DEEP LEARNING FRAMEWORKS

Where we are and where we should be going

JACK LEE | University of Toronto
AMY WANG | Huawei Canada

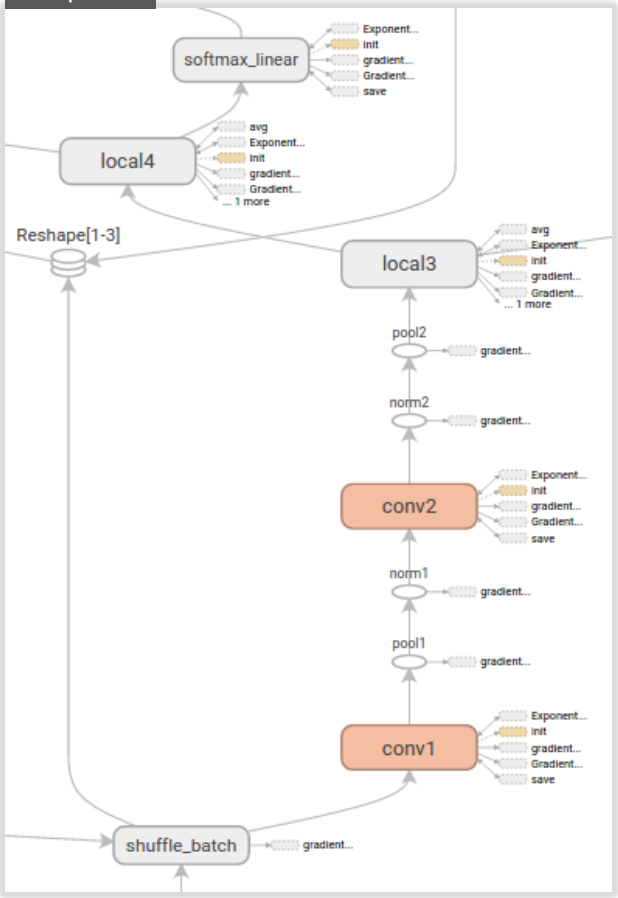
BACKGROUND

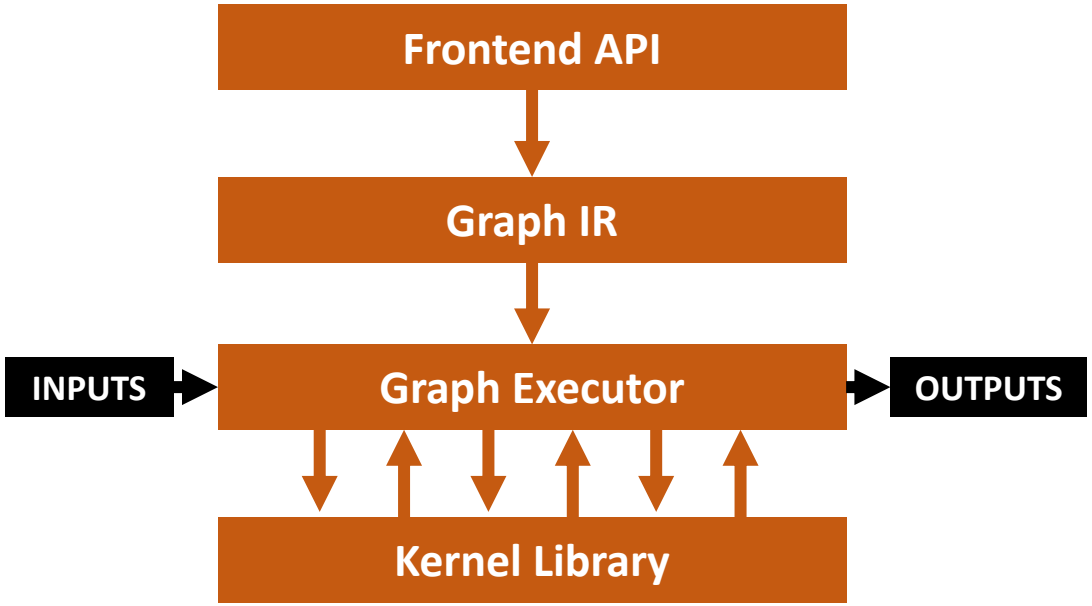
Architecture

TensorFlow Frontend

```python
n = 10000
x = tf.constant(list(range(n)))
c = lambda i, x: i < n
b = lambda i, x: (tf.Print(i + 1, [i]), tf.Print(x + 1, [i], "x:"))
i, out = tf.while_loop(c, b, (0, x))
with tf.Session() as sess:
    print(sess.run(i))  # prints [0] ... [9999]
```

