Efficient ML on the Edge with Apache TVM

ODIW Workshop, co-located with MLSys 2021

Thierry Moreau, Andrew Reusch
OctoML Inc.

Agenda

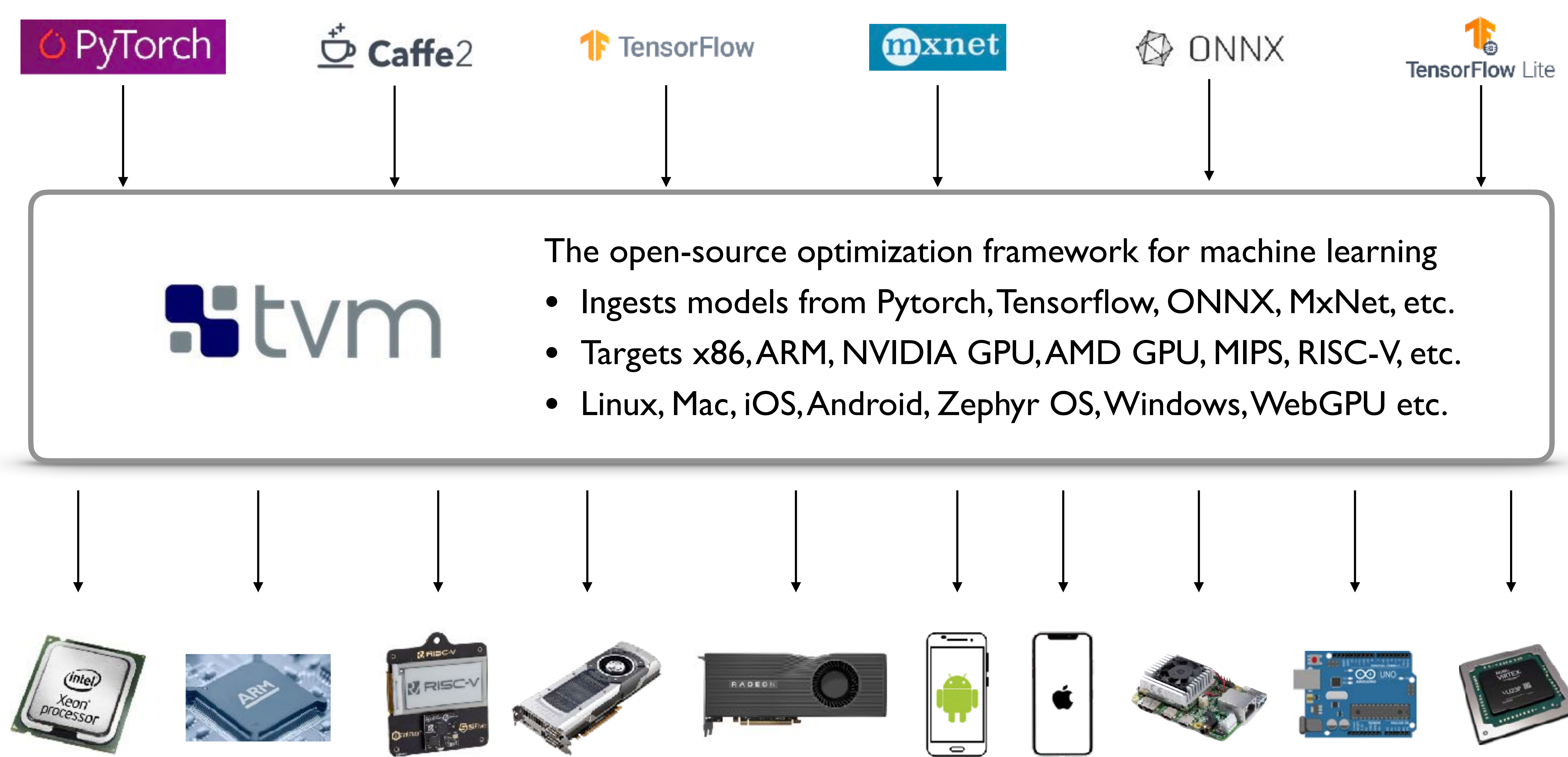
TVM Intro

TVM optimizations

uTVM - TVM for microcontrollers

Demo (15mins)

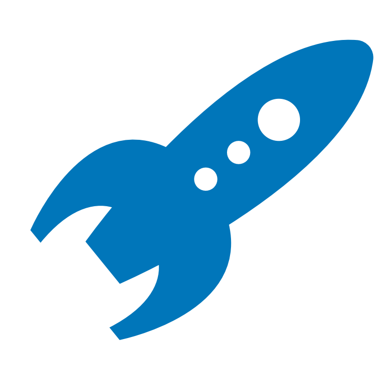
Introducing TVM



What problems does TVM address?

Portability:

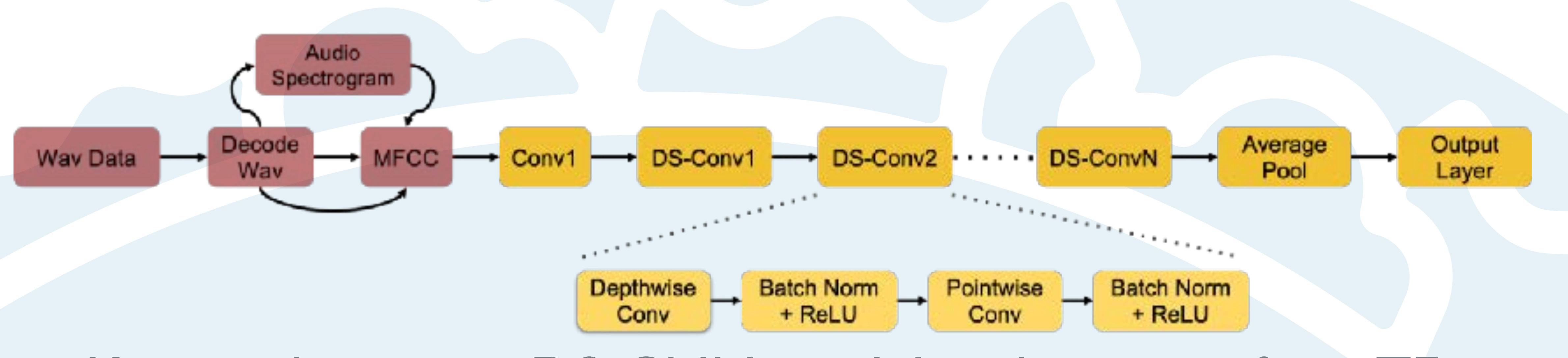
- When there are limited hardware options to deploy your model



Efficiency & Productivity:

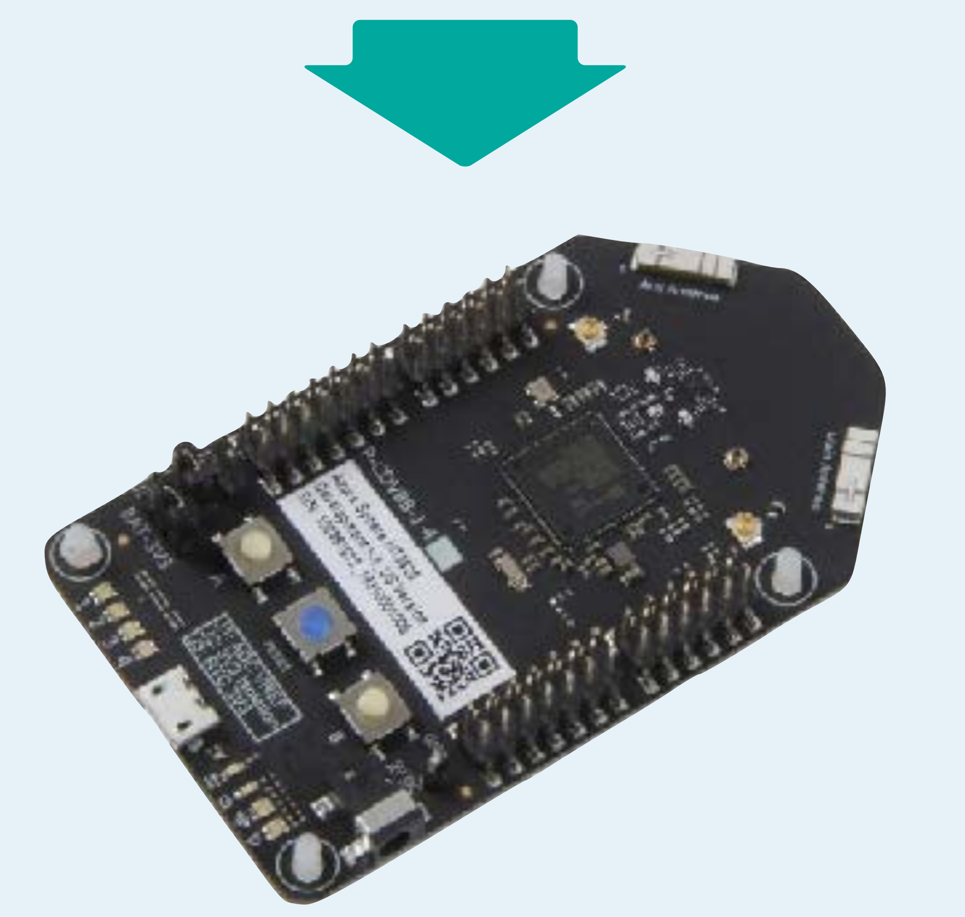
- When you need to squeeze as much efficiency out of your target platform

TVM for Portability



Keyword spotting DS-CNN model architecture from TF

Mehrdad Hessar, U.W. Ph.D.



AzureSphere: Microsoft's secure edge IoT device

limited SRAM, bare bones operating system, no C++ support, no dynamic linking …

Deploying deep learning on AzureSphere using TVM, Hessar et al., Blog post

TVM for Portability

- Use case #1: ML Engineer
  - *"I'm generally very happy with Tensorflow/Pytorch, but I was asked to deploy a model on a platform that is not/poorly supported by these frameworks."*
  - **Challenge**: non x86 ISAs, unsupported GPU/DSP/Accelerator, non-mainstream Operating System

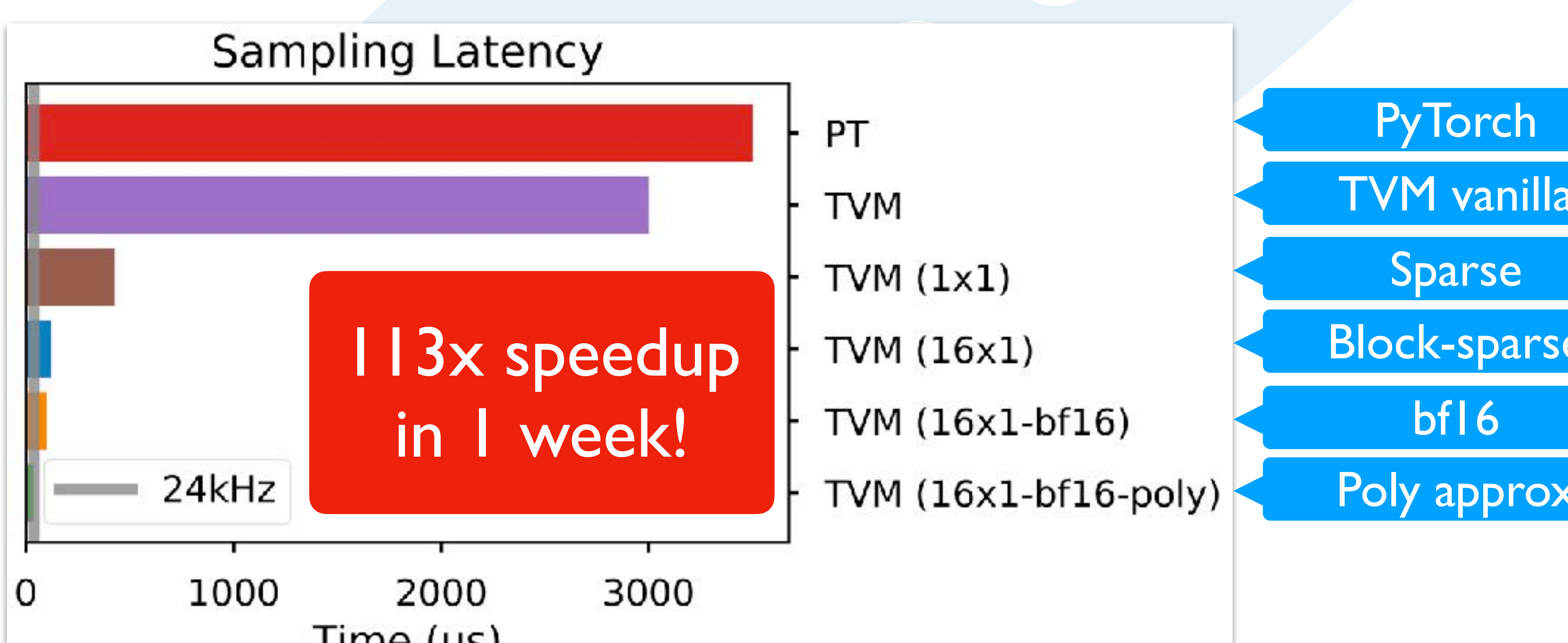
TVM provides the ability to deploy ML on platforms where deployment would be painstakingly difficult otherwise

