\Orchestrating a brighter world

NEC

SOL4VE: Bringing Deep Neural Networks to the NEC SX-Aurora TSUBASA

Dr. Nicolas Weber (NEC Labs Europe)

Motivation

“[BuzzWord] is like teenage sex. Everybody wants it, but no one knows how to do it!”

Motivation

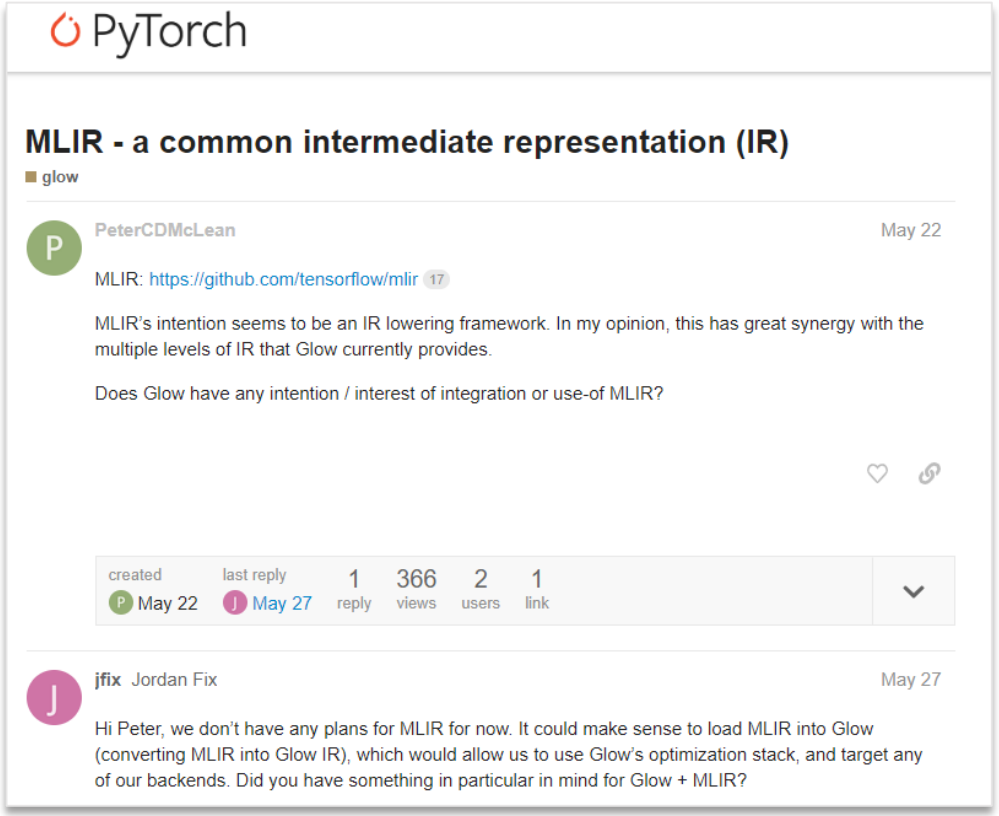
"[BuzzWord] is like teenage sex. Everybody wants it, but no one knows how to do it!"

Integration into existing frameworks is expensive

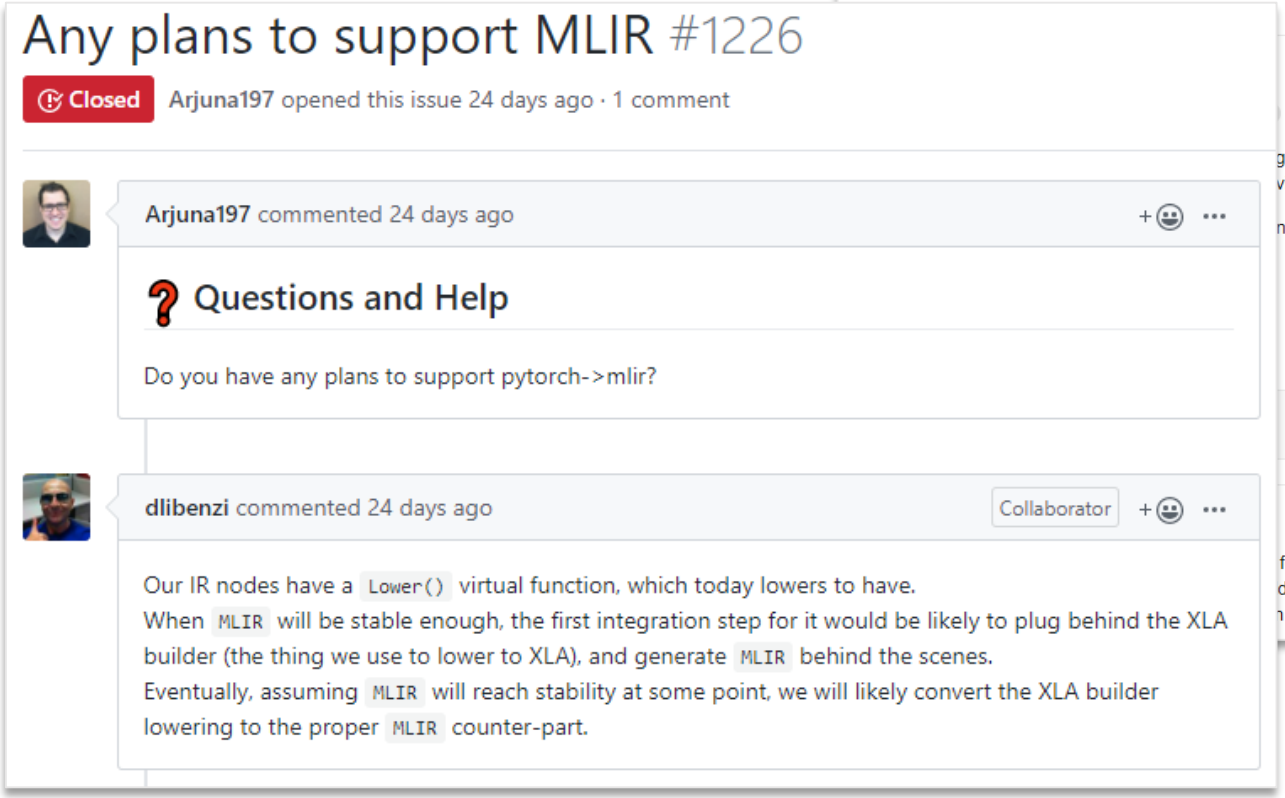
Each framework has its own internal and external APIs

- No common code base
- Approaches such as MLIR, ONNX, DLPack, … not widely adopted or very limited

Integration into existing frameworks is expensive



Integration into existing frameworks is expensive



Any plans to support MLIR #1226

Closed Arjuna197 opened this issue 24 days ago · 1 comment

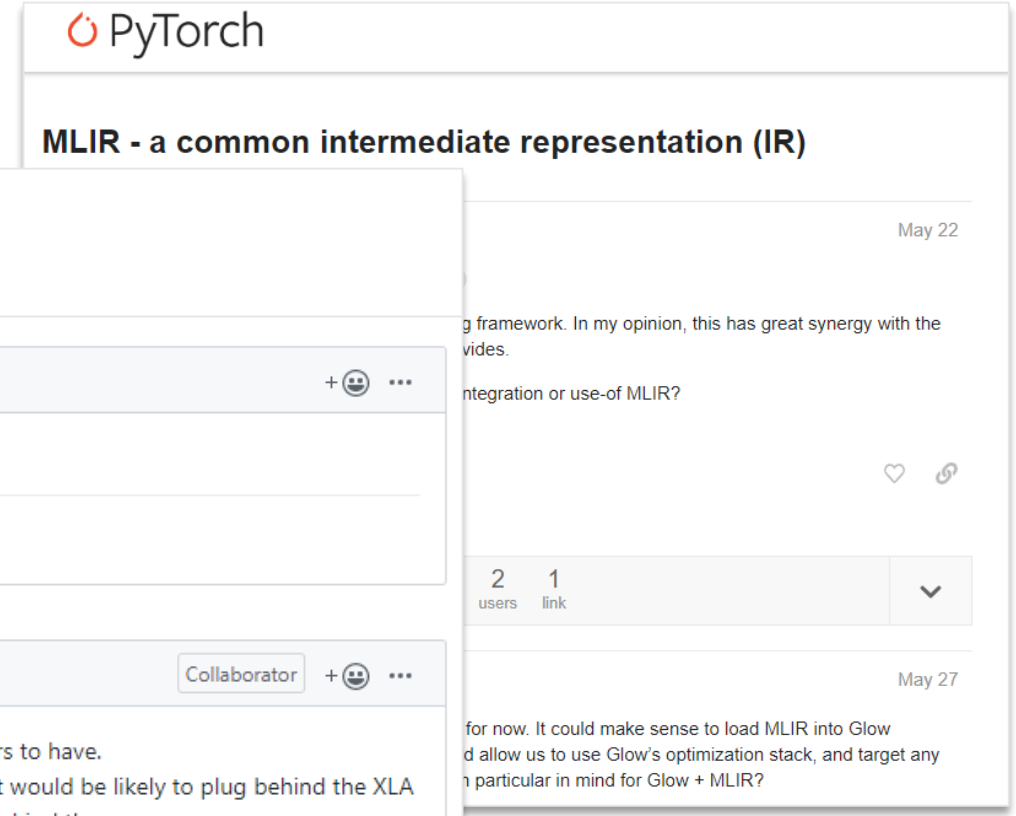
Arjuna197 commented 24 days ago

Questions and Help

Do you have any plans to support pytorch->mlir?

