Extending TVM with Dynamic Execution

Jared Roesch and Haichen Shen

Outline

- **Motivation for Dynamism**
- Representing Dynamism
- Executing Dynamism
- Evaluation

Dynamic Neural Networks

- Networks are exhibiting more and more dynamism
  - Dynamic inputs: batch size, image size, sequence length, etc.
  - Control-flow, recursion, conditionals and loops (in Relay today).
  - Dynamically sized tensors
    - Output shape of some ops are data dependent: arange, nms, etc.
    - Control flow: concatenation within a while loop
- A central challenge is how do we both **represent** and **execute** these networks.

```
fn network(input: Tensor<(n,3,1024,1024), float32>) -> … { … }
```

```
%t1: Tensor<(1), f32>
%t2 : Tensor<(10), f32>

if (%cond) { … } else { … } : Tensor<(?), f32>
```

```
%start,%stop, %step : i32

arange(%start, %stop, %step) : Tensor<(?), f32>
```

Dynamic Neural Networks

- A central challenge is how do we both **represent** and **execute** these networks.
- We will address these two challenges at various levels of the TVM stack and share initial promising results.

Outline

- Motivation for Dynamism
- **Representing Dynamism**
- Executing Dynamism
- Evaluation

Representing dynamics in TVM

- Add Relay support for dynamic dimension (Any-dim)
- Use shape functions to compute runtime shapes.
- Supporting Any in Tensor Expression (TE) IR.

Any: typing dynamic dimension in Relay

`Any`: represent an unknown dimension at compilation time.

Any: typing dynamic dimension in Relay

`Any`: represent an unknown dimension at compilation time.

Define a tensor type: `Tensor<(Any, 3, 32, 32), fp32>`

Any: typing dynamic dimension in Relay

Any: represent an unknown dimension at compilation time.

Define a tensor type: `Tensor<(Any, 3, 32, 32), fp32>`

Define type relation:

```
arange: fn(start:fp32, stop:fp32, step:fp32)
     -> Tensor<(Any), fp32>

broadcast: fn(Tensor<(Any, Any),fp32>, Tensor<(1, 8), fp32>)
     -> Tensor<(Any, 8), fp32>
```

Valid only when Any = 1 or 8

How to compute and check shape dynamically?

Challenges

- Static type checking cannot eliminate all errors
- Type checking system too heavy weight for runtime

How to compute and check shape dynamically?

Challenges

- Static type checking cannot eliminate all errors
- Type checking system too heavy weight for runtime

Approach

- Instrument shape computing functions into the program

Instrumentation example

```
def @main(%x: Tensor[(?, ?), float32], %y: Tensor[(1, 2), float32]) ->
Tensor[(?, 2), float32] {
  add(%x, %y) /* ty=Tensor[(?, 2), float32] */
}
```

```
def @main(%x: Tensor[(?, ?), float32], %y: Tensor[(1, 2), float32]) ->
Tensor[(?, 2), float32] {
  %0 = shape_of(%x, dtype="int64")
  %1 = meta[relay.Constant][0] /* y.shape: [1, 2] */
  %2 = broadcast_shape_func(%0, %1)
  %tensor = alloc_tensor(%2, float32)
  add(%x, %y, %tensor)
}
```

