Torchy: A Tracing JIT Compiler for PyTorch

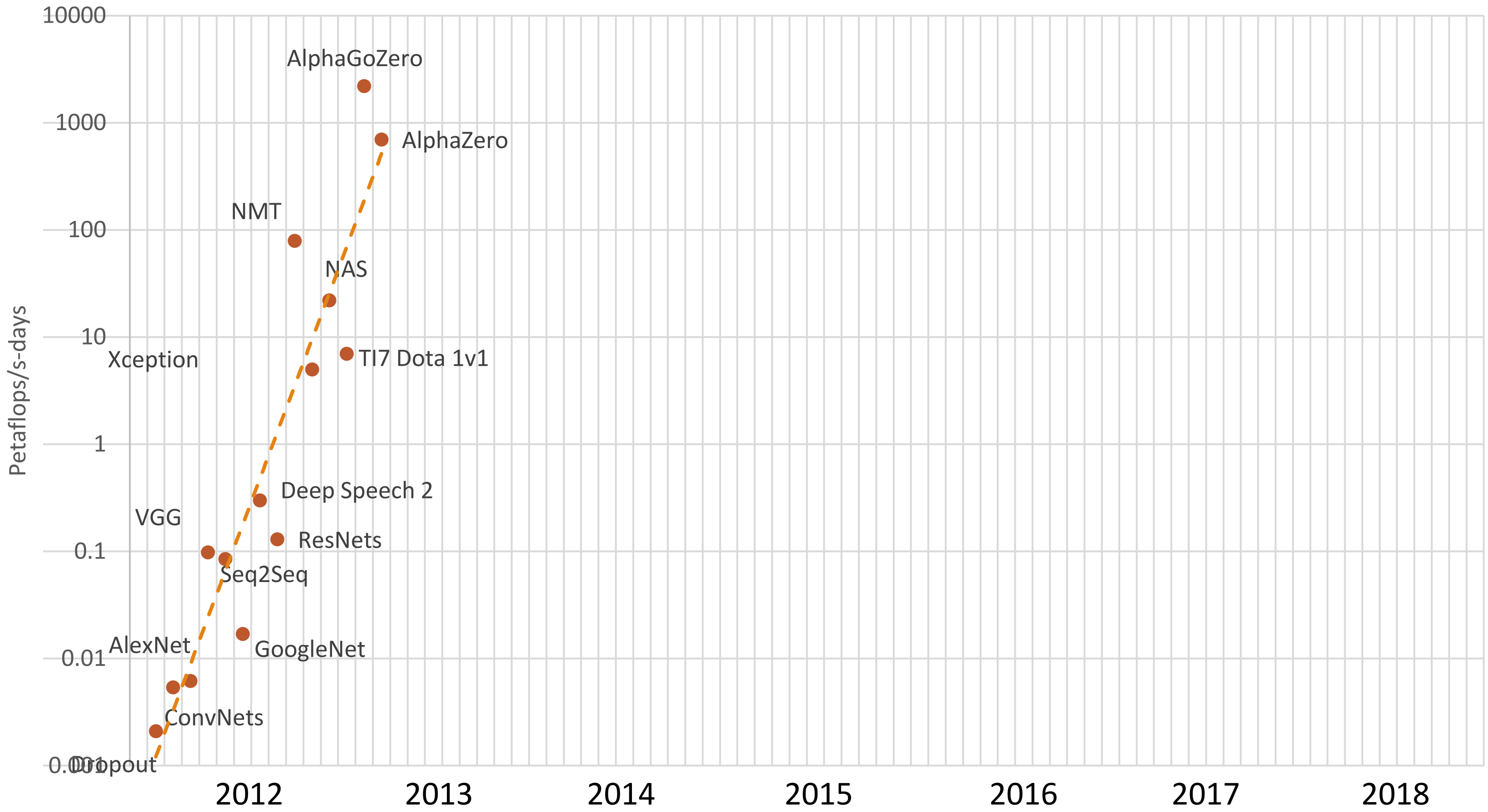
NUNO P. LOPES

UNIVERSITY OF LISBON

Machine Learning: Many useful applications

- Medical imaging
- Traffic prediction in navigation apps
- Text translation
- Grammar checking
- Fraud detection
- SPAM detection
- …

Computing demand for ML is exploding



FLOPS doubling every 3.4 months!

ML models run in frameworks

- First generation
  - Developer assembles model as a data-flow graph first
  - Hard for development & debugging
  - No support for dynamic models
  - TensorFlow 1

- Second generation / aka eager-mode or imperative
  - Instructions executed straight away
  - Easier
  - PyTorch, TensorFlow 2

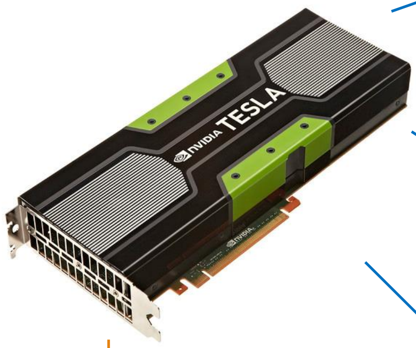
Eager-mode frameworks are amazing!

```
x = torch.tensor(((1.,2.), (3.,4.)))
y = torch.tensor(((5.,6.), (7.,8.)))

z = x.mul(y)
z = z.add(y)
x.add_(z)

print(x)
```

```
tensor([[11., 20.],
        [31., 44.]])
```

Eager-mode frameworks are slow! 🙄

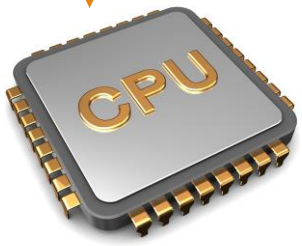
```
x = torch.tensor(((1.,2.), (3.,4.)))
y = torch.tensor(((5.,6.), (7.,8.)))

z = x.mul(y)
z = z.add(y)
x.add_(z)

print(x)
```




```
tensor([[ 5., 12.],
        [21., 32.]])
```

```
tensor([[10., 18.],
        [28., 40.]])
```

```
tensor([[11., 20.],
        [31., 44.]])
```




```
tensor([[11., 20.],
        [31., 44.]])
```

Eager-frameworks “hacks”

```
x = torch.tensor(((1.,2.), (3.,4.)))

z = x.transpose(0, 1)

z[0,0] = 42

print(z)
print(x)
```

```
tensor([[42.,  3.],
        [ 2.,  4.]])
```

```
tensor([[42.,  2.],
        [ 3.,  4.]])
```

