TVM at Facebook

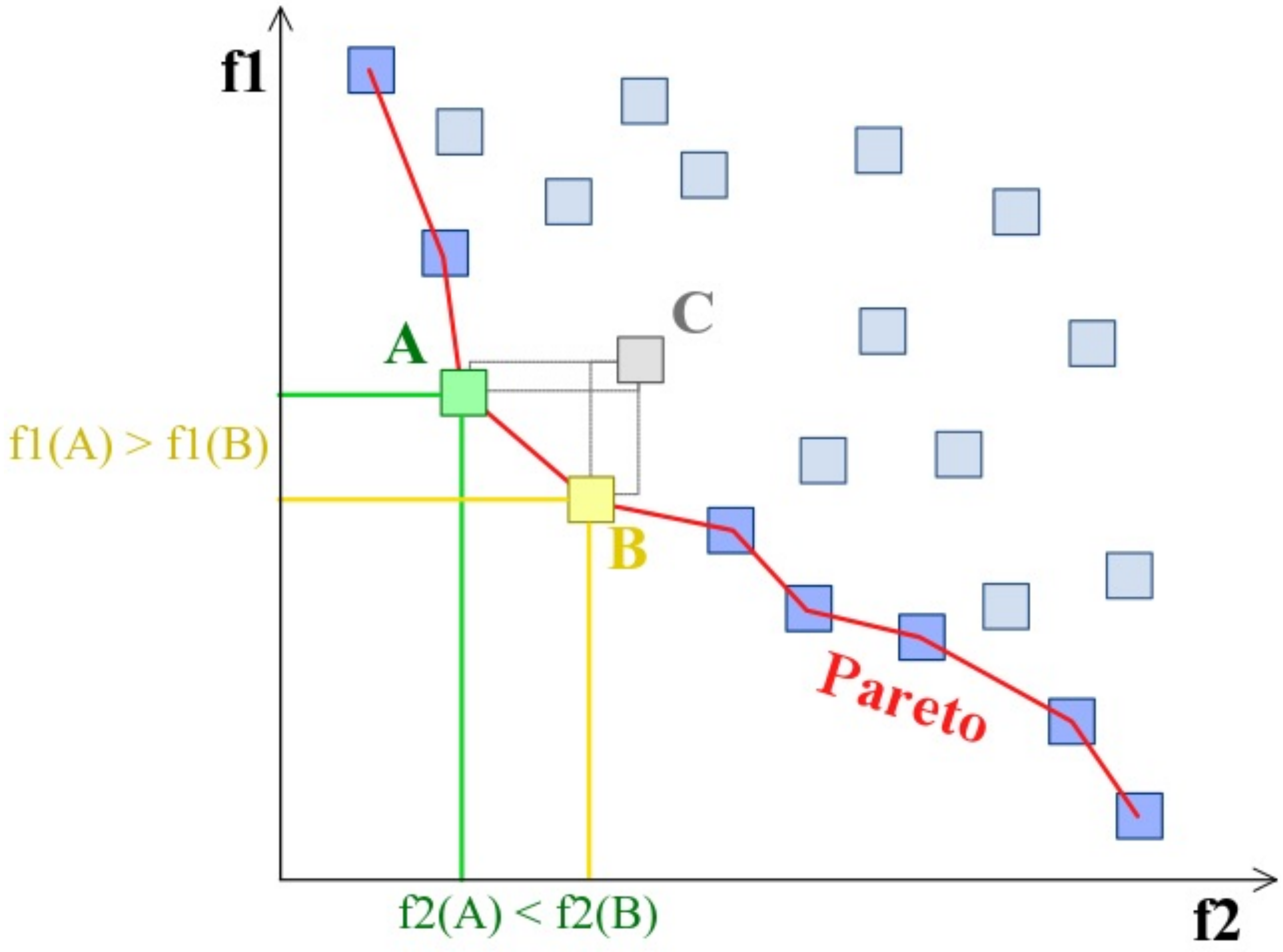
Lots of contributors at FB and elsewhere

TVM at Facebook

- Why TVM?
- Examples from Speech Synthesis
- Sparsity
- PyTorch

Why TVM for ML Systems?

- **Performance** matters
- **Flexibility** matters
- **Portability** matters



ML Systems at Facebook

- Heterogenous computing environment (CPU, GPU, Mobile, Accelerators, ...)
- Wide variety of workloads
- Rapidly increasing set of primitives
  - (over 500 in PyTorch alone)
- Exponential set of fusions
- Need **generalized** performance
- Need **flexibility** for new models