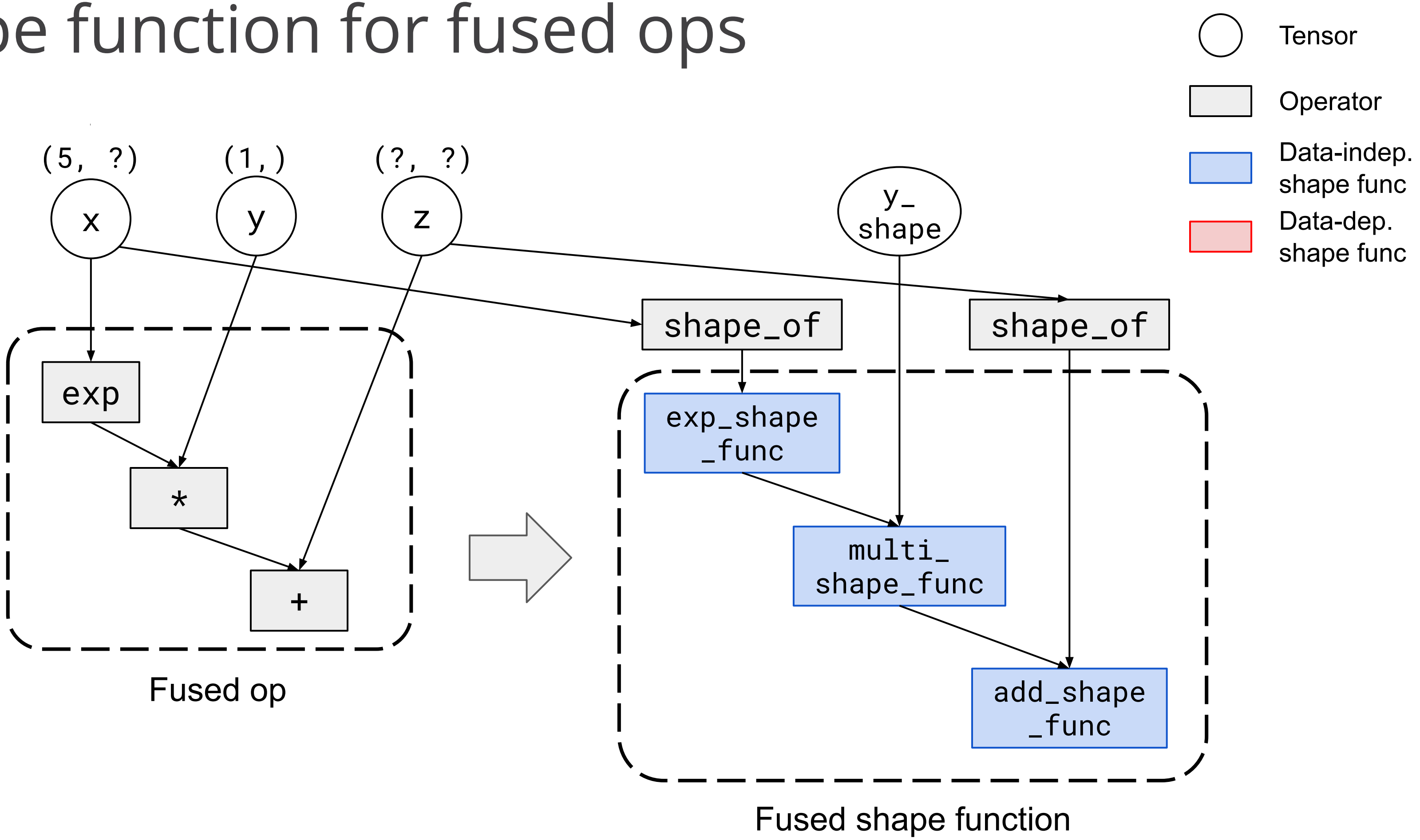
Shape function

- Register a shape function to each operator to check the type and compute the output shape

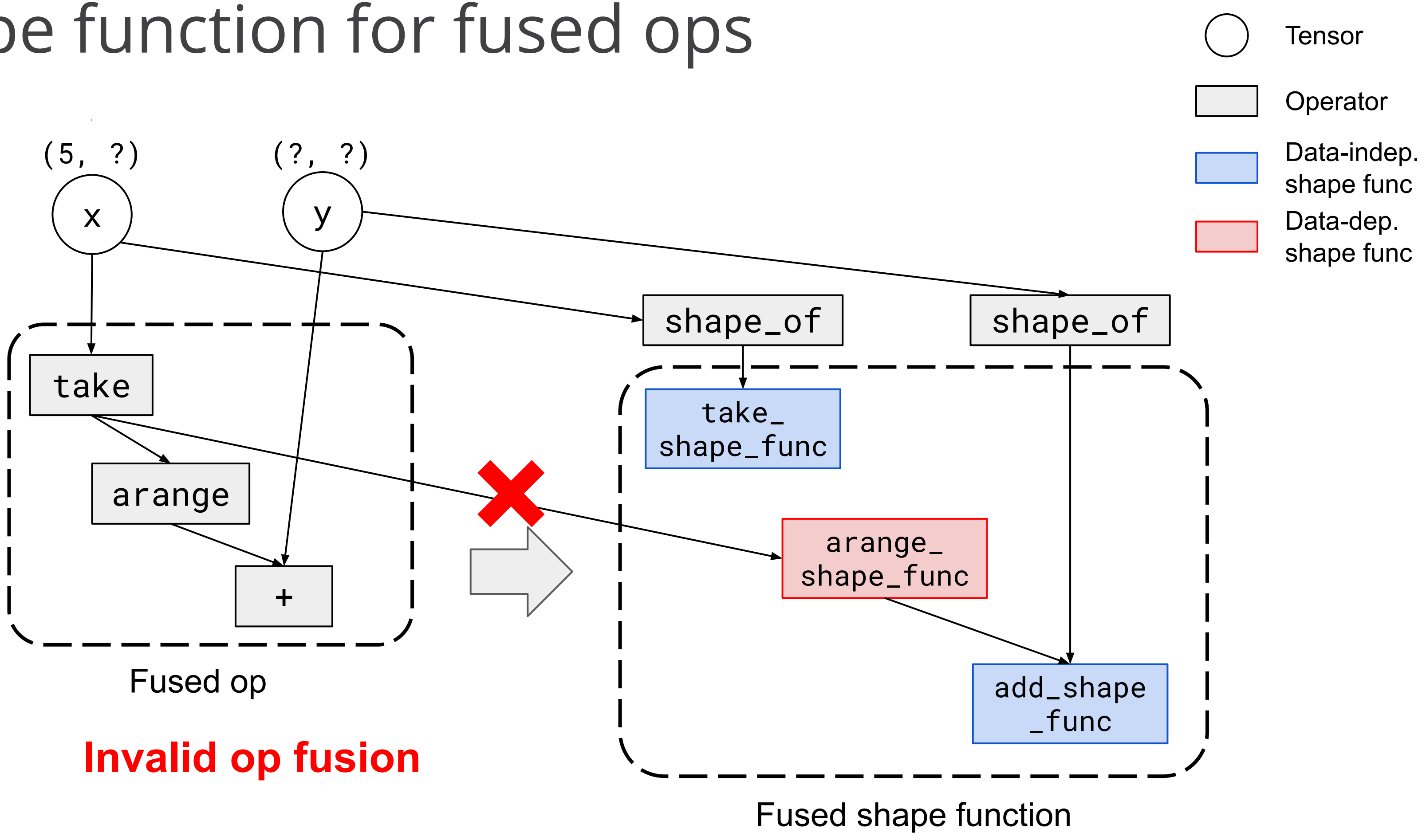
Shape function

- Register a shape function to each operator to check the type and compute the output shape
- Shape function has two modes
  (op_attrs, input_tensors, out_ndims) -> out_shape_tensors
  - Data independent
    (op_attrs, input_shapes, out_ndims) -> out_shape_tensors
  - Data dependent
    (op_attrs, input_data, out_ndims) -> out_shape_tensors

