensor API is the same as our Python API, so you can do preprocessing and post processing before calling the model.

```
# Python: save model
traced_resnet = torch.jit.trace(torchvision.models.resnet18(),
                                torch.rand(1, 3, 224, 224))
traced_resnet.save("serialized_resnet.pt")
```

```
// C++: load and run model
auto module = torch::jit::load("serialized_resnet.pt");
auto example = torch::rand({1, 3, 224, 224});
auto output = module->forward({example}).toTensor();
std::cout << output.slice(1, 0, 5) << '\n';
```

HARDWARE EFFICIENCY

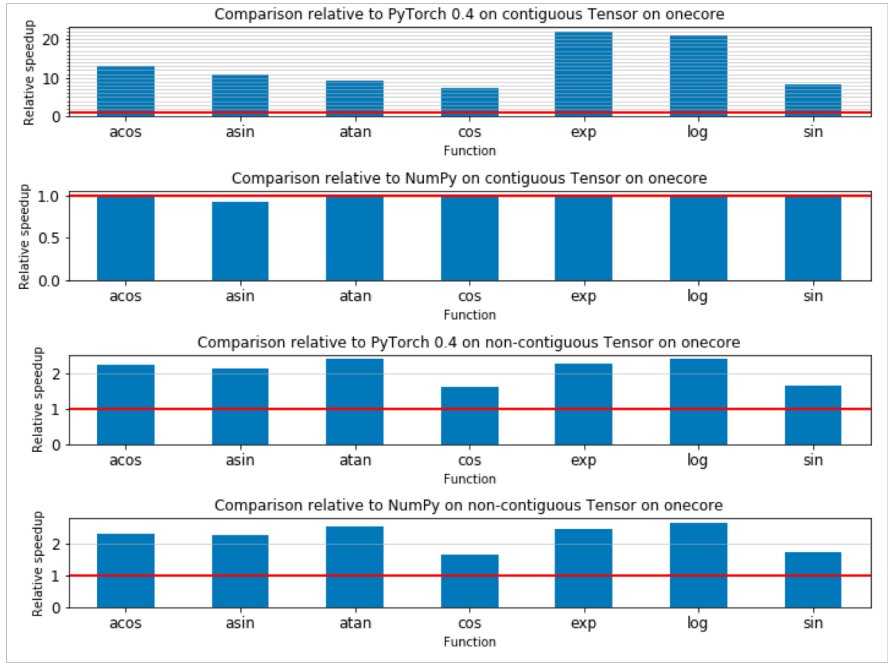
Faster operator performance

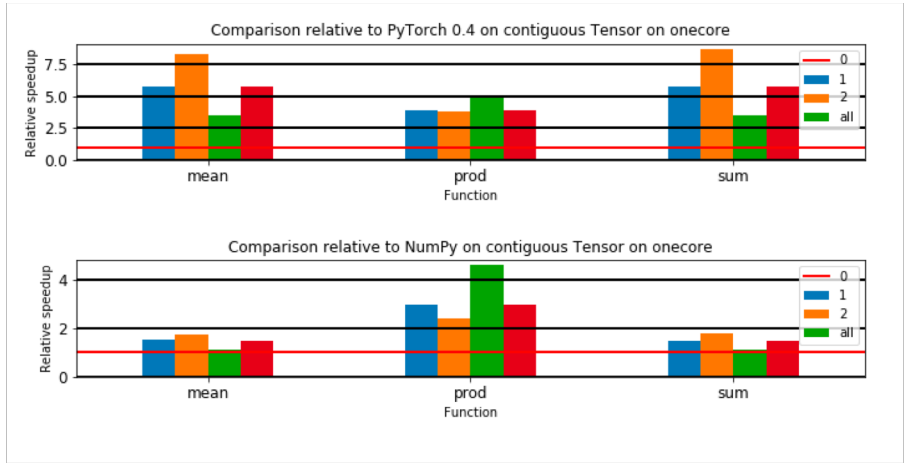
In PyTorch 1.0:

- Leveraging specialized libraries: MKL-DNN, CuDNN, etc
- Faster implementations for dozens of basic tensor operations

What's next:

- Exposing all of the best operator implementations from Caffe2





HARDWARE EFFICIENCY

Connecting to ONNX Ecosystem

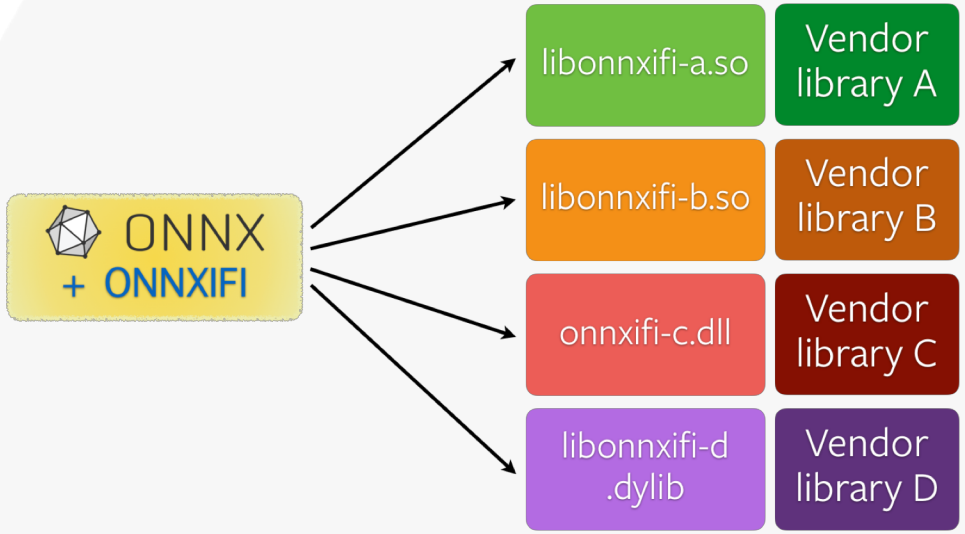
Vendor runtimes are best for running things fast.

In PyTorch 1.0:

- Export entire model to ONNX for inference

What's coming:

- ONNXIFI runtimes as part of bigger model through JIT



SCALABILITY

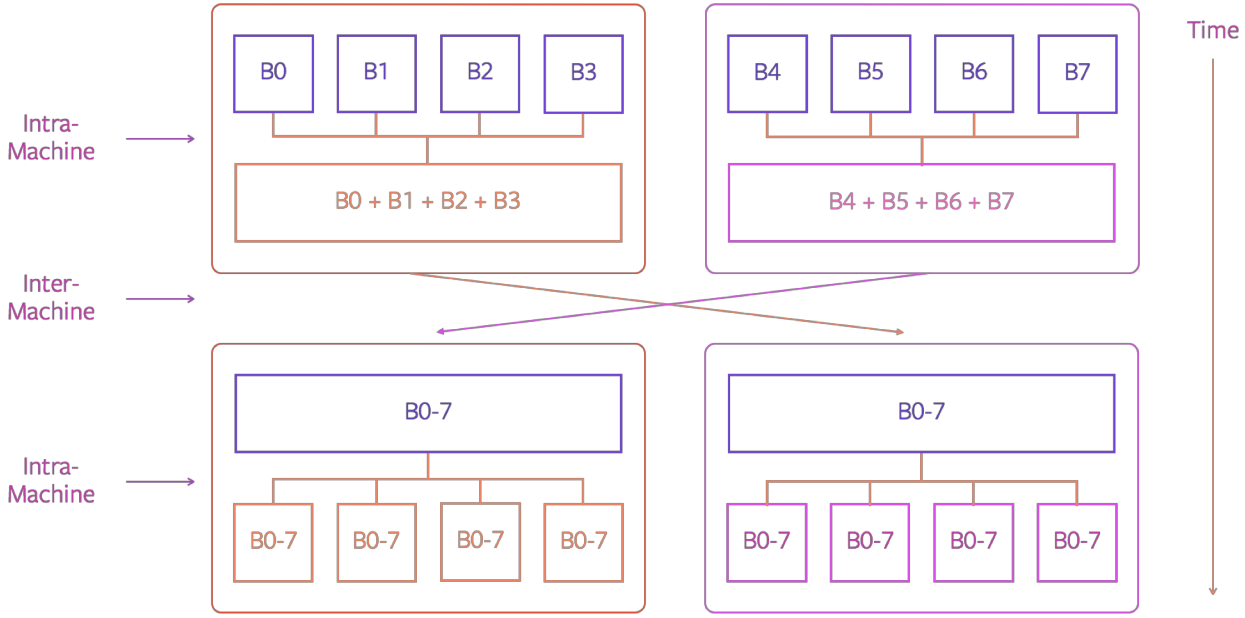
Distributed training

Challenges:

- Scaling to hundreds of GPUs
- Heterogeneous clusters, Ethernet/InfiniBand
- Potentially unreliable nodes

In PyTorch 1.0:

- Fully revamped distributed backend - c10d



SCALABILITY & CROSS-PLATFORM

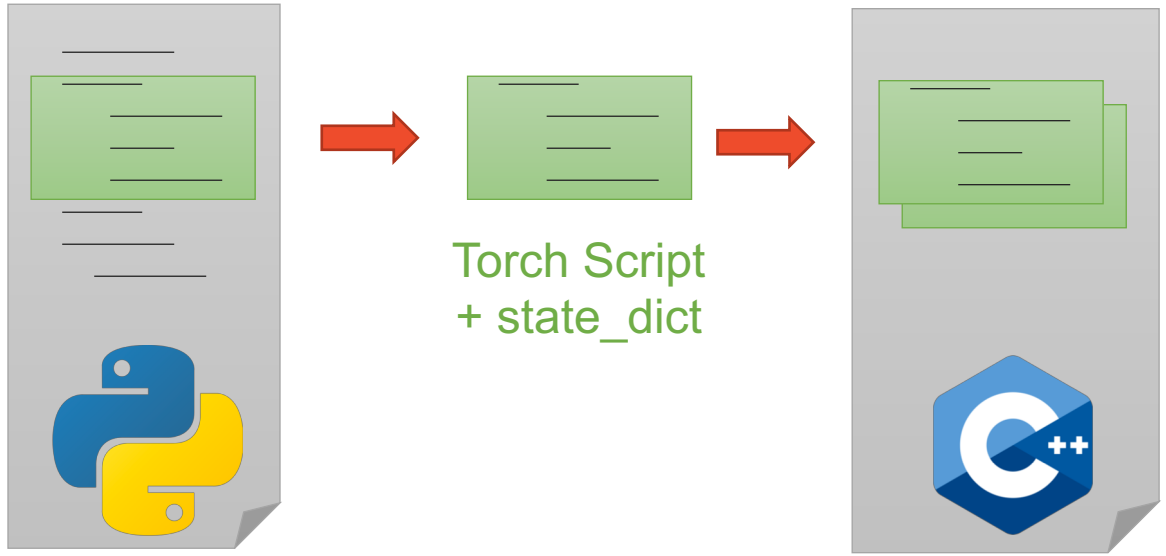
Deployment in C++

Often Python is not an option:

- High overhead on small models
- Multithreading services bottleneck on GIL
- Deployment service might be C++ only

In PyTorch 1.0:

- Convert inference part of the model to Torch Script
- Link with libtorch.so in your C++ application



PYTORCH DISTRIBUTED TRAINING

TENG LI
FACEBOOK AI

SIGNIFICANCE OF SCALABLE DISTRIBUTED TRAINING

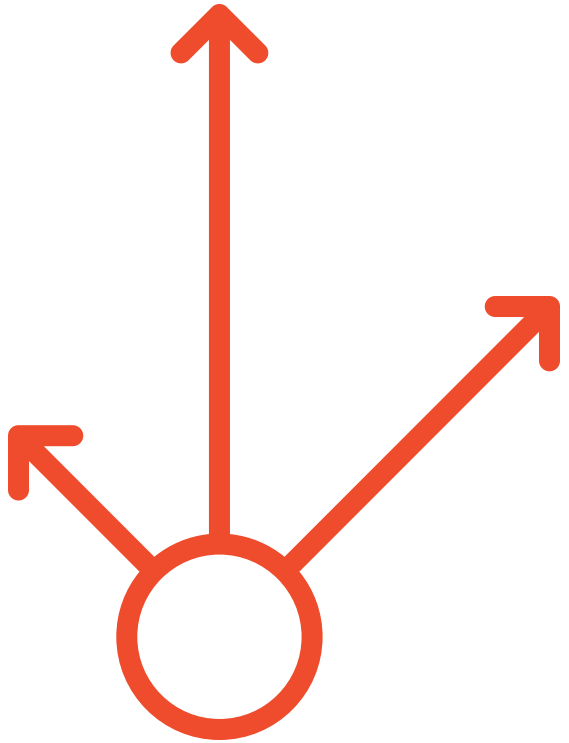
MORE COMPUTING POWER

MORE TRAINING DATA.
LARGER MODELS

- SIGNIFICANT TRAINING TIME SPEEDUPS
- GREAT EXTENT OF MODEL EXPLORATION

DISTRIBUTED – WHAT'S NEW?