TVM for Efficiency + Productivity

Facebook AI/Systems Engineer

Workload: WaveRNN style model architecture

- Compute dominated by GRU and FC layers
- 24kHz sampling frequency requires 40us inference net runtime
- Initial model runs in PT with 3,400us inference net runtime
- **85x slower than target**



PyTorch
TVM vanilla
Sparse
Block-sparse
bf16
Poly approx

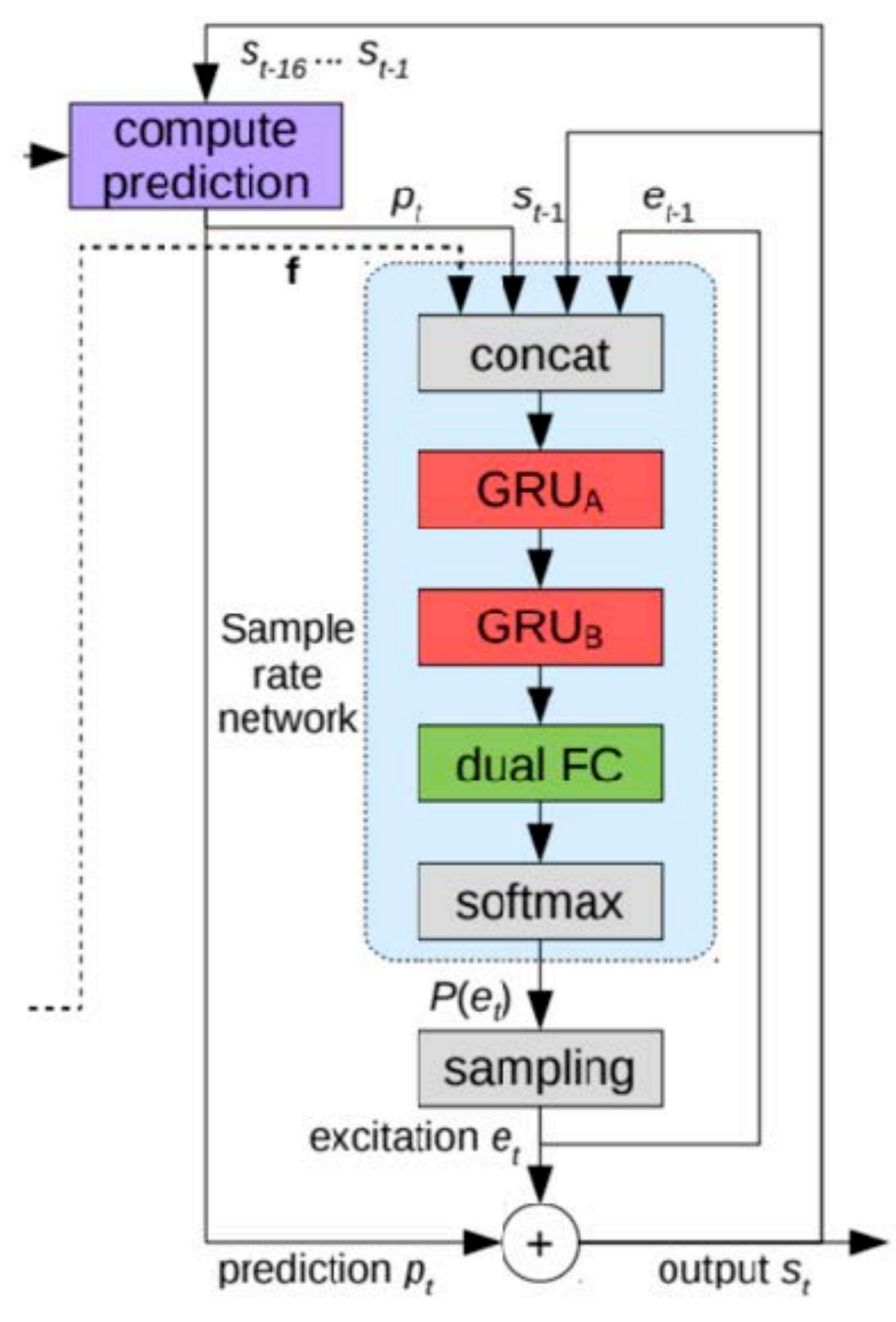


Image from LPCNet

TVM @ Facebook, Tulloch et al., TVM Conf 2019

TVM for Efficiency + Productivity

- Use case #2: ML-Sys Engineer
  - *"I am targeting a pretty mainstream system (e.g. x86, GPU) but I want to get the most performance out of my model. Every millisecond I shave off results in $$$ savings for your service that you've deployed in the cloud."*
  - **Challenge**: most frameworks rely on libraries that are optimized for the common workload. Framework have inefficient interpreters (1-2us per op. overhead on PT)

TVM provides model optimization capabilities to produce a library that is fined tune for your one model to run fast on your one platform

TVM Impact on Industry

Every "Alexa" wake-up today across all devices uses a model optimized with TVM

"[TVM enabled] real-time on mobile CPUs for free...We are excited about the performance TVM achieves." More than 85x speed-up for speech recognition model.

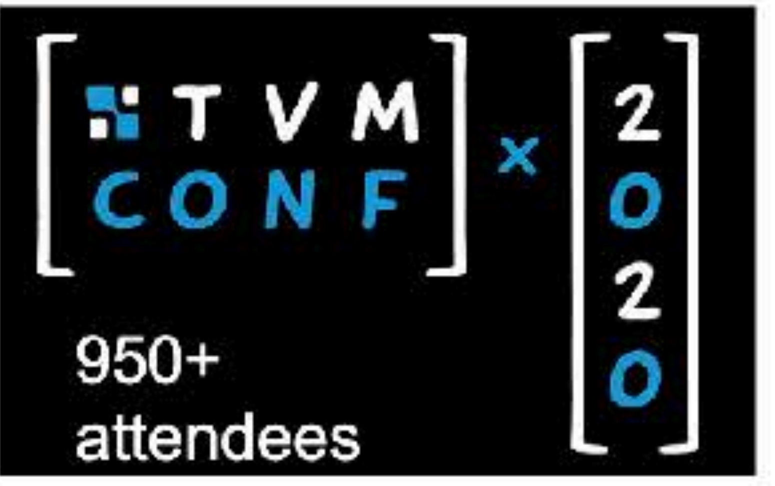
Bing query understanding: 112ms (Tensorflow) -> 34ms (TVM). QnA bot: 73ms->28ms (CPU), 10.1ms->5.5ms (GPU) - TVMconf 2019.

"TVM is key to ML Access on Hexagon" - Jeff Gehlhaar, VP Technology

Unified ML stack for CPU, GPU, NPU built on TVM. TVMconf 2020.

UNTETHER AI

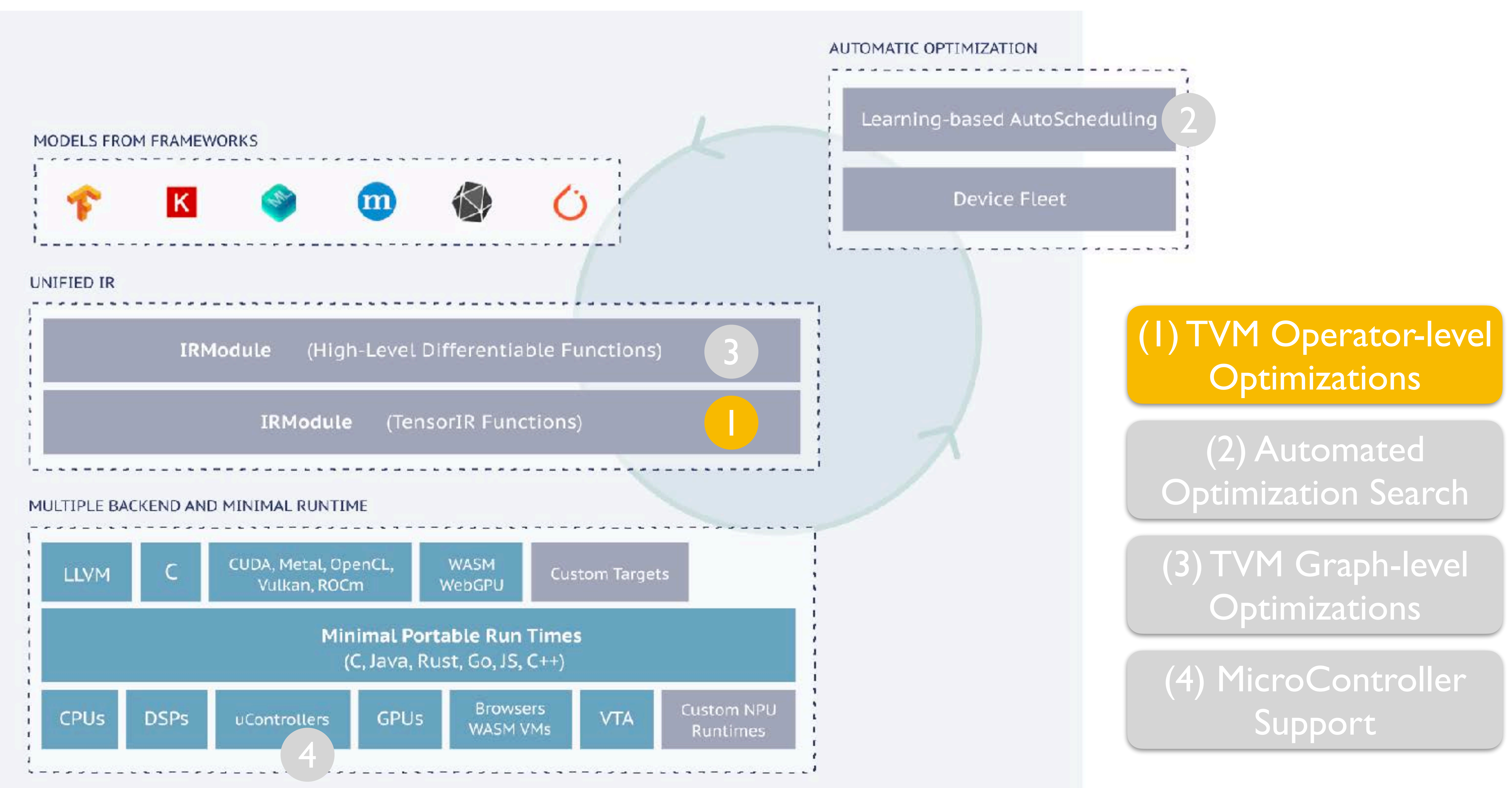
Open source
~500+ contributors from industry and academia.



