PAUL G. ALLEN SCHOOL
OF COMPUTER SCIENCE & ENGINEERING

Automatic Code Generation
TVM Stack

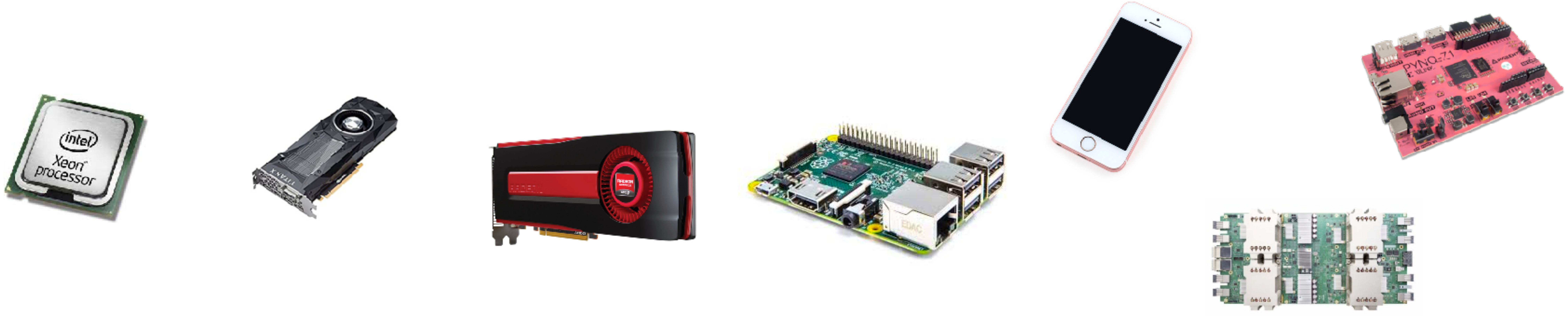
CSE 599W Spring

TVM stack is an active project by saml.cs.washington.edu
and many partners in the open source community

The Gap between Framework and Hardware

Frameworks Caffe2 CNTK

Each backend to a new software stack on top of it!



Compiler's Perspective to this Problem

Frameworks

Caffe2

CNTK

Express computation

Intermediate Representation (s)