- A brand new **performance-driven** distributed backend: **C10D**



HIGHLIGHTS

PyTorch 1.0 Distributed

BRAND NEW BACKEND DESIGN

- Fully asynchronous backend library: C10D
- Both Python and C++ support
- Fully backward-compatible frontend python API

HIGHLY SCALABLE PERFORMANCE

- Near roofline performance on key workloads
- Data Parallel: Single-node, multi-GPUs
- Data Parallel: Multi-node, multi-GPUs

C10D LIBRARY

DESIGN AND FEATURES

- Backends
  - Gloo, NCCL, MPI
- Fully asynchronous collectives for all backends
- Both Python and C++ APIs
- Performance-driven design
  - Self-managed CUDA streams for parallel execution
- Upcoming
  - Fault tolerance with elasticity

FULLY ASYNC DESIGN

PYTHON API

```python
import torch
import torch.distributed as dist

# Options
opts = dist.AllreduceOptions()

# Creating the process group with store method
store = dist.FileStore("/tmp/test")
pg = dist.ProcessGroupNCCL(store, RANK, WORLD_SIZE)

# Kicking off work
# Assuming that tensors are a list of Tensors
works = []
for tensor in tensors:
    work = pg.allreduce([tensor], opts)
    works.append(work)

# Wait
for work in works:
    work.wait()
```

C++ API

```cpp
// Creating the process group with store method
auto store = std::make_shared<FileStore>("/tmp/test");
ProcessGroupNCCL pg(store, RANK, WORLD_SIZE);

// Kicking off work
// Assuming that tensors are a vector of at::Tensor
std::vector<std::shared_ptr<ProcessGroup::Work>> works;
for (auto i = 0; i < tensors.size(); ++i) {
  std::vector<at::Tensor> tmp = {tensors[i]};
  works.push_back(pg.allreduce(tmp));
}

// Wait
for (auto& work : works) {
  work->wait();
}
```

BACKWARD COMPATIBLE

torch.distributed

SYNC MODE

```
# Backward compatible synchronous collective op
torch.distributed.all_reduce(tensor, op, group, async_op=False)
```

ASYNC MODE

```
# New asynchronous collective op
work = torch.distributed.all_reduce(tensor, op, group, async_op=True)
work.wait()
```

DISTRIBUTED DATA PARALLEL

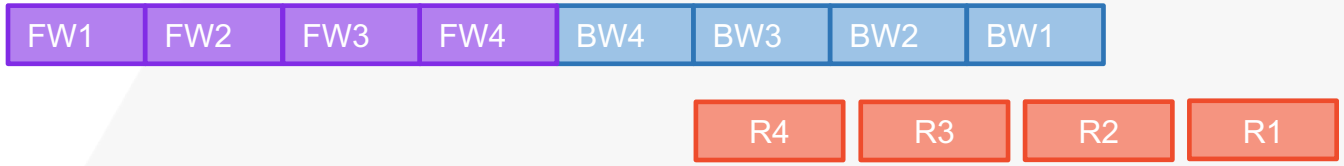
Performance-driven design

- Overlapping BWs with all-reductions
- Coalescing small tensors into buckets
  - A bucket is a big coalesced tensor

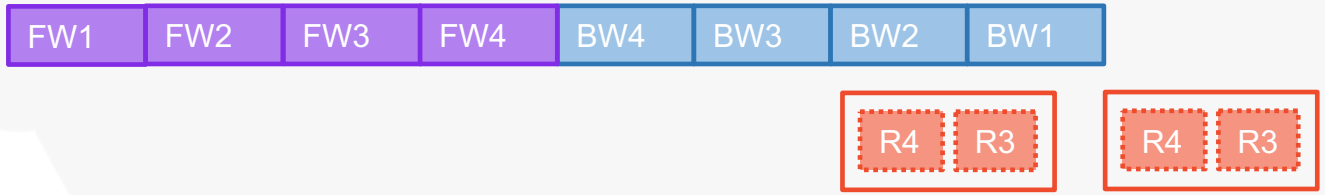
NO OVERLAPPING

An iteration: Forward (FW) -> Backward(BW) -> AllReduce(R)