Shape function for fused ops



Shape function for fused ops



Shape function example

Use hybrid script to write shape function

Type checking

Data independent

Input shape tensors

```
@script
def _concatenate_shape_func(inputs, axis):
    ndim = inputs[0].shape[0]
    out = output_tensor((ndim,), "int64")
    for i in const_range(ndim):
        if i != axis:
            out[i] = inputs[0][i]
            for j in const_range(1, len(inputs)):
                assert out[i] == inputs[j][i], "Dims mismatch in the inputs of concatenate."
        else:
            out[i] = int64(0)
            for j in const_range(len(inputs)):
                out[i] += inputs[j][i]
    return out

@_reg.register_shape_func("concatenate", False)
def concatenate_shape_func(attrs, input_shapes, _):
    axis = get_const_int(attrs.axis)
    return [_concatenate_shape_func(inputs, convert(axis))]
```

Shape function example

```
@script
def _arange_shape_func(start, stop, step):
    out = output_tensor((1,), "int64")
    out[0] = int64(ceil_div((int64(stop[0]) - int64(start[0])), int64(step[0])))
    return out


@_reg.register_shape_func("arange", True)  Data dependent
def arange_shape_func(attrs, input_data, _):
    return [_arange_shape_func(*input_data)]
```

Outline

- Motivation for Dynamism
- Representing Dynamism
- **Executing Dynamism**
- Evaluation

Executing dynamics in TVM

- By extending the IR we now can represent dynamic programs but *how do we execute them?*
- To handle flexibly executing dynamic programs we introduce the Relay virtual machine.
- We must also generate code which handles dynamic shapes in kernels (work-in-progress):
  - Kernel dispatch for a single op
  - Dispatch for a (sub-)expression