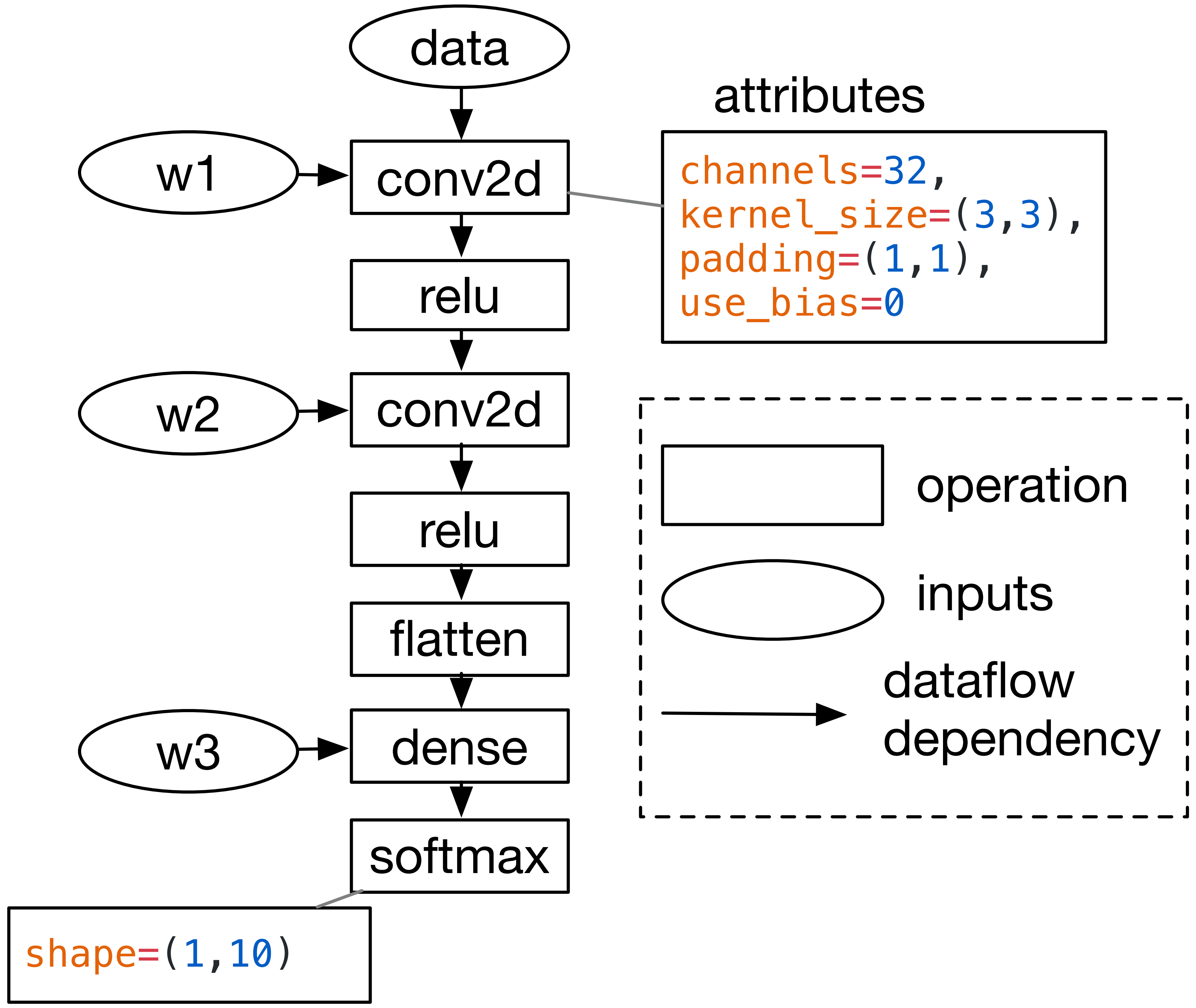
Reusable Optimizations

Code generation

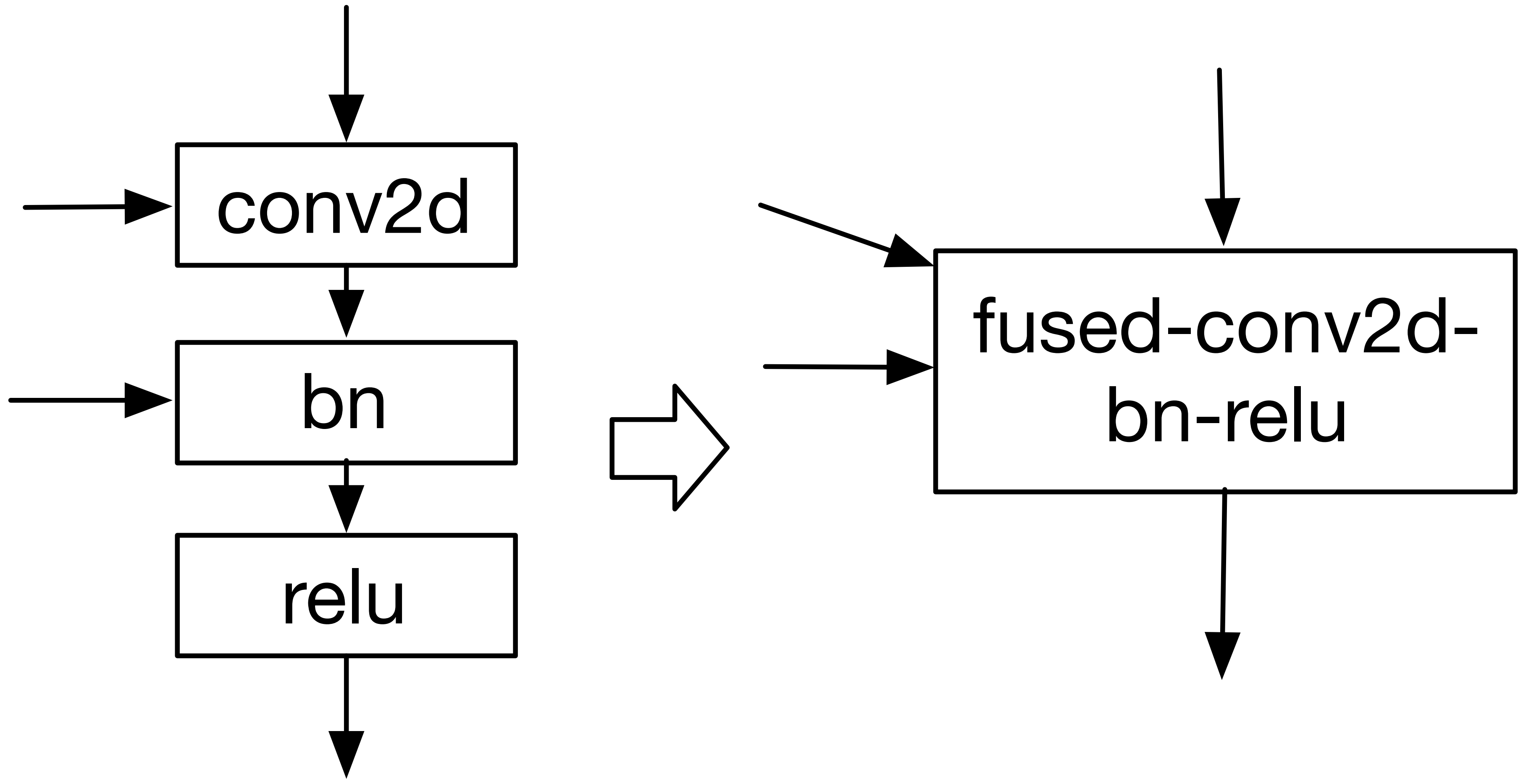
Hardware

Computational Graph as IR

Represent High level Deep Learning Computations



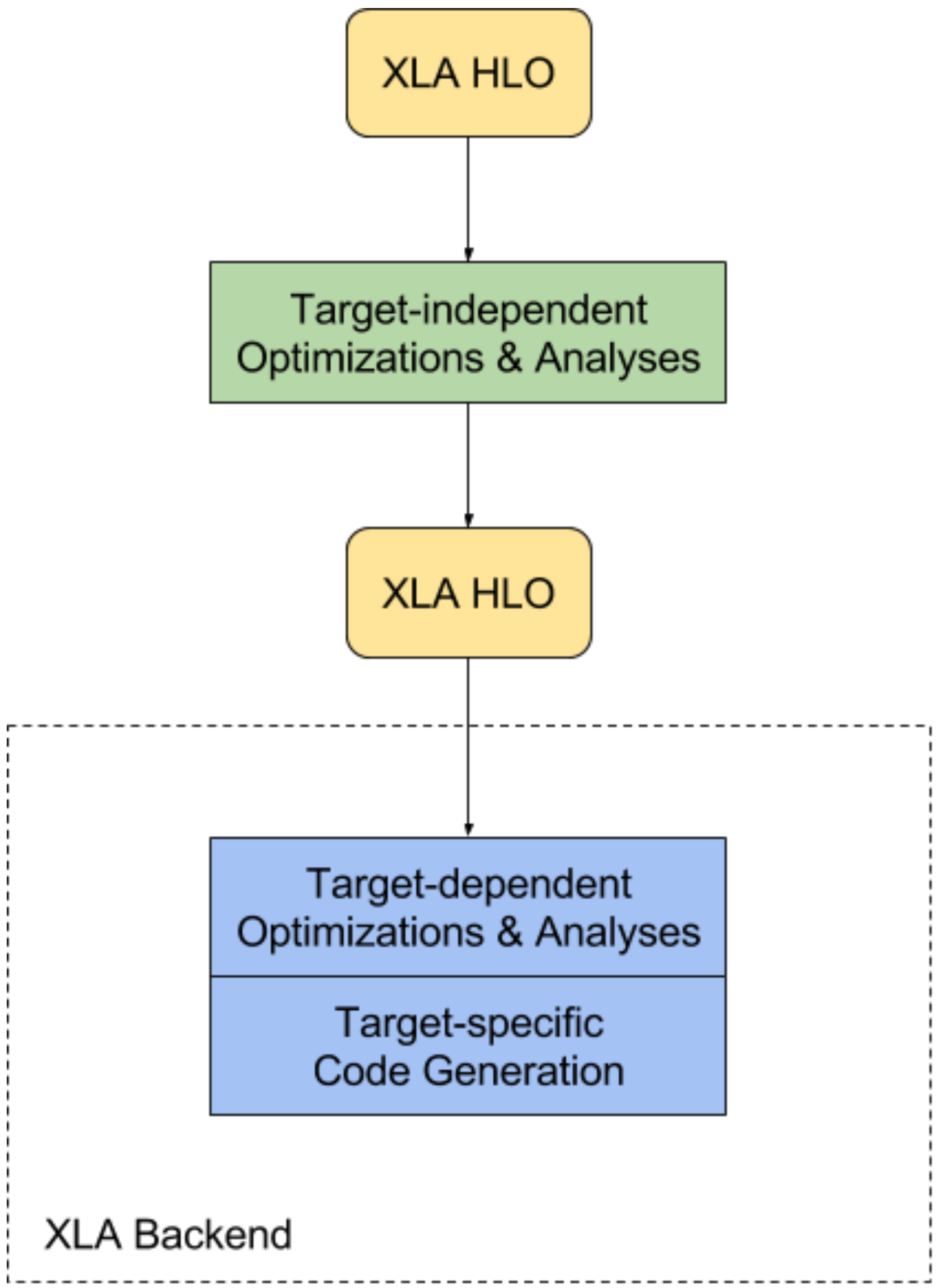
Effective Equivalent Transformations to Optimize the Graph



Approach taken by: TensorFlow XLA, Intel NGraph, Nvidia TensorRT

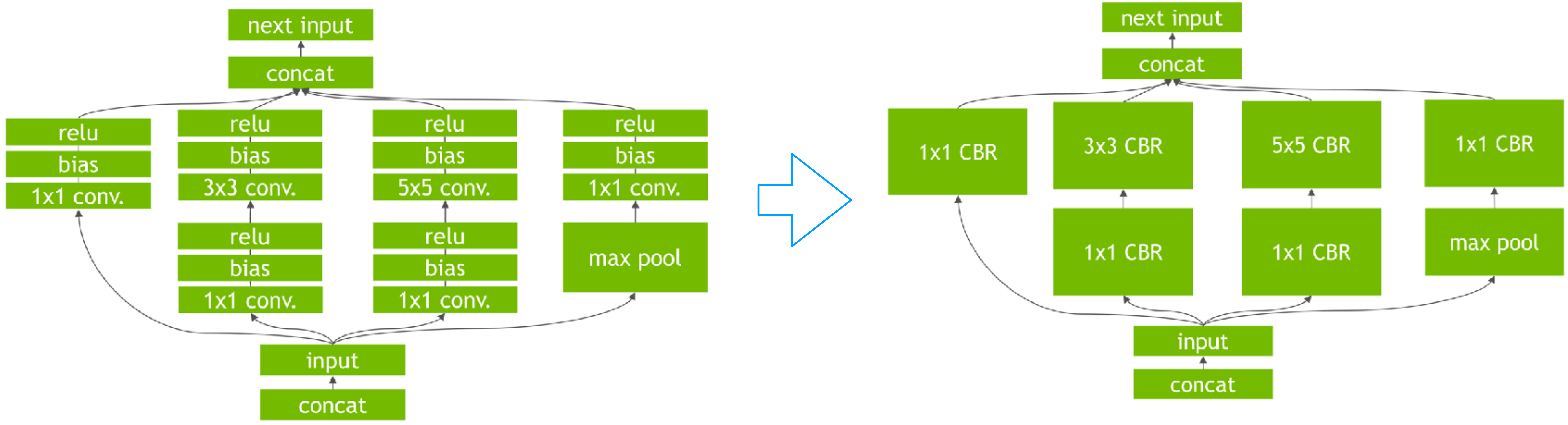
XLA: Tensorflow Compiler

- Constant shape dimension
- Data layout is specific
- Operations are low level tensor primitives
  - Map
  - Broadcast
  - Reduce
  - Convolution
  - ReduceWindow
  - ...