Graph IR





Kernel Implementation

```cpp
class ZeroOutOp : public OpKernel {
 public:
  explicit ZeroOutOp(OpKernelConstruction* context) : OpKernel(context) {}

  void Compute(OpKernelContext* context) override {
    // Grab the input tensor
    const Tensor& input_tensor = context->input(0);
    auto input = input_tensor.flat<int32>();

    // Create an output tensor
    Tensor* output_tensor = NULL;
    OP_REQUIRES_OK(context, context->allocate_output(0, input_tensor.shape(),
                                                     &output_tensor));

    auto output_flat = output_tensor->flat<int32>();

    // Set all but the first element of the output tensor to 0.
    const int N = input.size();
    for (int i = 1; i < N; i++) {
      output_flat(i) = 0;
    }

    // Preserve the first input value if possible.
    if (N > 0) output_flat(0) = input(0);
  }
};
```

MOTIVATIONS

Frontend Interface

TensorFlow Frontend

```
n = 10000
x = tf.constant(list(range(n)))
c = lambda i, x: i < n
b = lambda i, x: (tf.Print(i + 1, [i]), tf.Print(x + 1, [i], "x:"))
i, out = tf.while_loop(c, b, (0, x))
with tf.Session() as sess:
    print(sess.run(i))  # prints [0] ... [9999]
```

Autograph

```
@autograph.convert()
def fizzbuzz(i, n):
  while i < n:
    msg = ''
    if i % 3 == 0:
      msg += 'Fizz'
    if i % 5 == 0:
      msg += 'Buzz'
    if msg == '':
      msg = tf.as_string(i)
    print(msg)
    i += 1
  return i
```

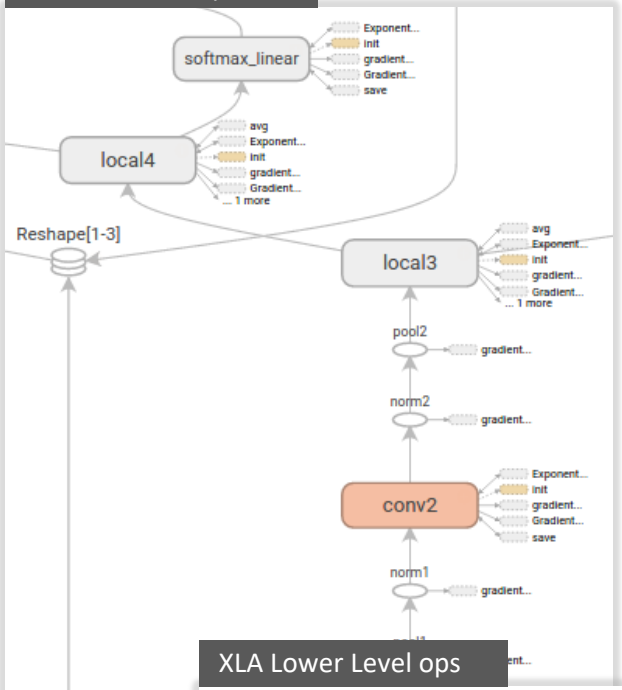
PyTorch Frontend

```
def forward(self, inputs, children, arities):

    i = self.wi_net(inputs)
    o = self.wo_net(inputs)
    u = self.wu_net(inputs)

    f_base = self.wf_net(inputs)
    fc_sum = Variable(inputs.data.new(self.memory_size).fill_(0))
    for k, child in enumerate(children):
        child_h, child_c = torch.chunk(child, 2, dim=1)
        i.add_(self.ui_nets[k](child_h))
        o.add_(self.uo_nets[k](child_h))
        u.add_(self.uu_nets[k](child_h))

        f = f_base
        for l, other_child in enumerate(children):
            other_child_h, _ = torch.chunk(child, 2, dim=1)
            f.add_(self.uf_nets[k][l](other_child_h))
        fc_sum.add(torch.sigmoid(f) * child_c)

    c = torch.sigmoid(i) * torch.tanh(u) + fc_sum
    h = torch.sigmoid(o) * torch.tanh(c)
    return torch.cat([h, c], dim=1)
```