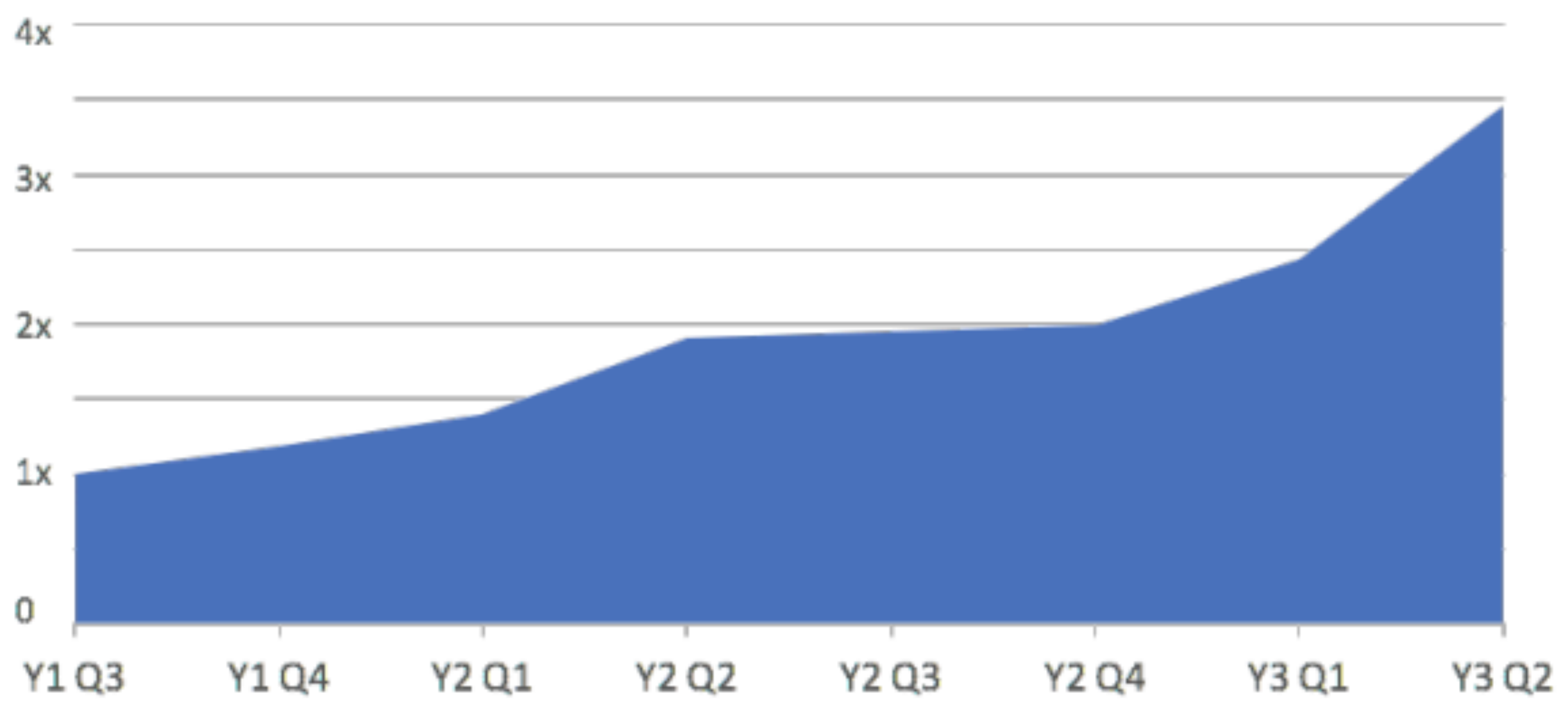


Figure 1: Server demand for DL inference across data centers

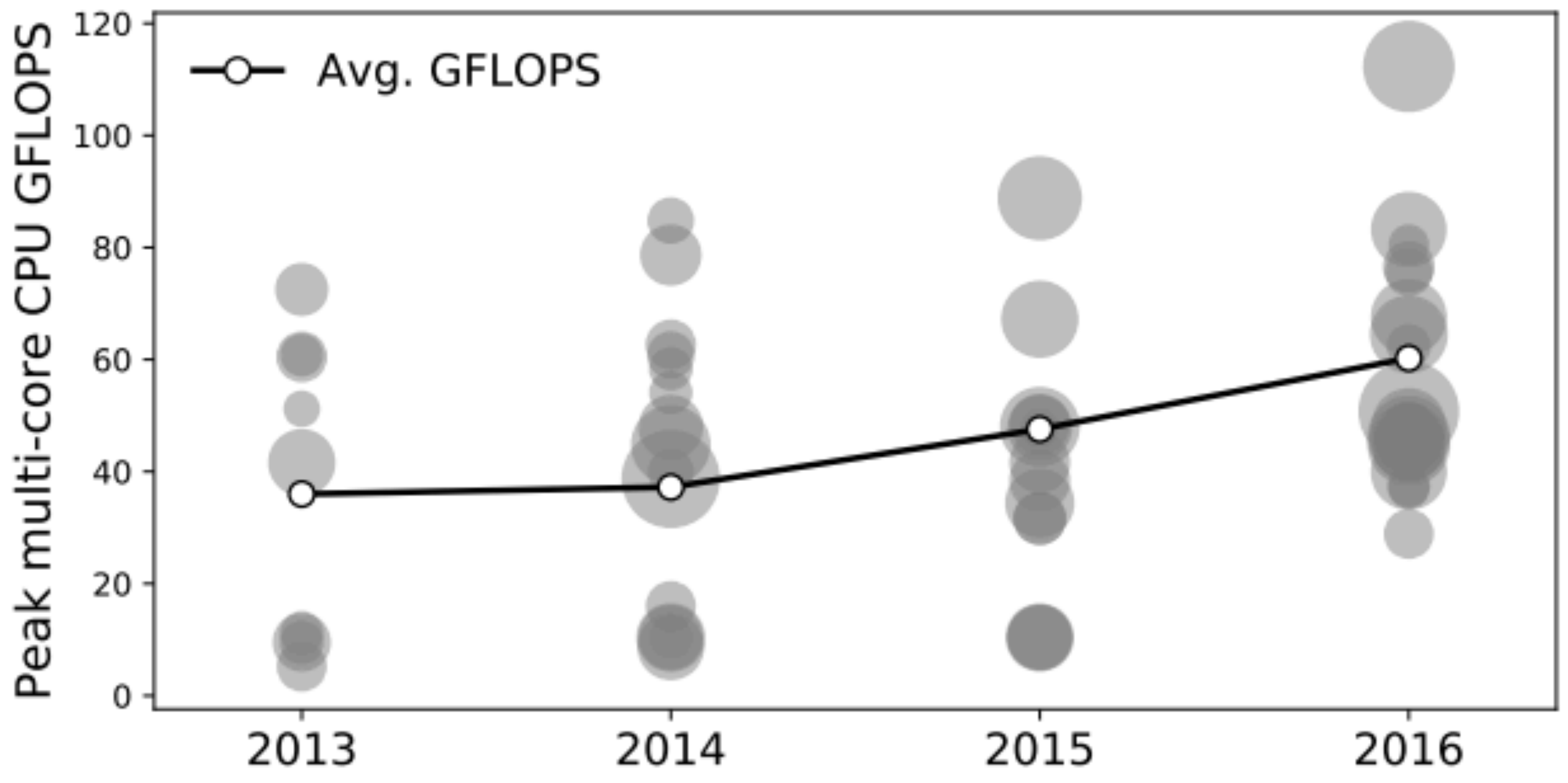


Figure 1: The distribution of peak performance of smartphone SoCs running Facebook

Speech Synthesis with RNNs

- Huge progress since WaveNet (2016)
- SOTA with neural **autoregressive** models
- Very challenging from systems perspective
- **Sequential dependency** structure
- **Very high** sample rates (e.g 48kHz)