13% HW vendors, 35% research, **50%+ ML end user.**

Cross-product of {Models} x {Hardware} is large.
Strong community support makes diversity manageable and ensures future-proofness.

Overview of this talk



(1) TVM Operator-level Optimizations

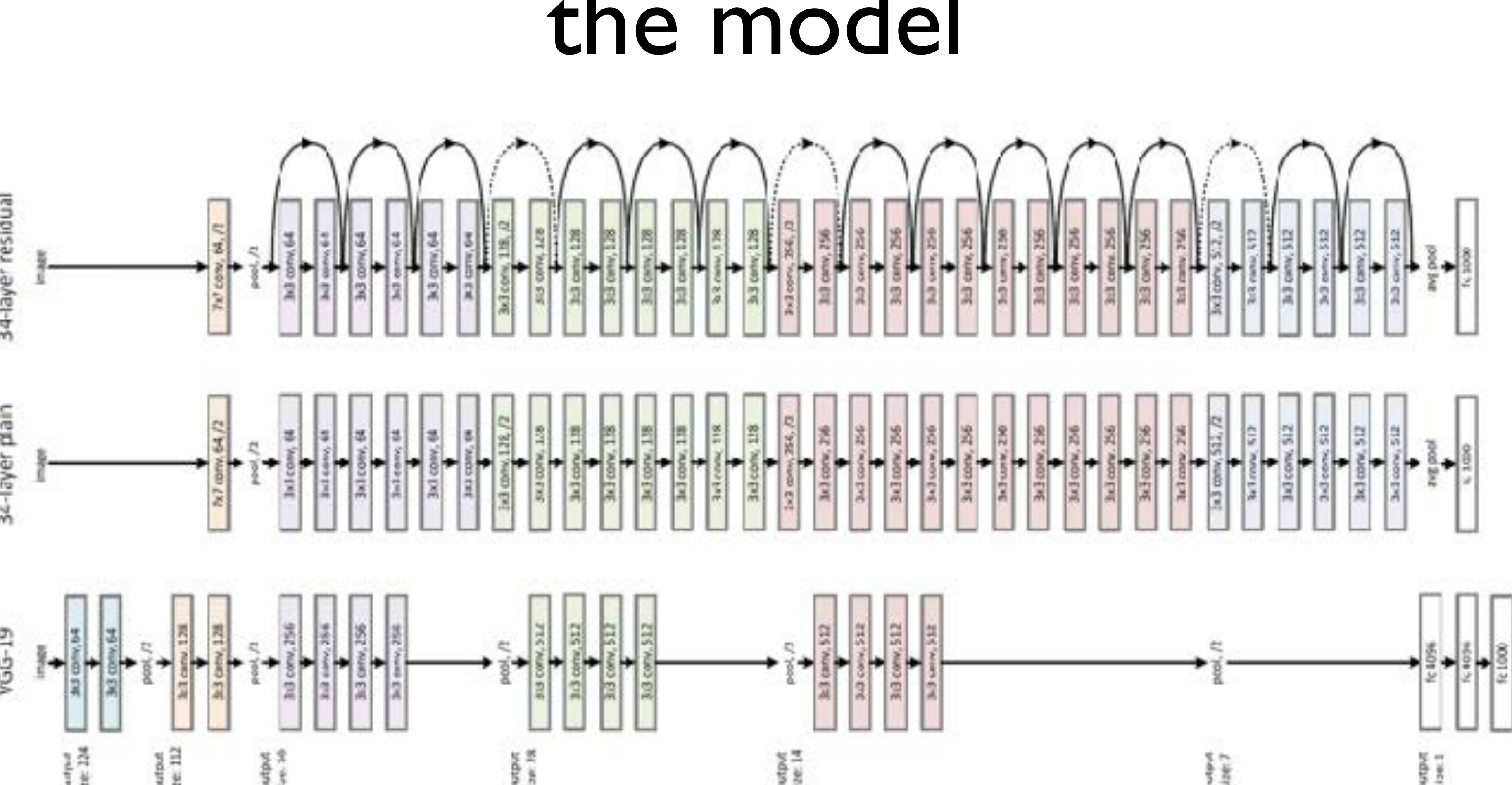
(2) Automated Optimization Search

(3) TVM Graph-level Optimizations

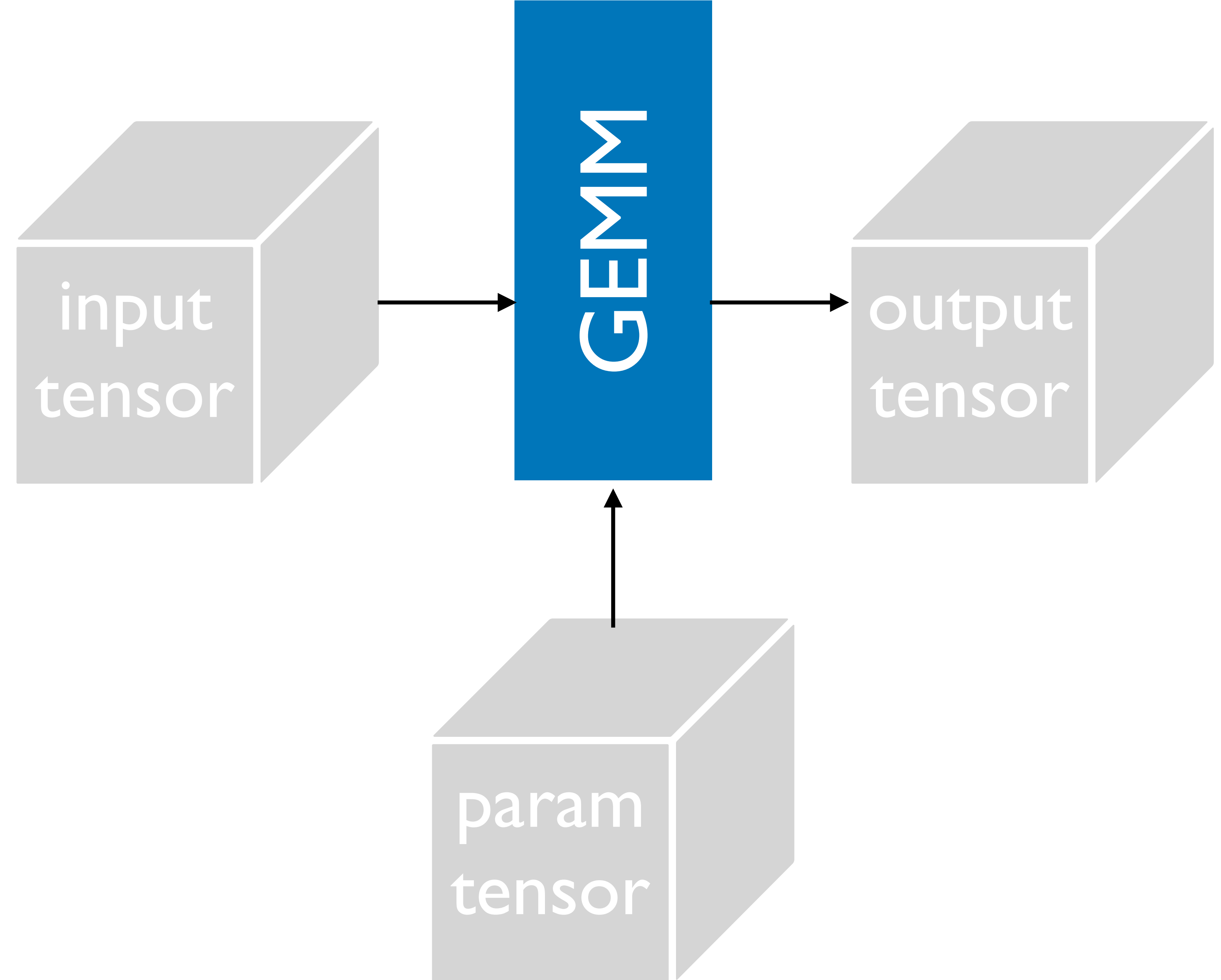
(4) MicroController Support

Dissecting the black box

the model



layer, a.k.a. operator



Example of a operator implementation

Naive Approach

```
void multiply(int mat1[][N],
              int mat2[][N],
              int res[][N])
{
    int i, j, k;
    for (i = 0; i < N; i++)
    {
        for (j = 0; j < N; j++)
        {
            res[i][j] = 0;
            for (k = 0; k < N; k++)
                res[i][j] += mat1[i][k] *
                             mat2[k][j];
        }
    }
}
```

10-100x faster

Optimized Approach (snippet from 1000s lines of code)

```
prefetcht0      PRESIZE * SIZE(%ebx, %edx, 2)
FLD     0 * SIZE(%ebx)          # at    = *(a offset  + 0 * lda)
fmul    %st(1),%st              # at1 *= bt1

prefetcht0      PRESIZE * SIZE(%ecx)
faddp   %st,%st(2)              # ct1 += at1
FLD     0 * SIZE(%ecx)          # at1   = *(a_offset2 + 0 * lda)

prefetcht0      PRESIZE * SIZE(%ecx, %edx, 2)
fmul    %st(1),%st              # at1 *= bt1
faddp   %st,%st(3)              # ct2 += at1

prefetcht0      PRESIZE * SIZE(%ebx)
FLD     0 * SIZE(%ebx, %edx, 2) # at    = *(a_offset  + 2 * lda)
fmul    %st(1),%st

faddp   %st,%st(4)
FLD     0 * SIZE(%ecx, %edx, 2) # at1   = *(a_offset2 + 2 * lda)
fmulp   %st, %st(1)

faddp   %st,%st(4)
FLD     1 * SIZE(%esi)
FLD     1 * SIZE(%ebx)          # at    = *(a_offset  + 0 * lda)

fmul    %st(1),%st              # at1 *= bt1
faddp   %st,%st(2)              # ct1 += at1
FLD     1 * SIZE(%ecx)          # at1   = *(a_offset2 + 0 * lda)
```

Halide programming model

Functional definition: what should this function do?

```
// The algorithm - no storage or order
blur_x(x, y) = (input(x-1, y) + input(x, y) + input(x+1, y))/3;
blur_y(x, y) = (blur_x(x, y-1) + blur_x(x, y) + blur_x(x, y+1))/3;
```

Schedule definitions: how should the function do it?

```
// The schedule - defines order, locality; implies storage
blur_y.tile(x, y, xi, yi, 256, 32)
      .vectorize(xi, 8).parallel(y);
blur_x.compute_at(blur_y, x).vectorize(x, 8);
```

TVM schedule: Matrix Multiplication

Tensor-Expression DSL defines the algorithm and the schedule

```python
# Algorithm
k = te.reduce_axis((0, K), "k")
A = te.placeholder((M, K), name="A")
B = te.placeholder((K, N), name="B")
C = te.compute((M, N), lambda x, y: te.sum(A[x, k] * B[k, y], axis=k), name="C")

# Default schedule
s = te.create_schedule(C.op)
```

vanilla schedule

Tensor-Level IR (TIR) program defines the low-level implementation (can be compiled down to LLVM IR, CUDA, OpenCL et.)

```
primfn(A_1: handle, B_1: handle, C_1: handle) -> ()
  attr = {"global_symbol": "main", "tir.noalias": True}
  buffers = {B: Buffer(B_2: Pointer(float32), float32, [1024, 1024], []),
             C: Buffer(C_2: Pointer(float32), float32, [1024, 1024], []),
             A: Buffer(A_2: Pointer(float32), float32, [1024, 1024], [])}
  buffer_map = {A_1: A, B_1: B, C_1: C} {
  for (x: int32, 0, 1024) {
    for (y: int32, 0, 1024) {
      C_2[((x*1024) + y)] = 0f32
      for (k: int32, 0, 1024) {
        C_2[((x*1024) + y)] = ((float32*)C_2[((x*1024) + y)] + ((float32*)A_2[((x*1024) + k)]*(float32*)B_2[((k*1024) + y)]))
      }
    }
  }
}
```