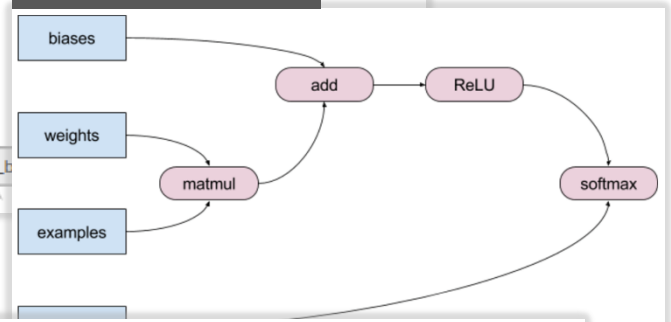
MOTIVATIONS

Graph Optimizations

TensorFlow Graph IR



XLA Lower Level ops



softmax = exp(logits) / reduce_sum(exp(logits), dim)

PyTorch Frontend

```
def forward(self, inputs, children, arities):

    i = self.wi_net(inputs)
    o = self.wo_net(inputs)
    u = self.wu_net(inputs)

    f_base = self.wf_net(inputs)
    fc_sum = Variable(inputs.data.new(self.memory_size).fill_(0))
    for k, child in enumerate(children):
        child_h, child_c = torch.chunk(child, 2, dim=1)
        i.add_(self.ui_nets[k](child_h))
        o.add_(self.uo_nets[k](child_h))
        u.add_(self.uu_nets[k](child_h))

        f = f_base
        for l, other_child in enumerate(children):
            other_child_h, _ = torch.chunk(child, 2, dim=1)
            f.add_(self.uf_nets[k][l](other_child_h))
        fc_sum.add(torch.sigmoid(f) * child_c)

    c = torch.sigmoid(i) * torch.tanh(u) + fc_sum
    h = torch.sigmoid(o) * torch.tanh(c)
    return torch.cat([h, c], dim=1)
```

AST-based JIT

```
import torch
@torch.jit.script
def foo(x, y):
    if x.max() > y.max():
        r = x
    else:
        r = y
    return r
```

Automatic differentiation every iteration.

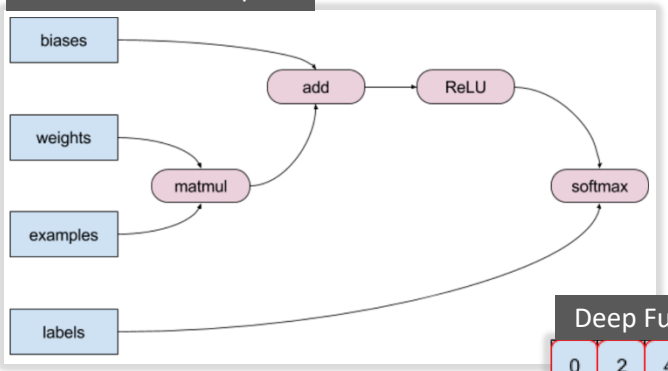
Trace-based JIT

```
import torch
def foo(x, y):
    return 2*x + y
traced_foo = torch.jit.trace(foo, (torch.rand(3),
torch.rand(3)))
```