dlibenzi commented 24 days ago Collaborator

Our IR nodes have a `Lower()` virtual function, which today lowers to have.
When `MLIR` will be stable enough, the first integration step for it would be likely to plug behind the XLA builder (the thing we use to lower to XLA), and generate `MLIR` behind the scenes.
Eventually, assuming `MLIR` will reach stability at some point, we will likely convert the XLA builder lowering to the proper `MLIR` counter-part.



PyTorch

MLIR - a common intermediate representation (IR)

May 22

g framework. In my opinion, this has great synergy with the vides.

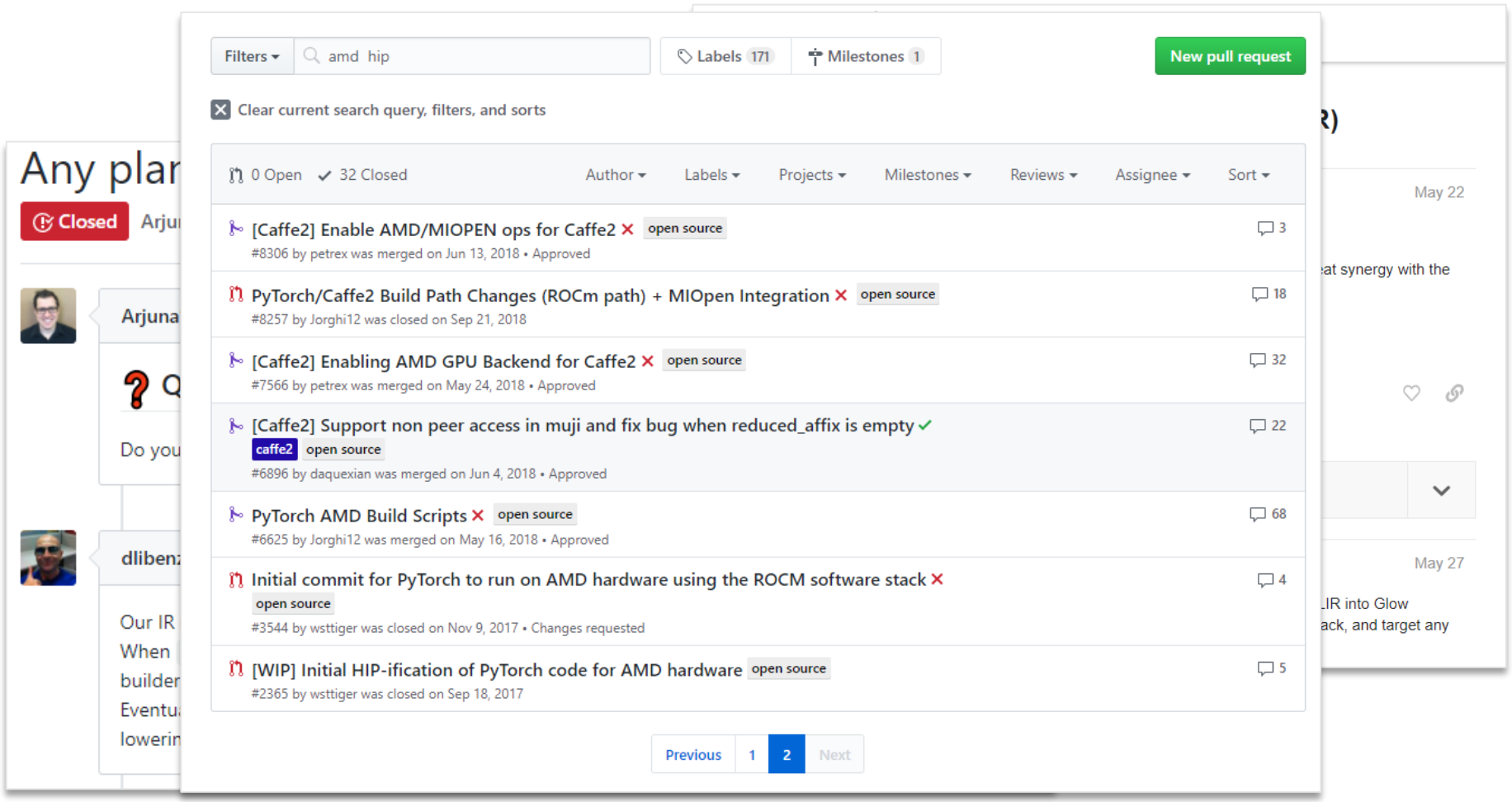
ntegration or use-of MLIR?

2 users 1 link

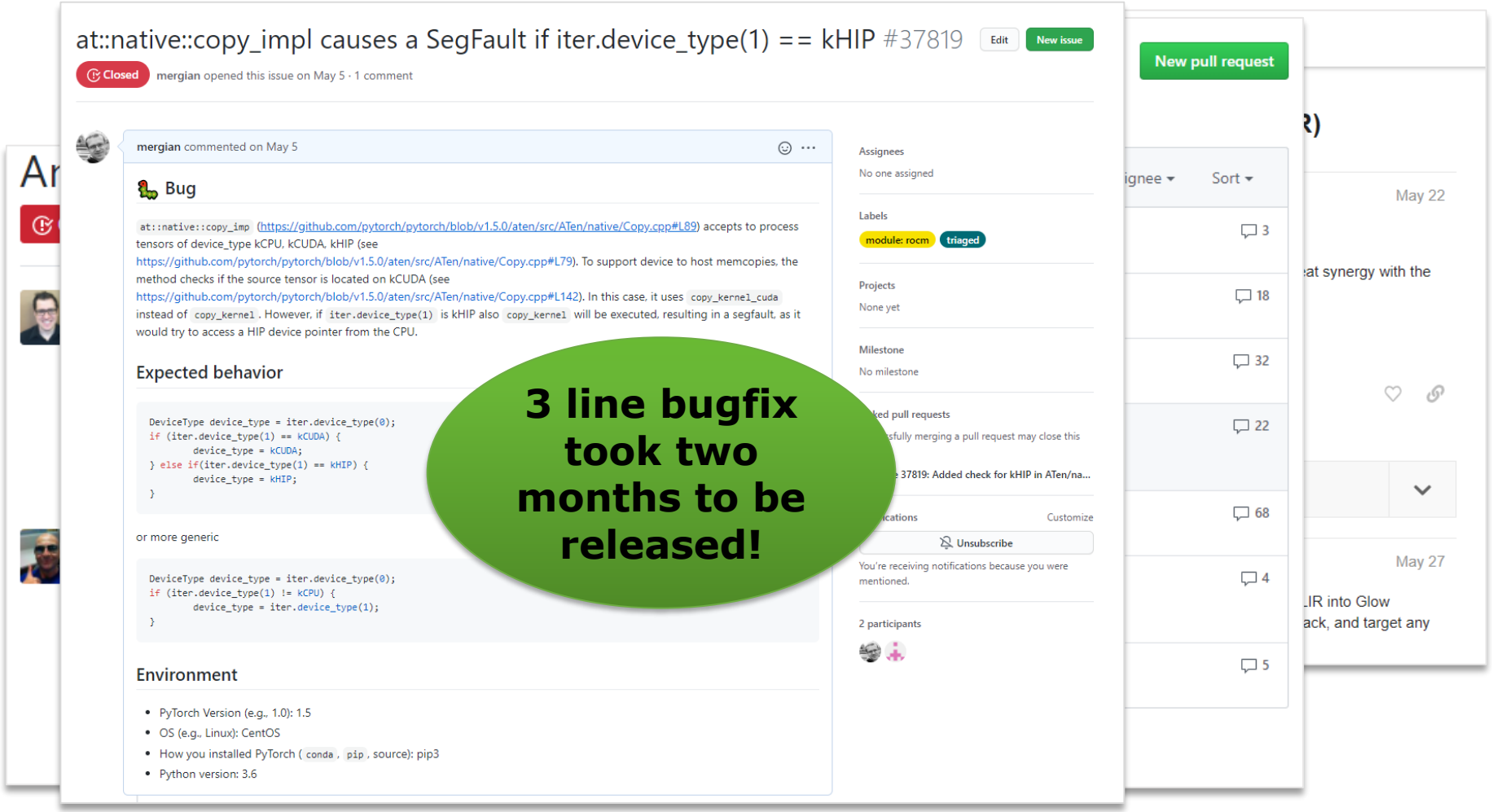
May 27

for now. It could make sense to load MLIR into Glow
d allow us to use Glow's optimization stack, and target any
n particular in mind for Glow + MLIR?