3 nested loops

TVM schedule: Tiling and Reordering

vanilla schedule

compute tiling

split reduction axis

```
bn = 32
s = te.create_schedule(C.op)

# Blocking by loop tiling
xo, yo, xi, yi = s[C].tile(C.op.axis[0], C.op.axis[1], bn, bn)
(k,) = s[C].op.reduce_axis
ko, ki = s[C].split(k, factor=4)

# Hoist reduction domain outside the blocking loop
s[C].reorder(xo, yo, ko, ki, xi, yi)
```

6 nested loops

```
primfn(A_1: handle, B_1: handle, C_1: handle) -> ()
  attr = {"global_symbol": "main", "tir.noalias": True}
  buffers = {C: Buffer(C_2: Pointer(float32), float32, [1024, 1024], []),
             B: Buffer(B_2: Pointer(float32), float32, [1024, 1024], []),
             A: Buffer(A_2: Pointer(float32), float32, [1024, 1024], [])}
  buffer_map = {A_1: A, B_1: B, C_1: C} {
  for (x.outer: int32, 0, 32) {
    for (y.outer: int32, 0, 32) {
      for (x.inner.init: int32, 0, 32) {
        for (y.inner.init: int32, 0, 32) {
          C_2[((((x.outer*32768) + (x.inner.init*1024)) + (y.outer*32)) + y.inner.init)] = 0f32
        }
      }
      for (k.outer: int32, 0, 256) {
        for (k.inner: int32, 0, 4) {
          for (x.inner: int32, 0, 32) {
            for (y.inner: int32, 0, 32) {
              C_2[((((x.outer*32768) + (x.inner*1024)) + (y.outer*32)) + y.inner)] = ((float32*)C_2[((((x.outer*32768) + (x.inner*1024)) + (y.outer*32)) + y.inner)] +
((float32*)A_2[((((x.outer*32768) + (x.inner*1024)) + (k.outer*4)) + k.inner)]*(float32*)B_2[((((k.outer*4096) + (k.inner*1024)) + (y.outer*32)) + y.inner)]))
            }
          }
        }
      }
    }
  }
}
```

TVM schedule: Vectorization

vanilla schedule

compute tiling

split reduction axis

vectorize

```python
s = te.create_schedule(C.op)
xo, yo, xi, yi = s[C].tile(C.op.axis[0], C.op.axis[1], bn, bn)
(k,) = s[C].op.reduce_axis
ko, ki = s[C].split(k, factor=4)

s[C].reorder(xo, yo, ko, ki, xi, yi)

# Vectorization
s[C].vectorize(yi)
```

5 nested loops

vectorization

```
primfn(A_1: handle, B_1: handle, C_1: handle) -> ()
  attr = {"global_symbol": "main", "tir.noalias": True}
  buffers = {C: Buffer(C_2: Pointer(float32), float32, [1024, 1024], []),
             B: Buffer(B_2: Pointer(float32), float32, [1024, 1024], []),
             A: Buffer(A_2: Pointer(float32), float32, [1024, 1024], [])}
  buffer_map = {A_1: A, B_1: B, C_1: C} {
  for (x.outer: int32, 0, 32) {
    for (y.outer: int32, 0, 32) {
      for (x.inner.init: int32, 0, 32) {
        C_2[ramp((((x.outer*32768) + (x.inner.init*1024)) + (y.outer*32)), 1, 32)] = broadcast(0f32, 32)
      }
      for (k.outer: int32, 0, 256) {
        for (k.inner: int32, 0, 4) {
          for (x.inner: int32, 0, 32) {
            C_2[ramp((((x.outer*32768) + (x.inner*1024)) + (y.outer*32)), 1, 32)] = ((float32x32*)C_2[ramp((((x.outer*32768) + (x.inner*1024)) + (y.outer*32)), 1, 32)] +
(broadcast((float32*)A_2[((((x.outer*32768) + (x.inner*1024)) + (k.outer*4)) + k.inner)], 32)*(float32x32*)B_2[ramp((((k.outer*4096) + (k.inner*1024)) + (y.outer*32)), 1, 32)]))
          }
        }
      }
    }
  }
}
```

TVM Schedule: After Several Steps

Before

```python
s = te.create_schedule(C.op)
```

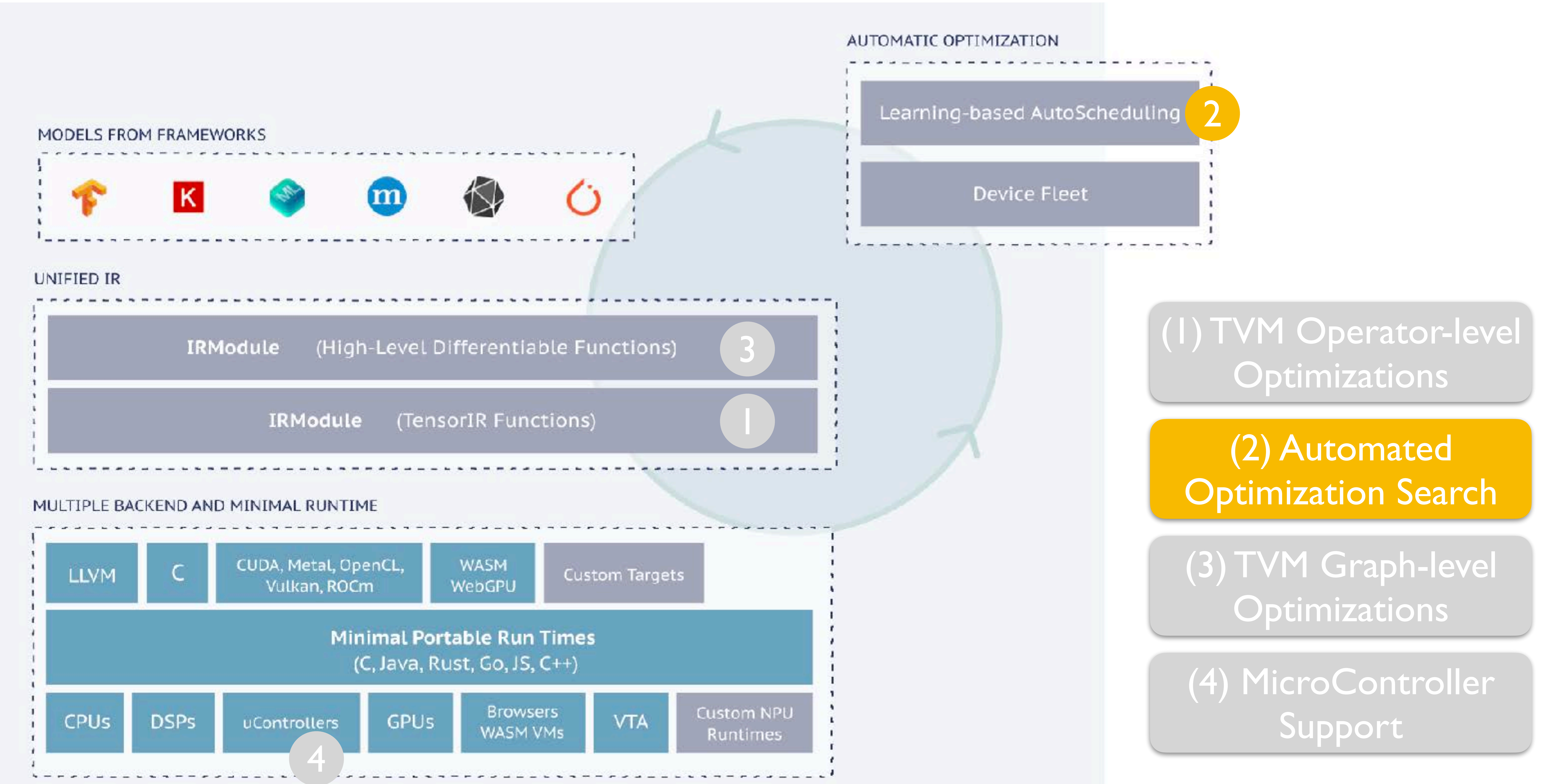
```
primfn(A_1: handle, B_1: handle, C_1: handle) -> ()
  attr = {"global_symbol": "main", "tir.noalias": True}
  buffers = {B: Buffer(B_2: Pointer(float32), float32, [1024, 1024], []),
             C: Buffer(C_2: Pointer(float32), float32, [1024, 1024], []),
             A: Buffer(A_2: Pointer(float32), float32, [1024, 1024], [])}
  buffer_map = {A_1: A, B_1: B, C_1: C} {
  for (x: int32, 0, 1024) {
    for (y: int32, 0, 1024) {
      C_2[((x*1024) + y)] = 0f32
      for (k: int32, 0, 1024) {
        C_2[((x*1024) + y)] = ((float32*)C_2[((x*1024) + y)] + ((float32*)A_2[((x*1024) + k)]*(float32*)B_2[((k*1024) + y)]))
      }
    }
  }
}
```

After: 200x faster

```python
s = te.create_schedule(C.op)

CC = s.cache_write(C, "global")

xo, yo, xi, yi = s[C].tile(C.op.axis[0], C.op.axis[1], bn, bn)

s[CC].compute_at(s[C], yo)

xc, yc = s[CC].op.axis

(k,) = s[CC].op.reduce_axis
ko, ki = s[CC].split(k, factor=4)
s[CC].reorder(ko, xc, ki, yc)
s[CC].unroll(ki)
s[CC].vectorize(yc)

# parallel
s[C].parallel(xo)

x, y, z = s[packedB].op.axis
s[packedB].vectorize(z)
s[packedB].parallel(x)
```

```
primfn(A_1: handle, B_1: handle, C_1: handle) -> ()
  attr = {"global_symbol": "main", "tir.noalias": True}
  buffers = {C: Buffer(C_2: Pointer(float32), float32, [1024, 1024], []),
             B: Buffer(B_2: Pointer(float32), float32, [1024, 1024], []),
             A: Buffer(A_2: Pointer(float32), float32, [1024, 1024], [])}
  buffer_map = {A_1: A, B_1: B, C_1: C} {
  attr [packedB: Pointer(float32)] "storage_scope" = "global";
  allocate(packedB, float32x32, [32768]) {
    for (x: int32, 0, 32) "parallel" {
      for (y: int32, 0, 1024) {
        packedB[ramp(((x*32768) + (y*32)), 1, 32)] = (float32x32*)B_2[ramp(((y*1024) + (x*32)), 1, 32)]
      }
    }
    for (x.outer: int32, 0, 32) "parallel" {
      attr [C.global: Pointer(float32)] "storage_scope" = "global";
      allocate(C.global, float32, [1024]);
      for (y.outer: int32, 0, 32) {
        for (x.c.init: int32, 0, 32) {
          C.global[ramp((x.c.init*32), 1, 32)] = broadcast(0f32, 32)
        }
        for (k.outer: int32, 0, 256) {
          for (x.c: int32, 0, 32) {
            C.global[ramp((x.c*32), 1, 32)] = ((float32x32*)C.global[ramp((x.c*32), 1, 32)] + (broadcast((float32*)A_2[(((x.outer*32768) +
(x.c*1024)) + (k.outer*4))], 32)*(float32x32*)packedB[ramp(((y.outer*32768) + (k.outer*128)), 1, 32)]))
            C.global[ramp((x.c*32), 1, 32)] = ((float32x32*)C.global[ramp((x.c*32), 1, 32)] + (broadcast((float32*)A_2[((((x.outer*32768) +
(x.c*1024)) + (k.outer*4)) + 1)], 32)*(float32x32*)packedB[ramp((((y.outer*32768) + (k.outer*128)) + 32), 1, 32)]))
            C.global[ramp((x.c*32), 1, 32)] = ((float32x32*)C.global[ramp((x.c*32), 1, 32)] + (broadcast((float32*)A_2[((((x.outer*32768) +
(x.c*1024)) + (k.outer*4)) + 2)], 32)*(float32x32*)packedB[ramp((((y.outer*32768) + (k.outer*128)) + 64), 1, 32)]))
            C.global[ramp((x.c*32), 1, 32)] = ((float32x32*)C.global[ramp((x.c*32), 1, 32)] + (broadcast((float32*)A_2[((((x.outer*32768) +
(x.c*1024)) + (k.outer*4)) + 3)], 32)*(float32x32*)packedB[ramp((((y.outer*32768) + (k.outer*128)) + 96), 1, 32)]))
          }
        }
        for (x.inner: int32, 0, 32) {
          for (y.inner: int32, 0, 32) {
            C_2[((((x.outer*32768) + (x.inner*1024)) + (y.outer*32)) + y.inner)] = (float32*)C.global[((x.inner*32) + y.inner)]
          }
        }
      }
    }
  }
}
```