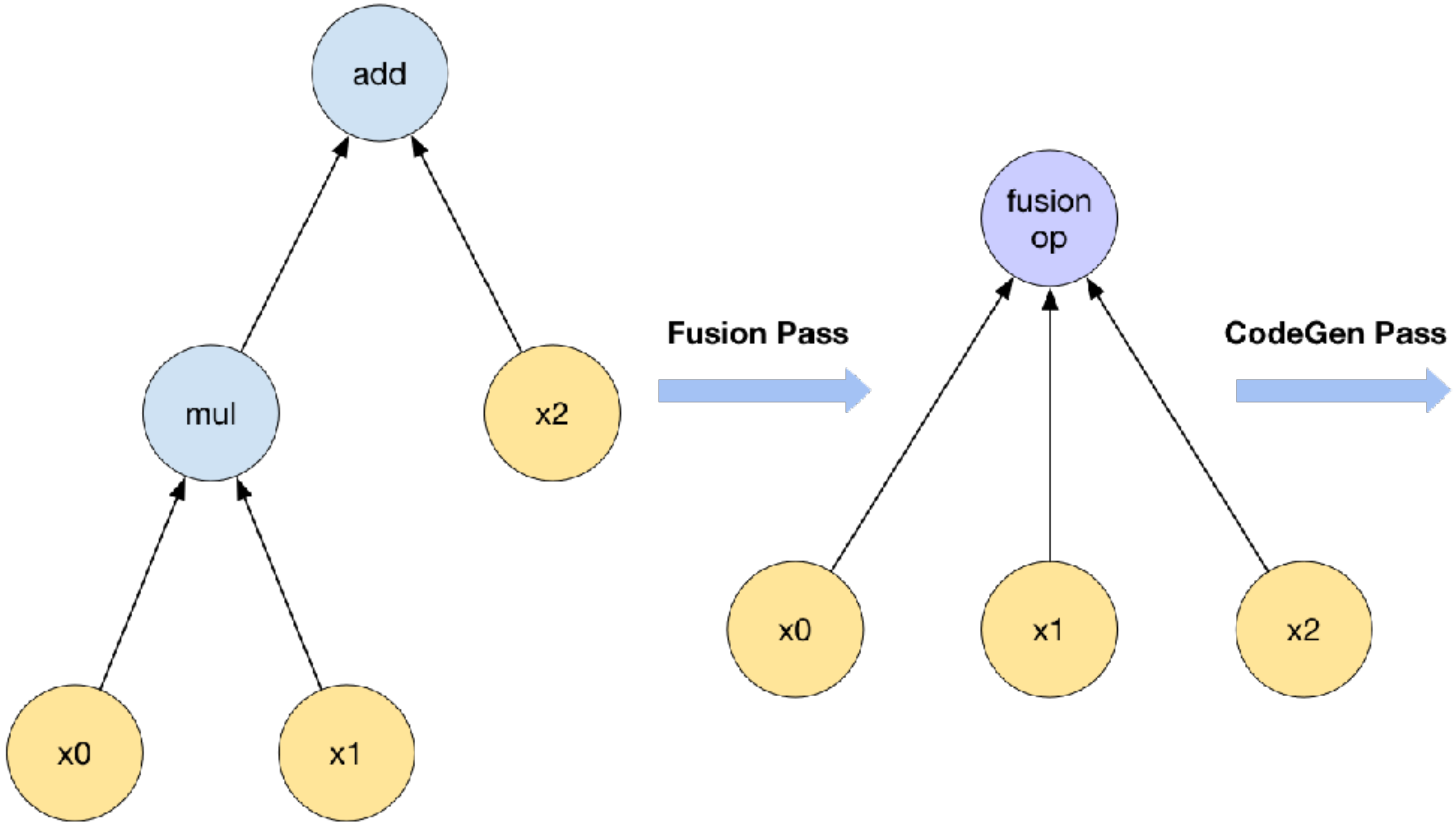
Source: Google

TensorRT: Rule based Fusion



Source: Nvidia

Simple Graph-based Element-wise Kernel Generator




```
extern "C" __global__ fusion_kernel (uint32_t num_element,
    float *x0, float *x1, float *x2, float *y) {
  int global_idx = blockIdx.x * blockDim.x + threadIdx.x;
  if (global_idx < num_element)
    y[global_idx] = (x0[global_idx] * x1[global_idx]) + x2[global_idx];
}
```

Two min Discussion

What are pros and cons of computational graph based approach

The Remaining Gap

Frameworks Caffe2 CNTK

Computational Graph Optimization

need to build and optimize operators for each hardware, variant of layout, precision, threading pattern …

Hardware

Tensor Level Optimizations

Frameworks

Caffe2

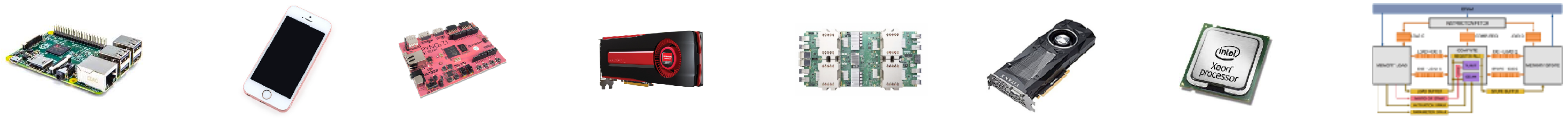
CNTK

Computational Graph Optimization

Tensor Expression Language

```
C = t.compute((m, n),
    lambda i, j: t.sum(A[i, k] * B[j, k], axis=k))
```

Hardware



Tensor Index Expression

Compute C = dot(A, B.T)

```
import tvm

m, n, h = tvm.var('m'), tvm.var('n'), tvm.var('h')
A = tvm.placeholder((m, h), name='A')
B = tvm.placeholder((n, h), name=‘B')


k = tvm.reduce_axis((0, h), name=‘k')
C = tvm.compute((m, n), lambda i, j: tvm.sum(A[i, k] * B[j, k], axis=k))
```

Inputs

Shape of C

Computation Rule

Tensor Expressions are Expressive

Affine Transformation

```
out = tvm.compute((n, m), lambda i, j: tvm.sum(data[i, k] * w[j, k], k))
out = tvm.compute((n, m), lambda i, j: out[i, j] + bias[i])
```

Convolution

```
out = tvm.compute((c, h, w),
  lambda i, x, y: tvm.sum(data[kc,x+kx,y+ky] * w[i,kx,ky], [kx,ky,kc]))
```

ReLU

```
out = tvm.compute(shape, lambda *i: tvm.max(0, out(*i))
```

Emerging Tools Using Tensor Expression Language

Halide: Image processing language

Loopy: python based kernel generator

TACO: sparse tensor code generator

Tensor Comprehension

Schedule: Tensor Expression to Code

Tensor Expression Language

```
C = t.compute((m, n),
     lambda i, j: t.sum(A[i, k] * B[j, k], axis=k))
```

↓

Schedule Optimizations

↓

Key Idea:
Separation of Compute
and Schedule
introduced by Halide

Hardware

Example Schedule Transformation

```
C = tvm.compute((n,), lambda i: A[i] + B[i])
s = tvm.create_schedule(C.op)
```

```
for (int i = 0; i < n; ++i) {
  C[i] = A[i] + B[i];
}
```

Example Schedule Transformation

```
C = tvm.compute((n,), lambda i: A[i] + B[i])
s = tvm.create_schedule(C.op)
xo, xi = s[C].split(s[C].axis[0], factor=32)
```

```
for (int xo = 0; xo < ceil(n / 32); ++xo) {
  for (int xi = 0; xi < 32; ++xi) {
    int i = xo * 32 + xi;
    if (i < n) {
      C[i] = A[i] + B[i];
    }
  }
}
```

Example Schedule Transformation

```
C = tvm.compute((n,), lambda i: A[i] + B[i])
s = tvm.create_schedule(C.op)
xo, xi = s[C].split(s[C].axis[0], factor=32)
s[C].recorder(xi, xo)
```

```
for (int xi = 0; xi < 32; ++xi) {
  for (int xo = 0; xo < ceil(n / 32); ++xo) {
    int i = xo * 32 + xi;
    if (i < n) {
      C[i] = A[i] + B[i];
    }
  }
}
```

Example Schedule Transformation

```
C = tvm.compute((n,), lambda i: A[i] + B[i])
s = tvm.create_schedule(C.op)
xo, xi = s[C].split(s[C].axis[0], factor=32)
s[C].recorder(xi, xo)
s[C].bind(xo, tvm.thread_axis("blockIdx.x")
s[C].bind(xi, tvm.thread_axis("threadIdx.x")
```

```
int i = threadIdx.x * 32 + blockIdx.x;
if (i < n) {
   C[i] = A[i] + B[i];
}
```

Key Challenge: Good Space of Schedule

Should contain any knobs that produces a logically equivalent program that runs well on backend models

Must contain the common manual optimization patterns

Need to actively evolve to incorporate new techniques

Two Min Discussions

What are useful program transformation that can be used a schedule primitive

TVM Schedule Primitives

Still constantly evolving

Tensor Expression Language

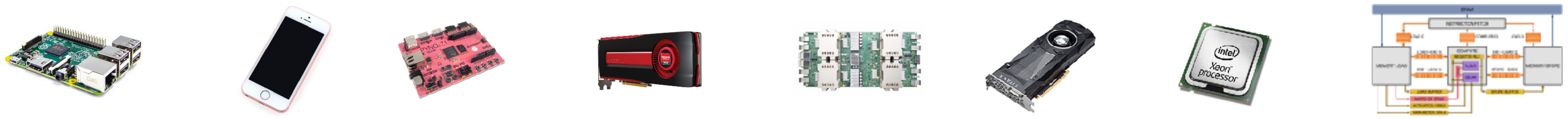
Primitives in prior works
Halide, Loopy

- Loop Transformations
- Thread Bindings
- Cache Locality

New primitives for GPU
Accelerators

- Thread Cooperation
- Tensorization
- Latency Hiding
- ...