Integration into existing frameworks is expensive



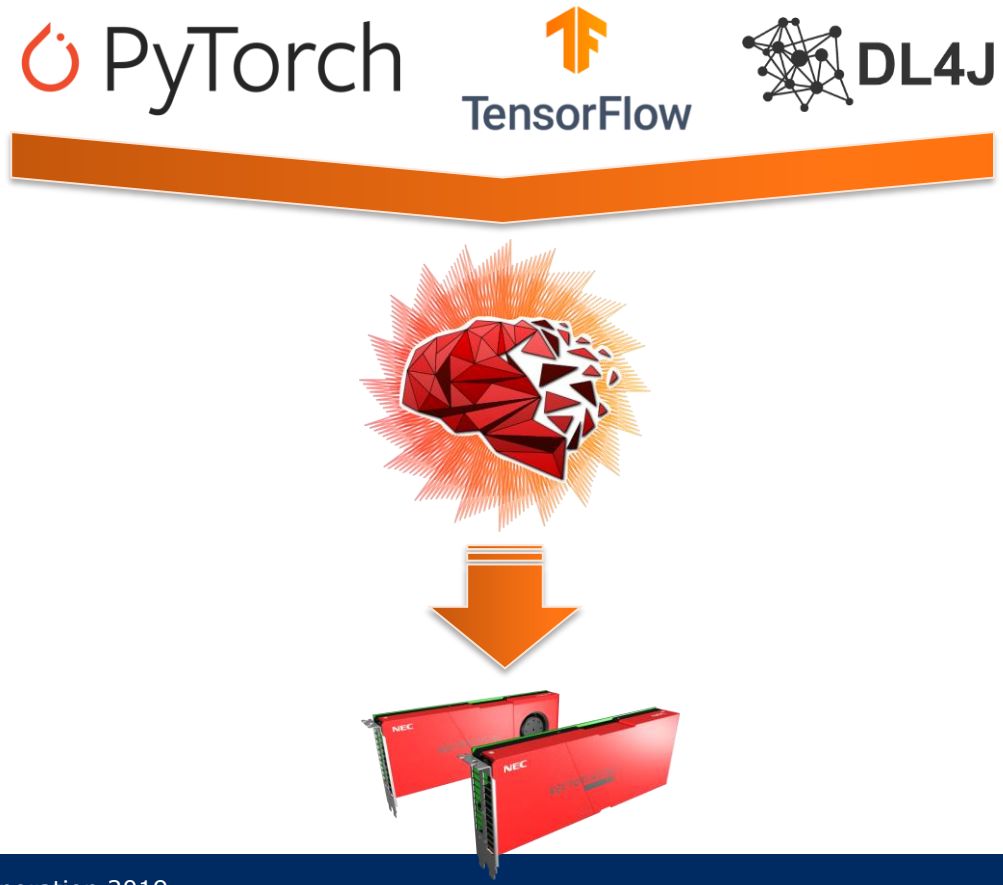
Integration into existing frameworks is expensive



The SOL-Project

SOL is a full stack AI acceleration middleware

- Add-on to AI frameworks that does not require any code changes to the framework
- Optimizations range from mathematical/algorithmic down to actual implementations/code generation



SOL in a nutshell

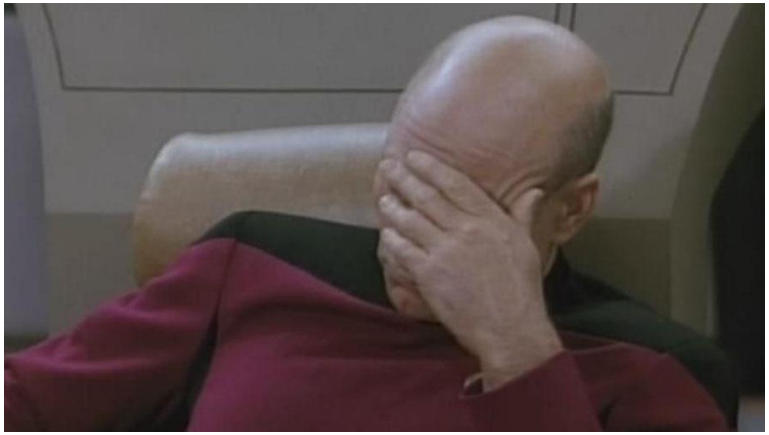
What data scientists see:

```
x = Conv(x, kernel=1x1, bias=True)
x = ReLU(x)
x = AvgPooling(x, kernel=13x13)
```

SOL in a nutshell

What data scientists see:

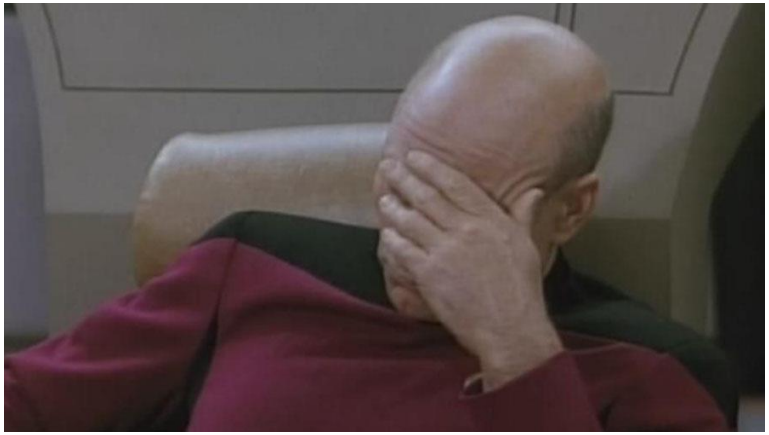
```
x = Conv(x, kernel=1x1, bias=True)
x = ReLU(x)
x = AvgPooling(x, kernel=13x13)
```



SOL in a nutshell

What data scientists see:

```
x = Conv(x, kernel=1x1, bias=True)
x = ReLU(x)
x = AvgPooling(x, kernel=13x13)
```



What HPC people see:

```
function(Conv):
    for(Batch, OutChannel, Y, X):
        for(InChannel, KernelY, KernelX):
            output[…] += input[…] * weight[…]
        output[…] += bias[…]

function(ReLU):
    for(Batch, OutChannel, Y, X):
        output[…] = max(0, input[…])

function(AvgPooling):
    for(Batch, OutChannel, Y, X):
        for(KernelY, KernelX):
            output[…] += input[…] / (13*13)
```

SOL in a nutshell (continued)