Overview of this talk



(1) TVM Operator-level Optimizations

(2) Automated Optimization Search

(3) TVM Graph-level Optimizations

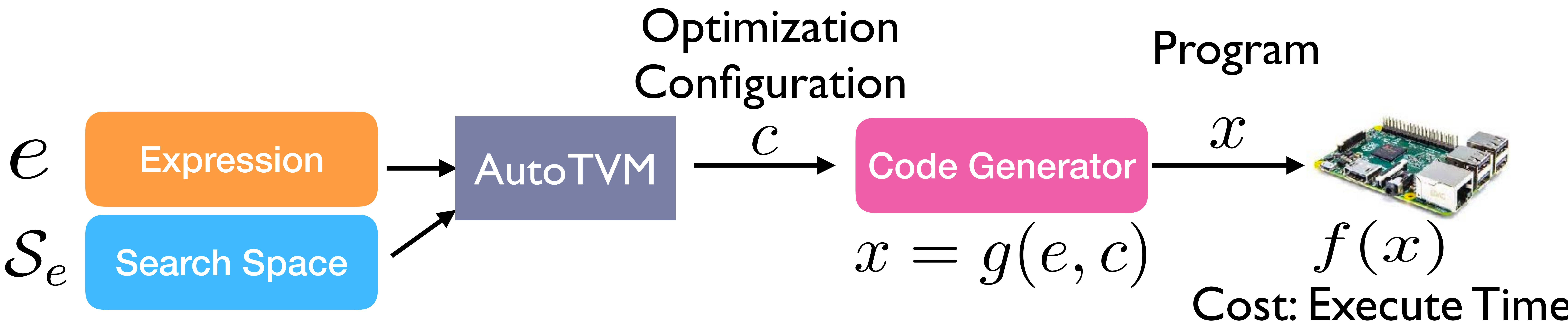
(4) MicroController Support

Taking optimization to the next level

- In the prior example we can quickly attain ~60% of MLK (hand tuned library) performance in a few lines code
- How to we take the performance a notch further with less “human in the loop”?

A single operator has a very large space of schedules to explore, often 1,000,000,000s of possible schedule implementations

Problem Formalization

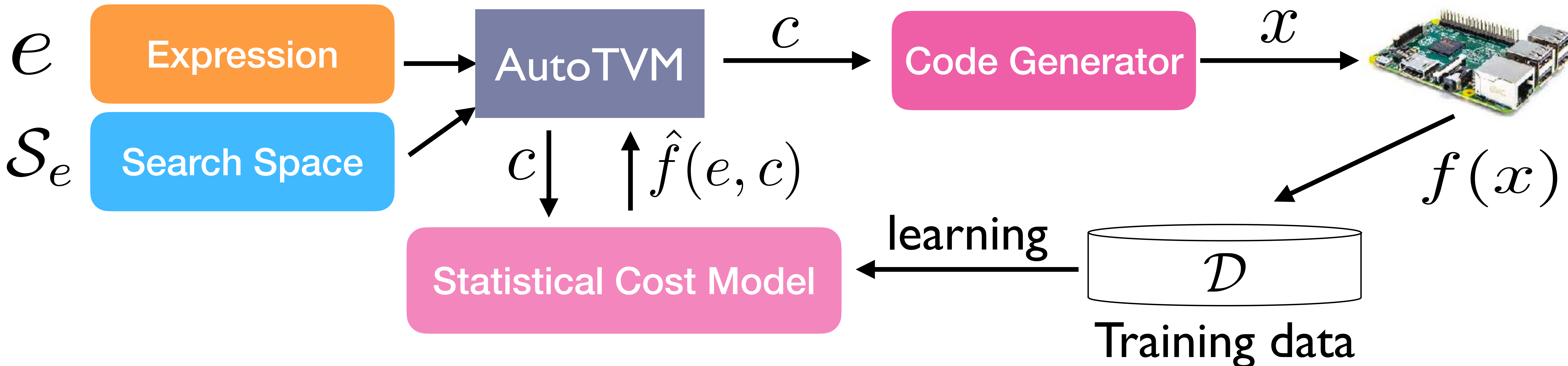


Objective $argmin_{c \in S_e} f(g(e, c))$

Learning to Optimize Tensor Programs. Chen et al. NeurIPS 18

ML based Models

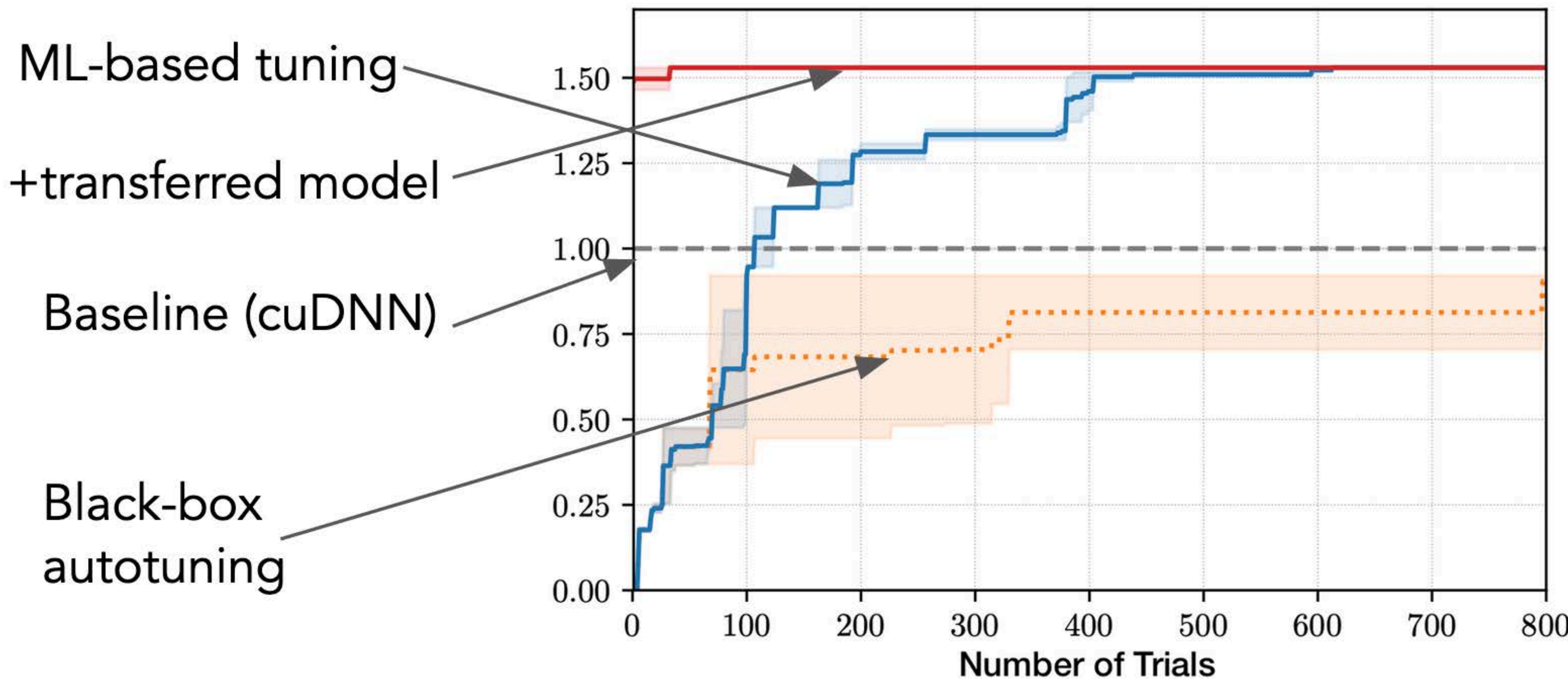
Use machine learning to learn a statistical cost model



Benefit: Automatically adapt to hardware type

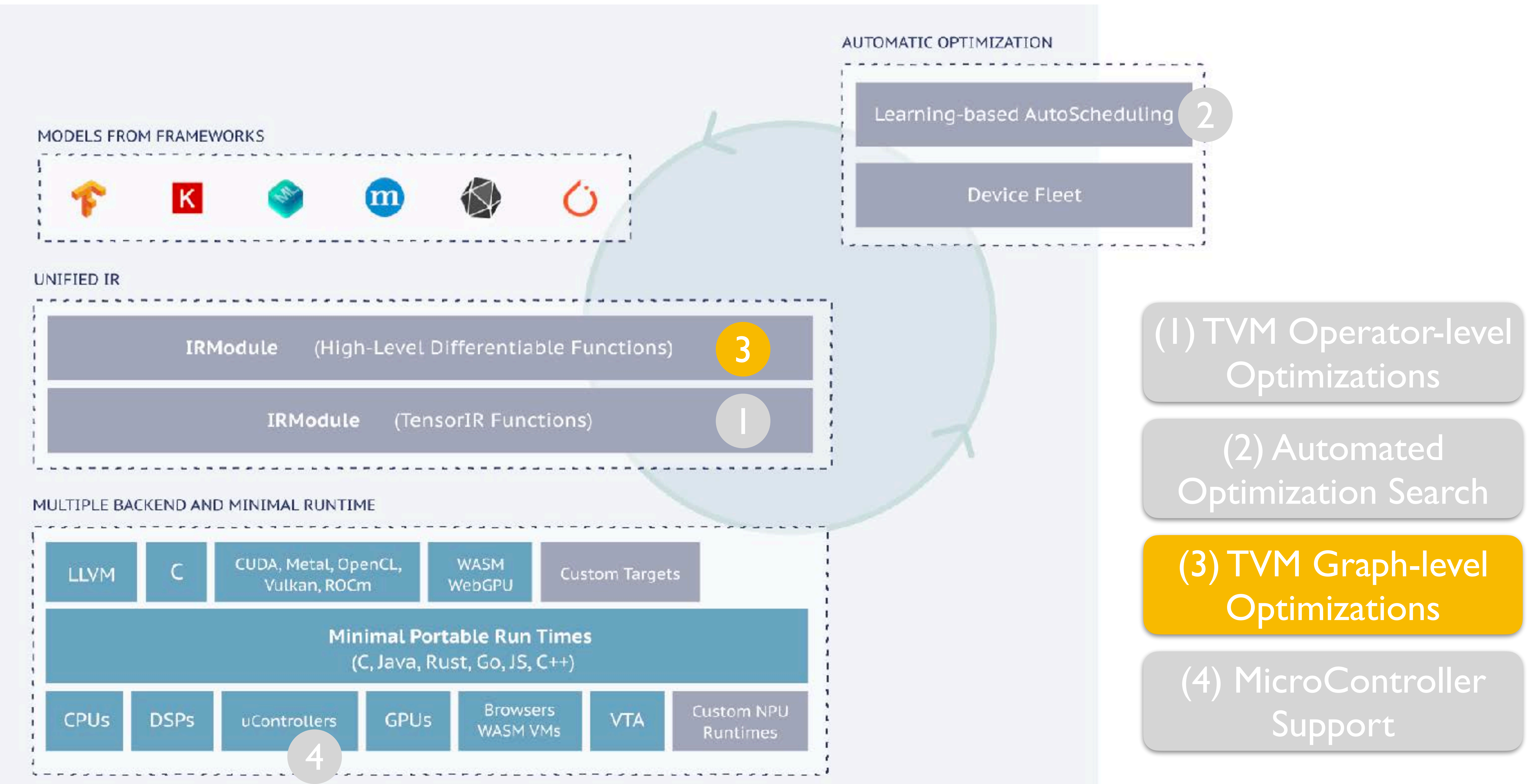
Learning to Optimize Tensor Programs. Chen et al. NeurIPS 18

AutoTVM: ML for ML



Automated solution outperforms vendor deep learning libraries!