Hardware



Schedule Space Exploration

Tensor Expression Language

```
C = t.compute((m, n),
    lambda i, j: t.sum(A[i, k] * B[j, k], axis=k))
```

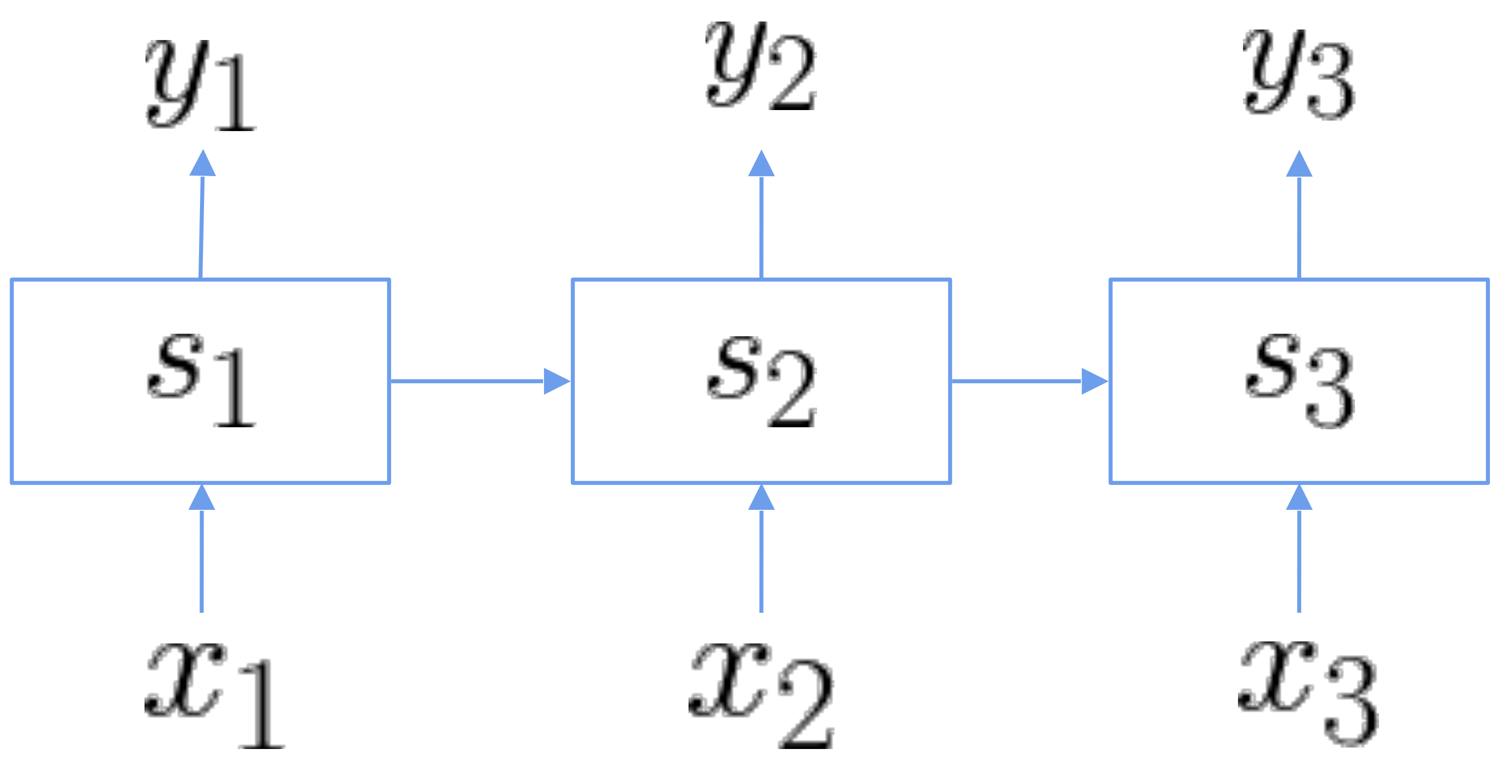
Schedule 1	Schedule 2	Schedule 3	....
↓	↓	↓	
Kernel 1	Kernel 2	Kernel 3	....

Make use of an AutoTuner

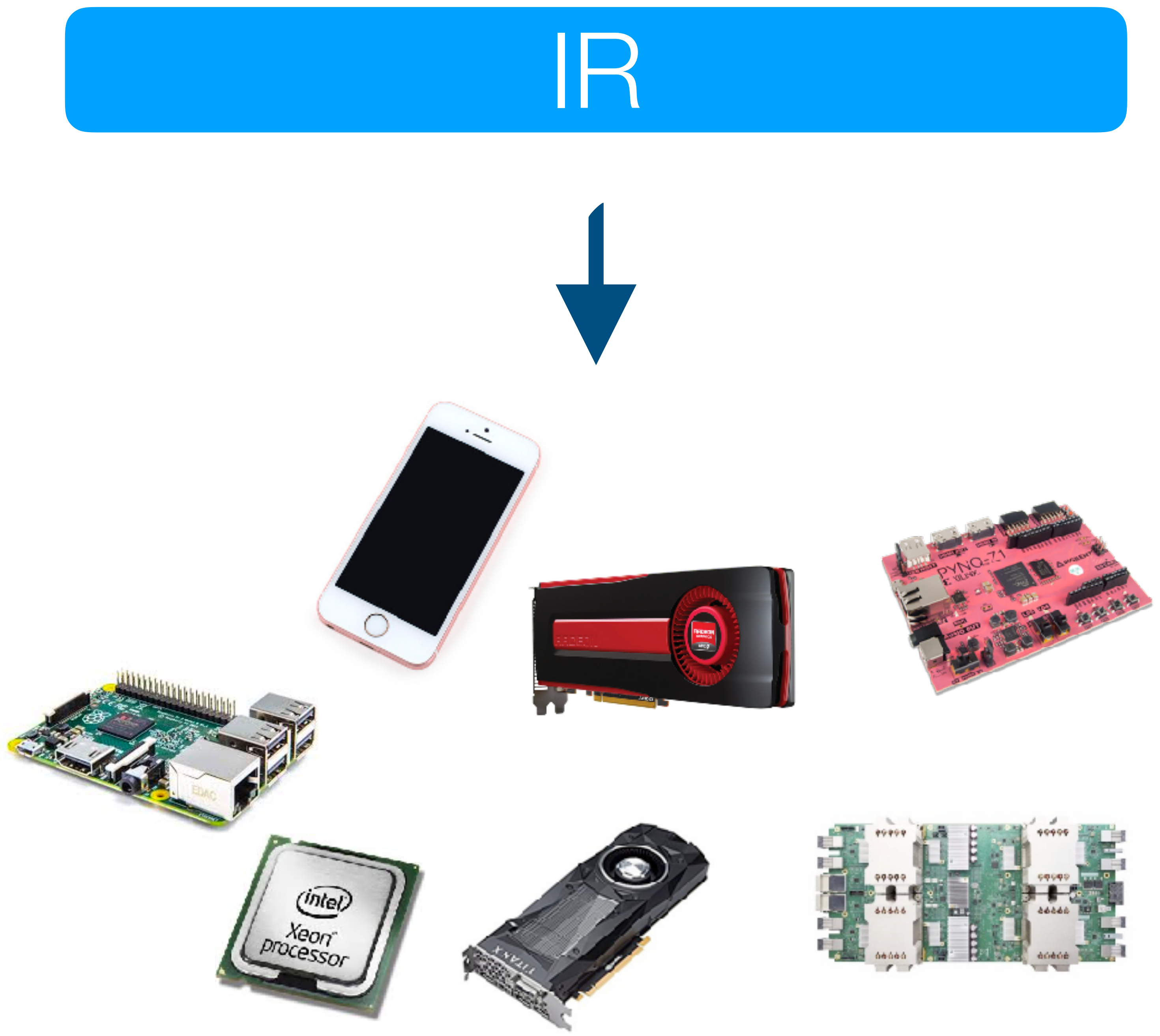
Extending Compute Primitives

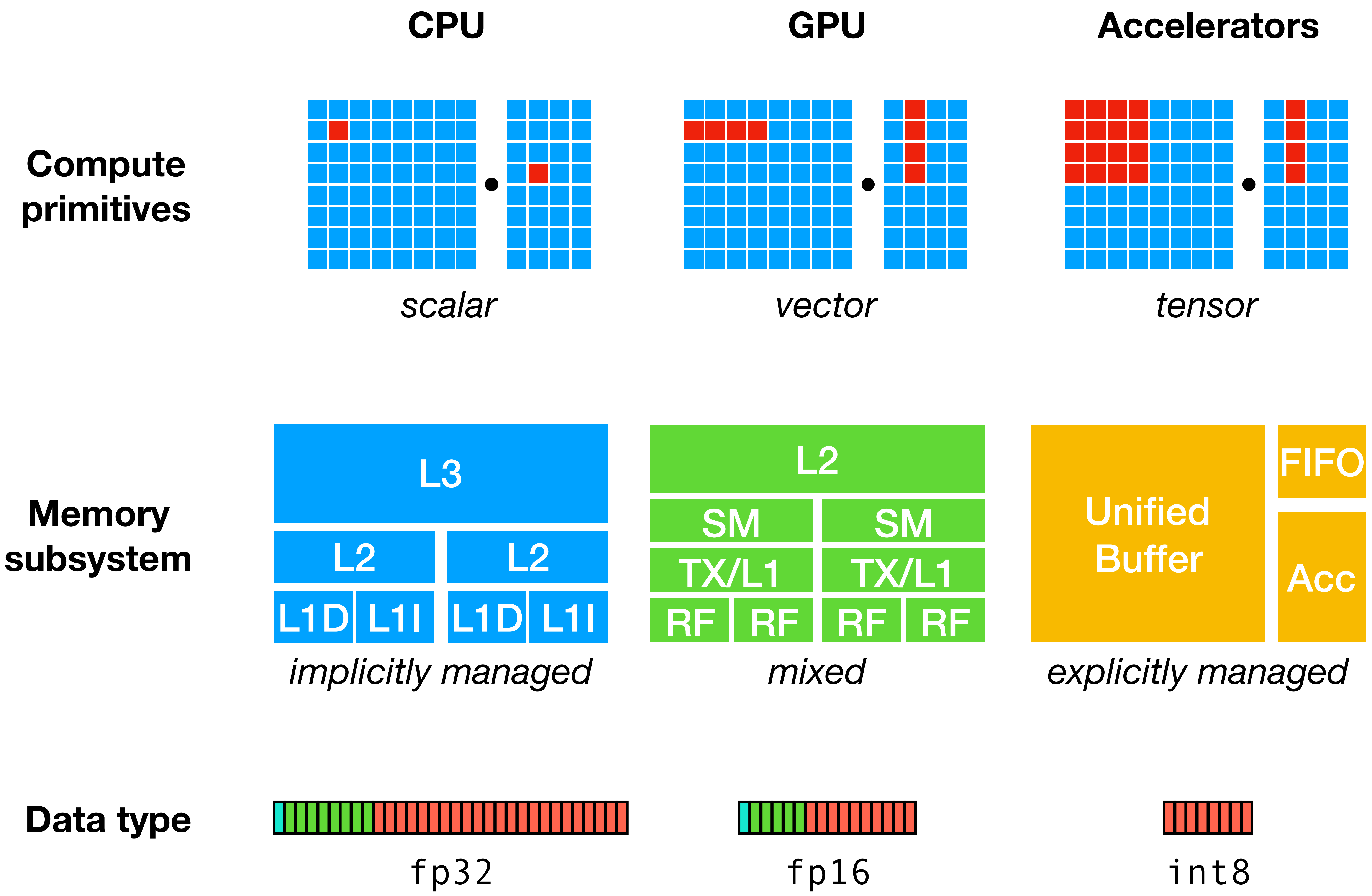
Symbolic Loop: Y = cumsum(X)

```
import tvm

m = tvm.var("m")
n = tvm.var("n")
X = tvm.placeholder((m, n), name="X")

s_state = tvm.placeholder((m, n))
s_init = tvm.compute((1, n), lambda _, i: X[0, i])
s_update = tvm.compute((m, n), lambda t, i: s_state[t-1, i] + X[t, i])

Y = tvm.scan(s_init, s_update, s_state, inputs=[X])
```