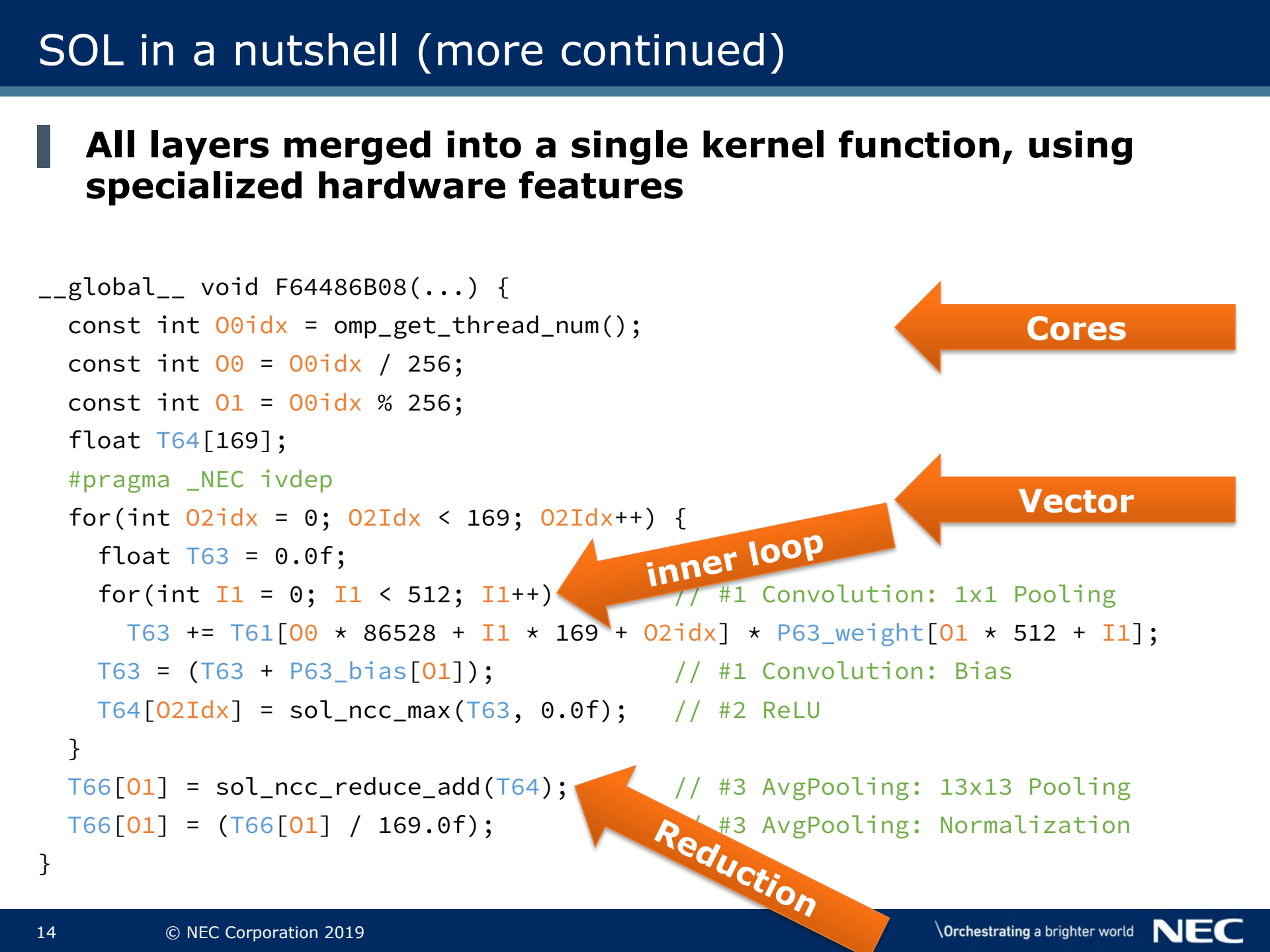
What we actually want:

```
function(FusedNetwork):
  for(Batch, OutChannel):
    float N[…]
    for(Y, X):
        for(InChannel, KernelY, KernelX):
            N[…] += input[…] * weight[…]
        N[…] += bias[…]
        N[…] = max(0, X)
    for(Y, X):
        for(KernelY, KernelX):
            output[…] += N[…] / (13*13)
```

SOL in a nutshell (more continued)

All layers merged into a single kernel function, using specialized hardware features

```
__global__ void F64486B08(...) {
  const int O0idx = omp_get_thread_num();
  const int O0 = O0idx / 256;
  const int O1 = O0idx % 256;
  float T64[169];
  #pragma _NEC ivdep
  for(int O2idx = 0; O2Idx < 169; O2Idx++) {
    float T63 = 0.0f;
    for(int I1 = 0; I1 < 512; I1++)
      T63 += T61[O0 * 86528 + I1 * 169 + O2idx] * P63_weight[O1 * 512 + I1];
    T63 = (T63 + P63_bias[O1]);                  // #1 Convolution: Bias
    T64[O2Idx] = sol_ncc_max(T63, 0.0f);         // #2 ReLU
  }
  T66[O1] = sol_ncc_reduce_add(T64);             // #3 AvgPooling: 13x13 Pooling
  T66[O1] = (T66[O1] / 169.0f);                  // #3 AvgPooling: Normalization
}
```



SOL Usage (PyTorch)

```
import torch
from torchvision import models


py_model  = models.__dict__["…"]()
input     = torch.rand(1, 32, 224, 224)

output    = py_model(input)
```

SOL Usage (PyTorch)

```
import torch
from torchvision import models
import sol.pytorch as sol

py_model  = models.__dict__["…"]()
input     = torch.rand(1, 32, 224, 224)
sol_model = sol.optimize(py_model, [input])
output    = sol_model(input)
```

How SOL integrates into the frameworks?

- SOL injects its optimized code as custom model into the framework

```
class SolLayer(torch.nn.Module):
      def  __init__(self):
            self.ParamA = …
            self.ParamB = …

      def forward(self, X):
            return sol.run(X, self.ParamA, self.ParamB)
```

framework handles model parameters!

SOL handles execution

Tested Neural Networks

Convolutional Neural Networks

- Alexnet
- SqueezeNet (1.0, 1.1)
- VGG + BN (11, 13, 16, 19)
- Resnet (18, 34, 50, 101, 152)
- Densenet (121, 161, 169, 201)
- Inception V3
- GoogleNet
- MobileNet (v1, v2)
- MNasNet (0.5, 0.75, 1.0, 1.3)
- ShuffleNet V2 (0.5, 1.0, 1.5, 2.0)
- ResNext (50, 101)
- WideResNet (50, 101)

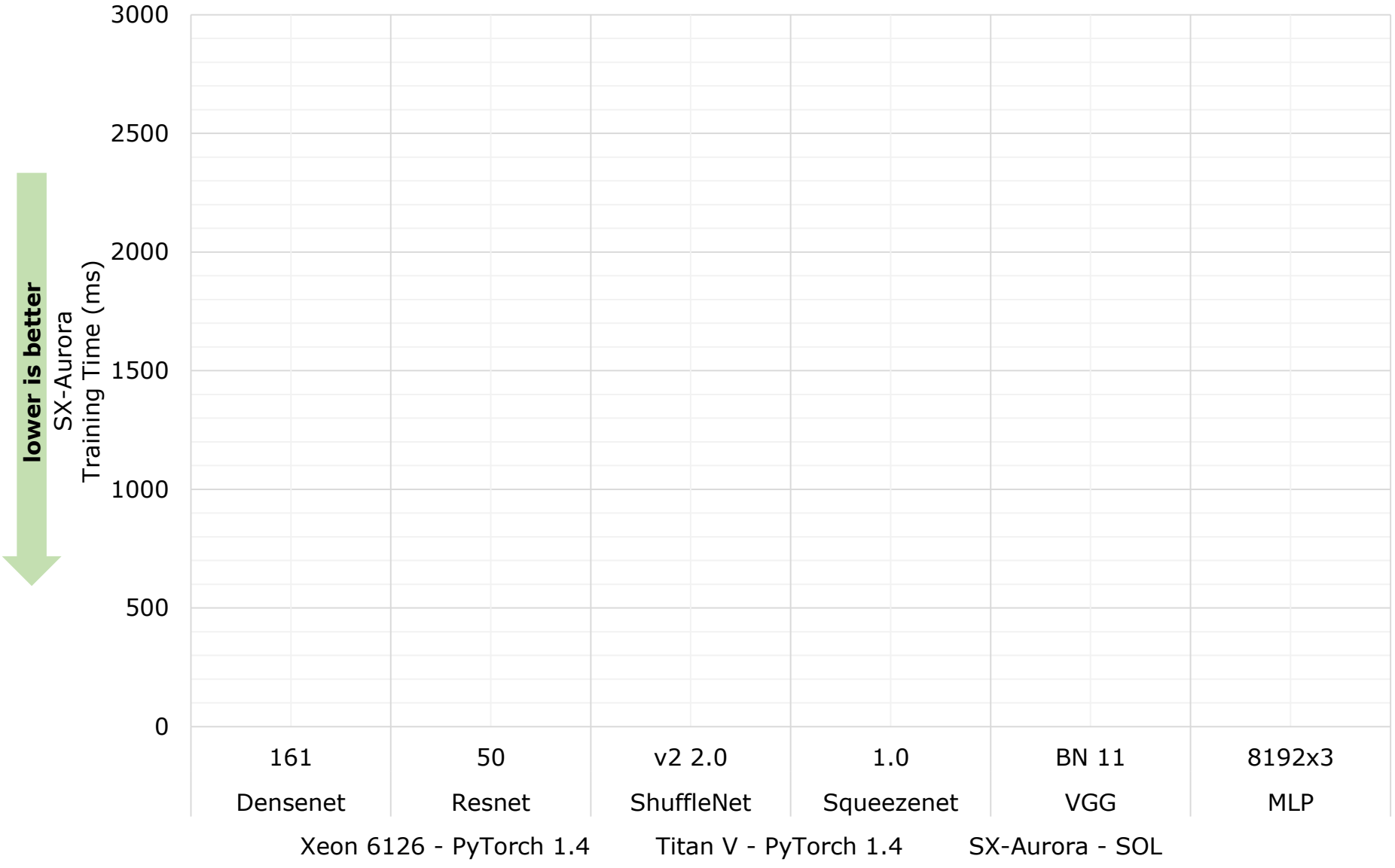
Multi Layer Perceptron (MLP)