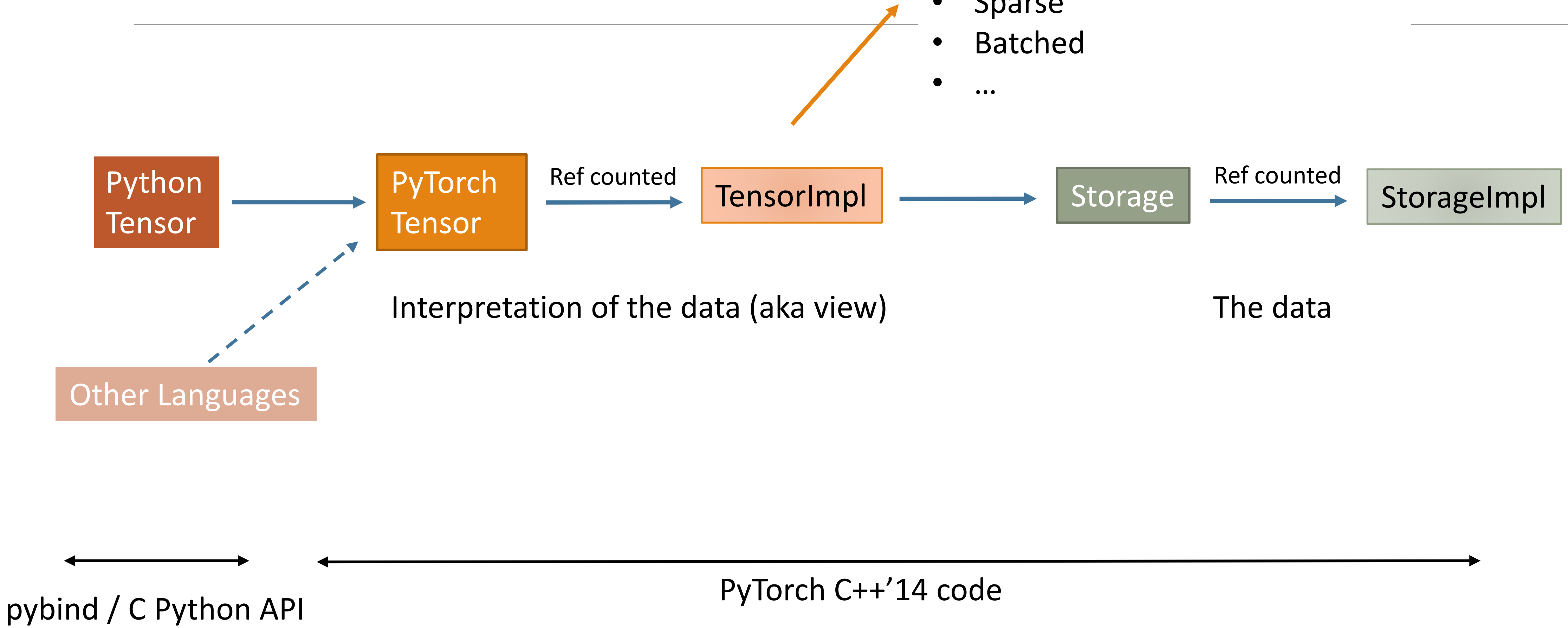
+ ever-increasing list of fused ops that users need to call manually

Transpose fuses marvelously with matmul!

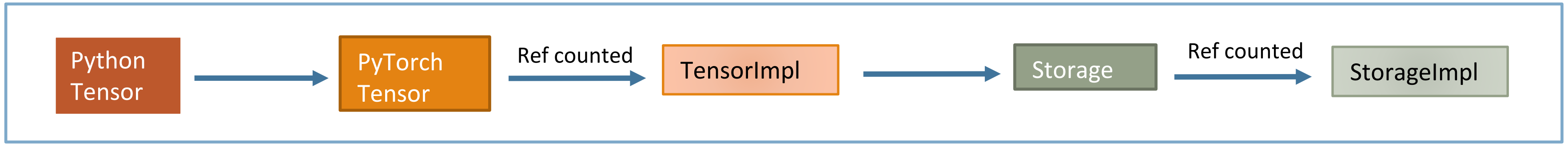
What’s a Tensor?

Multiple tensor implementations:

- Dense
- Sparse
- Batched
- …



Tensor creation




```
x = torch.tensor(((1.,2.), (3.,4.)))
y = torch.tensor(((5.,6.), (7.,8.)))

z = x.mul(y)

x.add_(z)

w = x.to(torch.float, copy=False)

z = x.transpose(0, 1)
```

New Tensor/TensorImpl/Storage/StorageImpl w/ default type & placed on default device

New Tensor/TensorImpl/Storage/StorageImpl w/ same type & device as inputs

Nothing new; override StorageImpl's data

Bump Tensor ref count if types match; new Tensor/Storage otherwise

New Tensor/TensorImpl; bump Storage ref count

Life of a PyTorch function call

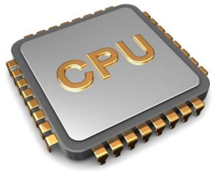
```
z = x.add(y)
```

<u>Dispatch:</u>
Operation = Add.Tensor
Op0 = Tensor, CPU, Float
Op1 = Tensor, CPU, Float

<u>Global dispatcher state:</u>
Default device = CPU
Default type = Float
Include dispatch key = None

Waterfall dispatcher

- VMap
  - Batched
    - Autocast
      - Tracer
        - Autograd
          - Backend Select
            - Devices (CPU, CUDA, etc)



Speeding up PyTorch today

- TorchScript Tracing (`torch.jit.trace`)
- TorchScript Compilation (`torch.jit.script`)
- ORT Module

Single device optimization

- Apex / autocast

Floating-point precision drop

- Deepspeed
- PyTorch's `DistributedDataParallel`
- Megatron (transformers only)
- Model parallelism by hand

Multi-device

TorchScript Tracing

- Function/module is executed (twice) with concrete inputs & operations recorded

```python
def f(x, y):
  z = x.add(y)
  z.add_(x)
  return x.mul(z)
```

```python
w = torch.tensor(…)
z = torch.tensor(…)
torch.jit.trace(f, (w, z))
```

<u>SSA-based IR:</u>

```python
def f(x: Tensor, y: Tensor) -> Tensor:
  z = torch.add(x, y, alpha=1)
  z0 = torch.add_(z, x, alpha=1)
  return torch.mul(x, z0)
```

Tracing input-dependent code

```
def RAdam(wd, N_sma, …):
  if wd != 0:
    p_data_fp32.add_(p_data_fp32, alpha=-wd * lr)

  # more conservative since it's an approximated value
  if N_sma >= 5:
    denom = exp_avg_sq.sqrt().add_(eps)
    p_data_fp32.addcdiv_(exp_avg, denom, value=-step_size)
  else:
    p_data_fp32.add_(exp_avg, alpha=-step_size)
```

- There are 4 possible different traces depending on the input!
- But TorchScript Tracing only supports single-trace functions.

TorchScript Compilation

- Compiler from Python AST to an SSA-based IR (the same used by tracing)
- Supports functions with control-flow
- But no support for too many Python features (only tensor inputs, no lambdas, no union types, etc, etc) – by design!

- Many real codebases are too pythonic. Will never work with TorchScript!

Not speeding up PyTorch today!

- TorchScript Tracing (`torch.jit.trace`)
- TorchScript Compilation (`torch.jit.script`)
- ORT Module

Too restrictive for real-world codebases

- Apex / autocast

Magic floating-point precision drop using a pre-defined list of ops

- Deepspeed
- PyTorch's `DistributedDataParallel`
- Megatron (transformers only)
- Model parallelism by hand

Limited applications or not automatic

Is PyTorch inherently inefficient?

Torchy

A TRACING JIT COMPILER FOR PYTORCH

Most Tensors are not observed

```
w = x.mul(y)
w = w.add(y)
w.add_(x)

print(w)
```

<u>Tensors are only observed:</u>

- Data access, e.g., for branching on data-dependent models
- Printing
- Some PyTorch functions query layout, size, etc for pre-dispatch optimization (a hack)

- Function from 2 tensors to another tensor
- Intermediate values of w not observed

Idea: delay execution until observation

```
x = torch.tensor(((1.,2.), (3.,4.)))
y = torch.tensor(((5.,6.), (7.,8.)))

w = x.mul(y)
w = w.add(y)
x.add_(w)

print(x)
```

Observable event!
Stop tracing and compute

Tracing JIT Compiler

```
w0 = x.mul(y)
w1 = w0.add(y)
x1 = x.add_(w1)
```


```
tensor([[11., 20.],
        [31., 44.]])
```