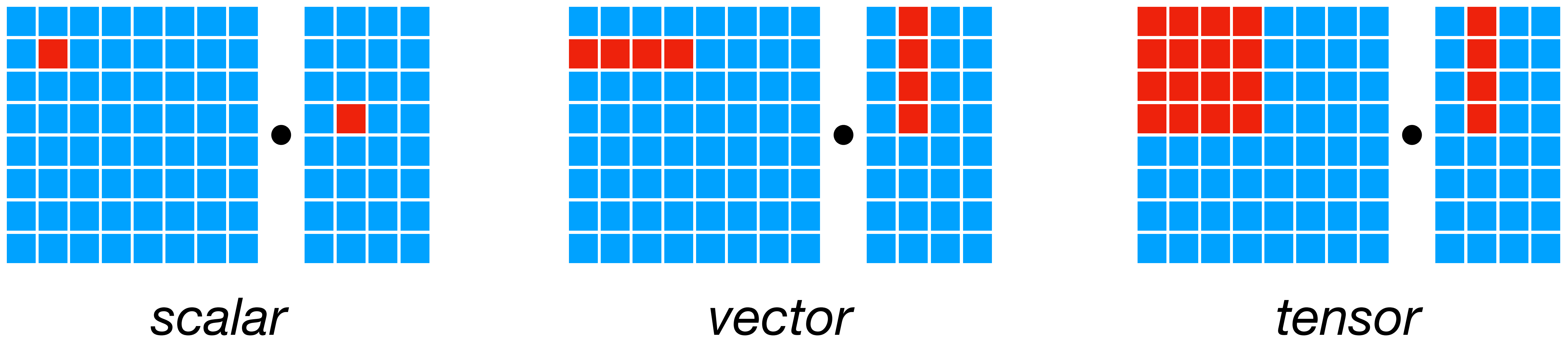
New Hardware Challenges





Tensorization Challenge

Compute primitives



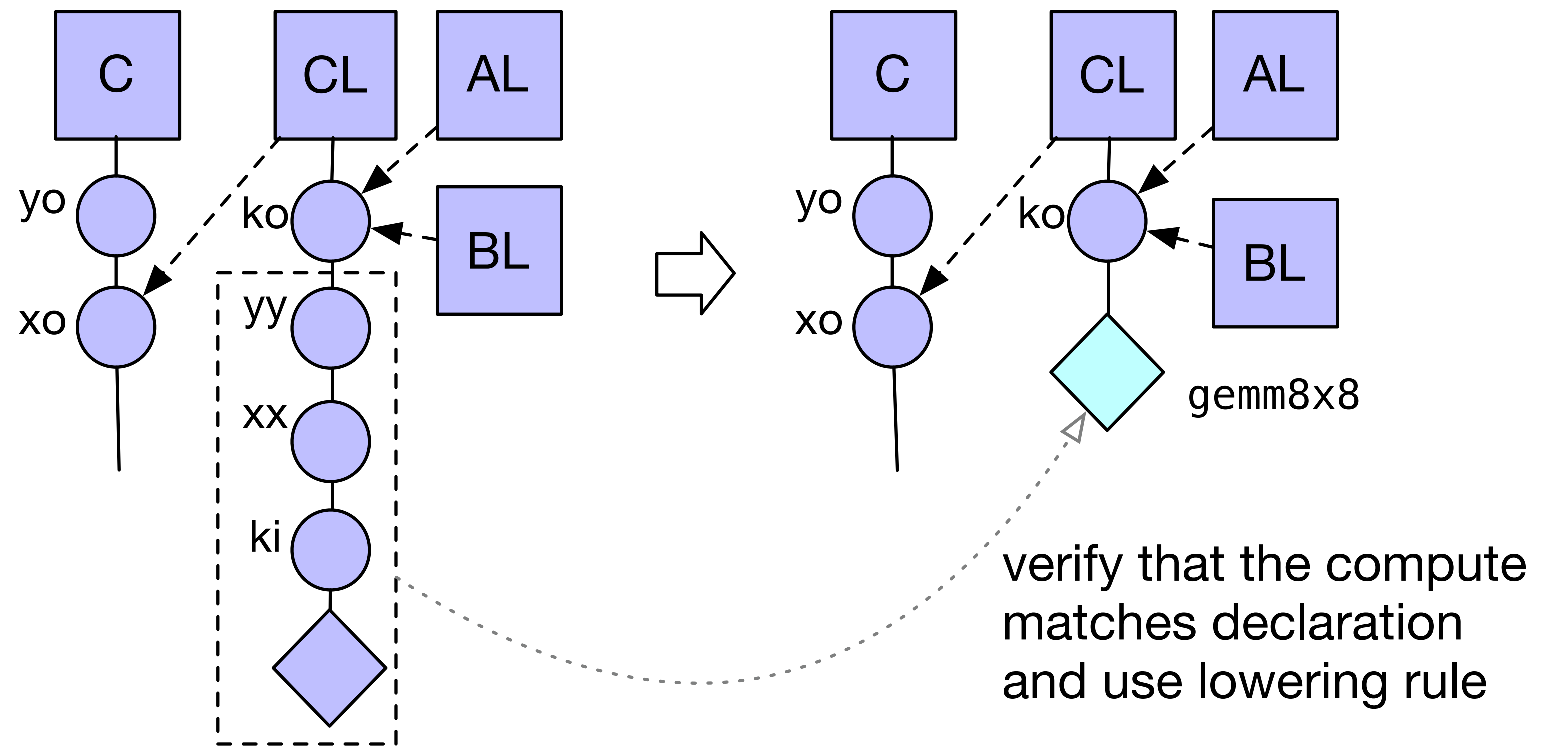
Hardware designer: declare tensor instruction interface

```
w, x = t.placeholder((8, 8)), t.placeholder((8, 8))
k = t.reduce_axis((0, 8))
y = t.compute((8, 8), lambda i, j:
              t.sum(w[i, k] * x[j, k], axis=k))

def gemm_intrin_lower(inputs, outputs):
   ww_ptr = inputs[0].access_ptr("r")
   xx_ptr = inputs[1].access_ptr("r")
   zz_ptr = outputs[0].access_ptr("w")
   compute = t.hardware_intrin("gemm8x8", ww_ptr, xx_ptr, zz_ptr)
   reset = t.hardware_intrin("fill_zero", zz_ptr)
   update = t.hardware_intrin("fuse_gemm8x8_add", ww_ptr, xx_ptr, zz_ptr)
   return compute, reset, update

gemm8x8 = t.decl_tensor_intrin(y.op, gemm_intrin_lower)
```