MOTIVATIONS

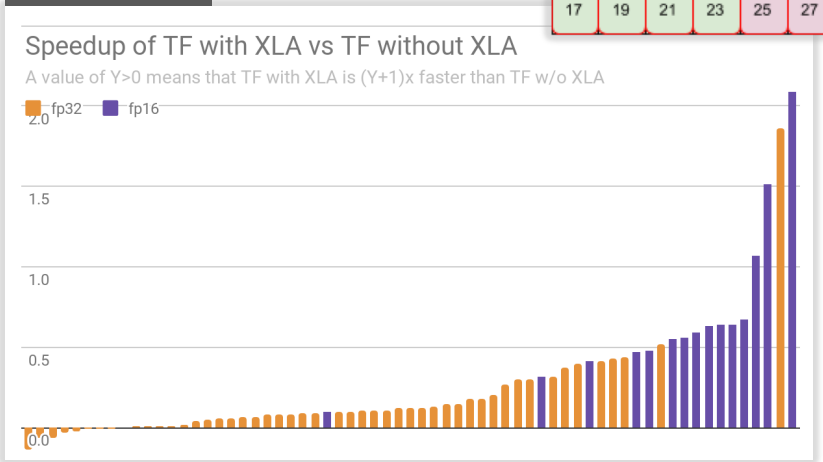
Kernel Specialization

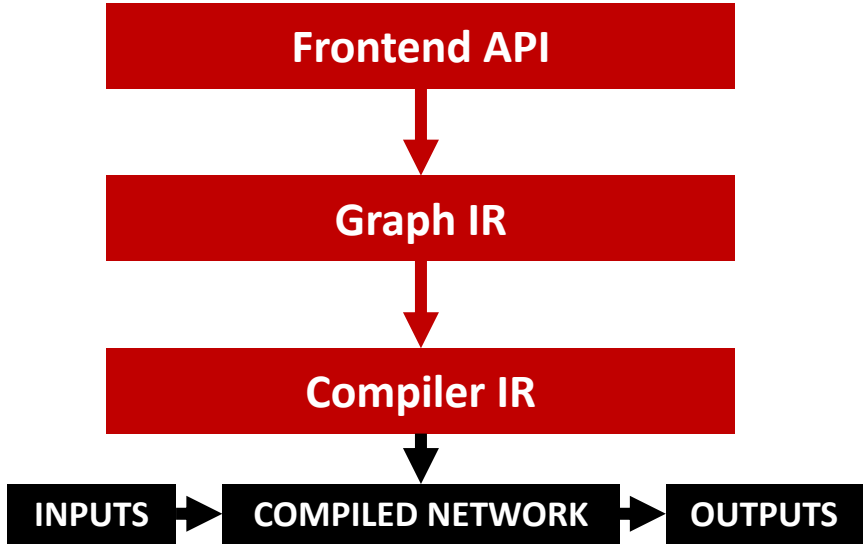
XLA Lower Level ops



Deep Fusion Tiling

Benchmarks





Graph Lowering

MOTIVATIONS

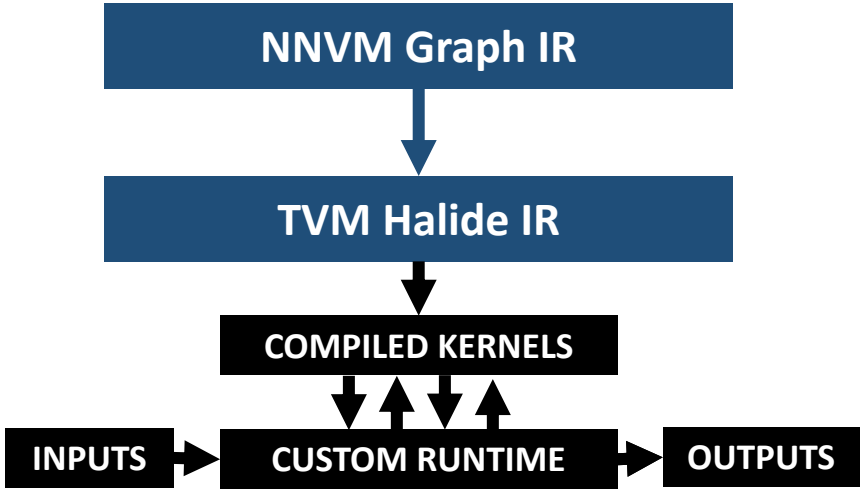
Kernel Specialization

NNVM API

```
x = sym.Variable("x")
y = sym.Variable("y")
z = sym.elemwise_add(x, sym.sqrt(y))
compute_graph = nnvm.graph.create(z)
print("-------compute graph-------")
print(compute_graph.ir())
```

TVM API

```
A = tvm.placeholder((m,), name='A')
B = tvm.compute((m,), lambda i: A[i]*2, name='B')

s = tvm.create_schedule(B.op)
xo, xi = s[B].split(B.op.axis[0], factor=32)
print(tvm.lower(s, [A, B], simple_mode=True))
```