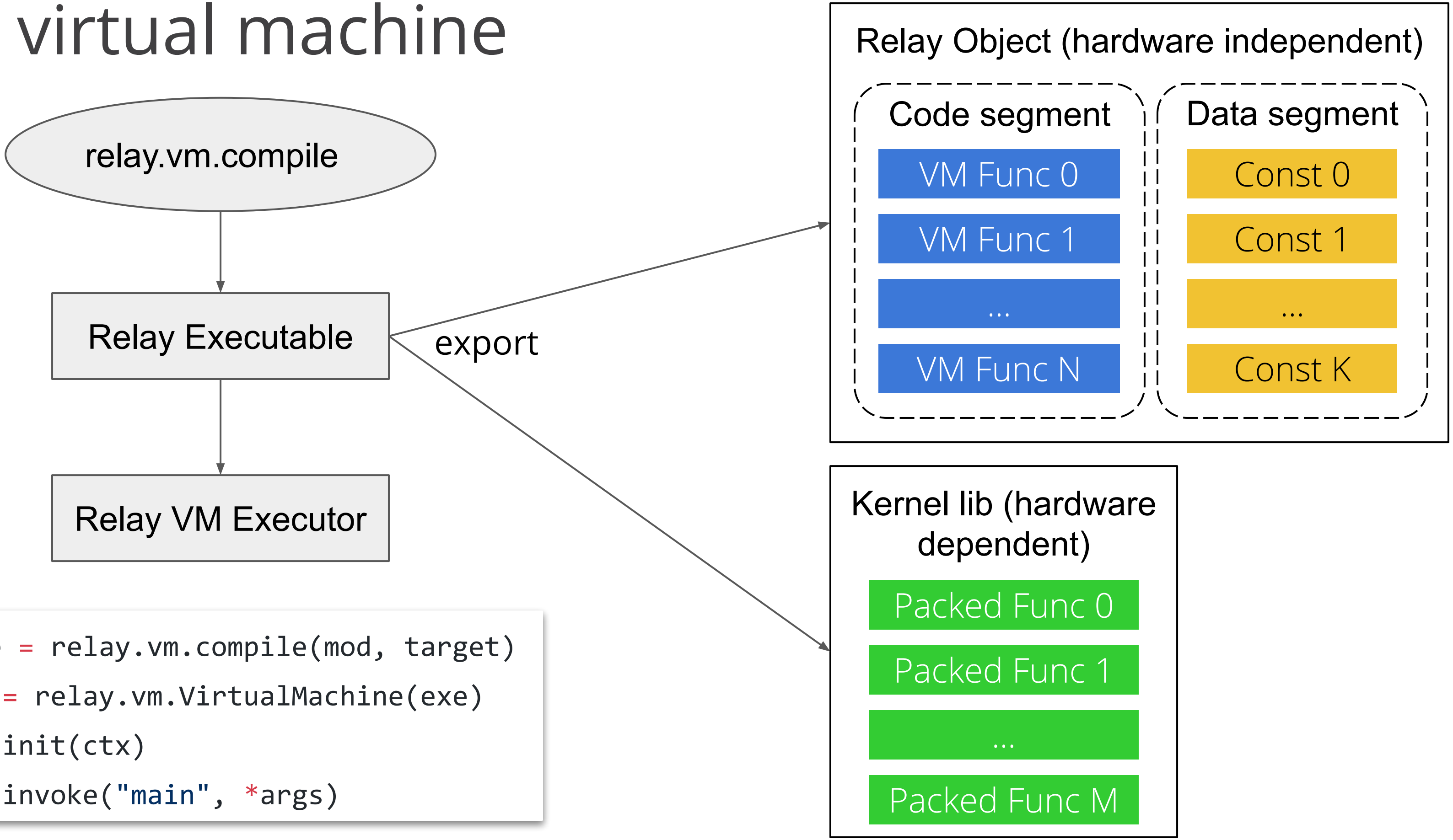
Previous approach: Graph Runtime

- Existing executors are based on a graph traversal style execution.
- Set up a graph of operators and push data along every edge, compute the operation, and flow forward until finished.
- Simple design enables simple memory allocation, and executor.
- Design is complicated by control, and dynamic shapes.

Enter the virtual machine

- Instead we take inspiration from full programming languages and design a VM.
- The VM has special considerations
  - Primitives are tensors, and instructions operate on tensors (CISC-style, no-scalar instructions)
  - Instructions normally built in (+, -, etc.) are realized by code generated via TVM.
  - Control handled in standard way in VM.
  - In contrast to AoT compilation, VM is flexible
    - graph dispatch and bucketing can be easily implemented.

Relay virtual machine




```python
exe = relay.vm.compile(mod, target)
vm = relay.vm.VirtualMachine(exe)
vm.init(ctx)
vm.invoke("main", *args)
```

VM bytecode

Instruction	Description
Move	Moves data from one register to another.
Ret	Returns the object in register result to caller’s register.
Invoke	Invokes a function at in index.
InvokeClosure	Invokes a Relay closure.
InvokePacked	Invokes a TVM compiled kernel.
AllocStorage	Allocates a storage block.
AllocTensor	Allocates a tensor value of a certain shape.
AllocTensorReg	Allocates a tensor based on a register.
AllocDatatype	Allocates a data type using the entries from a register.
AllocClosure	Allocates a closure with a lowered virtual machine function.
If	Jumps to the true or false offset depending on the condition.
Goto	Unconditionally jumps to an offset.
LoadConst	Loads a constant at an index from the constant pool.

Relay virtual machine

```
def @main(%i: int32) -> int32 {
 @sum_up(%i) /* ty=int32 */
}

def @sum_up(%i1: int32) -> int32 {
 %0 = equal(%i1, 0 /* ty=int32 */) /* ty=bool */;
 if (%0) {
   %i1
 } else {
   %1 = subtract(%i1, 1 /* ty=int32 */) /* ty=int32
*/;
   %2 = @sum_up(%1) /* ty=int32 */;
   add(%2, %i1) /* ty=int32 */
 }
}
```

```
sum_up:
alloc_storage 1 1 64 bool
alloc_tensor $2 $1 [] uint1
invoke_packed PackedFunc[0] (in: $0, out: $2)
load_consti $3 1
if $2 $3 1 2
goto 9
alloc_storage 4 4 64 int32
alloc_tensor $5 $4 [] int32
invoke_packed PackedFunc[1] (in: $0, out: $5)
invoke $6 VMFunc[0]($5)
alloc_storage 7 4 64 int32
alloc_tensor $8 $7 [] int32
invoke_packed PackedFunc[2] (in: $6, $0, out:
$8)
move $0 $8
ret $0

main:
invoke $1 VMFunc[0]($0)
ret $1
```