Overview of this talk



(1) TVM Operator-level Optimizations

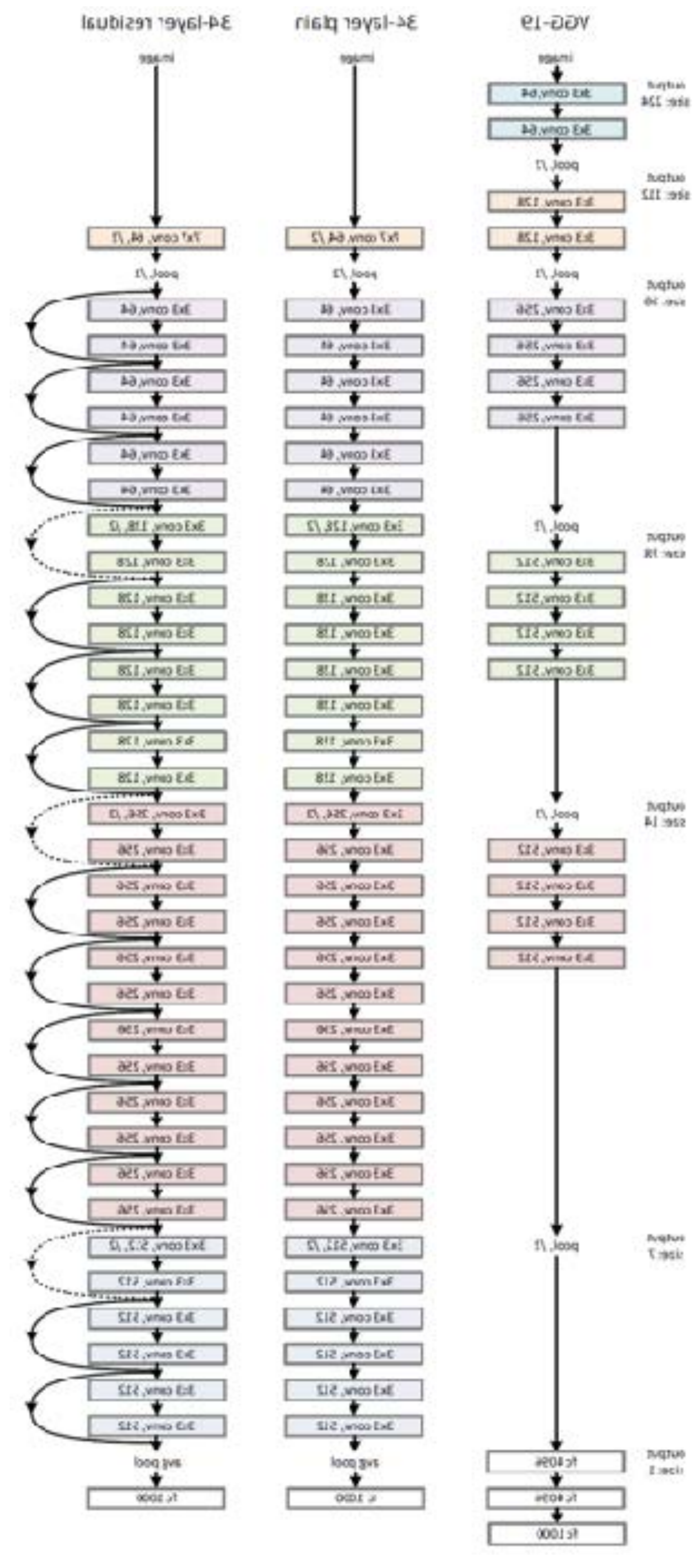
(2) Automated Optimization Search

(3) TVM Graph-level Optimizations

(4) MicroController Support

High-Level Data Flow Graph

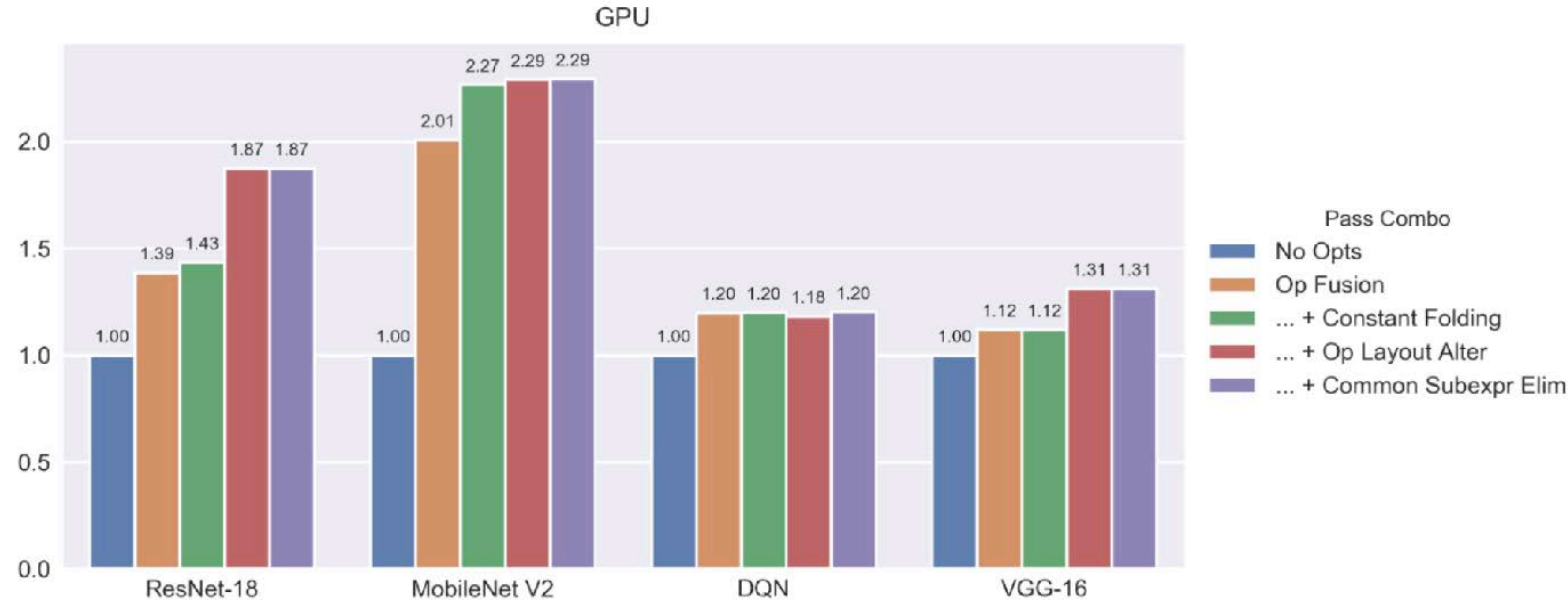
- TVM uses **Relay**, a functional and statically typed IR to describe ML computation
    - Relay implements common features in ML frameworks like quantization, and shape inference as standard compiler optimization passes



Relay: A High-Level Compiler for Deep Learning. Roesch et al. ArXiv 19

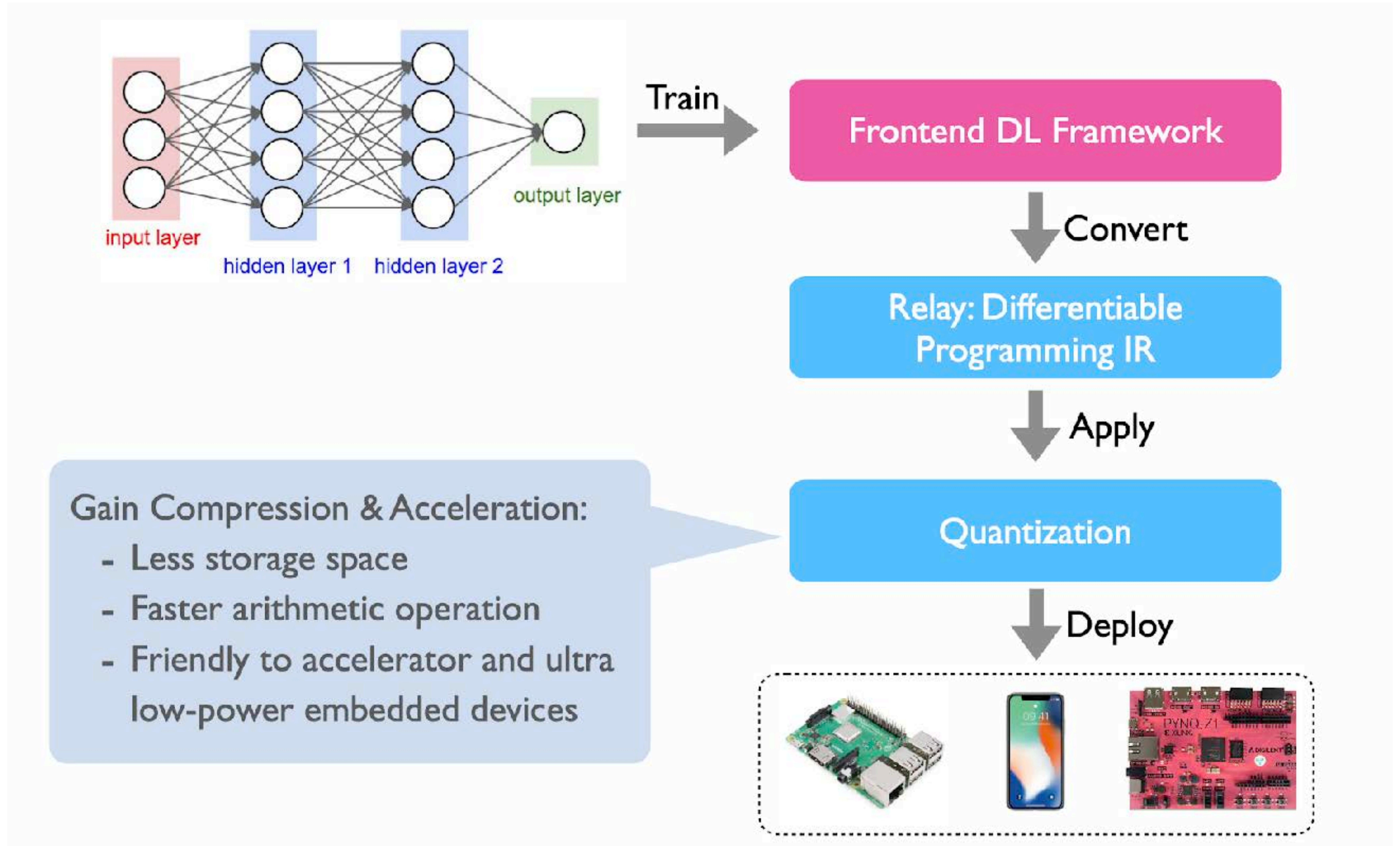
Graph-Level Optimization Passes

- Relay lets us compose a optimization passes to improve performance of DL workloads