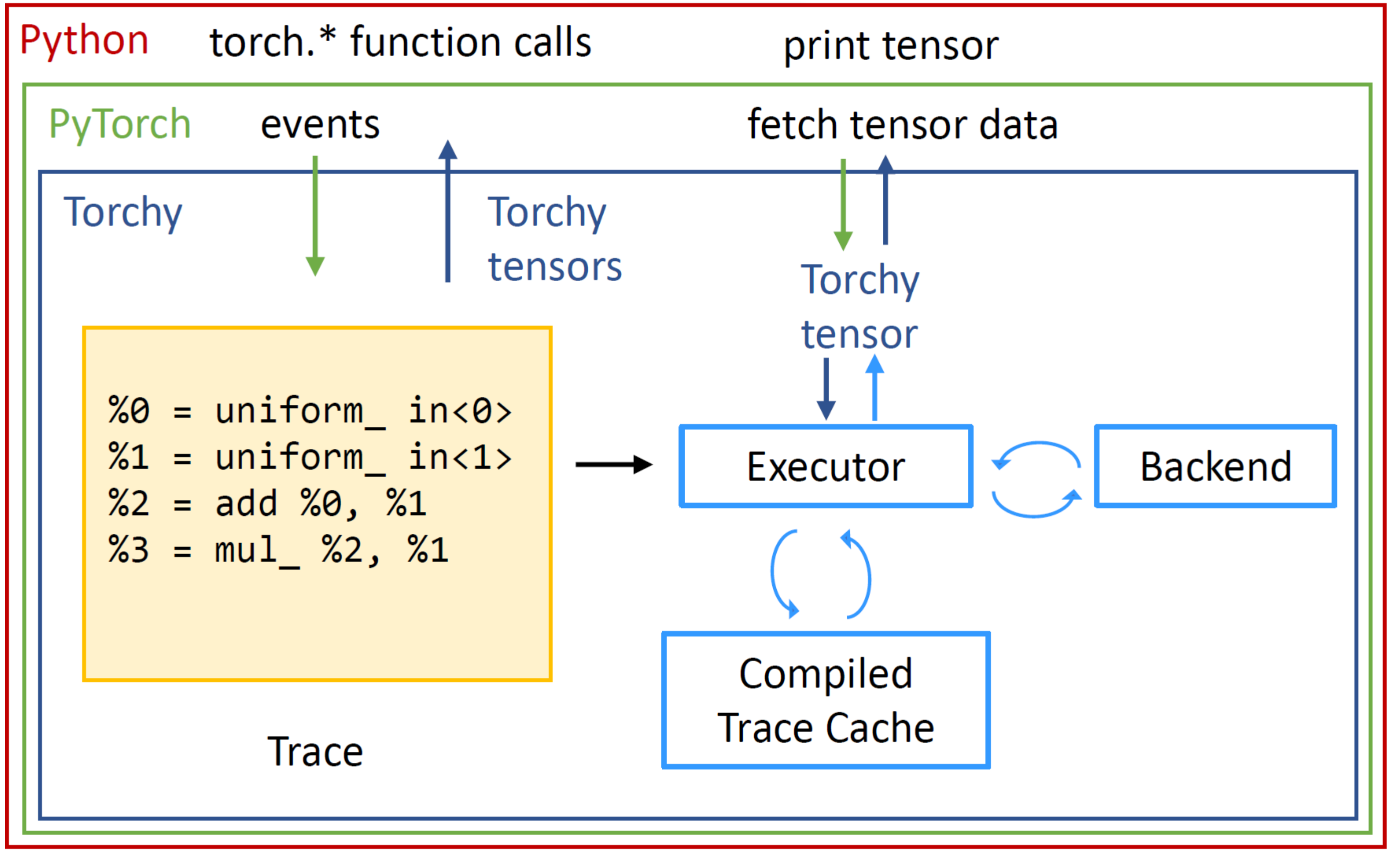
Tracing JIT compilers

- A tremendous success for JavaScript in the past decade
- Peek into the future as execution is delayed
- Detect which tensors are temporaries to help optimization

- Traces can be optimized before execution, or in background
- Traces repeat; optimization cost amortized

- Work with any codebase unmodified!

Torchy

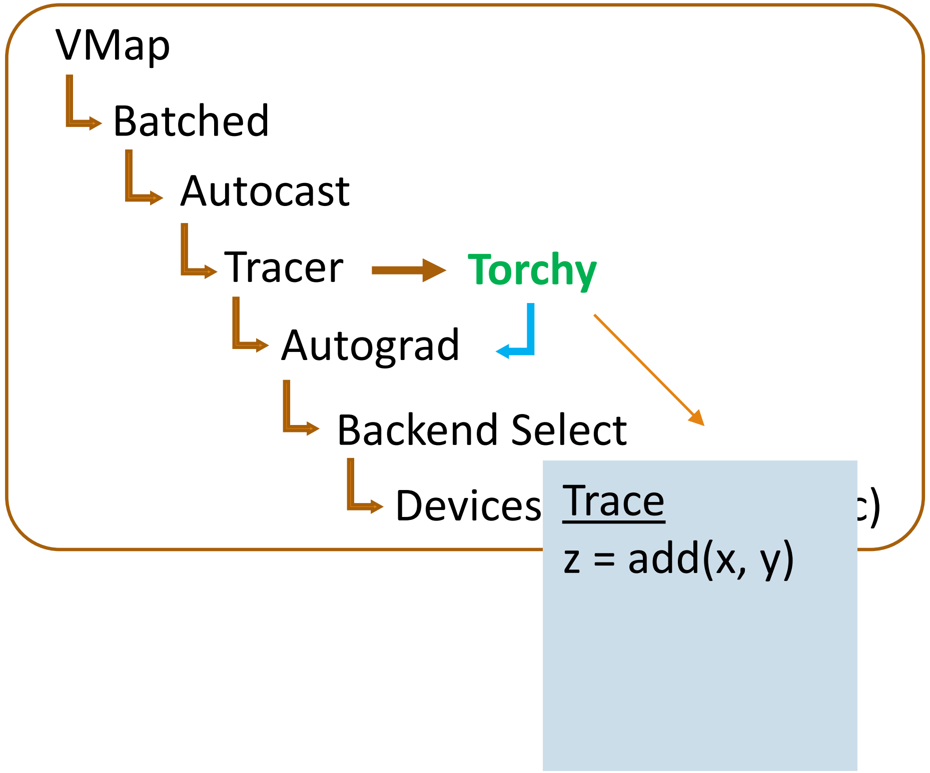


Python
torch.* function calls
print tensor
PyTorch
events
fetch tensor data
Torchy
Torchy
tensors
Torchy
tensor
%0 = uniform_ in<0>
%1 = uniform_ in<1>
%2 = add %0, %1
%3 = mul_ %2, %1
Executor
Backend
Compiled
Trace Cache
Trace

Intercepting PyTorch function calls

```
z = x.add(y)
```

Waterfall dispatcher



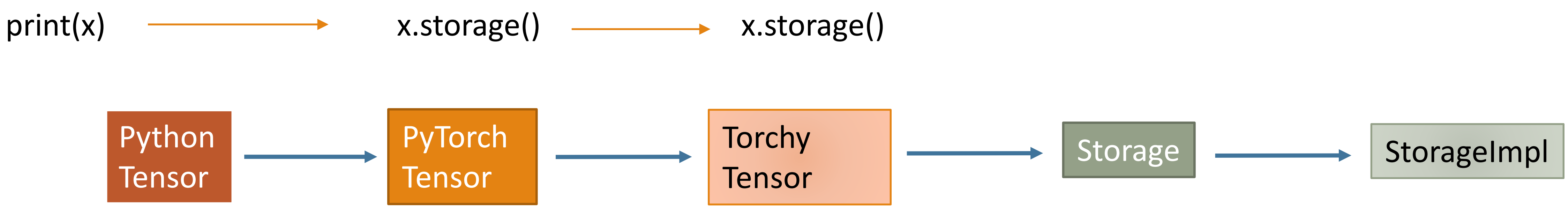
<u>Dispatch:</u>
Operation = Add.Tensor
Op0 = Tensor, CPU, Float
Op1 = Tensor, CPU, Float

<u>Global dispatcher state:</u>
Default device = CPU
Default type = Float
Include dispatch key = Torchy

```
import torchy
torchy.enable()
```

<u>Alternative backends:</u>
- Glow
- MLIR
- ONNX Runtime (ORT)
- TorchScript
- XLA

Intercepting non-dispatched events



Is tensor materialized?