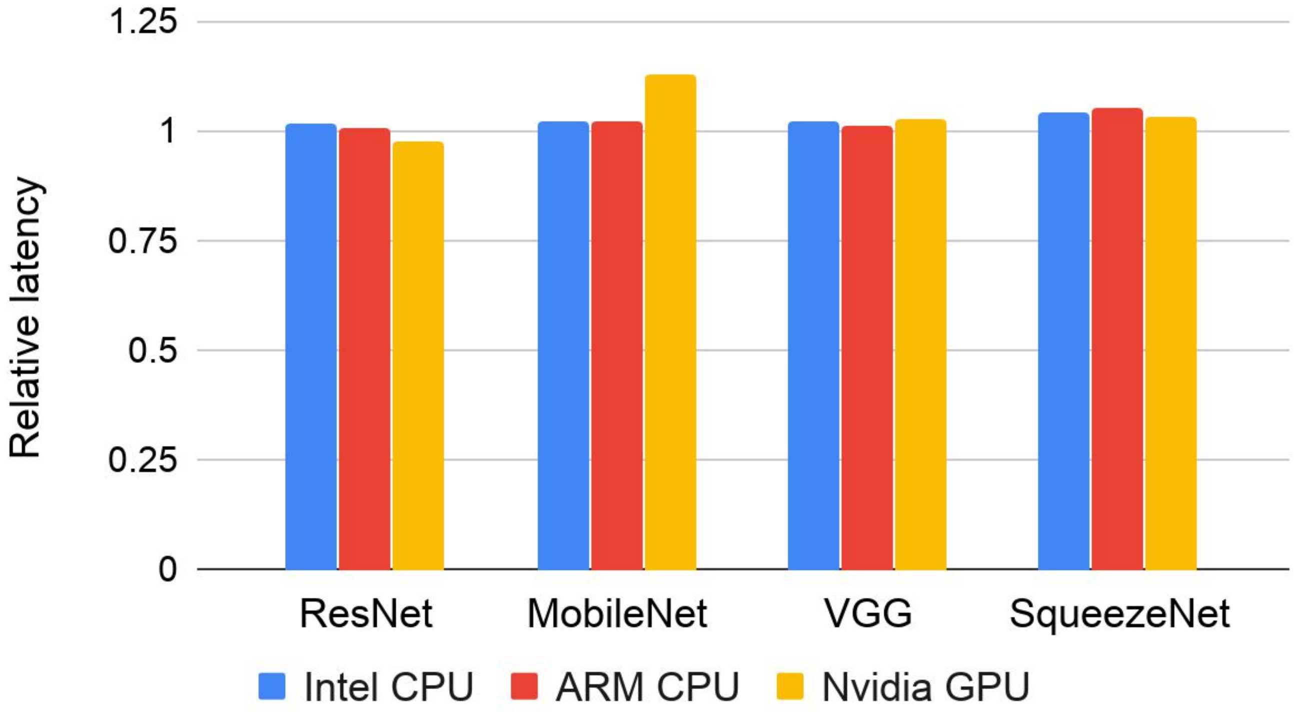
Generating code for dynamic shapes

- We now must solve the final problem of generating kernels that provide compelling performance for non-static shapes.
- The VM provides a framework for experimenting with different strategies, we will discuss in progress approaches:
    - Dynamic operator dispatch (WIP)
    - Graph Dispatch (https://github.com/apache/incubator-tvm/pull/4241)
- We believe there exists lots of future work in this area.

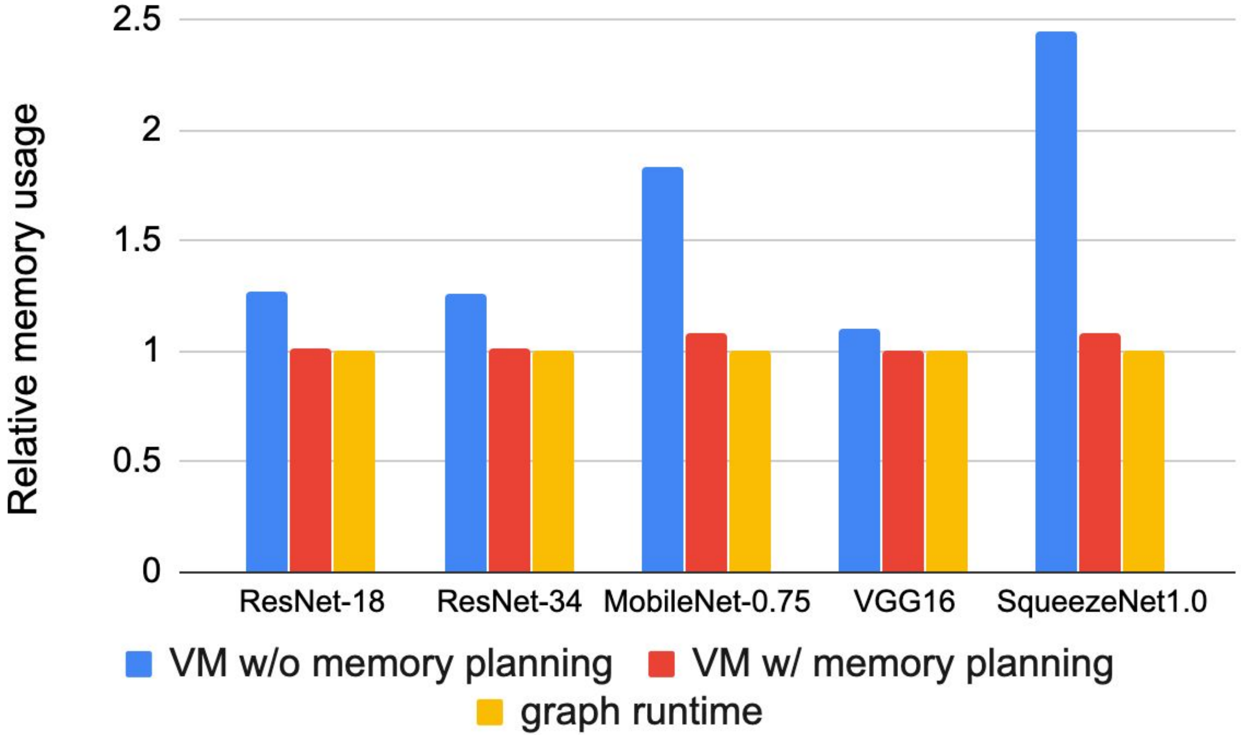
Outline

- Motivation for Dynamism
- Representing Dynamism
- Executing Dynamism
- **Evaluation**

Latency compared to graph runtime



Memory usage compared to graph runtime



Dynamic model performance

Unit: us/token	Intel CPU	ARM CPU
Relay VM	**38.7**	**186.5**
MXNet (1.6)	221.4	3681.4
Tensorflow (1.14)	247.5	-

LSTM model

Unit: us/token	Intel CPU	ARM CPU
Relay VM	**40.3**	**86.3**
PyTorch (1.3)	701.6	1717.1
TF Fold	209.9	-

Tree-LSTM model

BERT model performance

Unit: us/token	Intel CPU	ARM CPU	Nvidia GPU
Relay VM	501.3	**3275.9**	**79.4**
MXNet (1.6)	**487.1**	8654.7	113.2
Tensorflow (1.14)	747.3	-	118.4

Conclusions

- We have extended Relay/TVM with support for dynamic shapes.
- To support increased expressivity of Relay we have built a new execution mechanism the VM.
- We have begun exploring strategies for generating efficient kernels that support dynamic shapes with promising results.
- We believe the VM infrastructure can serve as a foundation for exploring future research into dynamic execution and code generation.