declare behavior

lowering rule to generate hardware intrinsics to carry out the computation

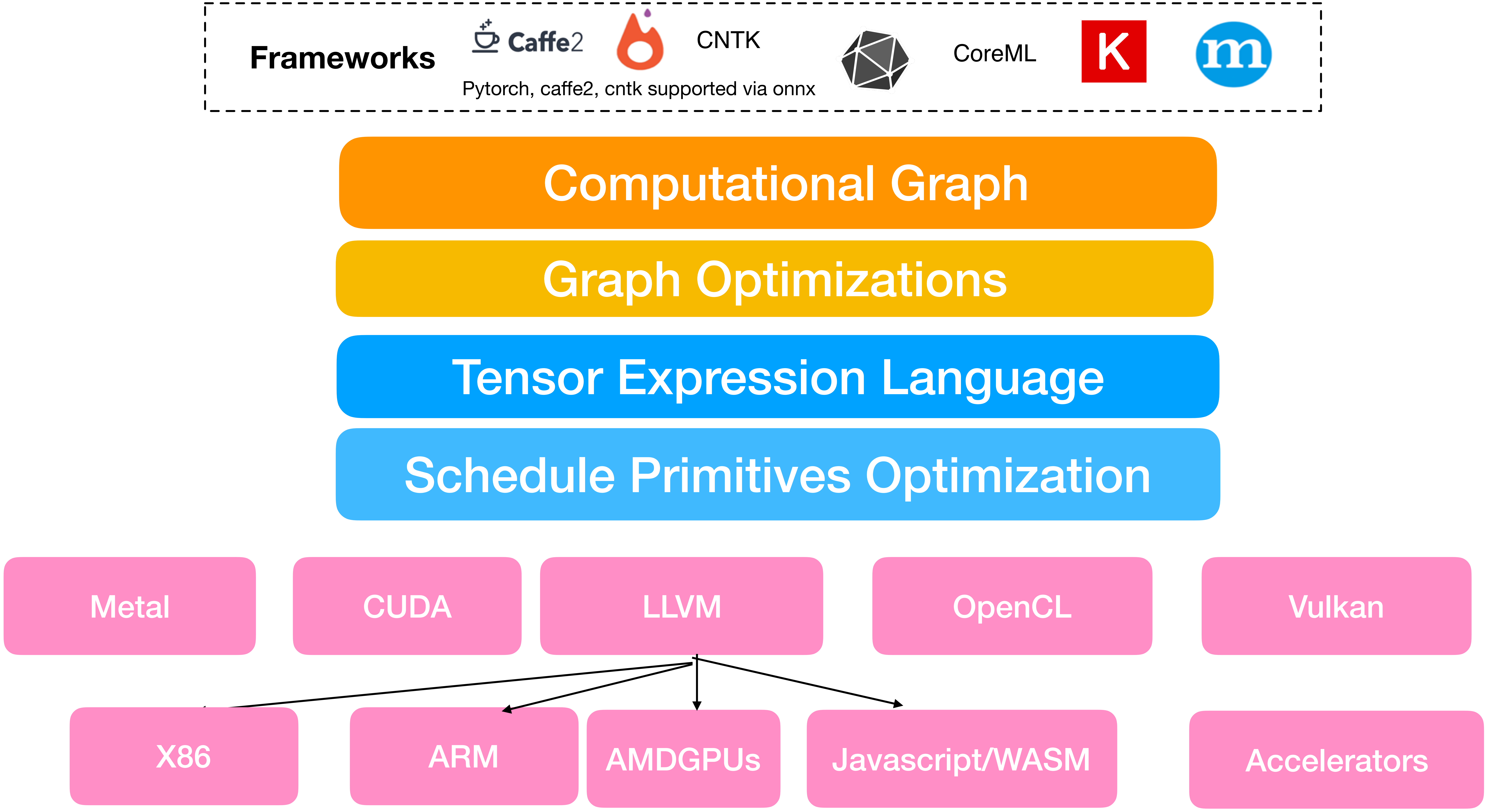
Tensorize: transform program to use tensor instructions



Two Min Discussions

We talked a lot of tensor expression language
what are the possible drawbacks about
what we talked about so far

Global View of TVM Stack



High Level Compilation Frontend

```python
module = runtime.create(graph, lib, tvm.gpu(0))
module.set_input(**params)
module.run(data=data_array)
output = tvm.nd.empty(out_shape, ctx=tvm.gpu(0))
module.get_output(0, output)
```

```python
import tvm
import nnvm.frontend
import nnvm.compiler

graph, params =
nnvm.frontend.from_keras(keras_resnet50)
graph, lib, params =
     nnvm.compiler.build(graph, target)
```

input

Deployable Module

prediction **tabby, tabby cat**

On languages and platforms you choose

Program Your Phone with Python from Your Laptop

RPC Server on Embedded Device

upload module to remote

get remote function

copy data to remote

get remote array handle

run function on remote

get profile statistics back

copy data back to host
for correctness verification

Compiler Stack

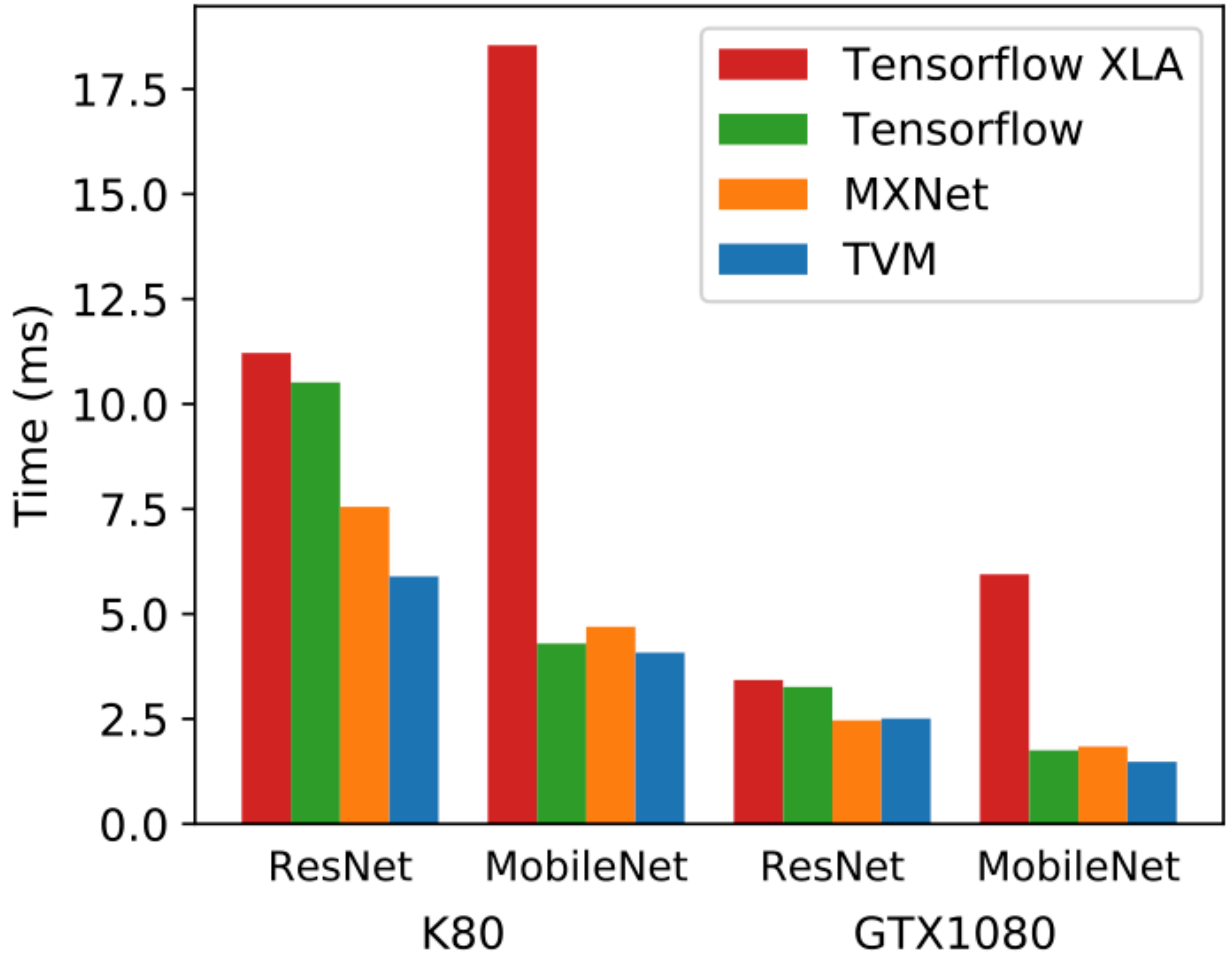
```
lib = t.build(s, [A, B],
    ‘llvm -target=armv7l-none-linux-gnueabihf‘,
     name='myfunc')
remote = t.rpc.connect(host, port)
lib.save(‘myfunc.o’)
remote.upload(‘myfunc.o’)
f = remote.load_module(‘myfunc.o')
ctx = remote.cpu(0)
a = t.nd.array(np.random.uniform(size=1024), ctx)
b = t.nd.array(np.zeros(1024), ctx)
remote_timer = f.time_evaluator(‘myfunc’, ctx, number=10)
time_cost = remote_timer(a, b)

np.testing.assert_equal(b.asnumpy(), expected)
```