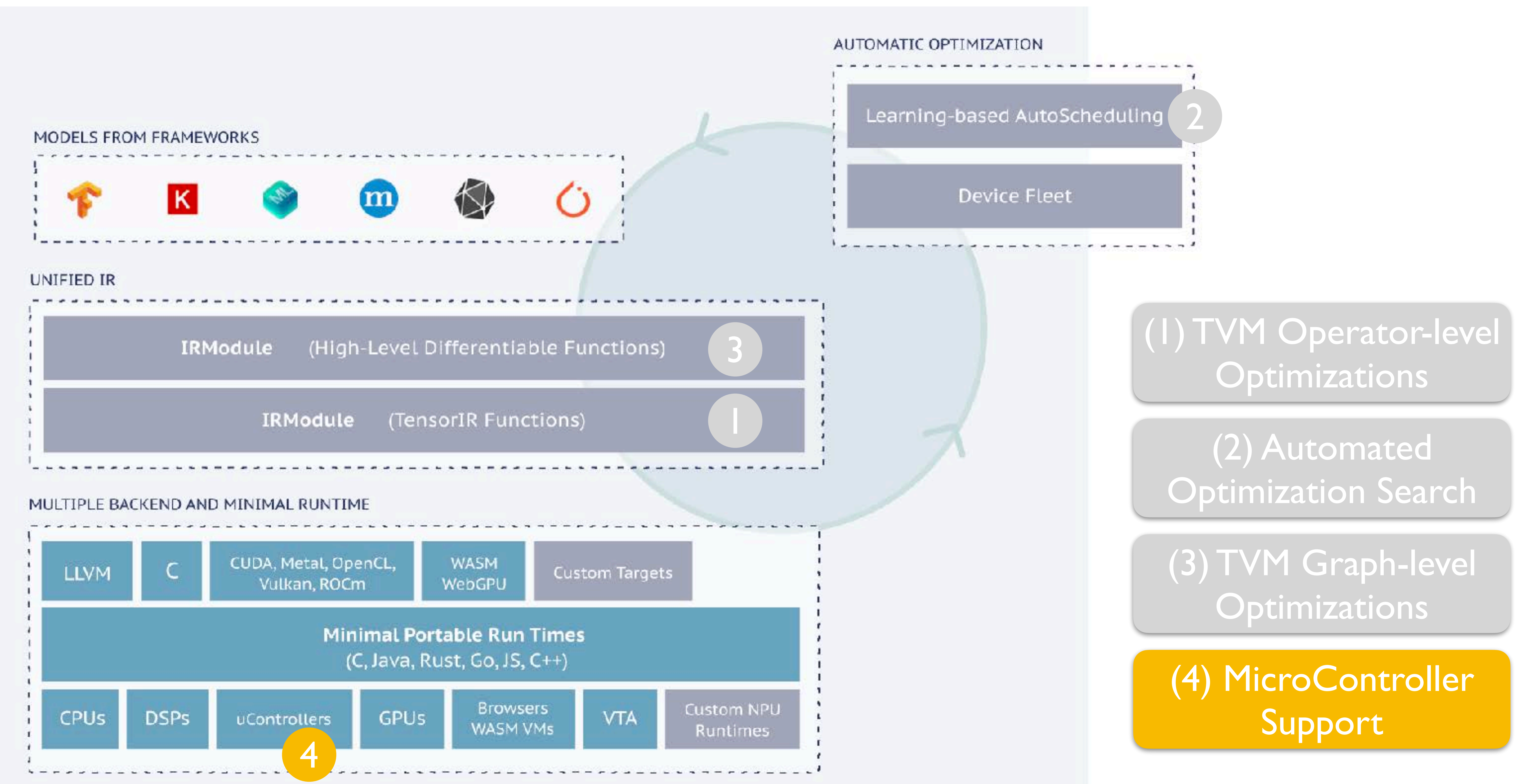
Relay: A High-Level Compiler for Deep Learning. Roesch et al. ArXiv 19

Graph-Level Optimization: Quantization



Credit: Ziheng Jiang, TVM Conference 2019

Overview of this talk



(1) TVM Operator-level Optimizations

(2) Automated Optimization Search

(3) TVM Graph-level Optimizations

(4) MicroController Support

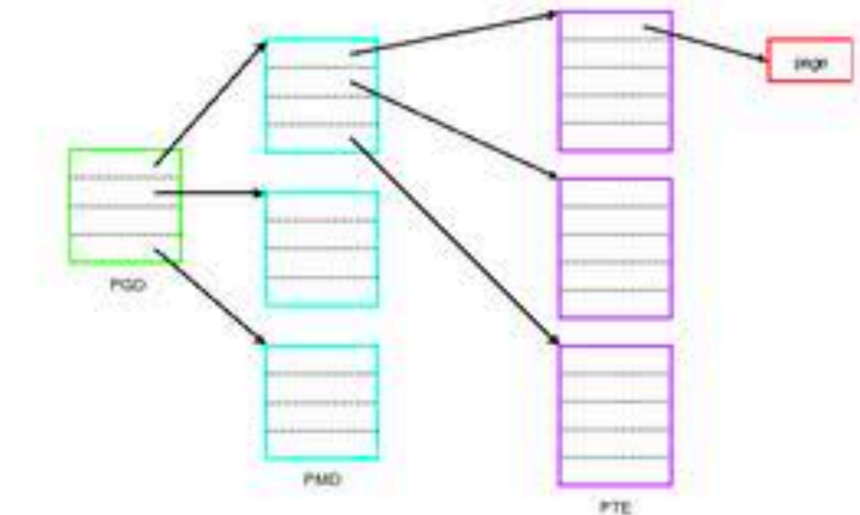
µController challenges

🚫 Operating Systems

- no files, DLLs, .so, memory mapping, kernels

🚫 Virtual Memory

- No malloc, C++ RAII, exceptions, …



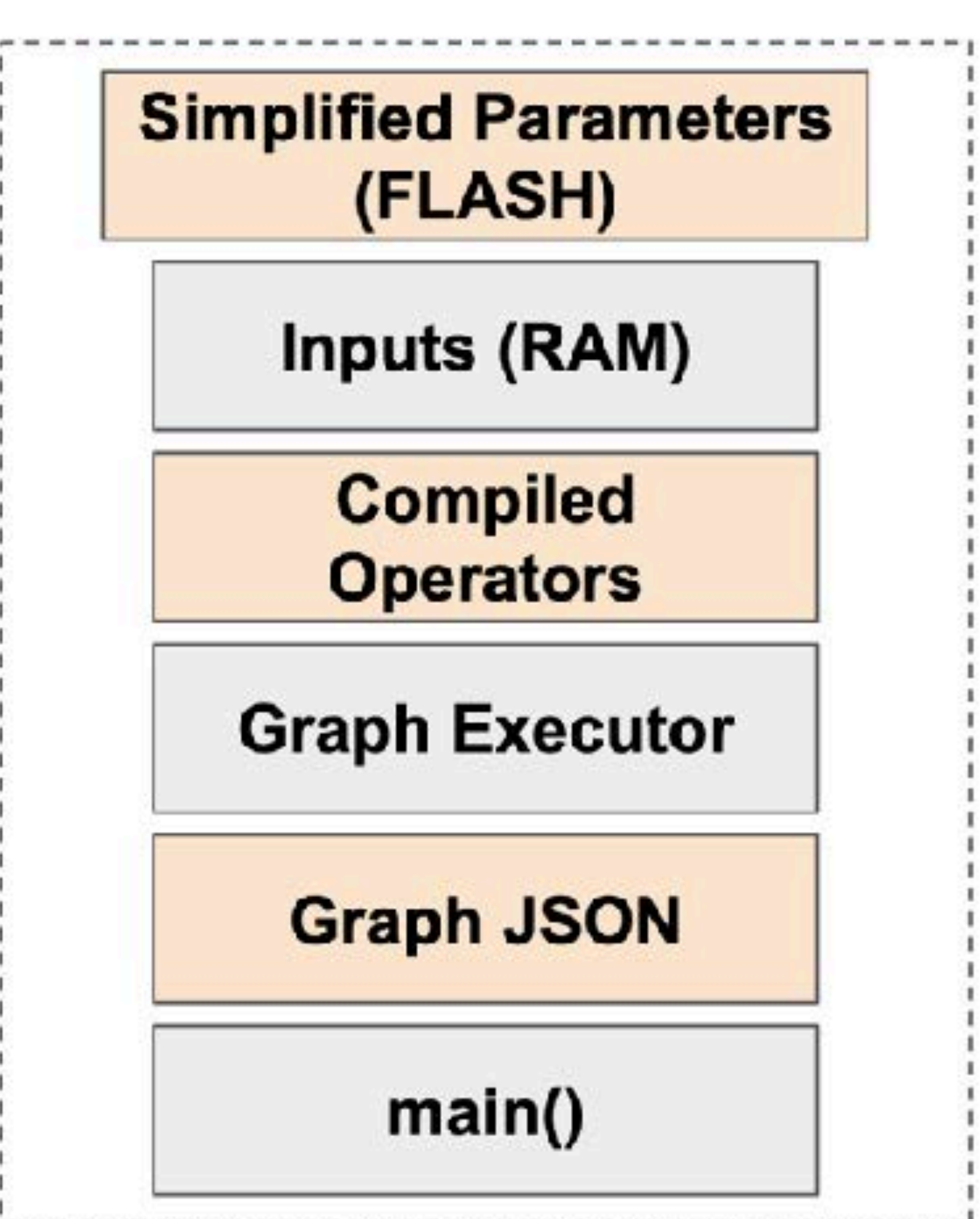
🚫 Advanced Programming Languages

- No Rust or Python required…
  (But we like those and you could use them!)

Really stringent memory and compute budgets!

µTVM: TVM for µControllers

- µTVM can be used with only the standard C library
- C/LLVM IR code-generator for GCC / LLVM
  - This lets us target ARM, MIPS, RISC-V etc. architectures
  - Highly transparent/auditable output
- µTVM leverages RTOS platforms for portability



uTVM 15 min Tutorial

- Carefully prepared by my colleague Andrew Reusch!
- Great way to recap everything we’ve gone through so far into a single demo
- Reproduce it here: https://github.com/areusch/microtvm-blogpost-eval

uTVM - Things to look forward to

Heterogeneous execution to accelerators (e.g. Ethos U-55)

More flexible memory planner

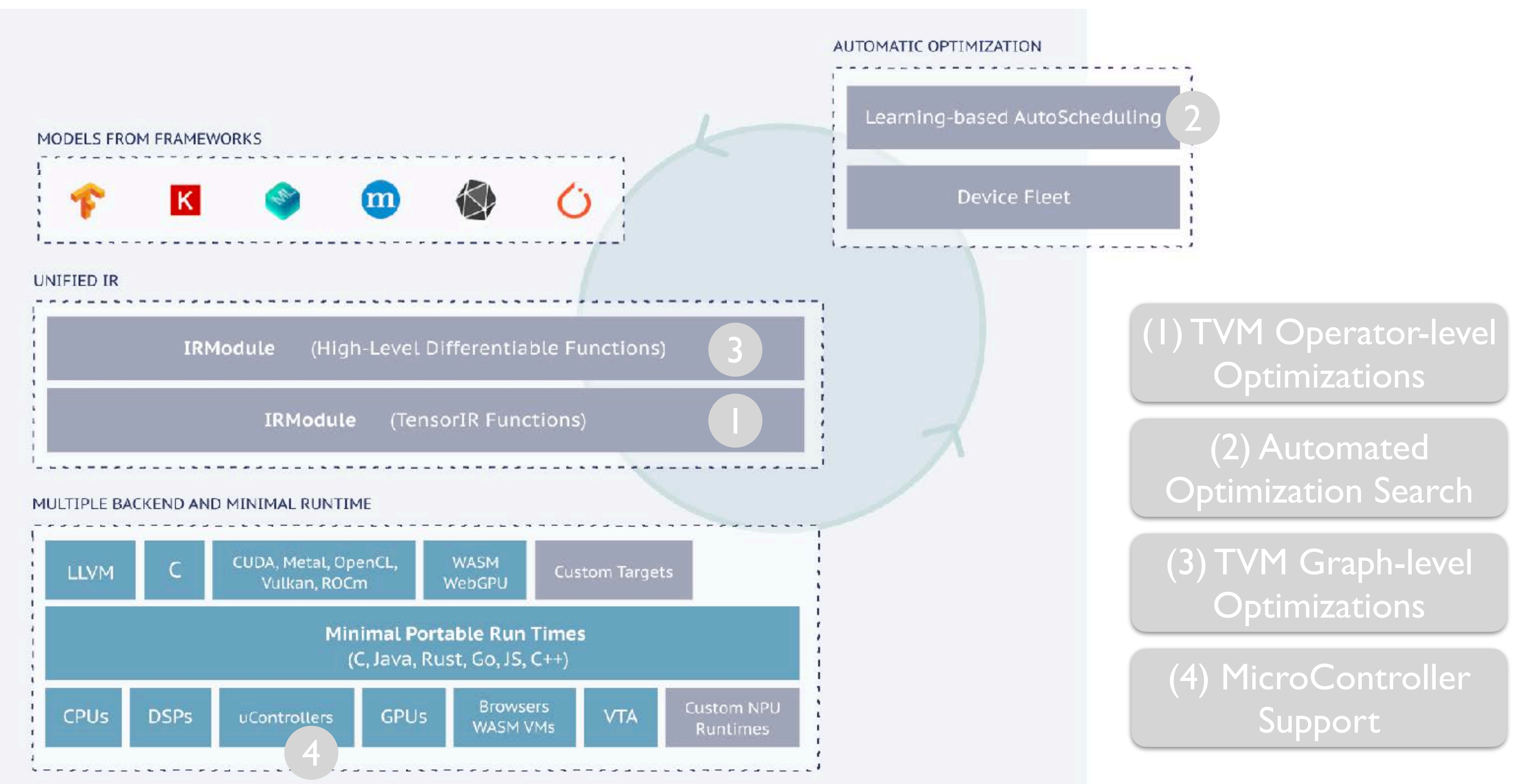
Multi-ISA support (e.g. RISC-V)