- Yes: behave like a normal tensor
- No: flush trace & act normally

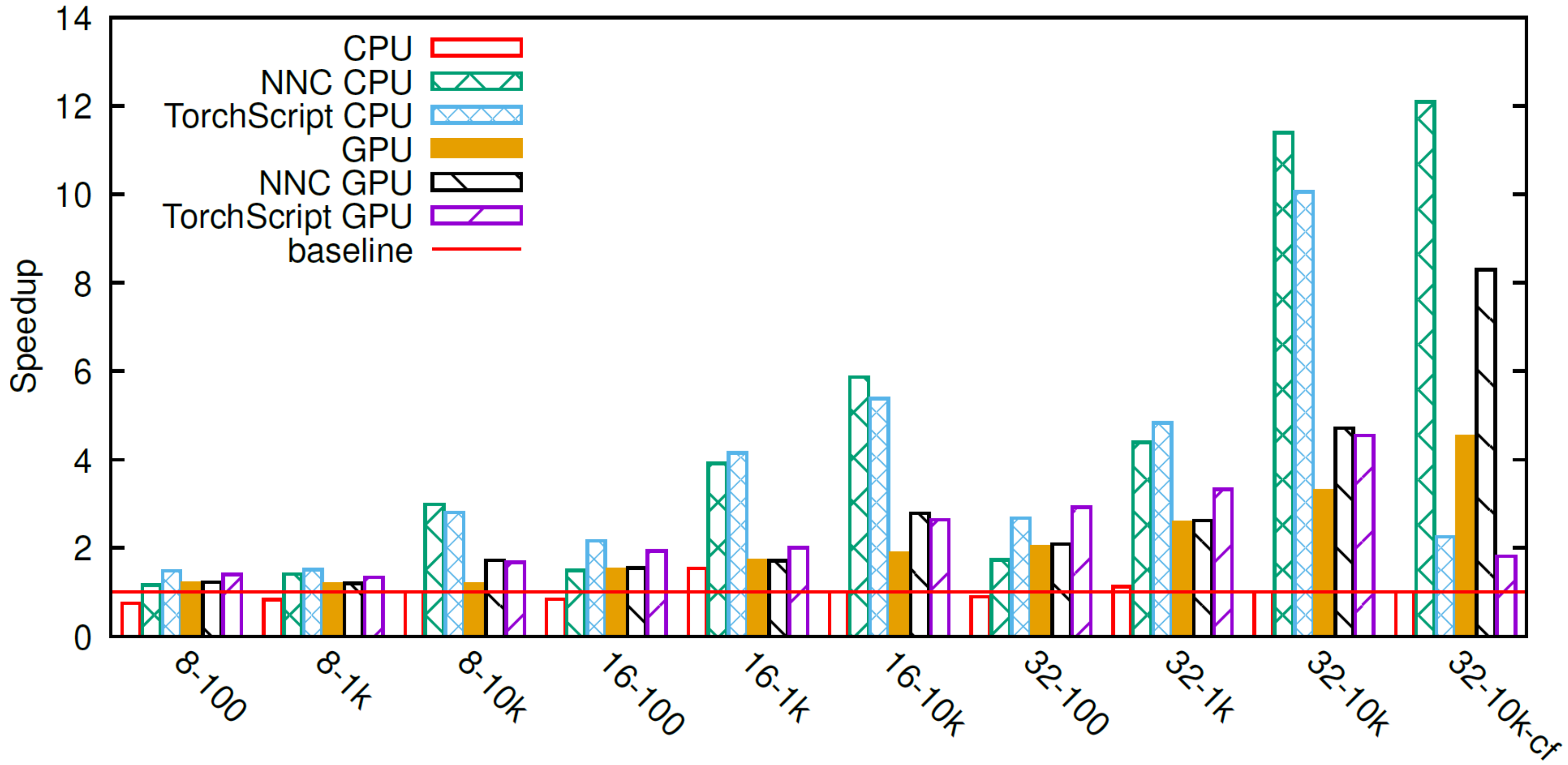
Why a new tracing JIT vs improving TorchScript?

- Single dataflow graph doesn’t work for dynamic models
- Run-time statistics enable new features & optimizations
- Trace across function boundaries

Microbenchmarks

- Code with 8, 16, 32 elementwise operations
- Square matrices, n=100, 1k, 10k
- Straight-line code & with control-flow

Microbenchmarks: results

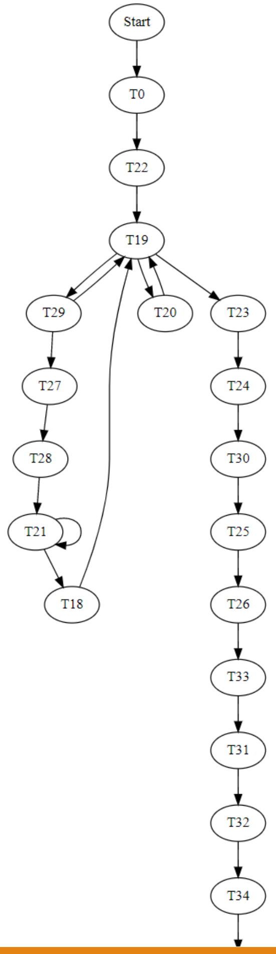


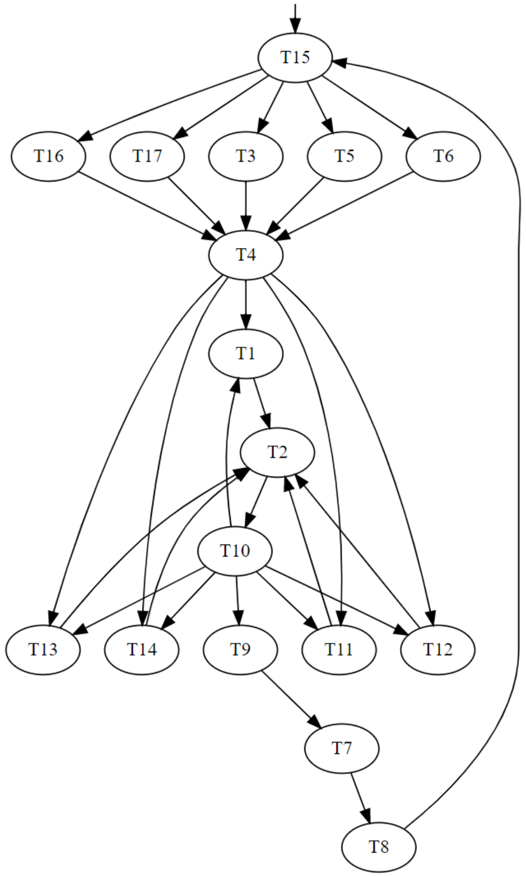
Early experiments

- Run 1,000 inference queries over:
  - TorchVision: ResNet-18, ResNeXt, MobileNet v3 Large
  - Hungging Face: Bert Base/Large, GPT-2, RoBERTa Large
- PyTorch 1.9+
- 12 CPU cores