Thank you!

Acknowledgement

Outline

- Dynamic motivations
  - NLP, NMS, control, data structures
  - Integration with external code and runtimes
- Existing solution: graph runtime
  - Challenges with graph runtime
- Enter VM
  - Designed to be scaffold to build new dynamic functionality consisting of compiler and runtime improvements
- VM design
- Extensions
- Results
- Future Work
  - Dispatch, strategies?

Existing solution: graph runtime

Challenges:

-

- Control flow (if, loop, etc)
- Dynamic shapes
  - Dynamic inputs: batch size, image size, sequence length, etc.
  - Output shape of some ops are data dependent: arange, nms, etc.
  - Control flow: concatenate within a while loop

Limitation of TVM/graph runtime

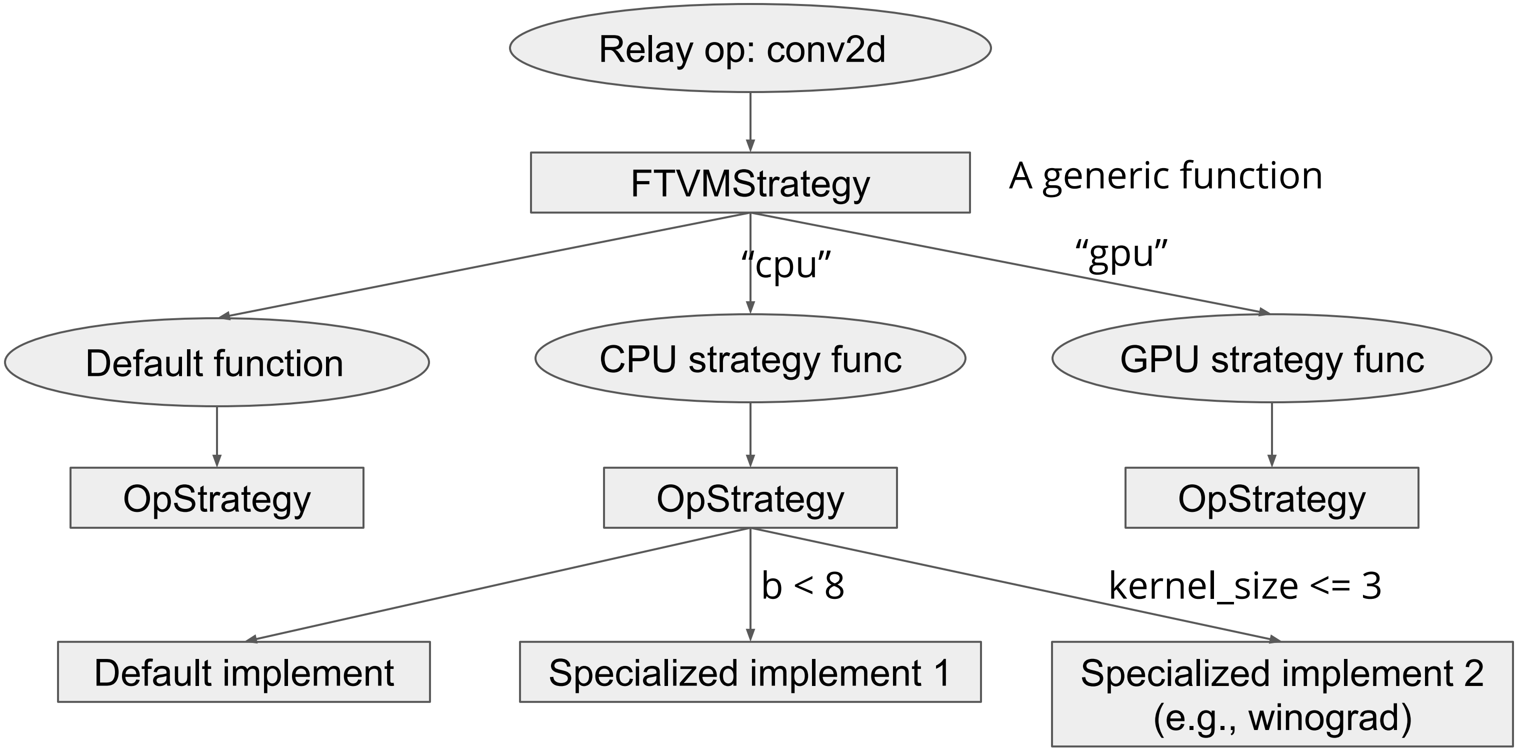
- Cannot compile and run dynamic models

Backup

Dynamic codegen: op dispatch (proposal)

- Goal: support codegen for dynamic shape
- Challenges
  - Single kernel performs poor across different shapes
  - Different templates for the same op
  - TVM compute and schedule are coupled together

Dynamic codegen: kernel dispatch (proposal)



Data structure

```
class SpecializedConditionNode : public Node {
  Array<Expr> conditions;
};

class OpImplementNode : public relay::ExprNode {
  FTVMCompute fcompute;
  FTVMSchedule fschedule;
  SpecializedCondition condition; // optional
};

class OpStrategyNode : public relay::ExprNode {
  OpImplement default_implement;
  Array<OpImplement> specialized_implements;
};

class OpStrategy : public relay::Expr {
  void RegisterDefaultImplement(FTVMCompute fcompute, FTVMSchedule fschedule, bool allow_override=false);
  void RegisterSpecializedImplement(FTVMCompute fcompute, FTVMSchedule fschedule,
                                    SpecializedCondition condition);
};
```

How to register a strategy?