Increased model coverage

Improve developer experience (generate Arduino, Zephyr projects)

> For more details, see the MicroTVM Roadmap

Overview of this talk



(1) TVM Operator-level Optimizations

(2) Automated Optimization Search

(3) TVM Graph-level Optimizations

(4) MicroController Support

A lot of topics not covered here…

- RPC communication mechanism for programming edge devices
- GPU, tensor core support
- Support for hardware accelerators, FPGAs
- Sparse operator support
- Int 4 quantization, ultra low bit width quantization
- Training, auto differentiation

Where to learn more about TVM?

- TVM Website is a good starting point: https://tvm.apache.org/
  - Documentation, tutorials, very active discuss forums
- We have hosted 3 yearly TVM conferences!
  - Great resource for past talks and presentations from academia/industry
  - Lots of very encouraging success stories

Accelerate time-to-market of ML models while reducing R&D costs

Automate benchmarking, optimization, and packaging

Eliminate time spent conforming to vendor specific software stacks

Build your model once, run anywhere

Cross platform portability

One platform for all your cloud, mobile and embedded platforms

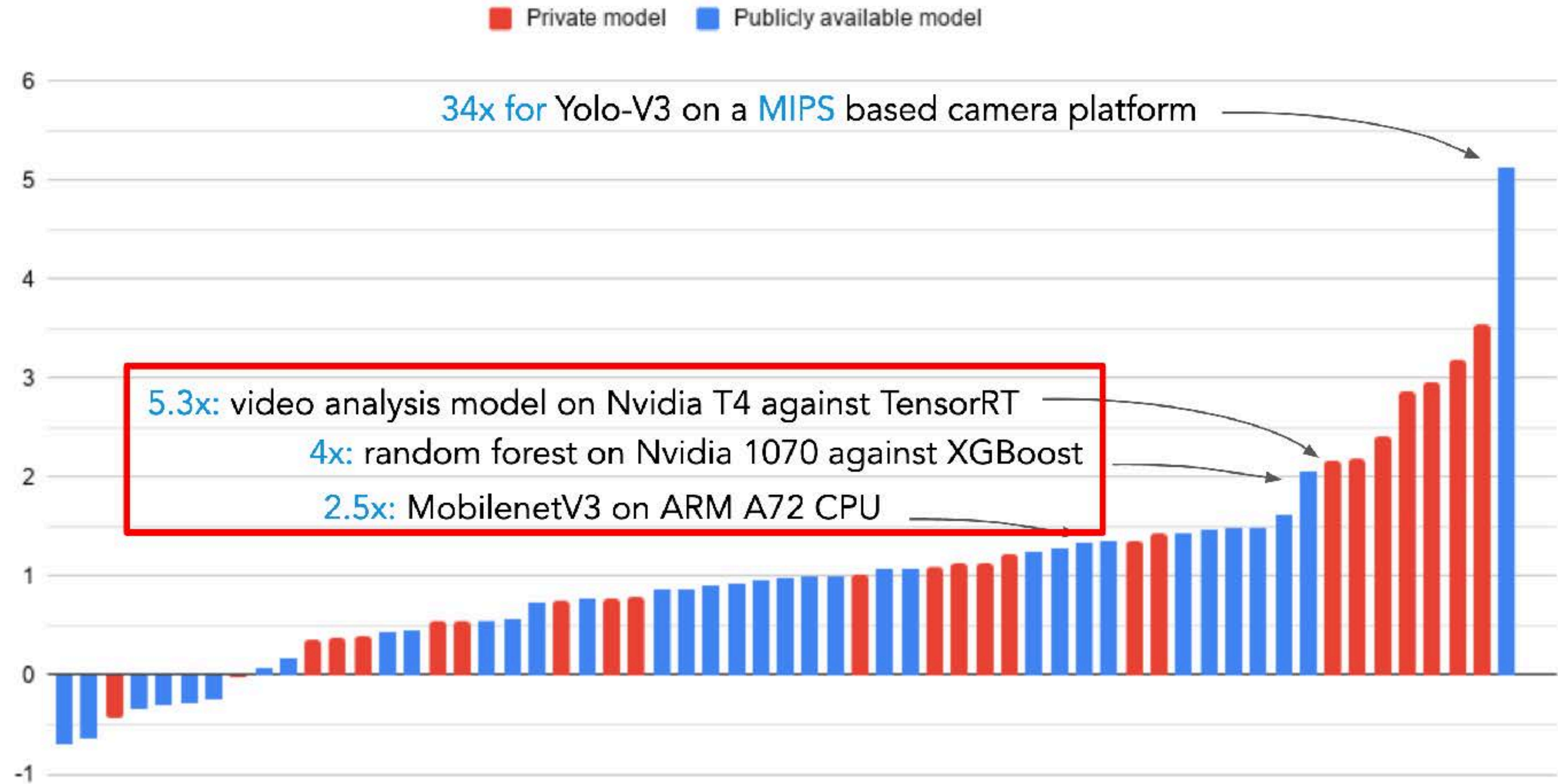
Cut operational ML costs

Find the right platform to optimize $/inference

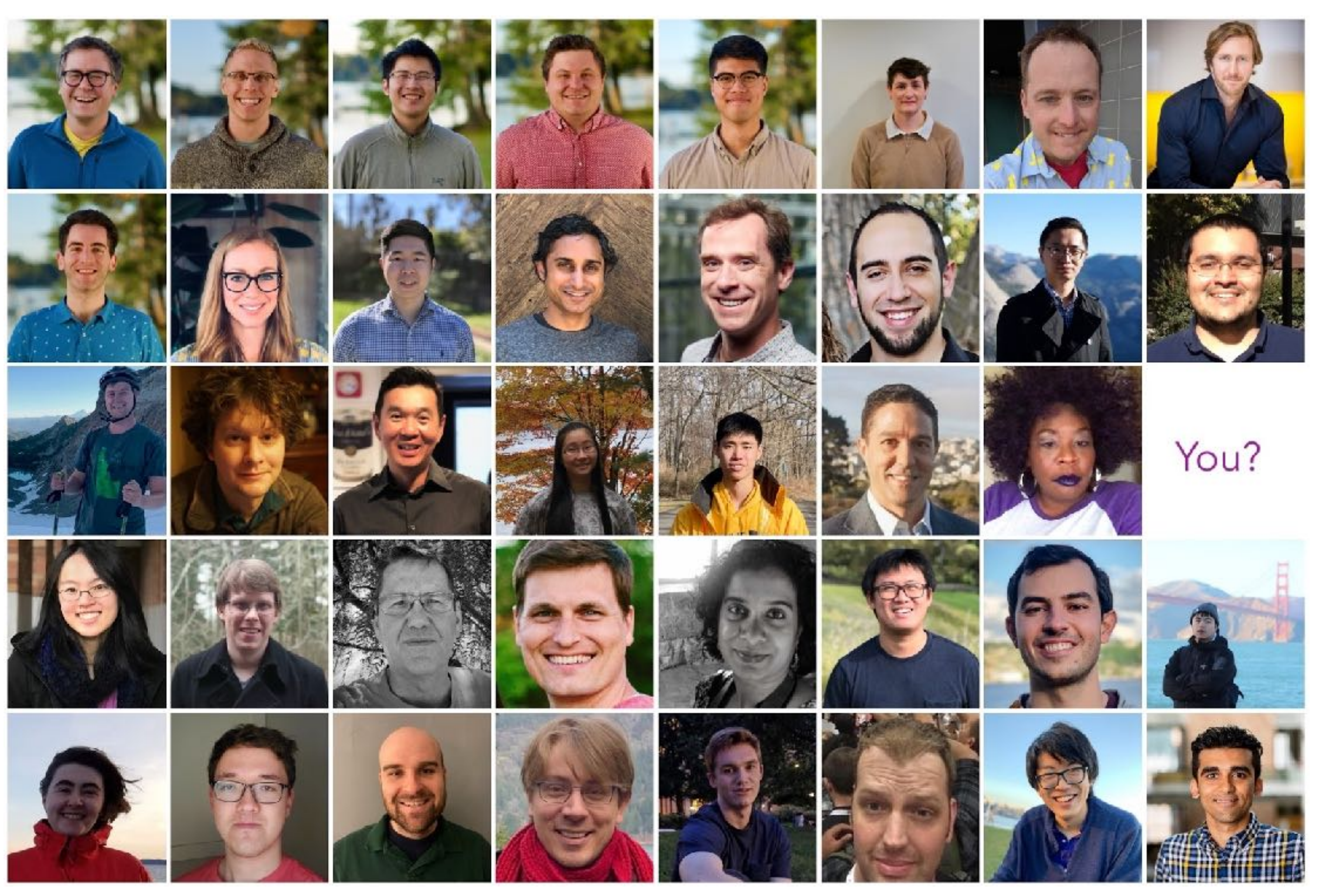
Better utilize your existing hardware by applying ML to your ML

Enable previously infeasible ML applications

Applying TVM optimizations



… and we're hiring!



https://octoml.ai/careers

Resources

- Good reads on TVM
  - TVM Paper, Chen et al., OSDI 2018: https://arxiv.org/abs/1802.04799
  - AutoTVM Paper, Chen et al., NeurIPS 2018: https://arxiv.org/abs/1805.08166
  - Relay Paper, Roesch et al., ArXiv 2019: https://arxiv.org/abs/1904.08368
  - VTA Paper, Moreau et al., IEEE Micro 2019: https://arxiv.org/abs/1807.04188
  - Ansor Paper, Zheng et al., OSDI 2020: https://arxiv.org/abs/2006.06762
  - Quantization Paper, Jain et al., ArXiv 2020: https://arxiv.org/pdf/2006.10226.pdf

Resources

- Tutorials:
  - Optimize GEMM on CPU to get 200x speedup
  - Automatically optimize a network on ARM CPU
  - AutoQuantize a neural network in TVM
  - Micro TVM tutorial
- Azuresphere Demo Blogpost

TVM is a community effort, and much of this work credit goes to Tianqi Chen, Jared Roesch, Andrew Reusch and many amazing TVM contributors and collaborators at the University of Washington, OctoML, ARM (Ramana Radhakrishnan & team), AWS, Berkeley etc.