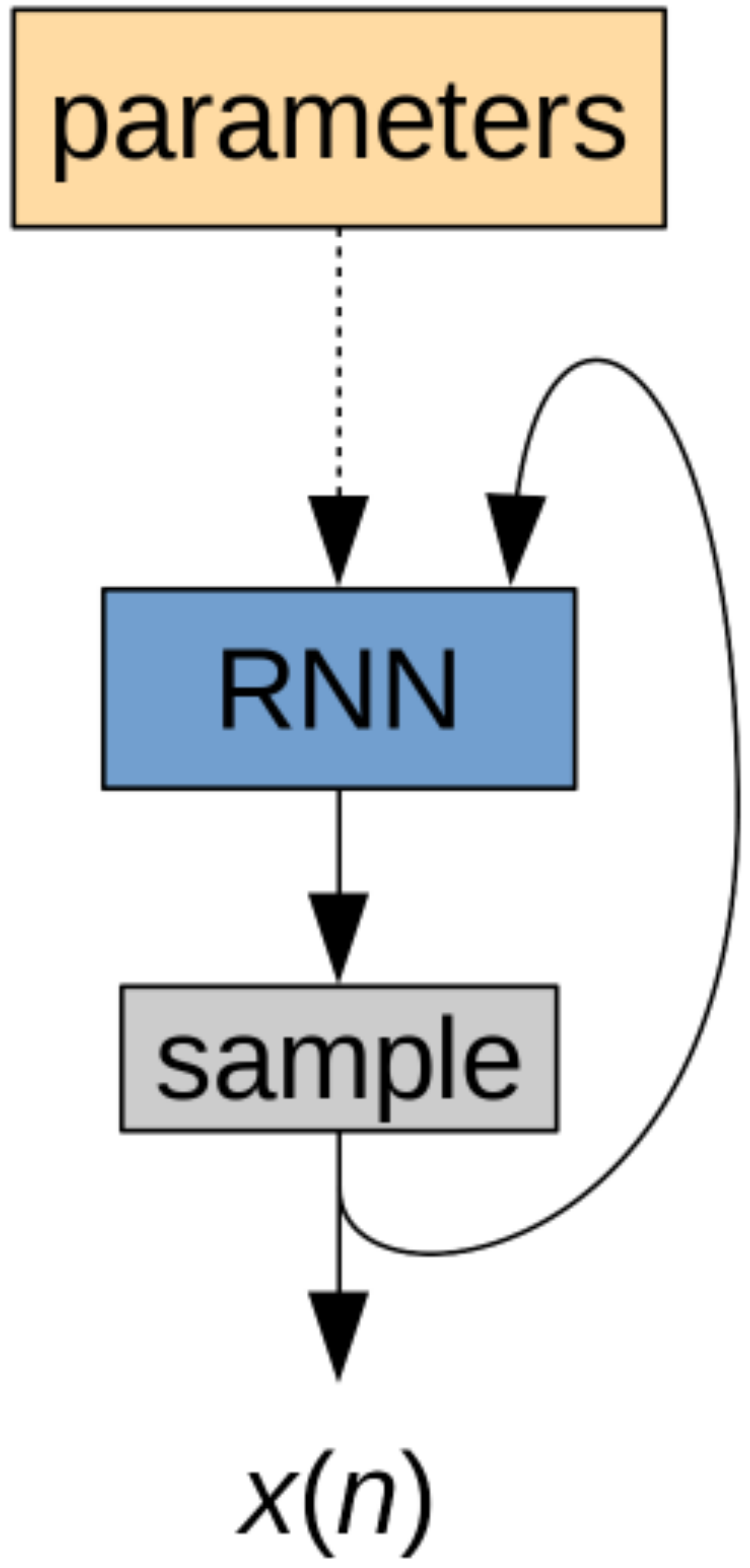


Image from LPCNet

TVM for Speech Synthesis

- WaveRNN-style model architecture

- Compute dominated by GRU and FC layers

- 24kHz sampling frequency requires **40us** sampling net runtime

- Initial model with **3,400us** sampling net runtime

- **85x slower than target**

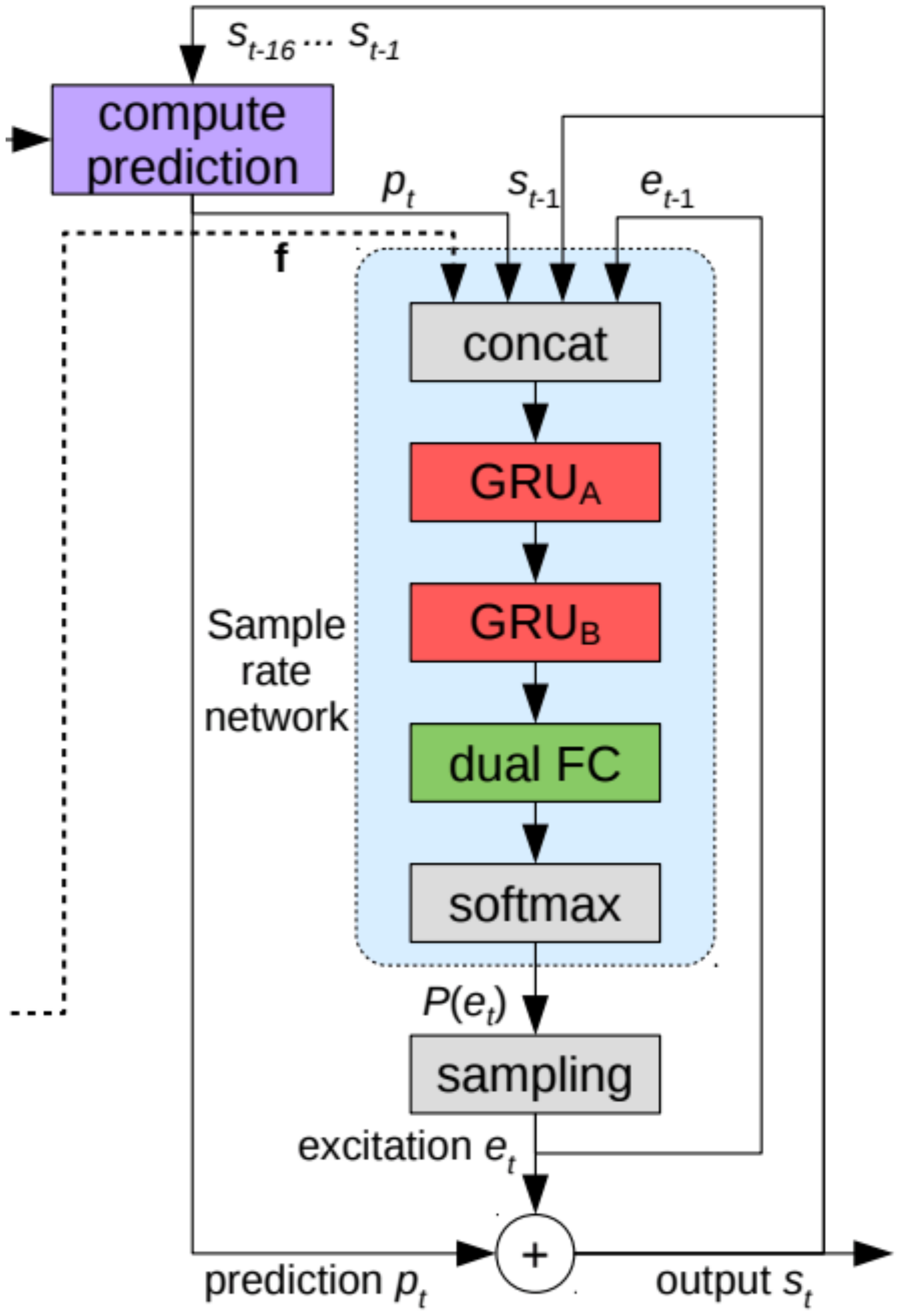


Image from LPCNet

TVM for low-hanging fruit

- Per-operator **framework overhead** (1-2us) means interpreter is infeasible
- Eliminate framework operator overhead via whole-graph compilation
- Substantial improvements for **memory-bound operations** (GEMV, elementwise)
- Still not enough...

```
fn (%X: Tensor[(1, 10), float32],
    %y: Tensor[(30, 10), float32])
    -> Tensor[(1, 10), float32] {
  %0 = nn.dense(%X, %y, units=None)
  %1 = split(%0, indices_or_sections=int64(3), axis=1)
  %2 = %1.0
  %3 = sigmoid(%2)
  %4 = %1.1
  %5 = tanh(%4)
  %6 = %1.2
  %7 = exp(%6)
  %8 = multiply(%5, %7)
  %9 = add(%3, %8)
  %9
}
```