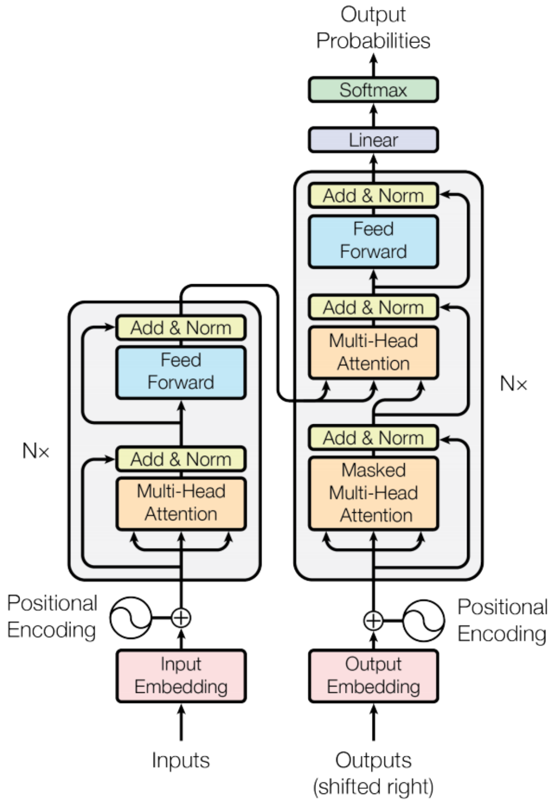
Bert from the compiler's perspective

Weight initialization

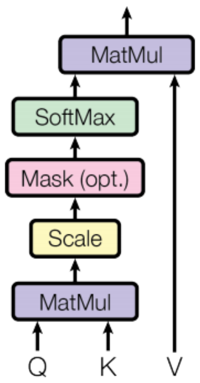




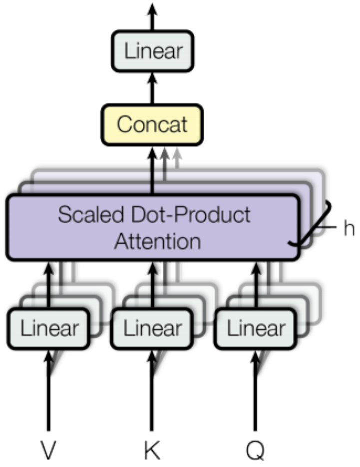
Model

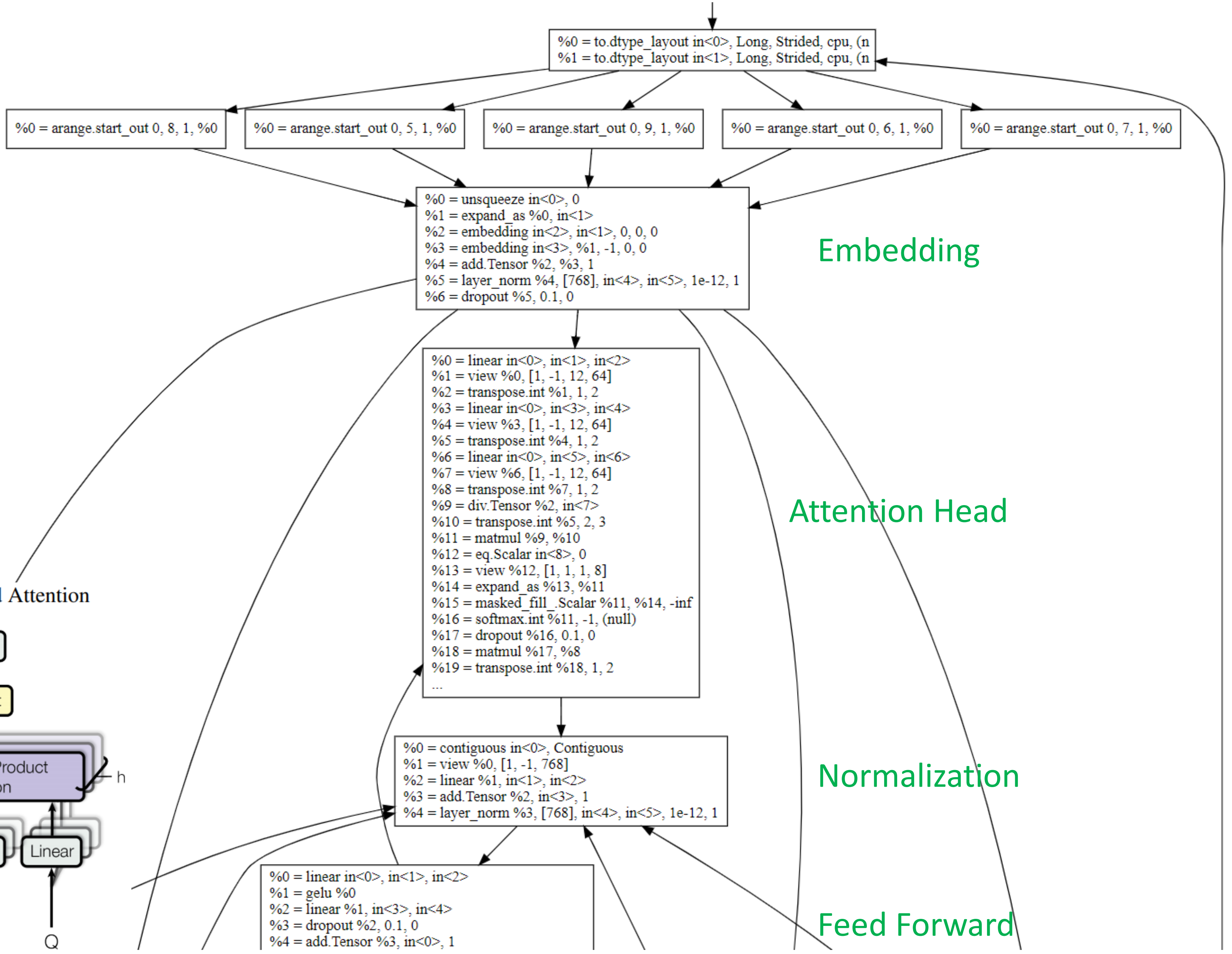


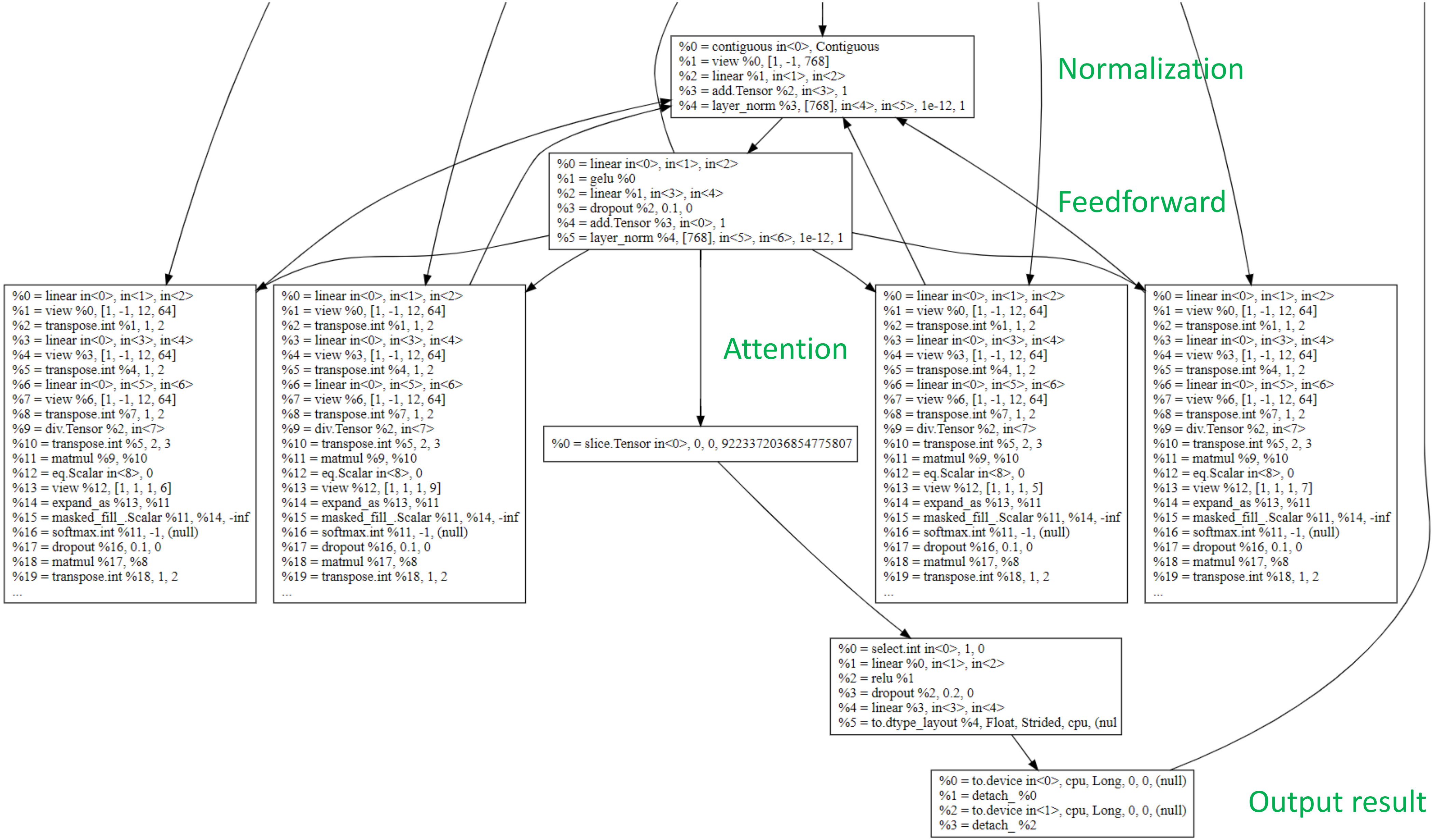
Scaled Dot-Product Attention



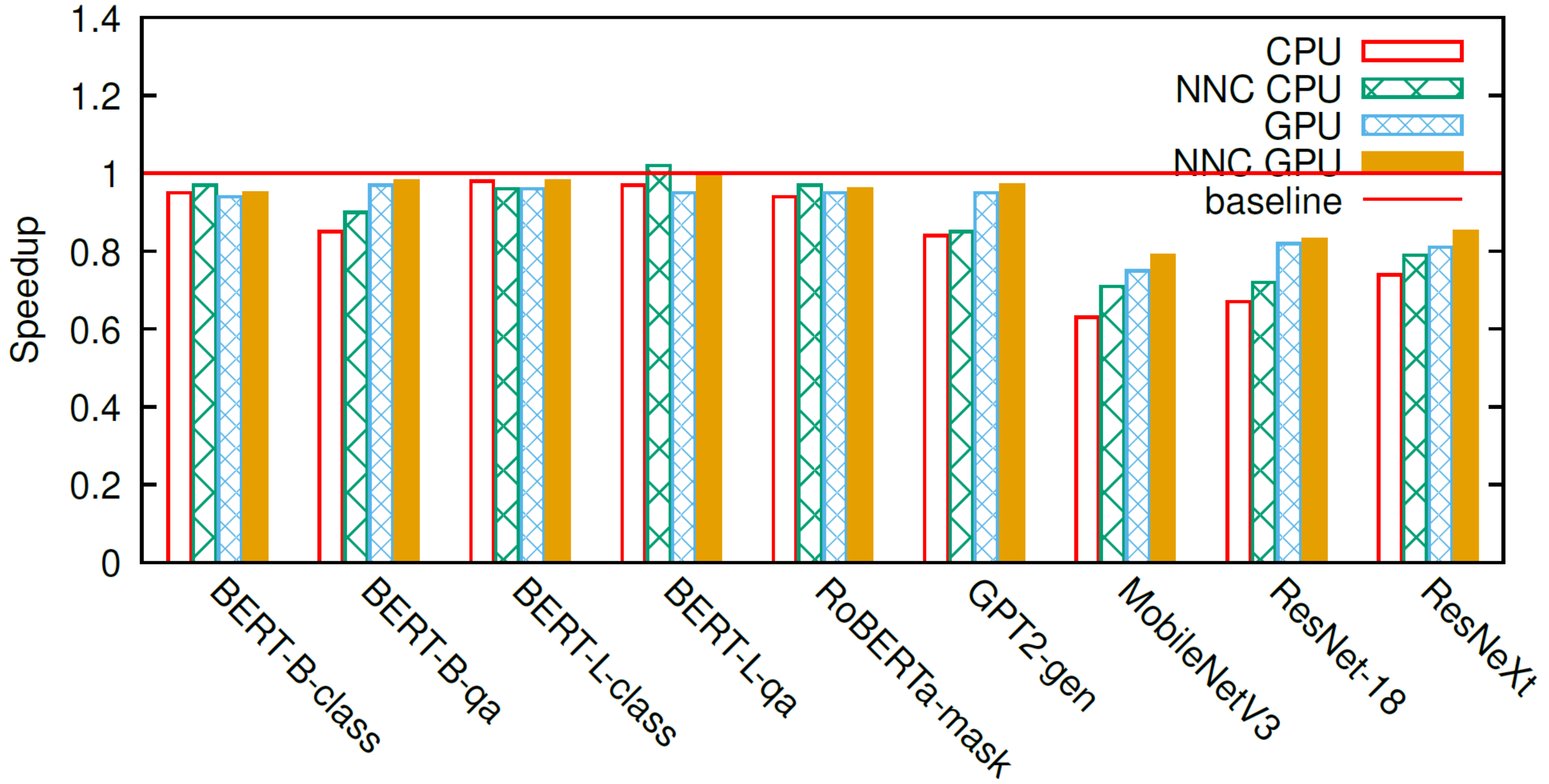
Multi-Head Attention






Normalization
%0 = contiguous in<0>, Contiguous
%1 = view %0, [1, -1, 768]
%2 = linear %1, in<1>, in<2>
%3 = add.Tensor %2, in<3>, 1
%4 = layer_norm %3, [768], in<4>, in<5>, 1e-12, 1
Feedforward
%0 = linear in<0>, in<1>, in<2>
%1 = gelu %0
%2 = linear %1, in<3>, in<4>
%3 = dropout %2, 0.1, 0
%4 = add.Tensor %3, in<0>, 1
%5 = layer_norm %4, [768], in<5>, in<6>, 1e-12, 1
Attention
%0 = linear in<0>, in<1>, in<2>
%1 = view %0, [1, -1, 12, 64]
%2 = transpose.int %1, 1, 2
%3 = linear in<0>, in<3>, in<4>
%4 = view %3, [1, -1, 12, 64]
%5 = transpose.int %4, 1, 2
%6 = linear in<0>, in<5>, in<6>
%7 = view %6, [1, -1, 12, 64]
%8 = transpose.int %7, 1, 2
%9 = div.Tensor %2, in<7>
%10 = transpose.int %5, 2, 3
%11 = matmul %9, %10
%12 = eq.Scalar in<8>, 0
%13 = view %12, [1, 1, 1, 6]
%14 = expand_as %13, %11