Linear/Logistic Regression

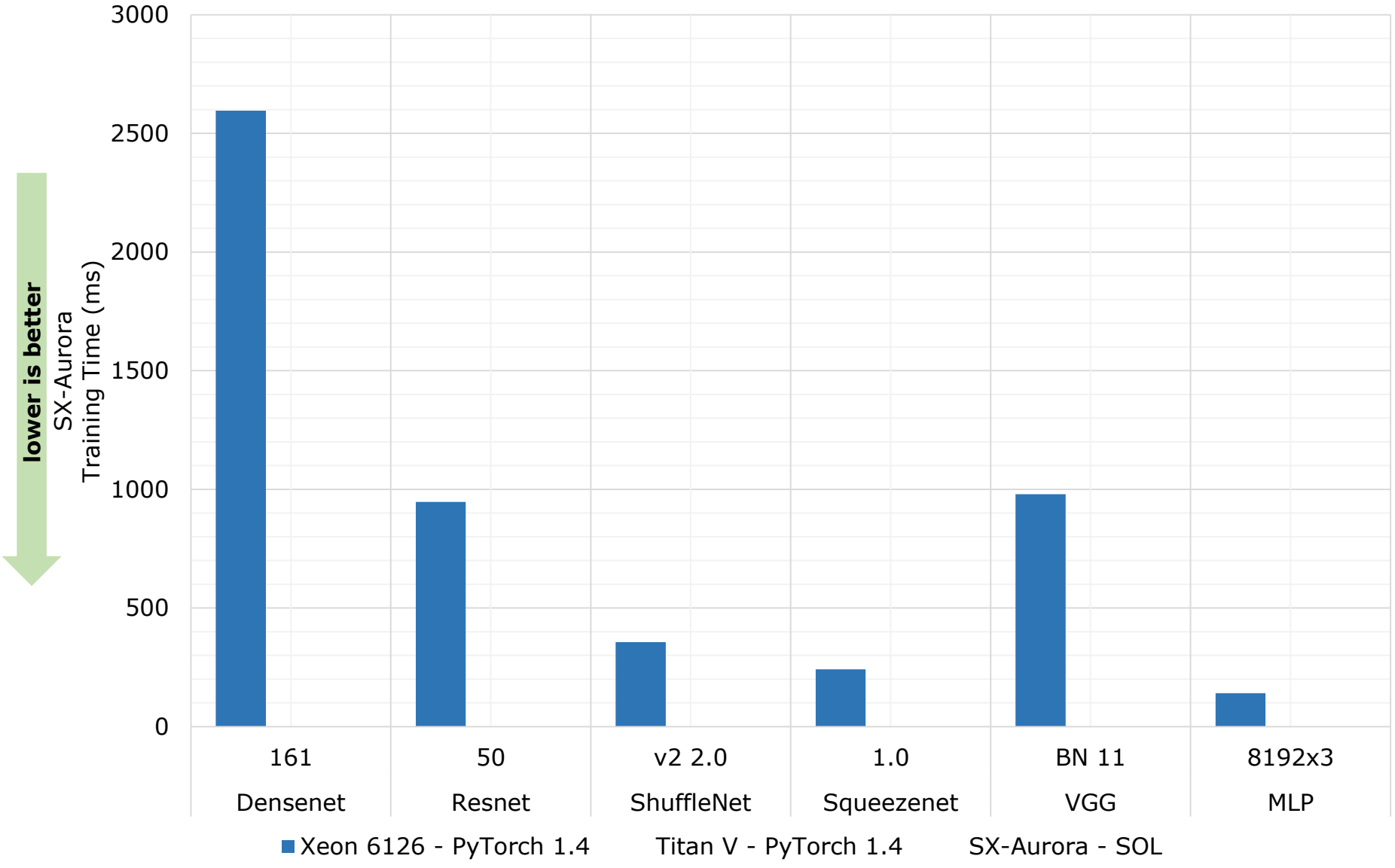
Natural Language Processing

- BERT (PyTorchic + HuggingFace implemenations)
- *GPT-2 (in upcoming v0.3.0 release)*
- *LSTM+GRU (coming in Q4 2020)*

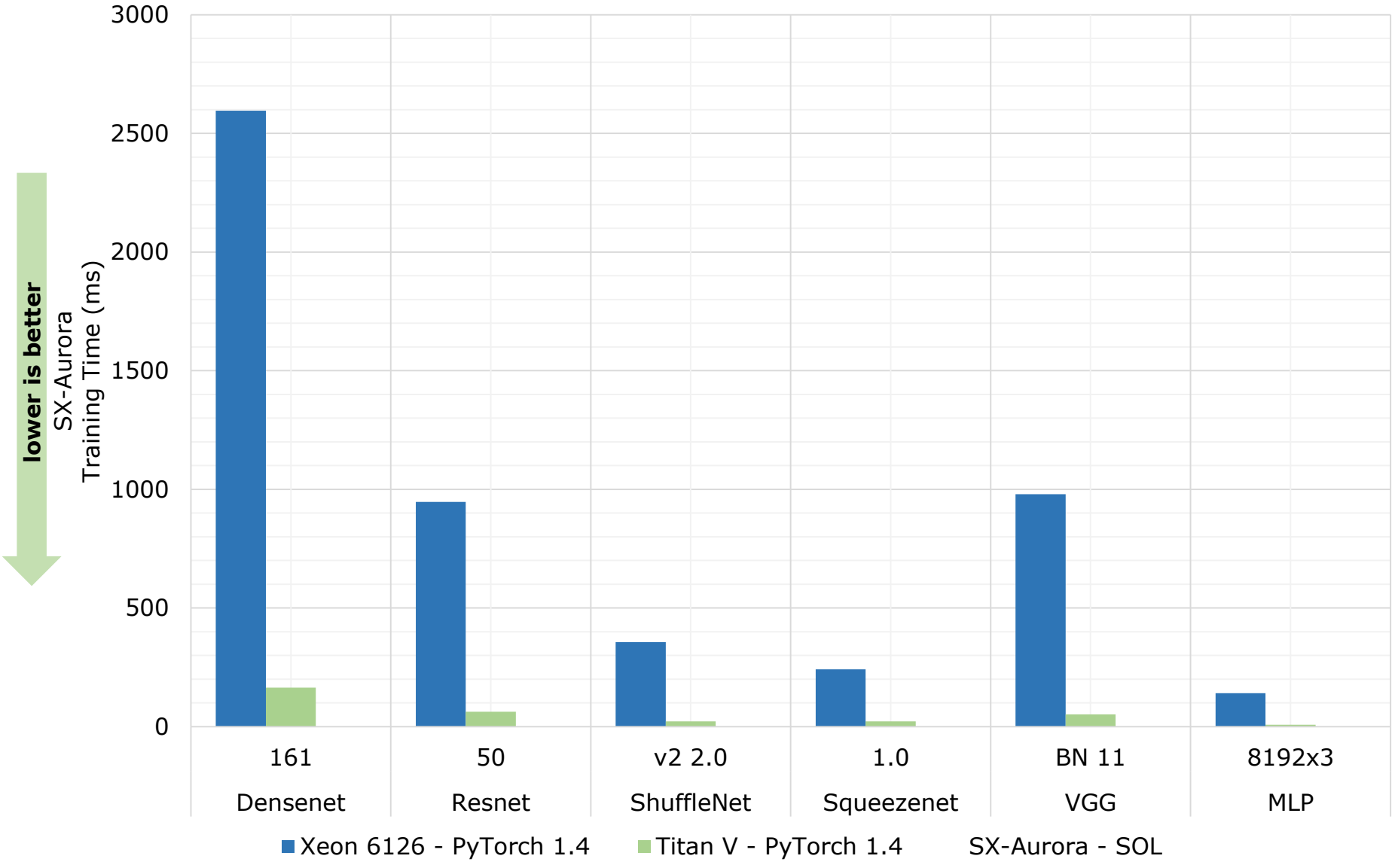
Training Performance (CNN BS=16, MLP BS=64)