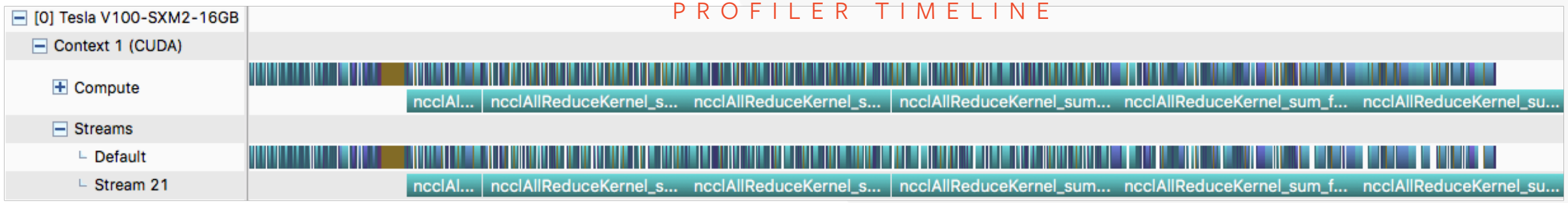
OVERLAPPING BACKWARD WITH REDUCE



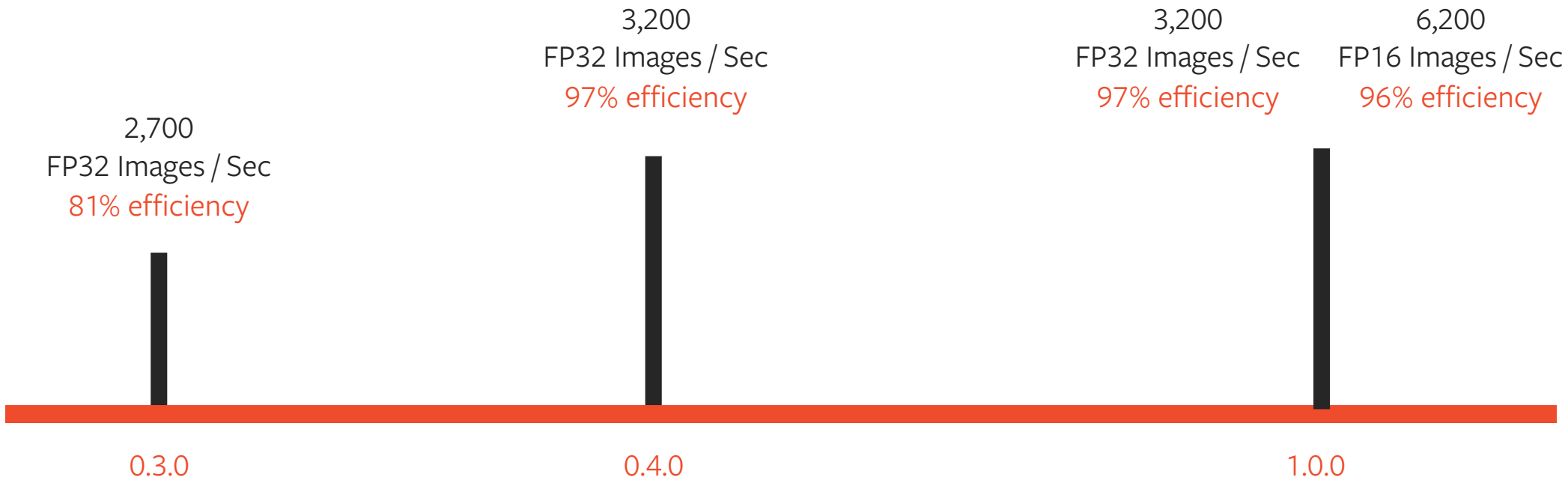
TENSOR COALESCING/ BUCKETING