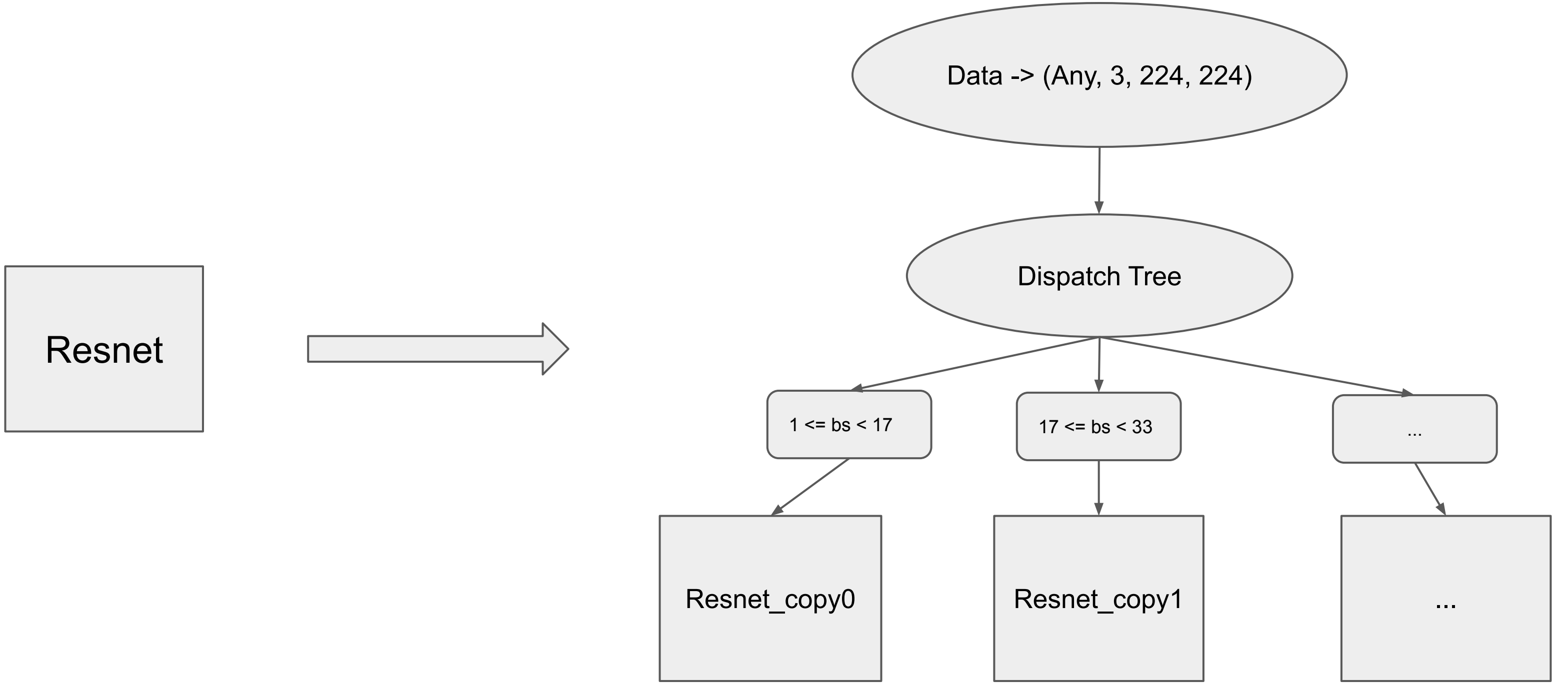
```python
@conv2d_strategy.register("cpu")
def conv2d_strategy_cpu(attrs, inputs, out_type, target):
    strategy = OpStrategy()
    layout = attrs.data_layout
    if layout == "NCHW":
        oc, ic, kh, kw = inputs[1].shape
        strategy.register_specialized_implement(wrap_compute_conv2d(topi.x86.conv2d_winograd),
                                                topi.x86.conv2d_winograd,
                                                [kh <= 3, kw <= 3])
        strategy.register_default_implement(wrap_compute_conv2d(topi.x86.conv2d_nchw),
                                            topi.x86.schedule_conv2d_nchw)
    elif layout == "NHWC":
        strategy.register_default_implement(wrap_compute_conv2d(topi.nn.conv2d_nhwc),
                                            topi.x86.schedule_conv2d_nhwc)
    elif layout == "NCHWc":
        strategy.register_default_implement(wrap_compute_conv2d(topi.nn.conv2d_nchwc),
                                            topi.x86.schedule_conv2d_nchwc)
    else: ...
    return strategy
```