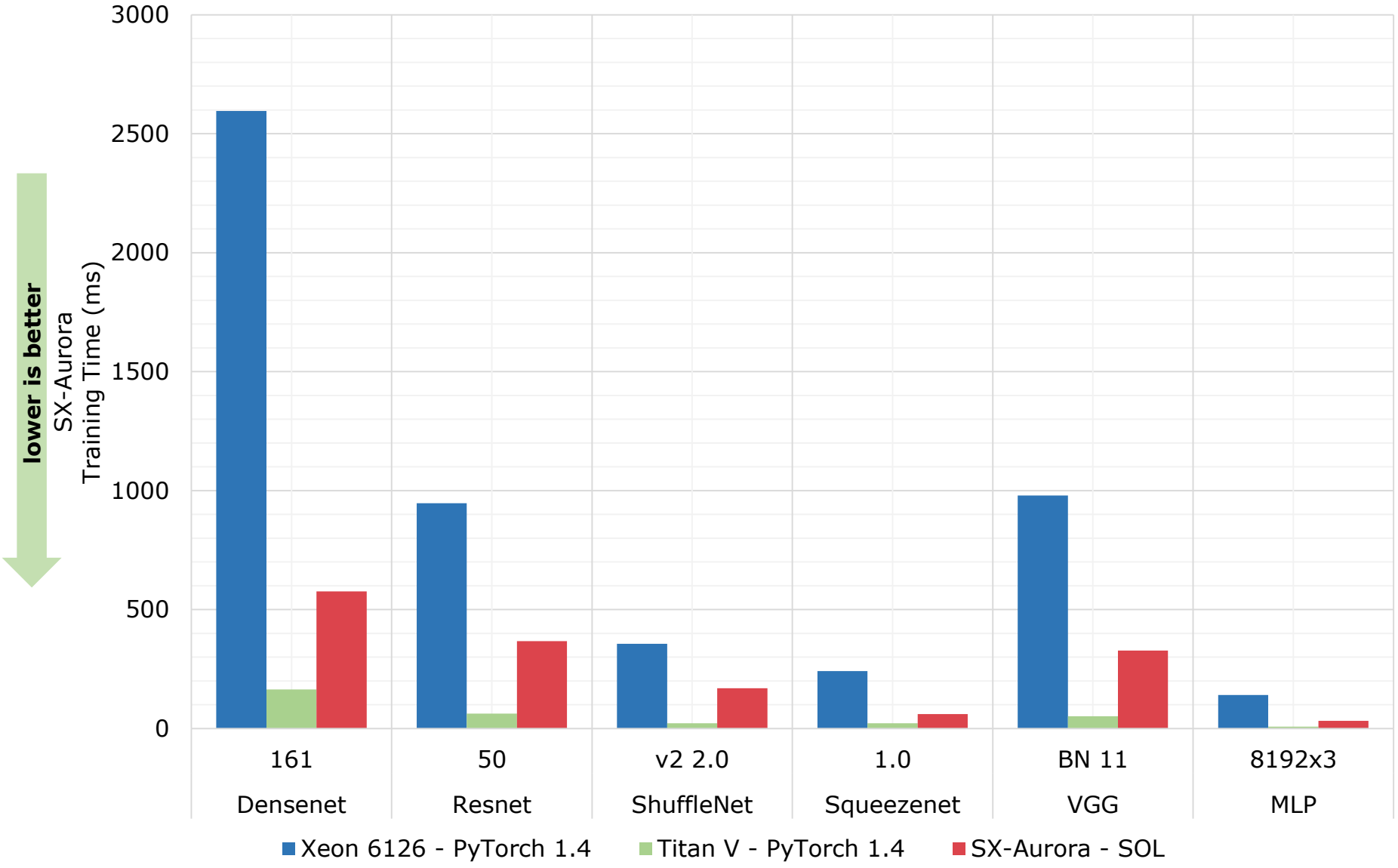
Training Performance (CNN BS=16, MLP BS=64)



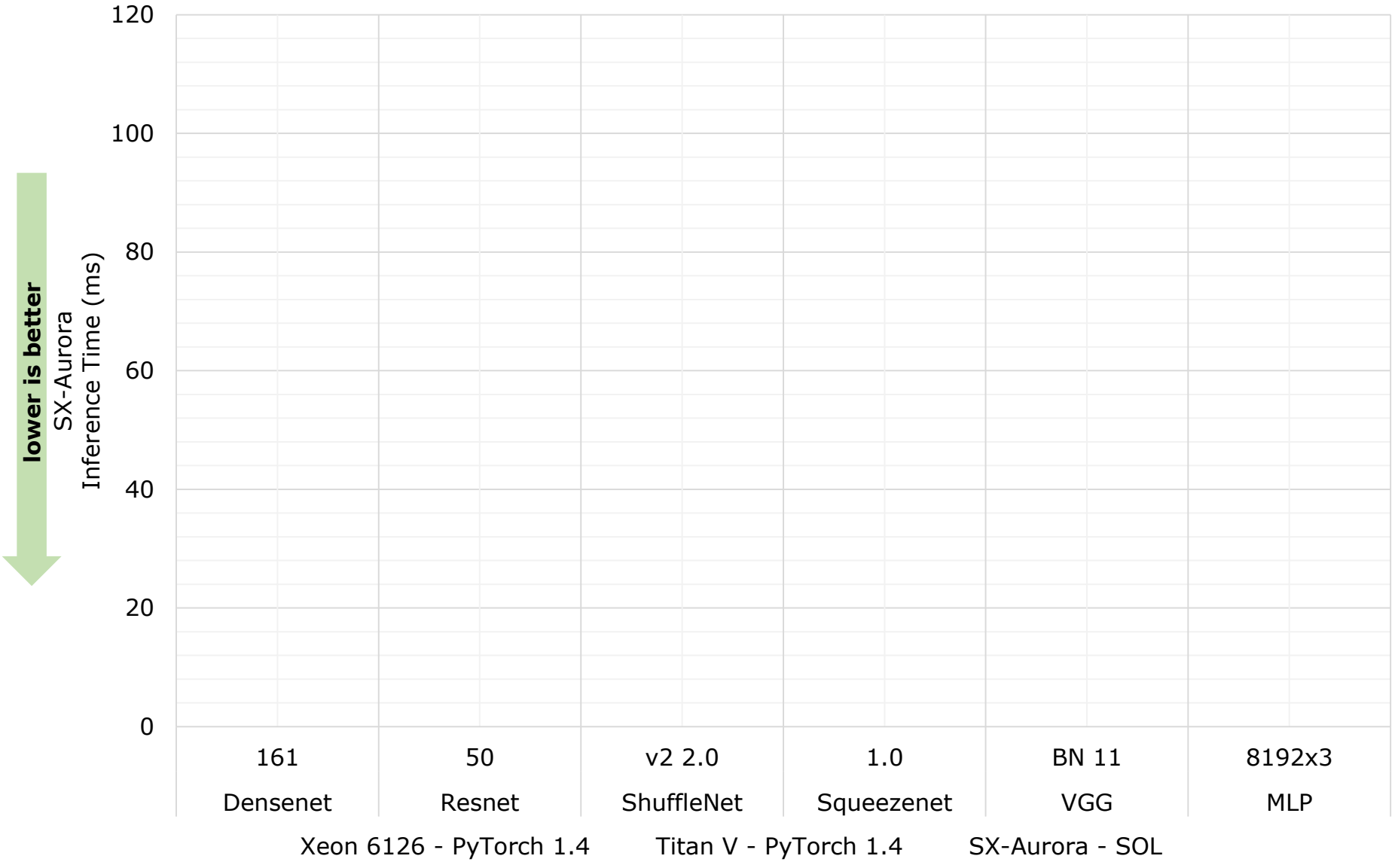
Training Performance (CNN BS=16, MLP BS=64)