%15 = masked_fill_.Scalar %11, %14, -inf
%16 = softmax.int %11, -1, (null)
%17 = dropout %16, 0.1, 0
%18 = matmul %17, %8
%19 = transpose.int %18, 1, 2
...
%0 = linear in<0>, in<1>, in<2>
%1 = view %0, [1, -1, 12, 64]
%2 = transpose.int %1, 1, 2
%3 = linear in<0>, in<3>, in<4>
%4 = view %3, [1, -1, 12, 64]
%5 = transpose.int %4, 1, 2
%6 = linear in<0>, in<5>, in<6>
%7 = view %6, [1, -1, 12, 64]
%8 = transpose.int %7, 1, 2
%9 = div.Tensor %2, in<7>
%10 = transpose.int %5, 2, 3
%11 = matmul %9, %10
%12 = eq.Scalar in<8>, 0
%13 = view %12, [1, 1, 1, 9]
%14 = expand_as %13, %11
%15 = masked_fill_.Scalar %11, %14, -inf
%16 = softmax.int %11, -1, (null)
%17 = dropout %16, 0.1, 0
%18 = matmul %17, %8
%19 = transpose.int %18, 1, 2
...
%0 = slice.Tensor in<0>, 0, 0, 9223372036854775807
%0 = linear in<0>, in<1>, in<2>
%1 = view %0, [1, -1, 12, 64]
%2 = transpose.int %1, 1, 2
%3 = linear in<0>, in<3>, in<4>
%4 = view %3, [1, -1, 12, 64]
%5 = transpose.int %4, 1, 2
%6 = linear in<0>, in<5>, in<6>
%7 = view %6, [1, -1, 12, 64]
%8 = transpose.int %7, 1, 2
%9 = div.Tensor %2, in<7>
%10 = transpose.int %5, 2, 3
%11 = matmul %9, %10
%12 = eq.Scalar in<8>, 0
%13 = view %12, [1, 1, 1, 5]