TVM for block-sparse kernels

- Need to reduce FLOPs significantly
- Need to reduce cache footprint
- Introduce block-sparsity in dense layers
  - cf WaveRNN, Sparse Transformers, etc
- Reduce storage footprint with int8/float16
- Substantial latency reduction
- Enables more aggressive fusion

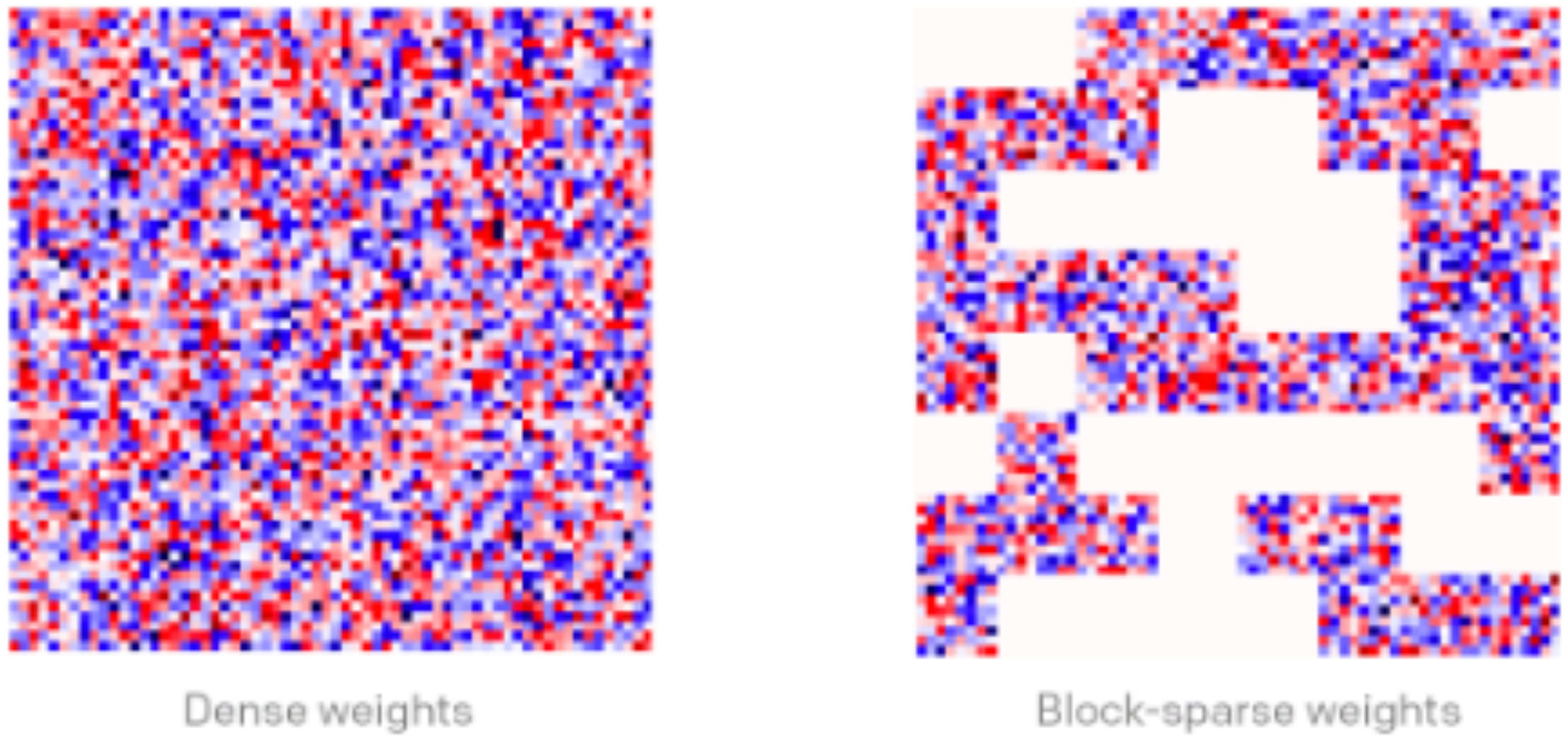


Image from [OpenAI]