PROFILER TIMELINE

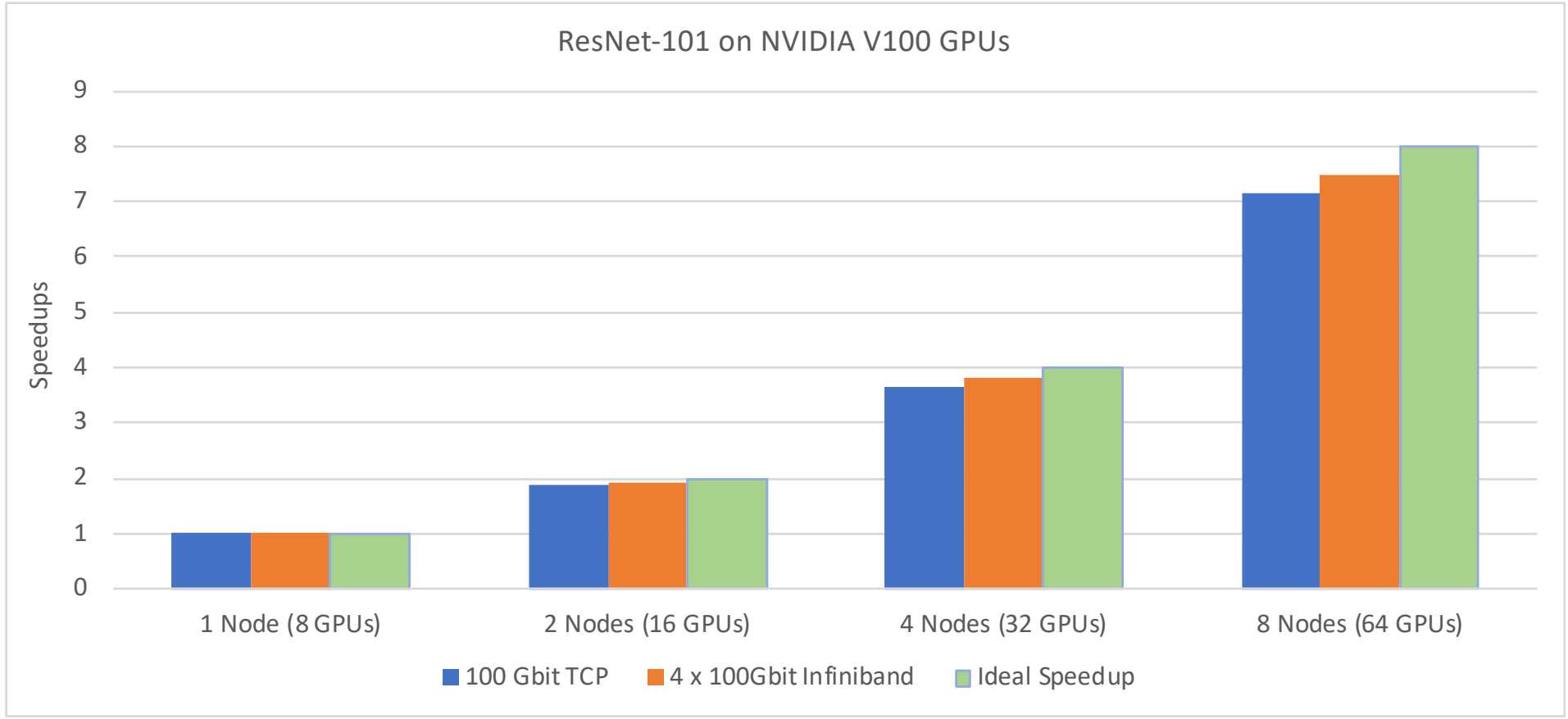


PERFORMANCE: SINGLE NODE DATA PARALLEL



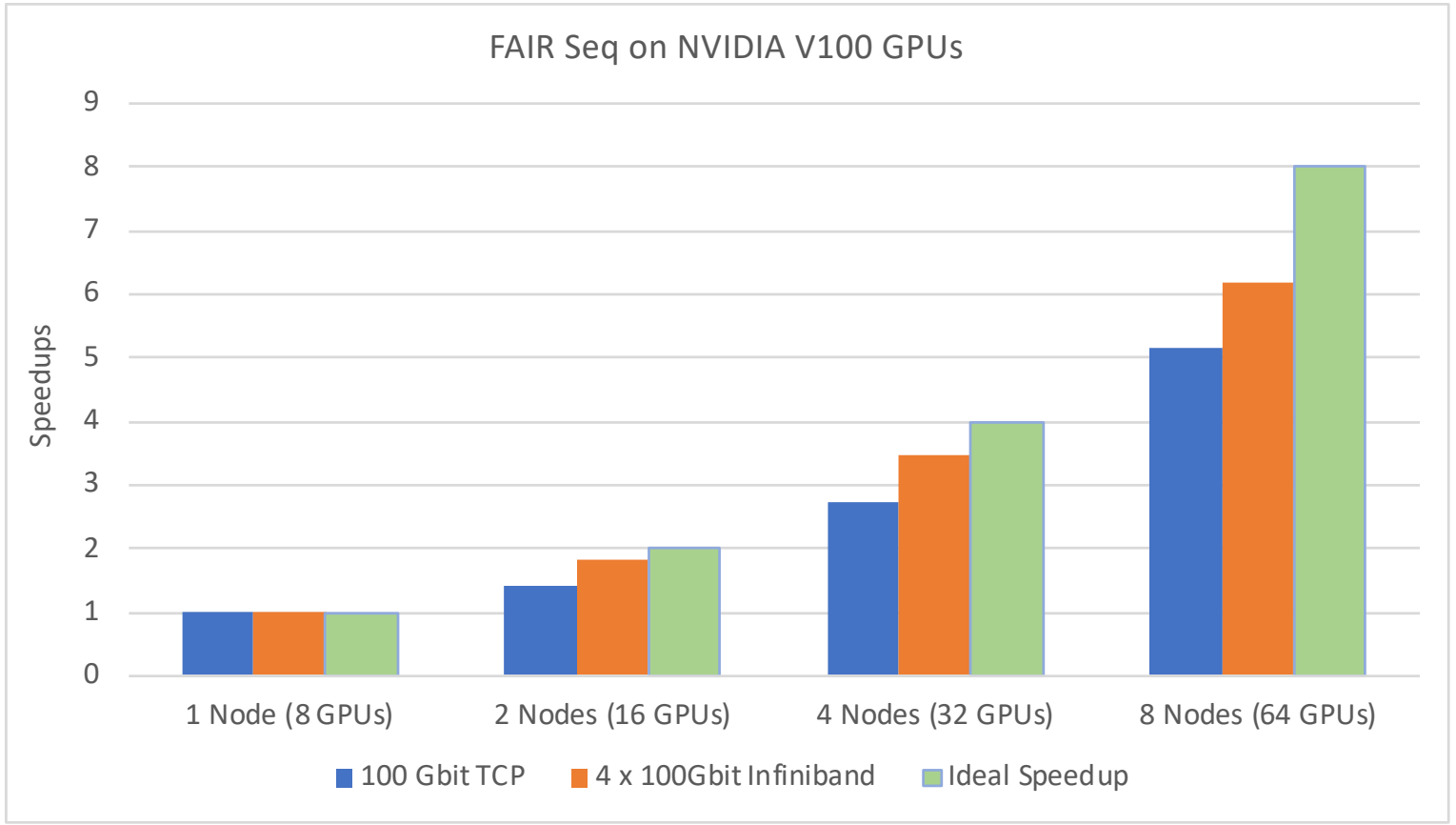