Compiler IR

```
produce C {
  for (i, 0, m) {
    for (j, 0, n) {
      C[((i*n) + j)] = (A[((i*n) + j)]*B[((i*n) + j)])
    }
  }
}
```

Generated GPU Code

```
extern "C" __global__ void myadd_kernel0( float* __restrict__ C,  float* __restrict__ A,  float* __res
  if (((int)blockIdx.x) < (n / 64)) {
    C[((((int)blockIdx.x) * 64) + ((int)threadIdx.x))] = (A[((((int)blockIdx.x) * 64) + ((int)threadId
  } else {
    if ((((int)blockIdx.x) * 64) < (n - ((int)threadIdx.x))) {
      C[((((int)blockIdx.x) * 64) + ((int)threadIdx.x))] = (A[((((int)blockIdx.x) * 64) + ((int)thread
    }
  }
}
```

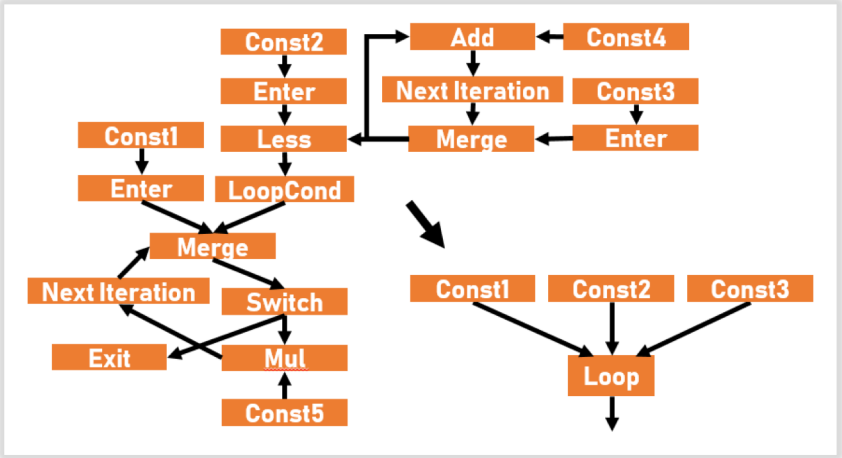
STATE OF THE ART SUMMARY

[redacted]

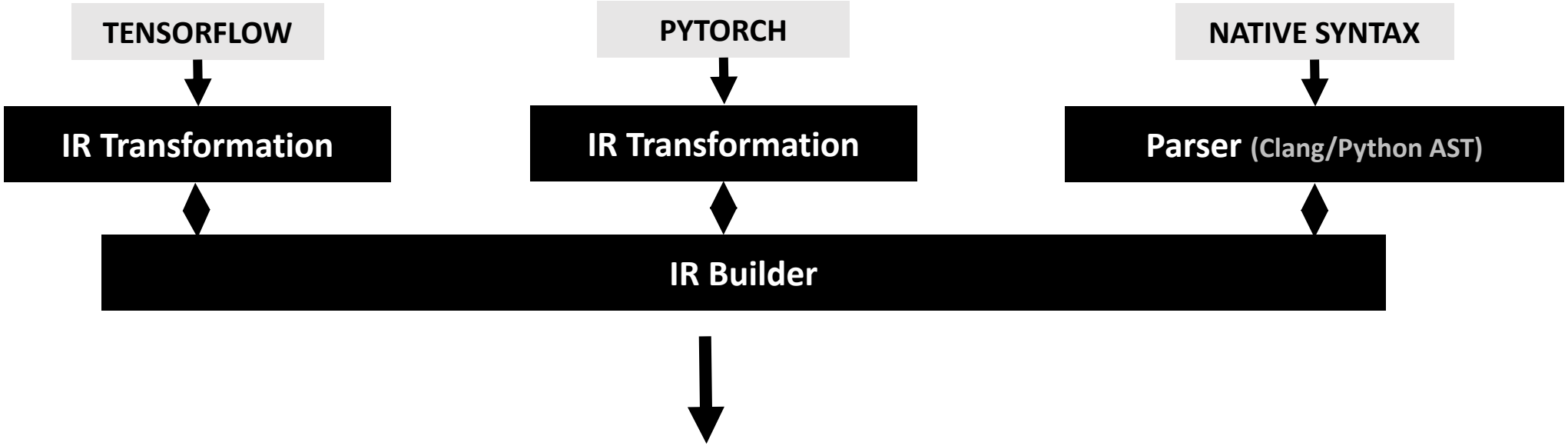
	TENSORFLOW	TENSORFLOW XLA	PYTORCH	PYTORCH - GLOW	NNVM + TVM
Staged Frontend			✗	✗	
Native Frontend	✗	✗			✗
Graph Optimization			✗		
Kernel Specialization	✗		✗		
Runtime Specialization	✗	✗	✗	✗	✗
Execution Level	C++	C++	Python	Machine Code	C++