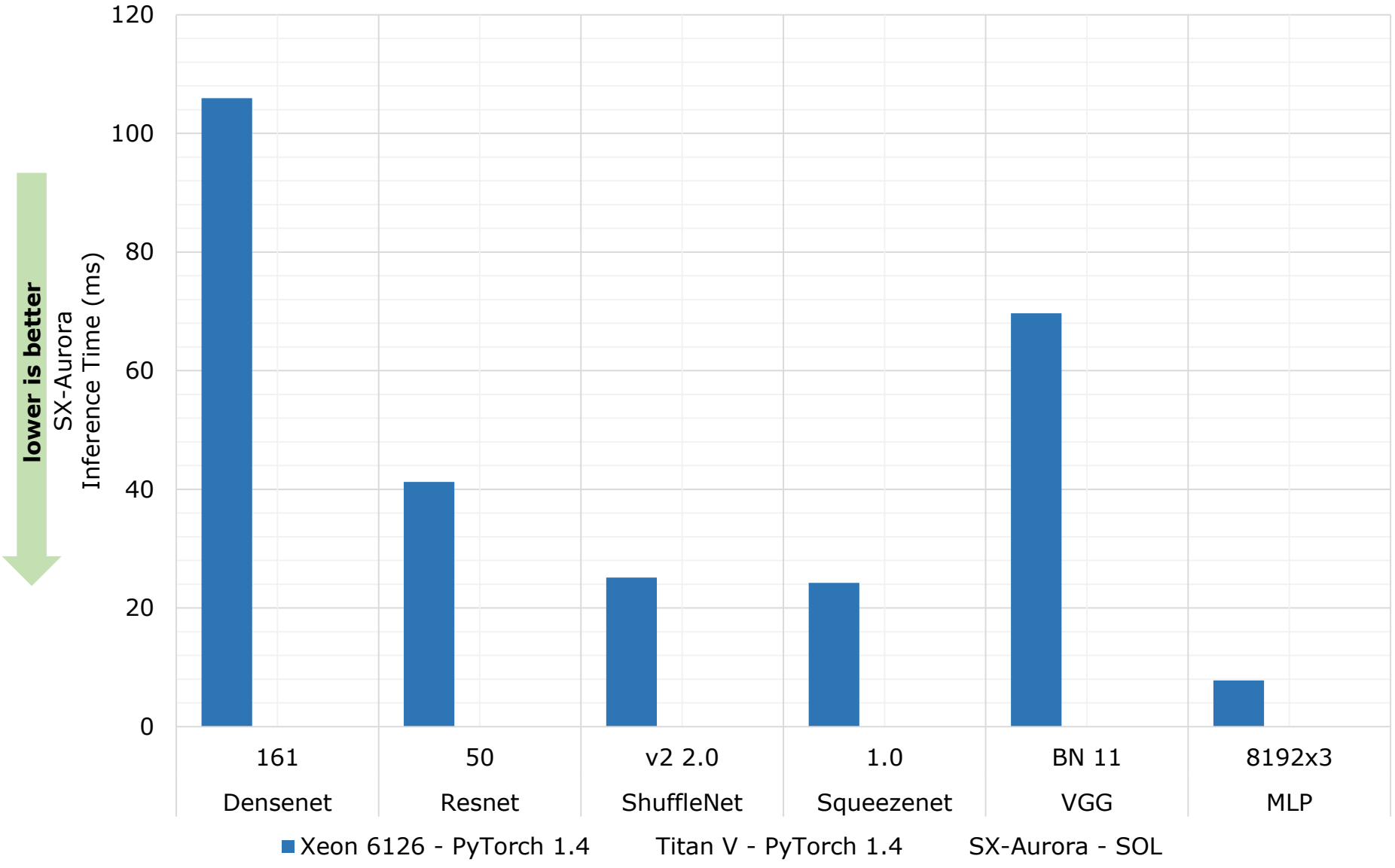
Training Performance (CNN BS=16, MLP BS=64)



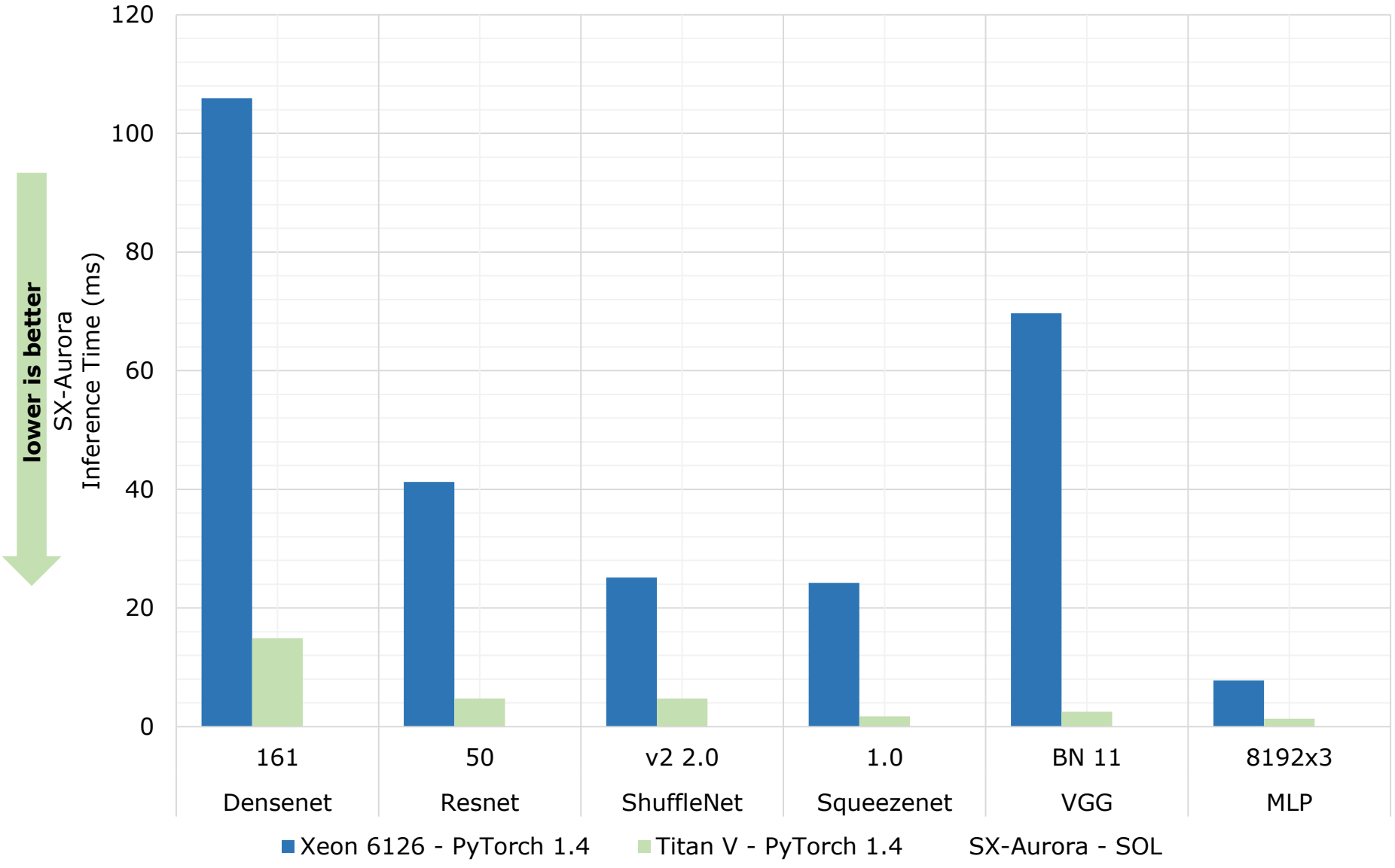
Inference Performance (BS=1)