ImageNet ResNet50 on NVIDIA DGX-1 with 8X V100 GPUs

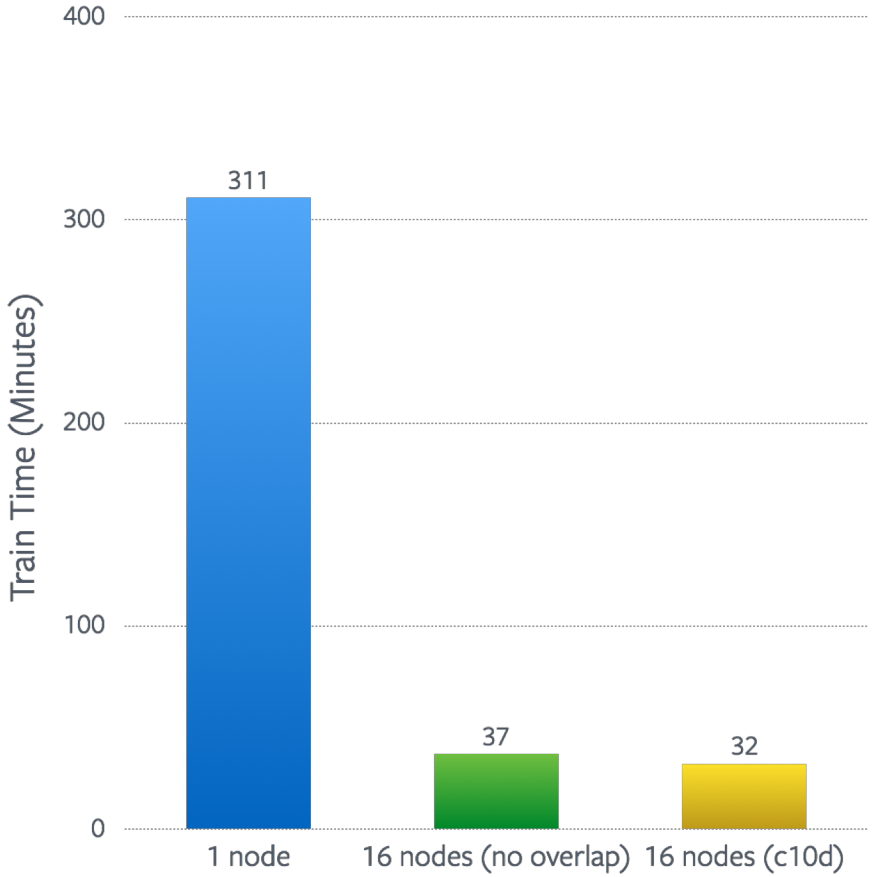
Distributed Training Performance – ResNet101