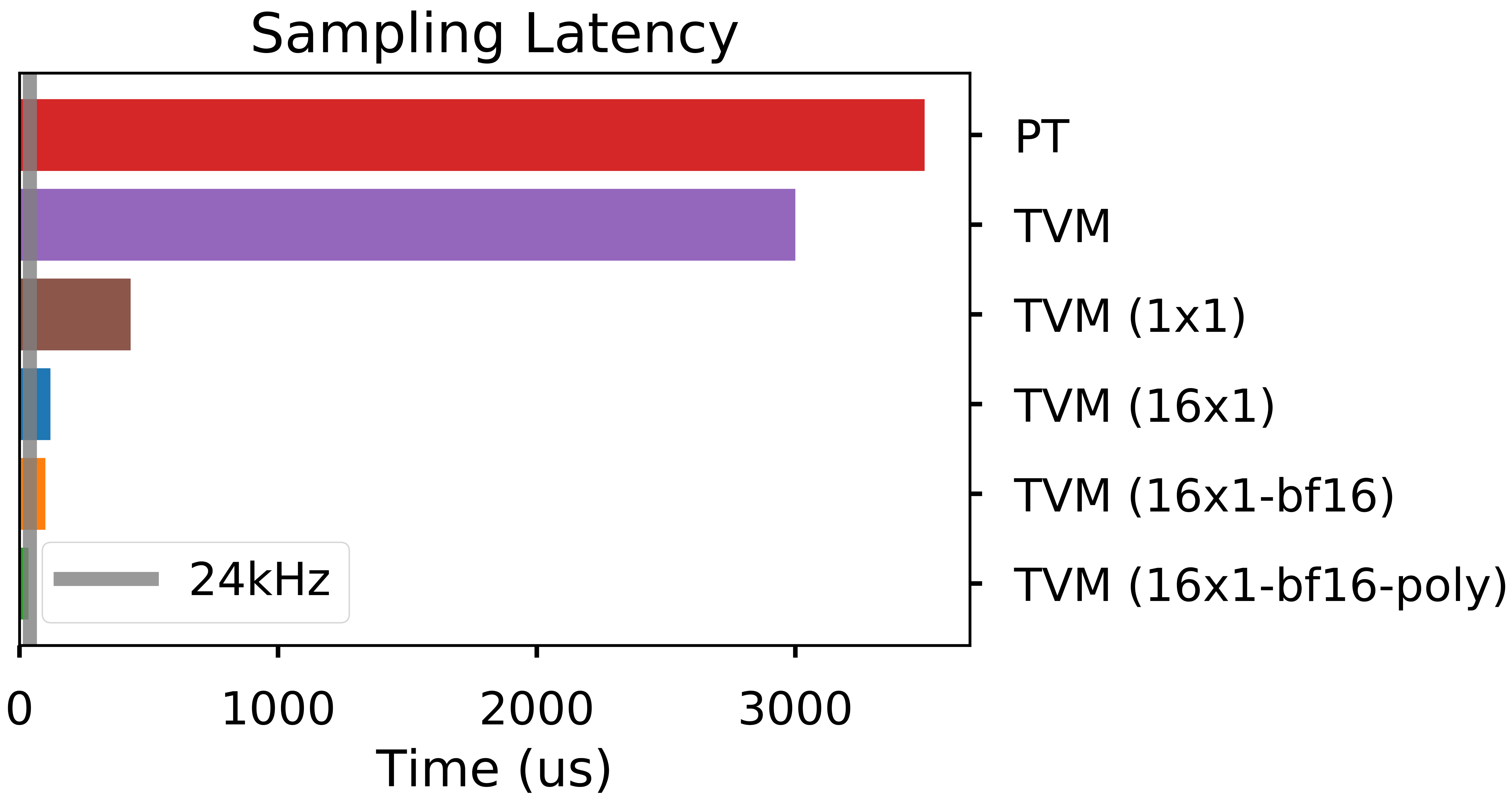
TVM for transcendentals

- Nonlinearity computation (exp, erf, tanh, sigmoid, etc) now **bulk of time**!
- Implemented as intrinsics, lowered to function calls (**no vectorization**)
- Replace with rational polynomial approximations

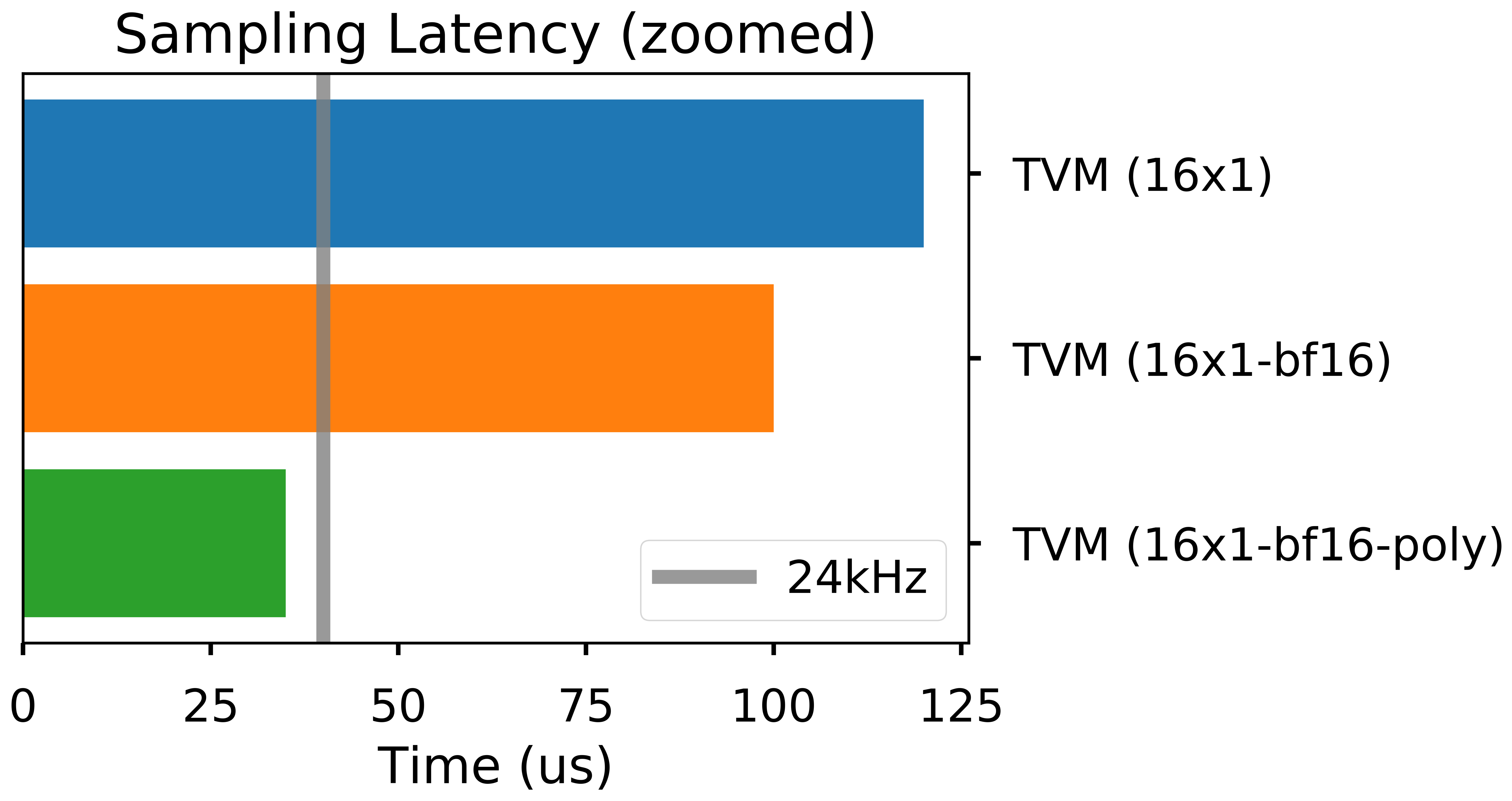
```
def approx_exp(x):
    x = relay.minimum(relay.maximum(x, C(-88.0)), C(88.0))
    x = C(127.0) + x * C(1.44268504)

    i = relay.cast(x, "int32")
    xf = relay.cast(i, "float32")
    x = x - xf
    Y = C(0.99992522) + x * (C(0.69583354) + x \
        * (C(0.22606716) + x * C(0.078024523)))
    exponent = relay.left_shift(i, relay.expr.const(23, "int32"))
    exponent = relay.reinterpret(exponent, "float32")
    return exponent * Y
```

TVM implementation details

- Add `relay.nn.sparse_dense` for block-sparse matrix multiplication (~50 lines of TVM IR)
- Add `relay.reinterpret` to implement transcendental approximations in frontend (~10 lines of Relay IR)
- Add knobs for tuning TVM multithreading runtime
- Use AutoTVM to generate lookup table for architecture search
- **All in less than 1 week!**



Sampling Latency
PT
TVM
TVM (1x1)
TVM (16x1)
TVM (16x1-bf16)
TVM (16x1-bf16-poly)
24kHz
0
1000
2000
3000
Time (us)


Sampling Latency (zoomed)
TVM (16x1)
TVM (16x1-bf16)
TVM (16x1-bf16-poly)
24kHz
0
25
50
75
100
125
Time (us)

TVM results

- TVM sampling model running in **30us on single server CPU core**
- Beat hand-written, highly optimized baselines (https://github.com/mozilla/LPCNet) by ~40% on server CPUs
- Bonus: **Real-time on mobile CPUs for "free"**

Sparsity

Regularization

L1 regularization

- Has been around for a long time!