%14 = expand_as %13, %11
%15 = masked_fill_.Scalar %11, %14, -inf
%16 = softmax.int %11, -1, (null)
%17 = dropout %16, 0.1, 0
%18 = matmul %17, %8
%19 = transpose.int %18, 1, 2
...
%0 = linear in<0>, in<1>, in<2>
%1 = view %0, [1, -1, 12, 64]
%2 = transpose.int %1, 1, 2
%3 = linear in<0>, in<3>, in<4>
%4 = view %3, [1, -1, 12, 64]
%5 = transpose.int %4, 1, 2
%6 = linear in<0>, in<5>, in<6>
%7 = view %6, [1, -1, 12, 64]
%8 = transpose.int %7, 1, 2
%9 = div.Tensor %2, in<7>
%10 = transpose.int %5, 2, 3
%11 = matmul %9, %10
%12 = eq.Scalar in<8>, 0
%13 = view %12, [1, 1, 1, 7]
%14 = expand_as %13, %11
%15 = masked_fill_.Scalar %11, %14, -inf
%16 = softmax.int %11, -1, (null)
%17 = dropout %16, 0.1, 0
%18 = matmul %17, %8
%19 = transpose.int %18, 1, 2
...
%0 = select.int in<0>, 1, 0
%1 = linear %0, in<1>, in<2>
%2 = relu %1
%3 = dropout %2, 0.2, 0
%4 = linear %3, in<3>, in<4>
%5 = to.dtype_layout %4, Float, Strided, cpu, (nul
%0 = to.device in<0>, cpu, Long, 0, 0, (null)
%1 = detach_ %0
%2 = to.device in<1>, cpu, Long, 0, 0, (null)
%3 = detach_ %2
Output result

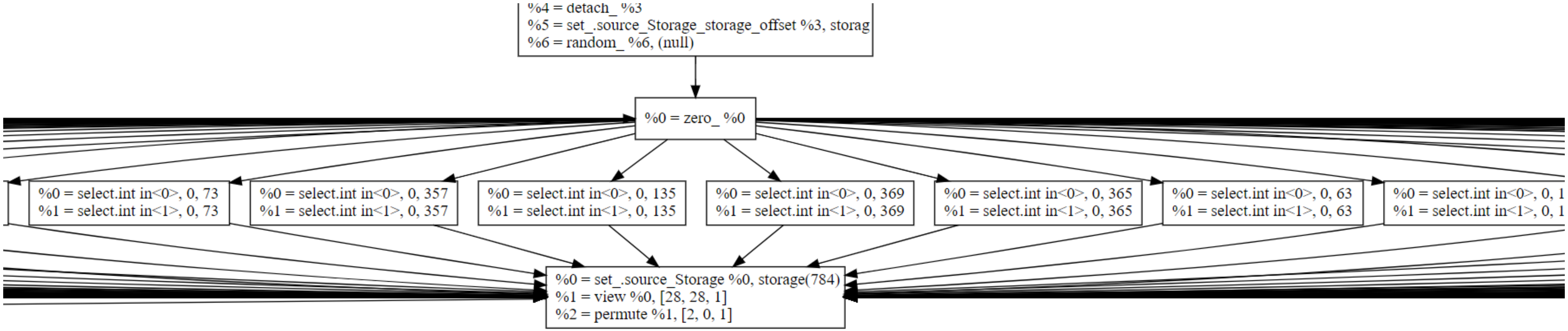
Early Results



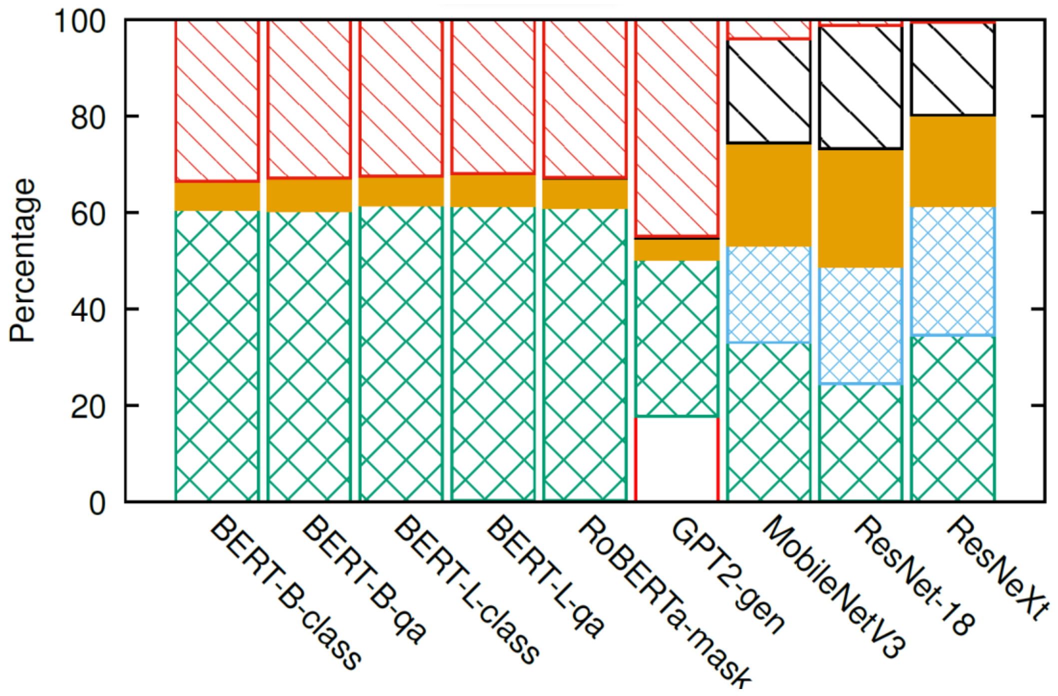
How many traces?