Some Fun Results

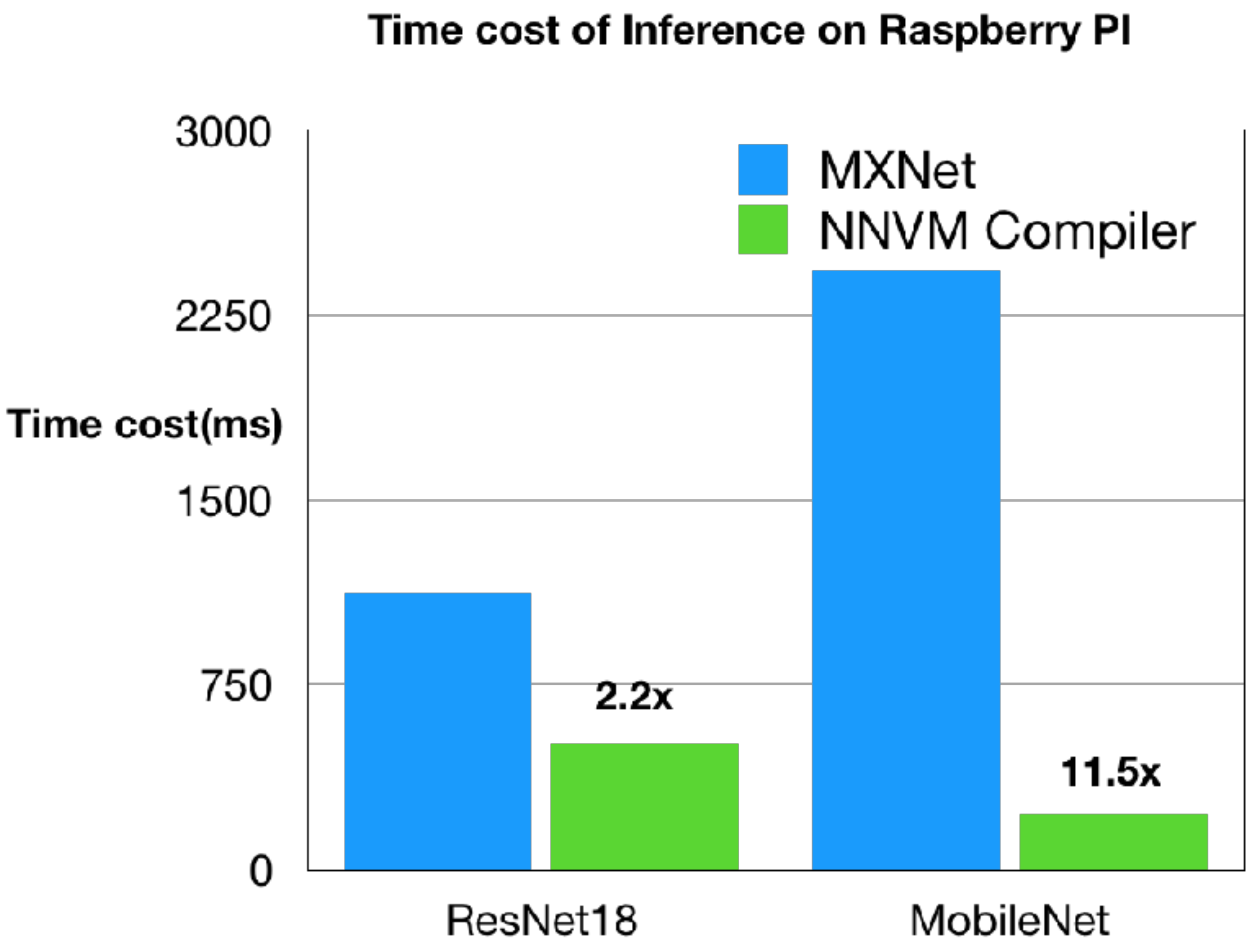
Compare TVM Stack solution to
Existing solutions which relies on manually optimized libraries

End to End Performance across Hardwares

Nvidia GPUs

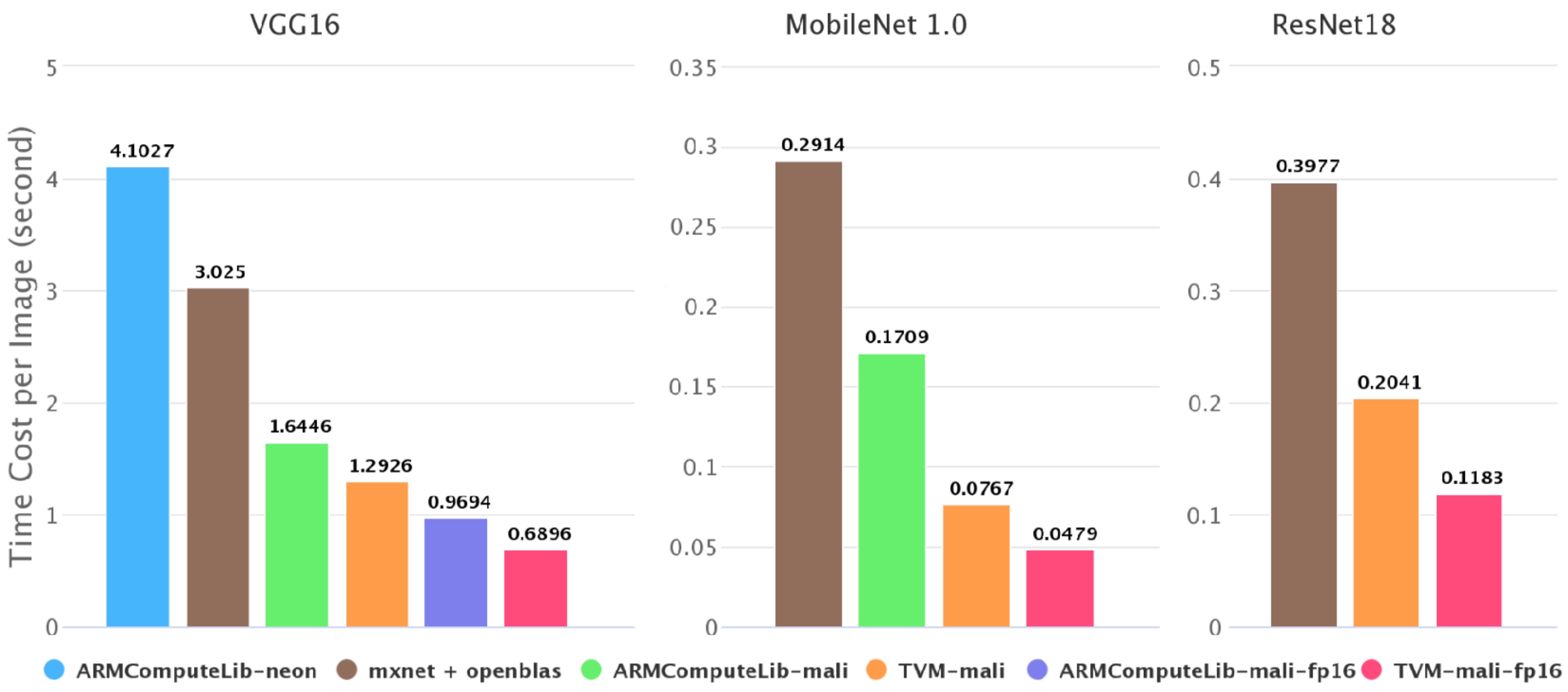


Raspberry Pis



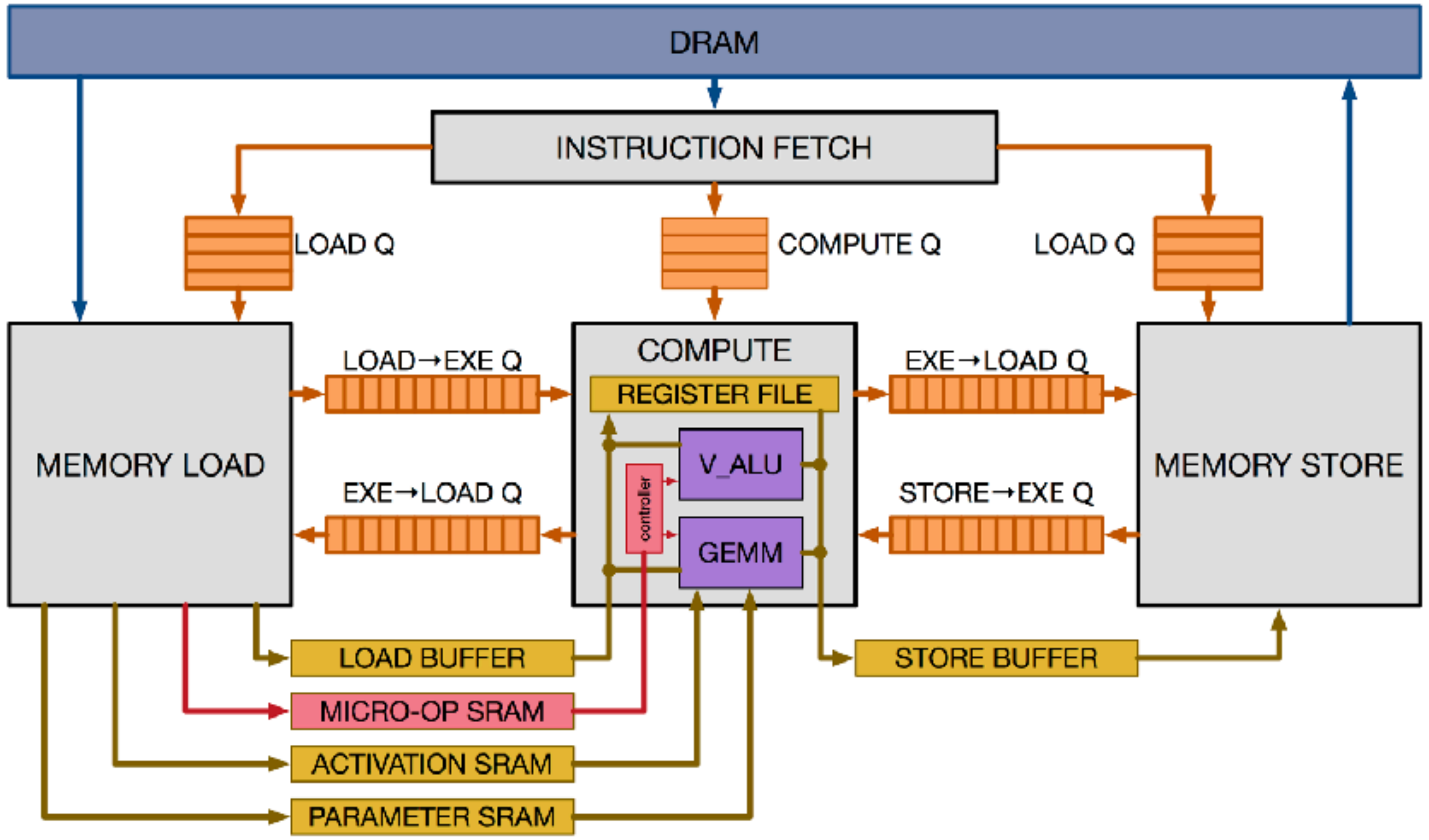
Credit: Leyuan Wang(AWS/UCDavis), Yuwei Hu(TuSimple), Zheng Jiang(AWS/FDU), Lianmin Zheng(SJTU)