More complex loss terms

- *Alternating Direction Method of Multipliers for Sparse Convolutional Neural Networks* (2016) Farkhondeh Kiaee, Christian Gagné, and Mahdieh Abbasi

Lotto Ticket Hypothesis

The Lottery Ticket Hypothesis: Finding Sparse, Trainable Neural Networks (2018)
Jonathan Frankle, Michael Carbin [https://arxiv.org/pdf/1803.03635.pdf]

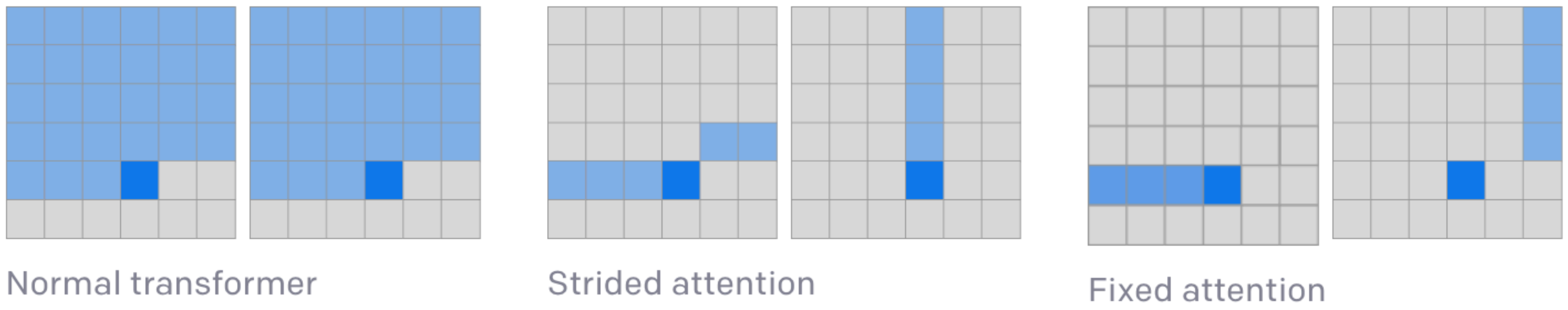
“We find that a standard pruning technique naturally uncovers subnetworks whose initializations made them capable of training effectively.”

“dense, randomly-initialized, feed-forward networks contain subnetworks (”winning tickets”) that - when trained in isolation - reach test accuracy comparable to the original network in a similar number of iterations”

Factorization

Open AI Sparse transformers (2019) [https://openai.com/blog/sparse-transformer/]

- Strided and fixed attentions as two-step sparse factorizations of normal attention



Rewon Child, Scott Gray, Alec Radford, Ilya Sutskever

Factorization

Butterfly Matrices (2019) [https://dawn.cs.stanford.edu/2019/06/13/butterfly/]

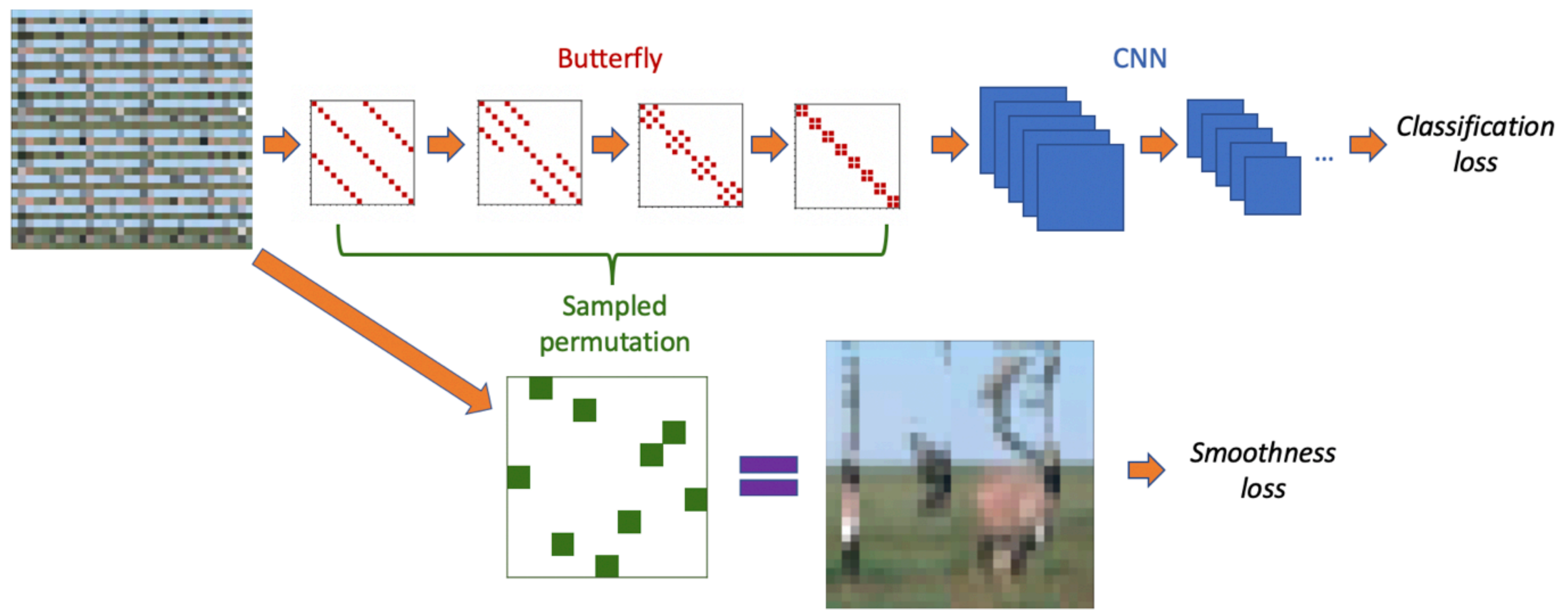


Figure 7. Schematic overview of permutation-learning CNN.