Distributed Training Performance – FAIR Seq

Bonjour à tous ! → Hello everybody!





- **311 minutes – 32 minutes**, by going from 1 to 16 NVIDIA DGX-1 nodes (8 to 128 NVIDIA V100 GPUs)
- **19% performance gain** (1.53M – 1.82M Words Per Second on 16 nodes), thanks to c10d DDP overlapping

PYTORCH 1.0

Try it out

ALL NEW FEATURES

PyTorch1.0 Stable Release

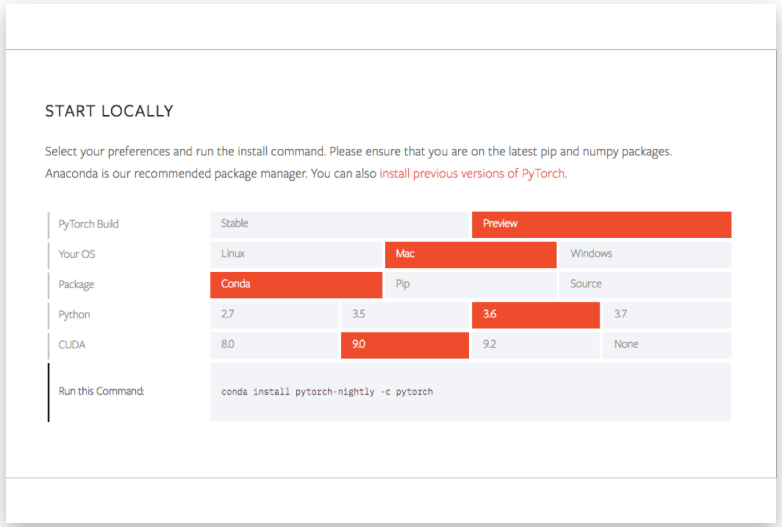
- torch.distributed
- torch.nn.parallel.DistributedDataParallel

OLD DISTRIBUTED

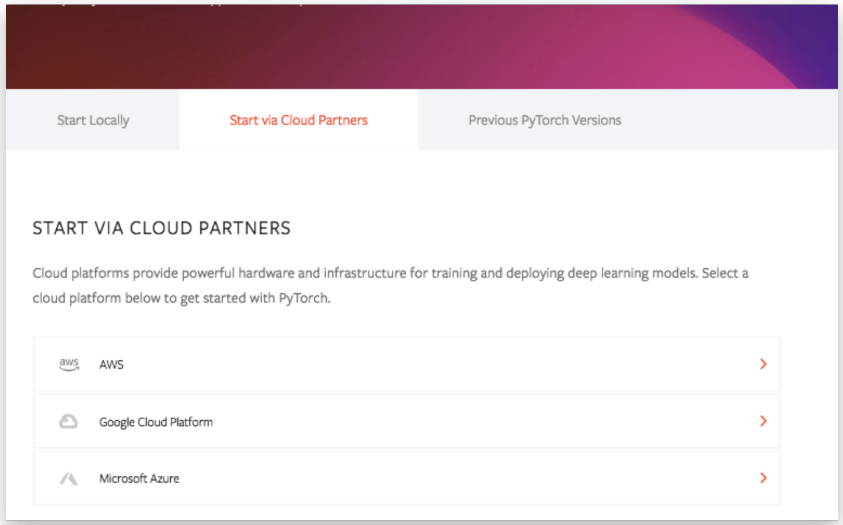
Deprecated to

- torch.distributed.deprecated
- torch.nn.parallel.deprecated.DistributedDataParallel

GET STARTED



LOCAL INSTALL



CLOUD PARTNERS

THANKS