Model	Unique Traces wo/ inference
BERT Base	87
BERT Large	87
RoBERTa Large	87
GPT-2	246
ResNet-18	1048
ResNeXt	1065
MobileNet v3 large	1124

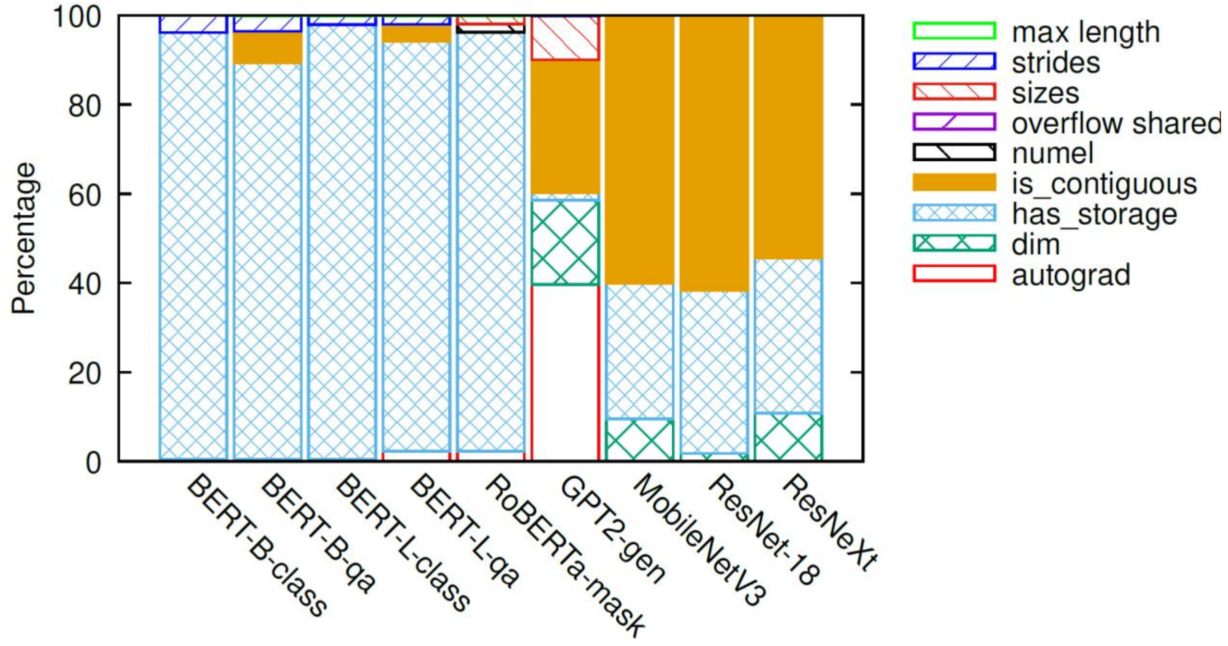
ResNet trace explosion



Flush Reasons

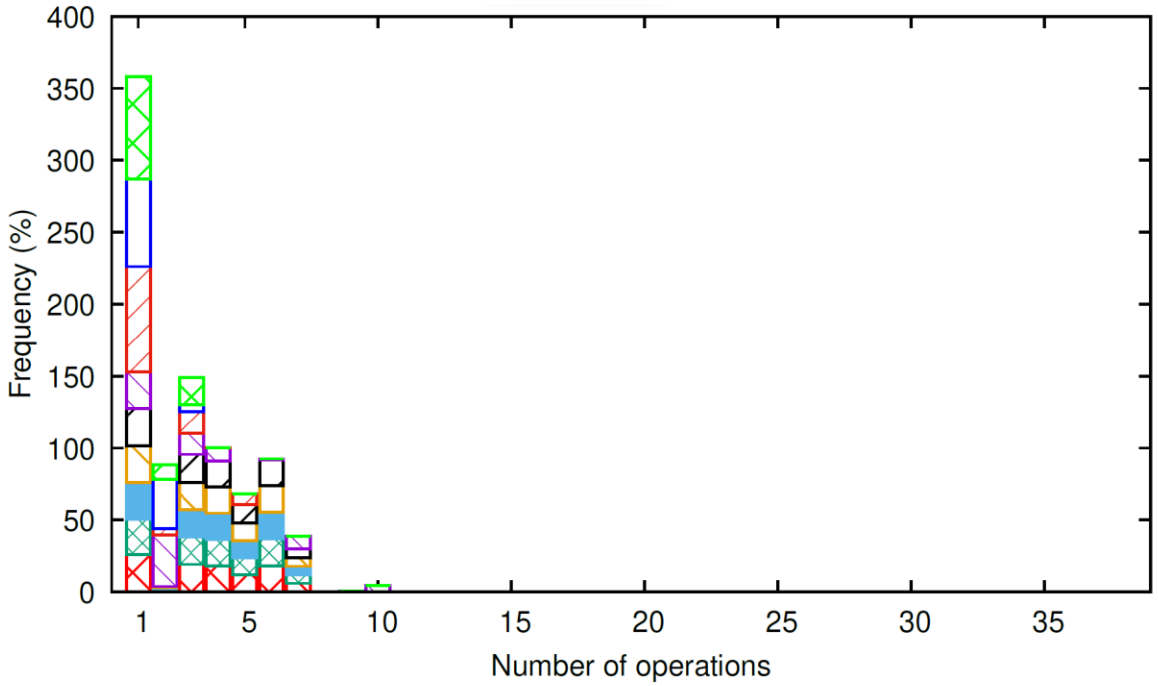


Without shape inference

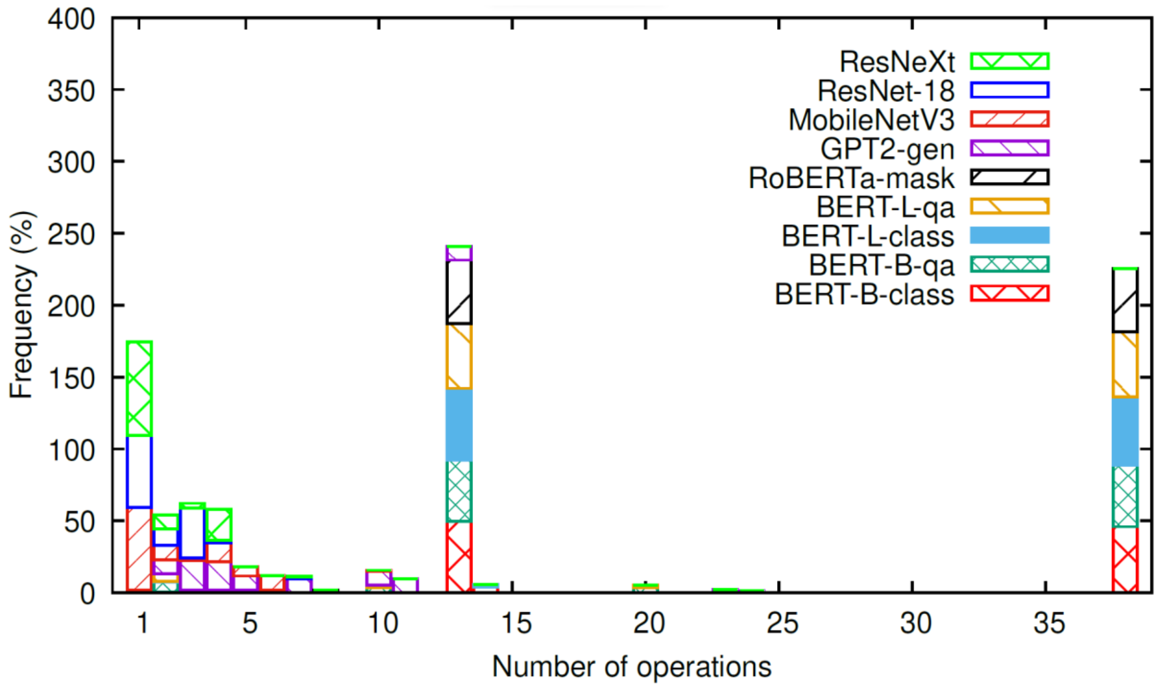


With shape inference (partial)

Trace sizes



Without shape inference



With shape inference (partial)

Summary: Torchy

- Acceleration for dynamic PyTorch programs through JIT compilation
- Converts programs into small-ish straight-line programs (traces)
- Optimizes and runs each trace with the best backend
- Zero code changes! Just ‘pip install torchy’