Tri Dao, Albert Gu, Matthew Eichhorn, Megan Leszczynski, Nimit Sohoni, Amit Blonder, Atri Rudra, and Chris Ré

PyTorch Training Support

Pruning API [https://github.com/pytorch/pytorch/issues/20402]

Pruning tutorial [https://github.com/pytorch/tutorials/pull/605]

Large suite of techniques pre-built

- Random, L1, Ln
- Structured, unstructured, channel-wise
- Custom mask-based

Work done by Michela Paganini

Inference Performance

- Work by Aleks Zi and Jongsoo Park [github.com/pytorch/FBGEMM]

- Embed weights directly into the code
  - Currently using asmjit
- What would multiply out to a zero is simply never loaded
  - Skips MACs

```
vbroadcastss ymm7, [rdi+840]
vbroadcastss ymm6, [rdi+844]
vbroadcastss ymm5, [rdi+848]
vbroadcastss ymm4, [rdi+860]
vbroadcastss ymm3, [rdi+868]
vbroadcastss ymm2, [rdi+876]
vbroadcastss ymm1, [rdi+912]
vbroadcastss ymm0, [rdi+932]
vfmadd231ps ymm11, ymm7, yword [L2+9952]
vfmadd231ps ymm12, ymm6, yword [L2+9984]
vfmadd231ps ymm11, ymm5, yword [L2+10016]
vfmadd231ps ymm12, ymm4, yword [L2+10048]
vfmadd231ps ymm13, ymm3, yword [L2+10080]
vfmadd231ps ymm12, ymm2, yword [L2+10112]
vfmadd231ps ymm11, ymm1, yword [L2+10144]
vfmadd231ps ymm8, ymm0, yword [L2+10176]
vbroadcastss ymm7, [rdi+972]
vbroadcastss ymm6, [rdi+1016]
vbroadcastss ymm5, [rdi+1020]
vfmadd231ps ymm11, ymm7, yword [L2+10208]
vfmadd231ps ymm10, ymm6, yword [L2+10240]
vfmadd231ps ymm9, ymm5, yword [L2+10272]
; ...
L1:
ret
align 32
L2:
db 14EE6EC414EE6EC414EE6EC414EE6EC4
db 08547044085470440854704408547044
db FBA176C4FBA176C4FBA176C4FBA176C4
db 6D1673C46D1673C46D1673C46D1673C4
db 38D3724438D3724438D3724438D37244
db 59A56DC459A56DC459A56DC459A56DC4
db 68BA794468BA794468BA794468BA7944
; ...
```

Experimenting With Perf

Batch size 1, 256x256 weights, 90% unstructured sparsity: **2.3x faster**

11 -> 26 effective GFlops

Batch size 1, 256x256 weights, 80% 1x8 blocked sparsity: **6.3x faster**

11 -> 70 effective GFlops

Model system co-design, next steps

Sparsity is easy to achieve at train time

- Free performance at inference time
- Exploration into train time performance (lotto tickets, Open AI blocksparse)

Suddenly, the weights of the model directly impact performance

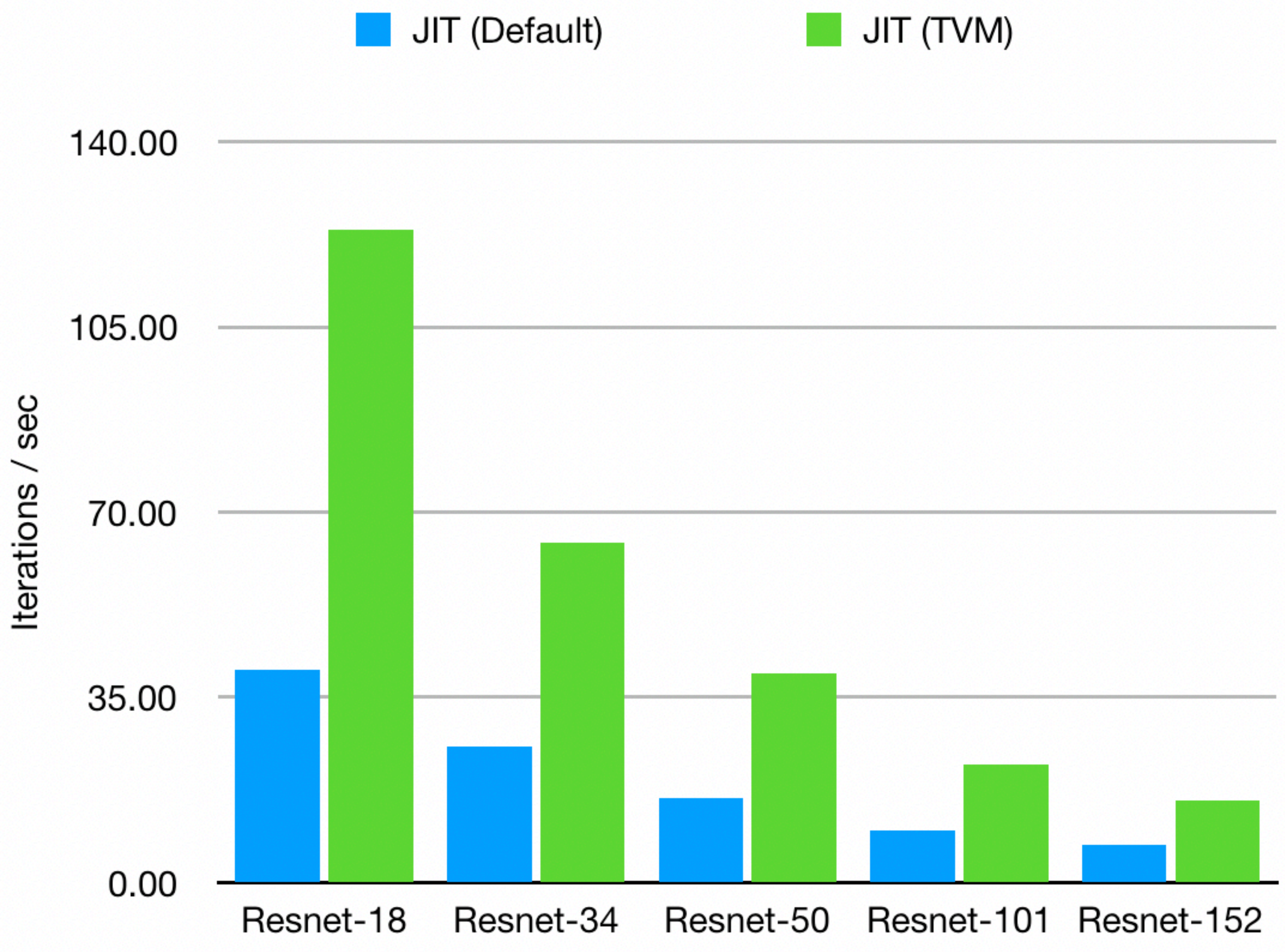
- Benefit: we can transparently speed up models
- Challenge: we should provide perf-visibility to model engineers

TVM - PyTorch Integration

github.com/pytorch/tvm

- Repository that lowers TorchScript graphs to Relay
- Work done by Kimish Patel, Lingyi Liu, Wanchao Liang, Yinghai Lu and others