THE DVM FRONTEND

Deep Learning Compilation Framework

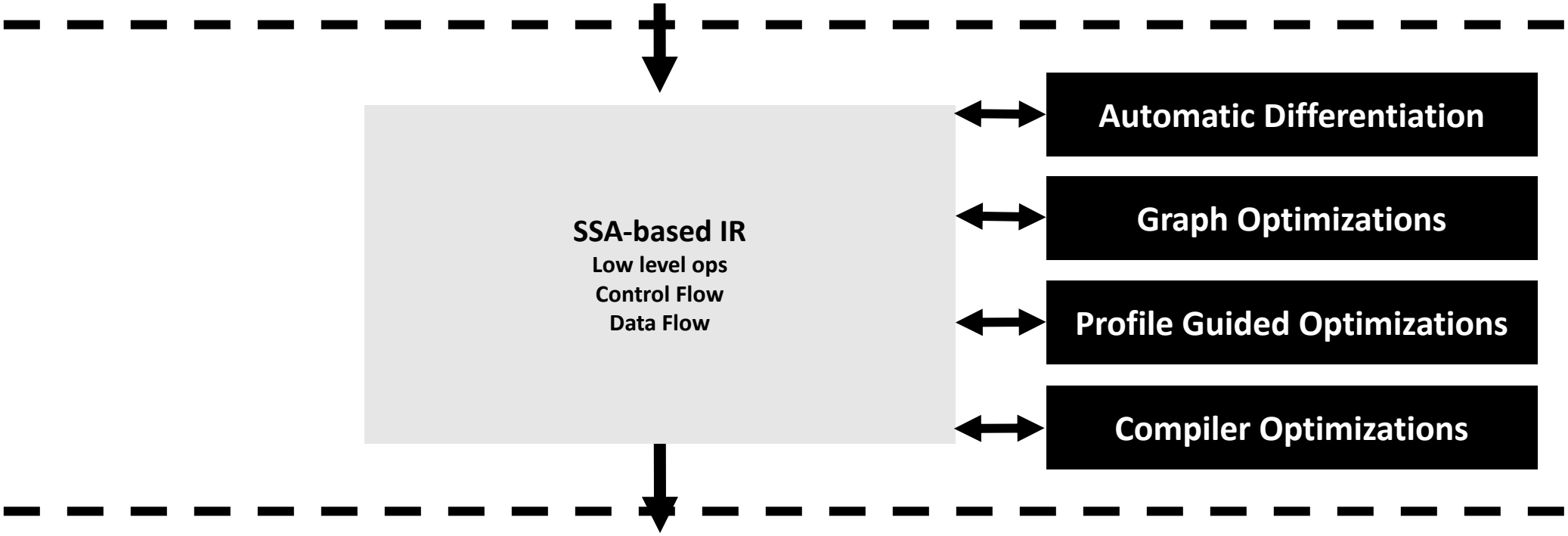



```
let linspace = np.linspace(start: 0, stop: 1, num: 10, endpoint: true, retstep: false)
let colorMap = plt.get_cmap("RdBu")
let colors = np.r_[linspace, linspace]
let mappedColors = colorMap(colors)
for element in 0..<10 {
    let xElementIndex = labels.scalars.enumerated().filter { $0.element == element }.map { 2 * $0.offset + 0 }
    let yElementIndex = labels.scalars.enumerated().filter { $0.element == element }.map { 2 * $0.offset + 1 }
    let x = image.enumerated().filter { xElementIndex.contains($0.offset) }.map { $0.element }
    let y = image.enumerated().filter { yElementIndex.contains($0.offset) }.map { $0.element }
    ax.scatter(x: x, y: y, color: mappedColors[element], label: "\(element)", s: 20, alpha: 0.8)
}
```