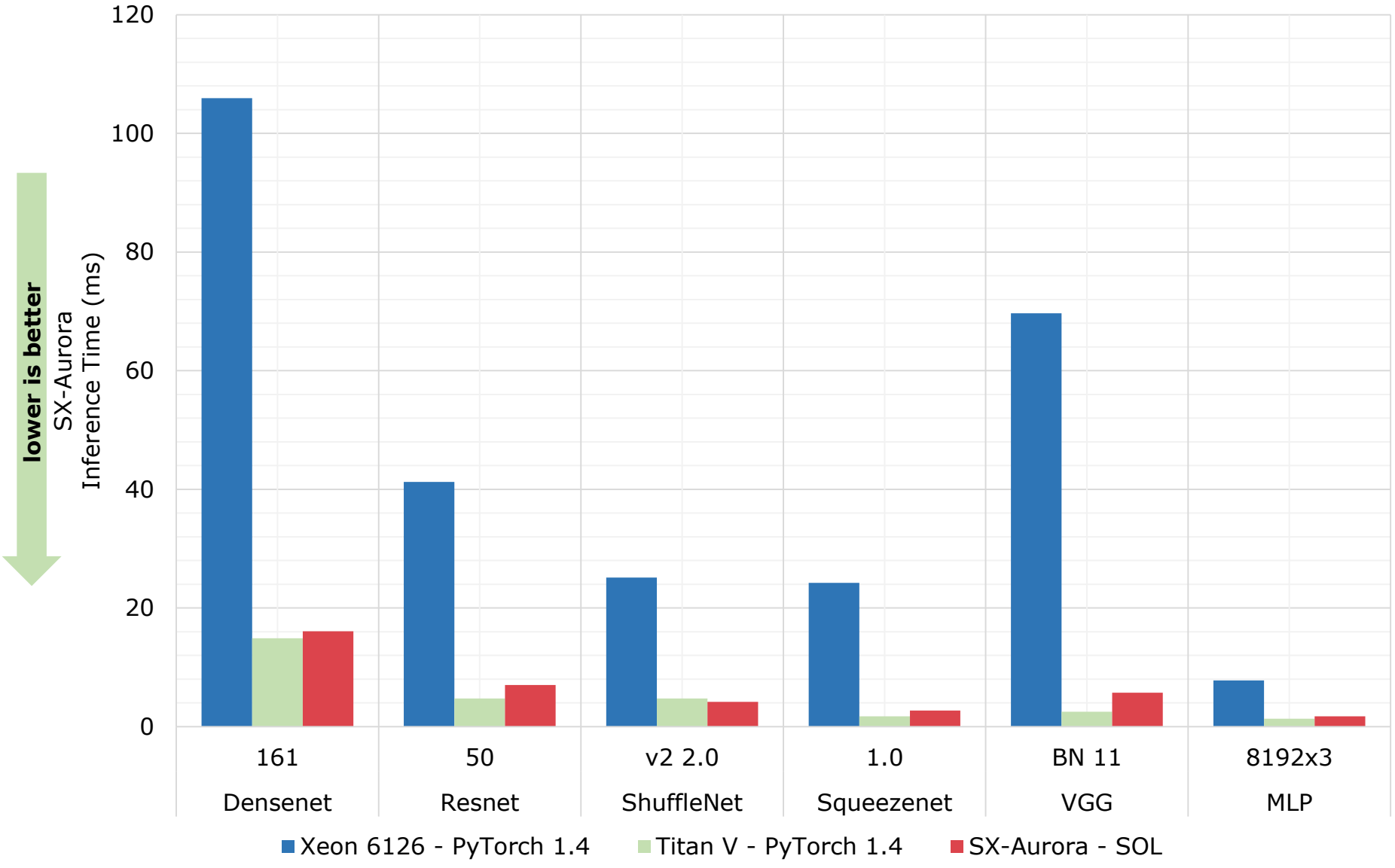
Inference Performance (BS=1)



Inference Performance (BS=1)



Inference Performance (BS=1)



How to use DNN in my own software?

Again, dozen of available tools...

- TF-Lite
- LibTorch
- ONNXRuntime
- OpenVino (only Intel)
- NGraph
- TVM
- TensorRT (only NVIDIA)
- SOL
- …

How to use DNN in my own software?

```
sol.deploy(trained_model, [input],
target=sol.deployment.shared_lib, device=sol.device.ve,
lib_name="MyNetwork", func_name="predict", …)
```

```
#ifndef __MyNetwork__
#define __MyNetwork__

#ifdef __cplusplus
extern "C" {
#endif

void predict_init(const int deviceIdx);
int  predict_seed(const int seed);
void predict     (void* ctx, const float* input, float** output);

#ifdef __cplusplus
}
#endif
#endif
```

SOL RoadMap

Status Quo:

- PyTorch and ONNX
- CNN, MLP, Transformer, …
- Training, Inference, Deployment
- …

2020:

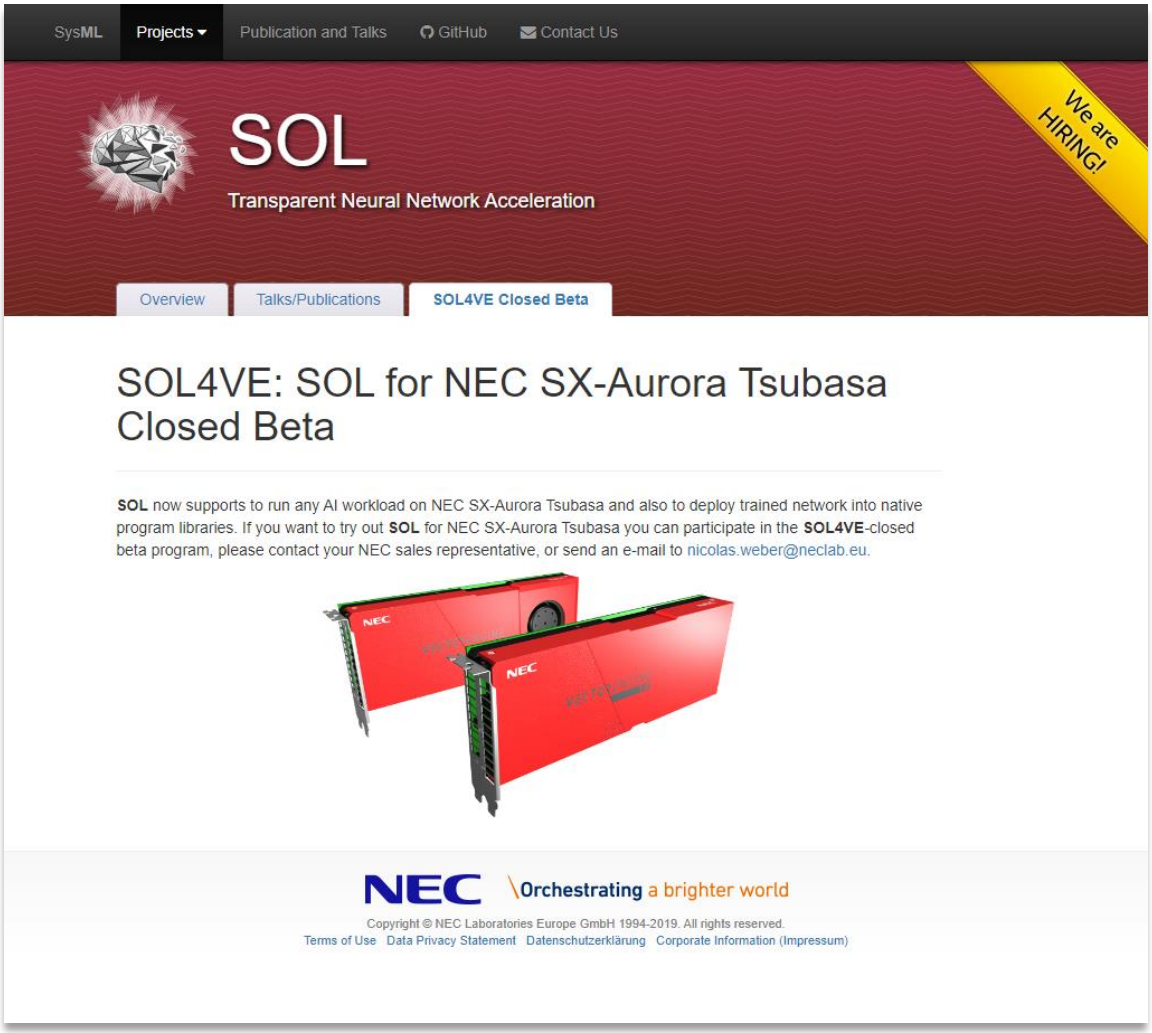
- DL4J (October)
- TensorFlow v2 (December)
- Recurrent Neural Networks (LSTM, GRU)
- torch.nn.DataParallel support for PyTorch

2021:

- Adjustable memory consumption during training (trading memory vs performance)
- User defined Custom Layers
- Algorithmic and internal code optimizations to improve performance
- NumPY support

“Can we try SOL4VE?”

Apply for the SOL4VE closed beta via your NEC Sales representative: **www.sol-project.org**



\Orchestrating a brighter world NEC

Dr. Nicolas Weber
Intelligent Software Systems Group
Senior Software Engineer
NEC Laboratories Europe
nicolas.weber@neclab.eu

www.sol-project.org