- See https://tvm.ai/2019/05/30/pytorch-frontend

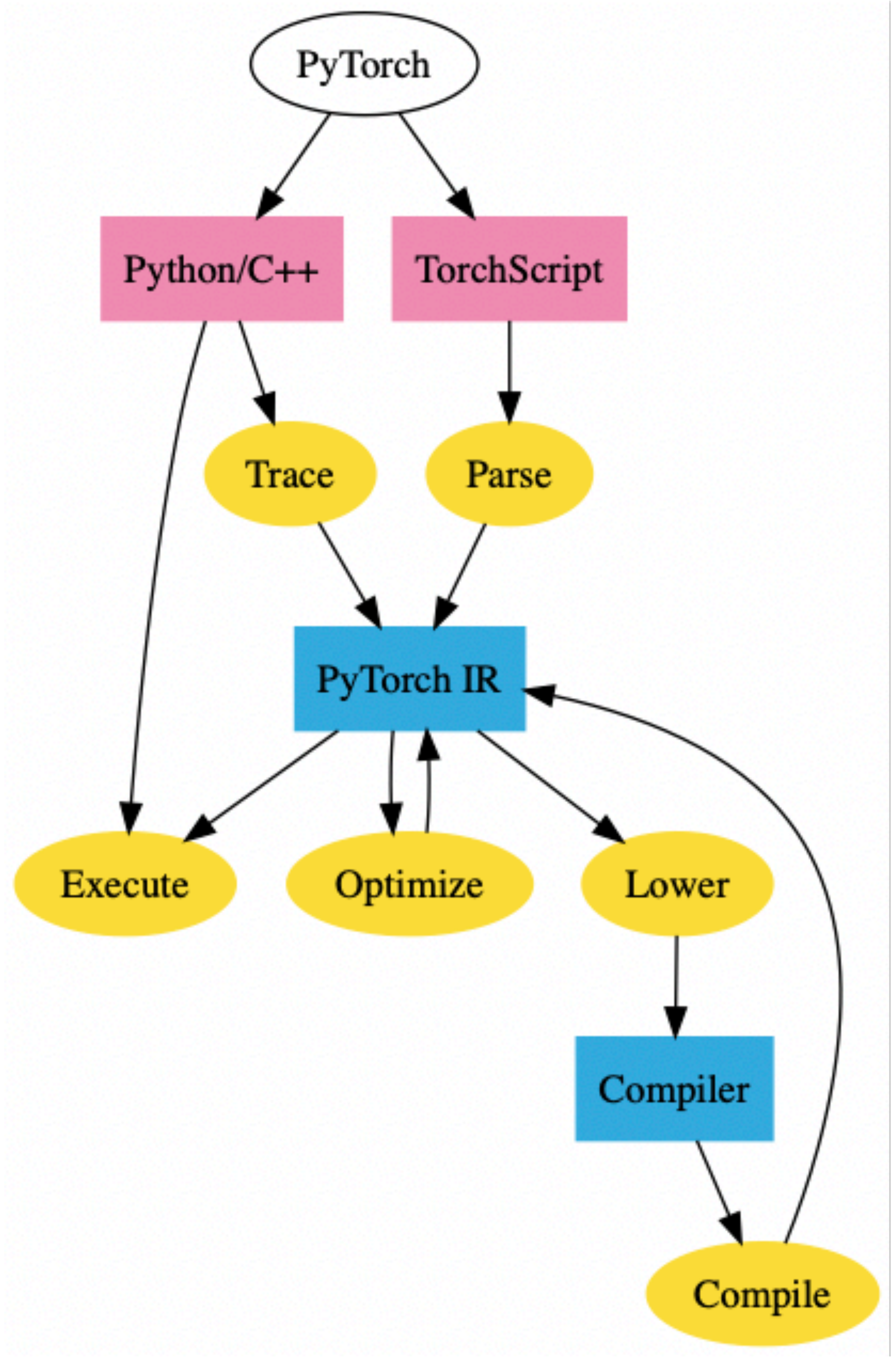


Optimizing Python isn’t fun

Python is too flexible to optimize directly
- Workloads being run aren’t complicated

TorchScript was developed to run models in C++
- Full Python-like language implementation
- Runtime

We want to flush out real performance
- Preserve PyTorch’s flexibility
- Easily enable fast backends like TVM



Lazy Tensors

Record computation

- Accumulate into a graph
- Execute as late as possible

On execution, try to compile

- Cache precompiled graphs

Limitations

- No control flow is captured
- Compilation latency can create perf cliffs

Profiling Executor

Record computation

- Execute immediately
- Accumulate statistics

After a couple of executions

- Rewrite the IR
- Optimize a stable subgraph

Limitations

- Multiple runs before performance
- Complicates the IR

```
graph(%a.1 : Tensor,
      %b.1 : Tensor,
      %c : Tensor):
  %27 : int = prim::BailoutTemplate_0()
  %25 : ProfiledTensor(dtype = Double, requires_grad = 0 , shape = (2, 2) = prim::BailOut[index=0](%27, %b.1, %a.1
  %26 : ProfiledTensor(dtype = Double, requires_grad = 0 , shape = (2, 2) = prim::BailOut[index=1](%27, %a.1, %25)
  %3 : int = prim::Constant[value=1]()
  %4 : int = prim::Constant[value=2]() # test_jit.py:4232:25
  %5 : int = prim::Constant[value=3]() # test_jit.py:4233:25
  %6 : int = prim::Constant[value=0]() # test_jit.py:4235:47
  %e.1 : ProfiledTensor(dtype = Double, requires_grad = 0 , shape = (2, 2) = aten::add(%25, %5, %3)
  %f.1 : ProfiledTensor(dtype = Double, requires_grad = 0 , shape = (2, 2) = aten::sub(%26, %25, %3)
  %14 : ProfiledTensor(dtype = Double, requires_grad = 0 , shape = (2, 2) = aten::add(%f.1, %e.1, %3)
  %16 : ProfiledTensor(dtype = Double, requires_grad = 0 , shape = (2, 2) = aten::clamp(%14, %6, %4)
  return (%16)
with prim::BailoutTemplate_0 = graph(%a.1 : Tensor,
      %b.1 : Tensor,
      %c : Tensor):
  %3 : ProfiledTensor(dtype = Double, requires_grad = 0 , shape = (2, 2) = prim::BailOut[index=0](%b.1, %a.1)
  %4 : ProfiledTensor(dtype = Double, requires_grad = 0 , shape = (2, 2) = prim::BailOut[index=1](%a.1, %3)
  %5 : int = prim::Constant[value=1]()
  %6 : int = prim::Constant[value=2]() # test_jit.py:4232:25
  %7 : int = prim::Constant[value=3]() # test_jit.py:4233:25
  %8 : int = prim::Constant[value=0]() # test_jit.py:4235:47
  %e.1 : ProfiledTensor(dtype = Double, requires_grad = 0 , shape = (2, 2) = aten::add(%3, %7, %5)
  %f.1 : ProfiledTensor(dtype = Double, requires_grad = 0 , shape = (2, 2) = aten::sub(%4, %3, %5)
  %11 : ProfiledTensor(dtype = Double, requires_grad = 0 , shape = (2, 2) = aten::add(%f.1, %e.1, %5)
  %12 : ProfiledTensor(dtype = Double, requires_grad = 0 , shape = (2, 2) = aten::clamp(%11, %8, %6)
  return (%12)
```

Next Steps

We are excited about the performance TVM achieves

We are working to more tightly integrate PyTorch and TVM

Big thanks to the community