Codegen for OpStrategy

- Each implementation defined will be compiled into a kernel in the module
- Dispatch logic will be compiled into another kernel as well

```
# pseudocode for dispatch kernel
def dispatch_kernel(*args):
   if specialized_condition1:
       specialized_kernel1(*args)
   elif specialized_condition2:
       specialized_kernel2(*args)
   ...
   else:
       default_kernel(*args) # corresponding to default implement
```

Dispatch a Whole Graph



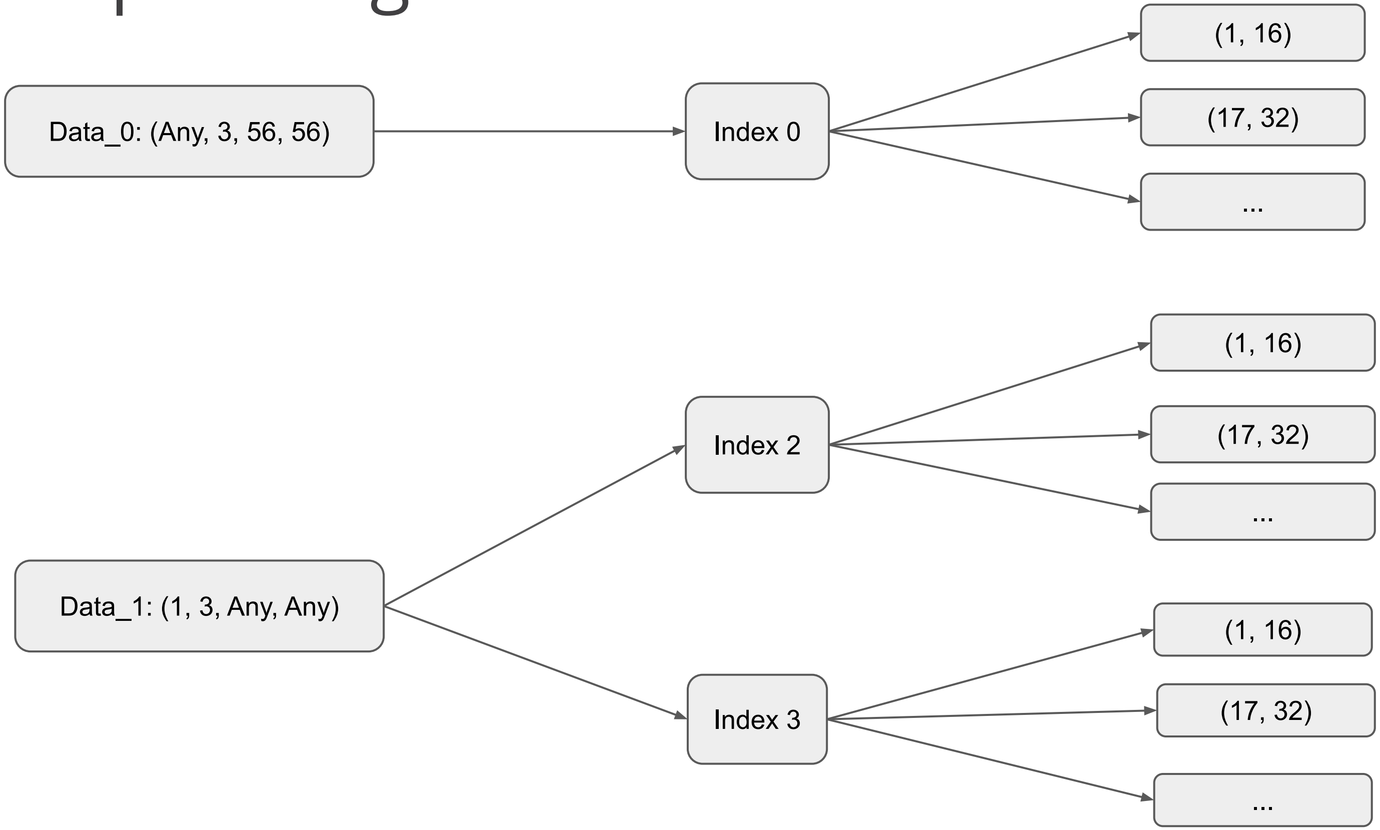
Why do we need graph dispatcher

1. Minimal overhead: only one dispatching operation is required for each inference.

2. Fit for operator such as conv2d_NCHWc. Graph tuning is well defined for each subgraph.

3. Avoid runtime layout tracking system for operator requires layout transformation to optimize.

Dispatching Function



API Example

```
input_name = "data"
input_shape = [tvm.relay.Any(), 3, 224, 224]
dtype = "float32"
block = get_model('resnet50_v1', pretrained=True)
mod, params = relay.frontend.from_mxnet(block, shape={input_name: input_shape}, dtype=dtype)
tvm.relay.transform.dispatch_global_func(mod, "main", {input_name: input_shape}, tvm.relay.vm.exp_dispatcher)
vmc = relay.backend.vm.VMCompiler()
with tvm.autotvm.apply_graph_best("resnet50_v1_graph_opt.log"):
    vm = vmc.compile(mod, "llvm")

vm.init(ctx)
vm.load_params(params)

data = np.random.uniform(size=(1, 3, 224, 224)).astype("float32")
out = vm.run(data)

data = np.random.uniform(size=(4, 3, 224, 224)).astype("float32")
out = vm.run(data)
```