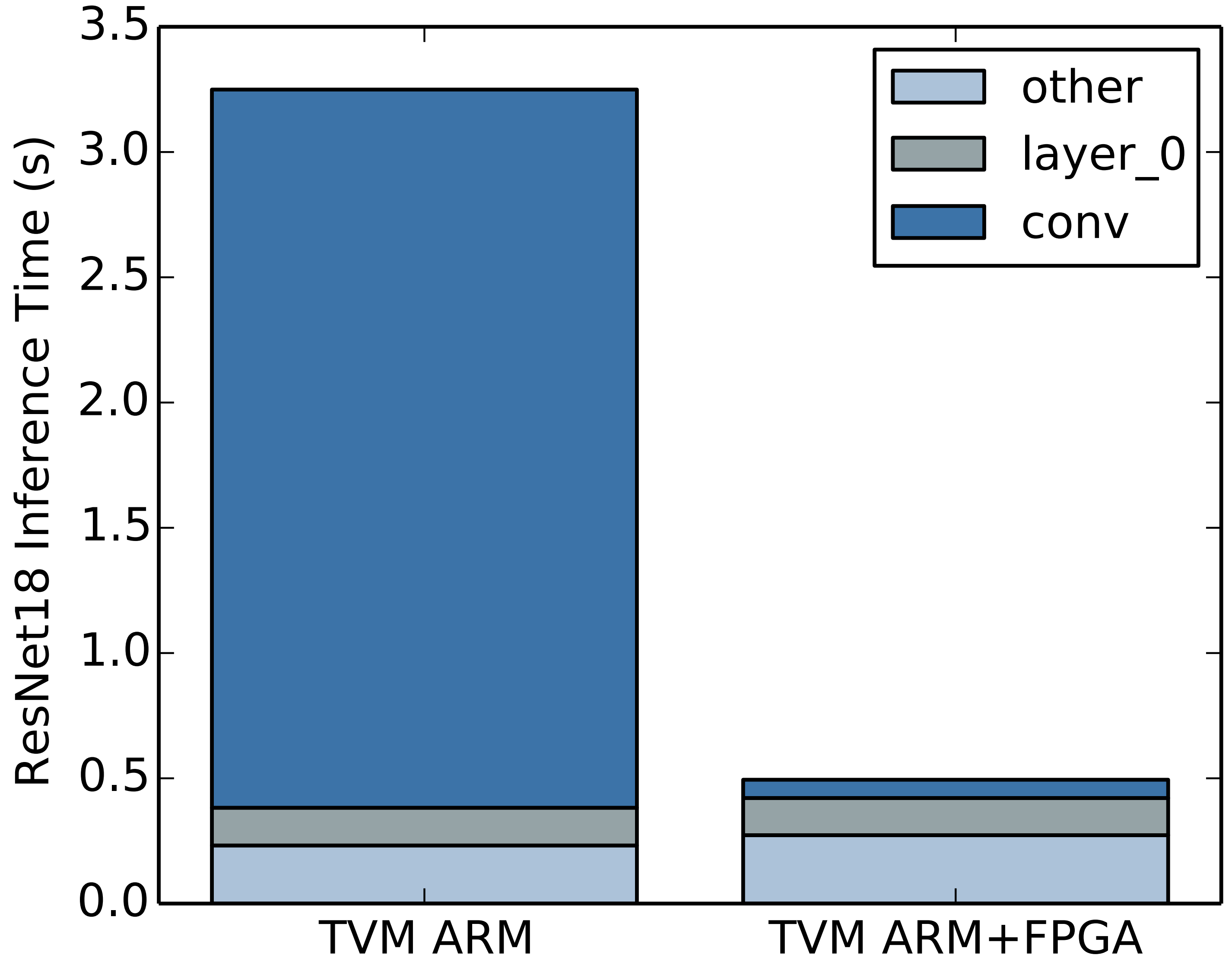
End to End Performance on Mobile GPUs(ARM Mali)

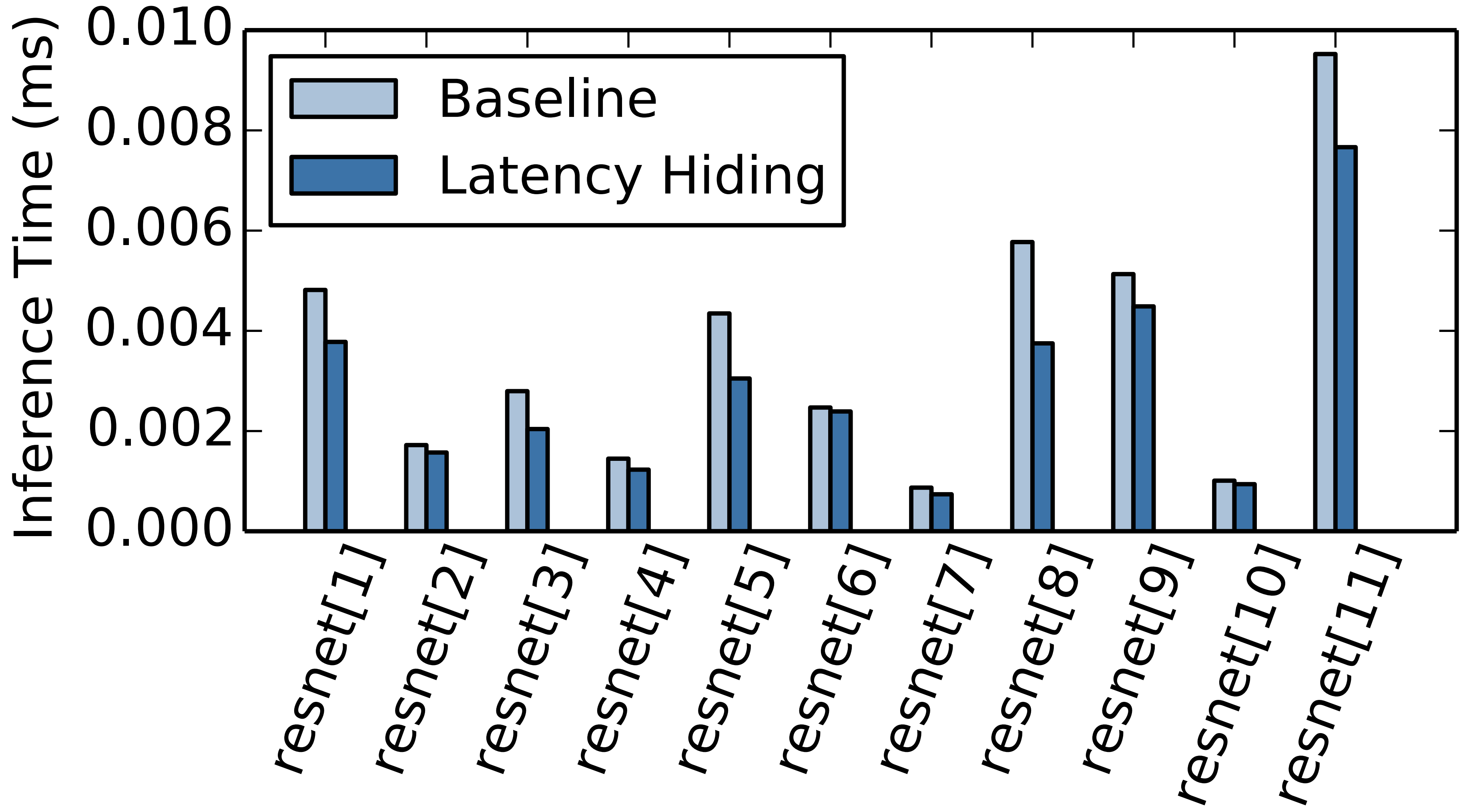


Credit: Lianmin Zheng(SJTU)

Support New Accelerators as Well







A Lot of Open Problems

Some examples questions:

Optimize for NLP models like RNN, attention

High dimensional convolutions

Low bit and mix precision kernels

More primitive support for accelerators

Tutorials from

tvmlang.org