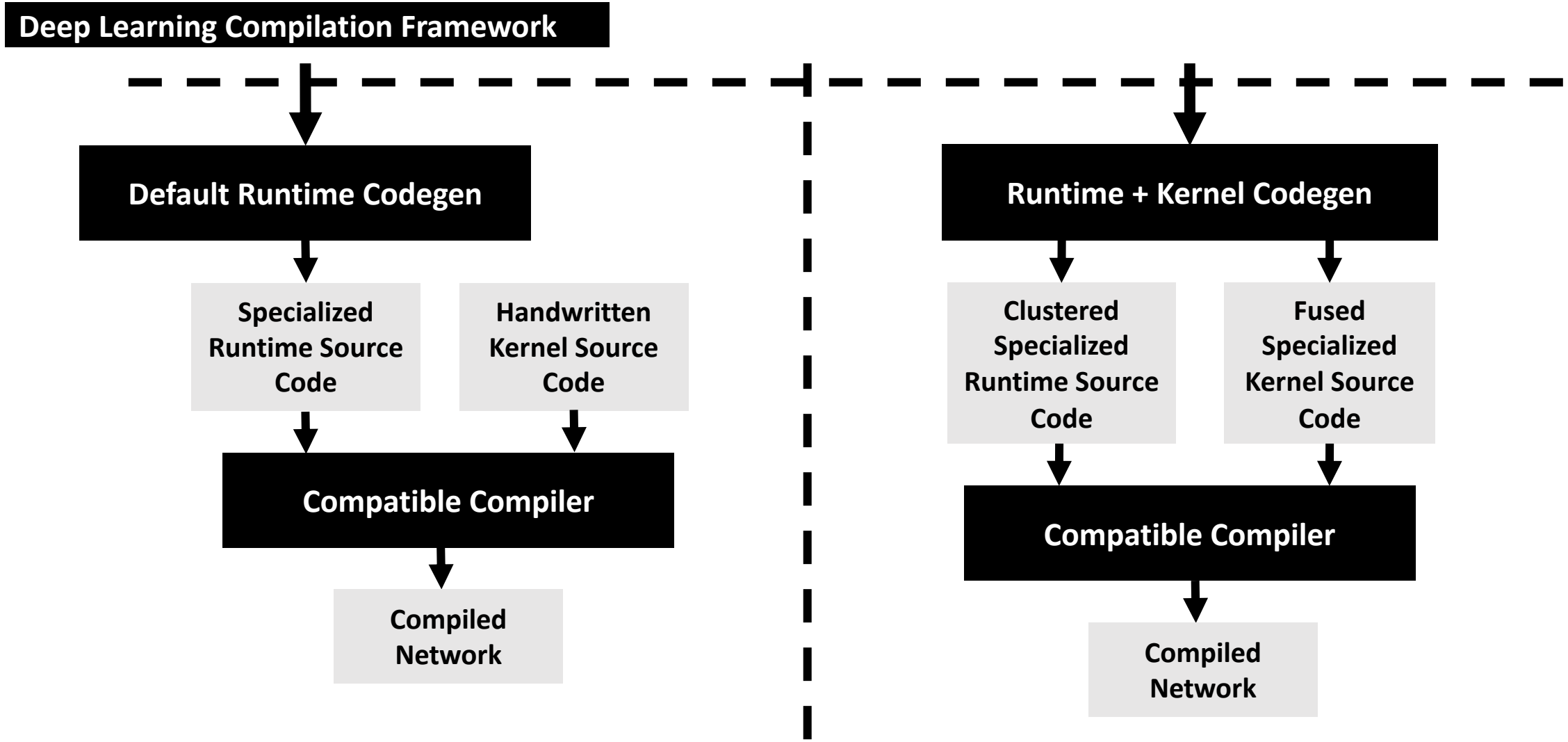


THE DVM MIDEND

Deep Learning Compilation Framework

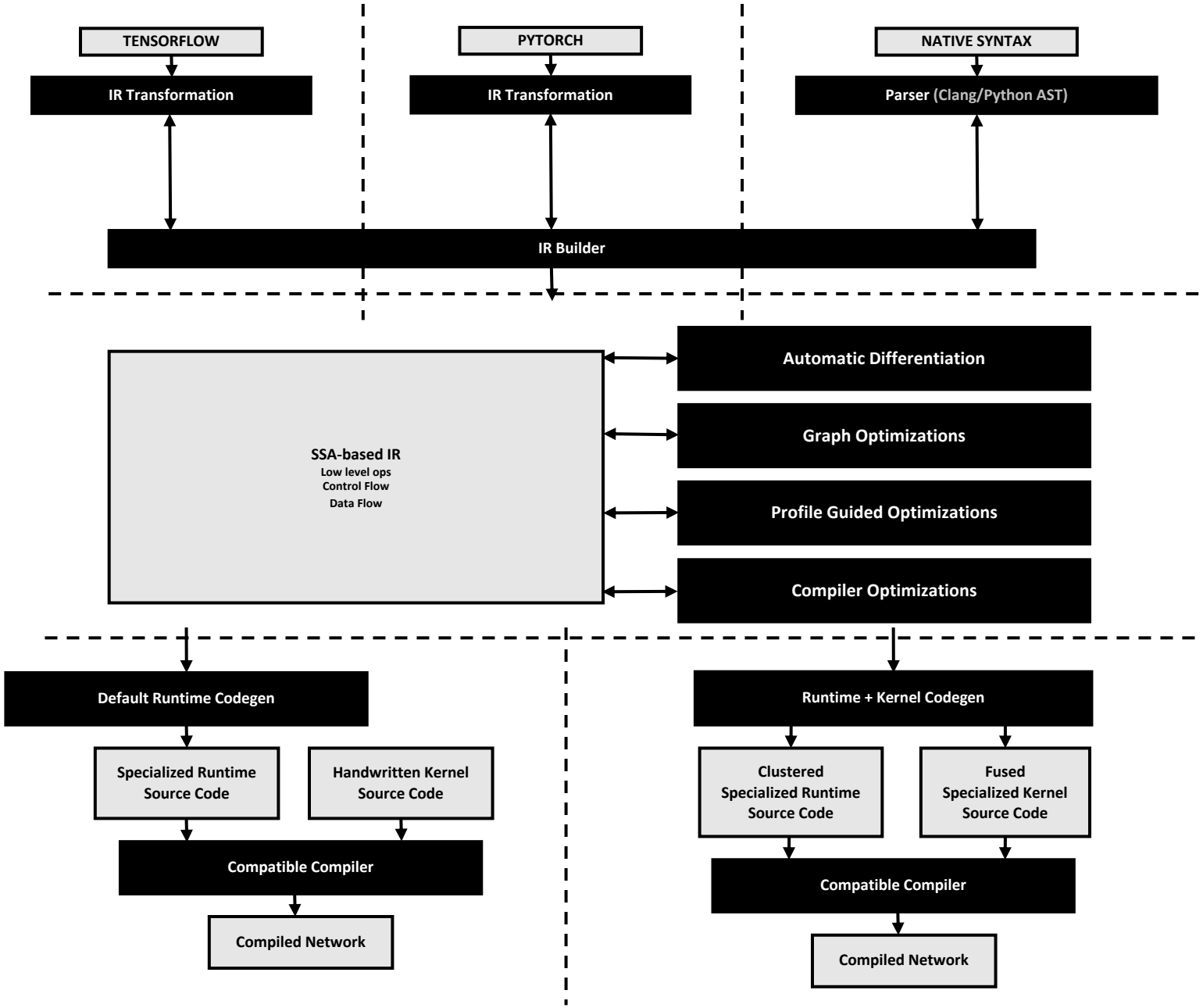


THE DVM BACKENDS



Q&A

[redacted]


TENSORFLOW
PYTORCH
NATIVE SYNTAX
IR Transformation
IR Transformation
Parser (Clang/Python AST)
IR Builder
SSA-based IR
Low level ops
Control Flow
Data Flow
Automatic Differentiation
Graph Optimizations
Profile Guided Optimizations
Compiler Optimizations
Default Runtime Codegen
Specialized Runtime Source Code
Handwritten Kernel Source Code
Compatible Compiler
Compiled Network
Runtime + Kernel Codegen
Clustered Specialized Runtime Source Code
Fused Specialized Kernel Source Code
Compatible Compiler
Compiled Network