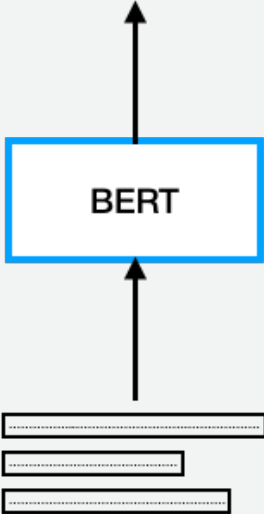




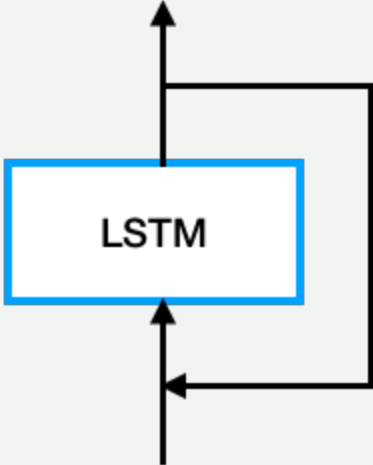
Nimble: Efficiently Compiling Dynamic Neural Networks for Model Inference

Haichen Shen*, Jared Roesch*, Zhi Chen, Wei Chen, Yong Wu,
Mu Li, Vin Sharma, Zachary Tatlock, Yida Wang

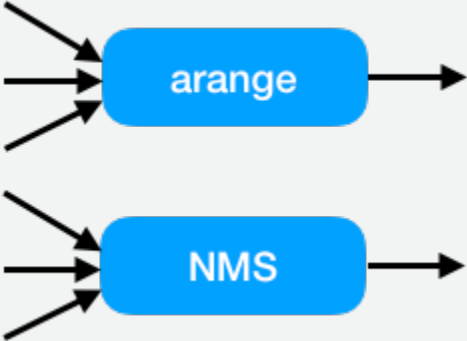
DNN models are exhibiting more dynamism



Dynamic input size



Control flow



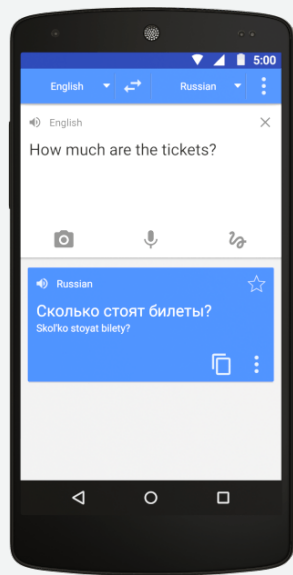
Dynamic output shapes



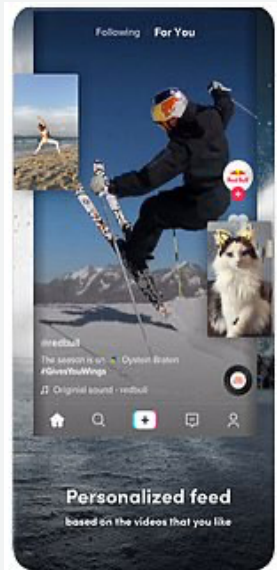
Dynamic model inference is an important workload



Smart speaker



Translation



Recommendation

Existing approaches to handle dynamism

1. Extend the representation: TensorFlow, MXNet
2. Rely on the host language: PyTorch, DyNet
3. Optimization for frameworks: TF Fold, JANUS

Limitation for inference

- ✗ Too heavyweight for model inference
- ✗ Lack portability: third-party libraries or Python
- ✗ Optimization doesn't apply to all types of models

Deep learning compilers are promising for model inference

XLA: Optimizing Compiler for Machine Learning

XLA (Accelerated Linear Algebra) is a domain-specific compiler for machine learning with potentially no source code changes.

The results are improvements in speed and memory usage: most inferences are 2x faster and use 50% less memory.

TVM: An Automated End-to-End Optimizing Compiler

Tianqi Chen¹, Thierry Moreau¹, Ziheng Jiang^{1,2}, Liang Song¹, Mehdi Amini¹, Mehdi Amini¹, Haichen Shen¹, Leyuan Wang^{4,2}, Yuwei Hu⁵, Luis Ceze¹, Paul G. Allen School of Computer Science & Engineering

² AWS, ³Shanghai Jiao Tong University, ⁴UC Berkeley

Glow: Graph Lowering Compiler Techniques for Neural Networks

Nadav Rotem, Jordan Fix, Saleem Abdulrasool, Garret Catron, Summer Deng, Roman Dzhabarov, Nick Gibson, James Hegeman, Meghan Lele, Roman Levenstein,

MLIR: A Compiler Infrastructure for the End of Moore's Law

Chris Lattner* Google Mehdi Amini Google Uday Bondhugula IISc Albert Cohen Google Andy Davis Google

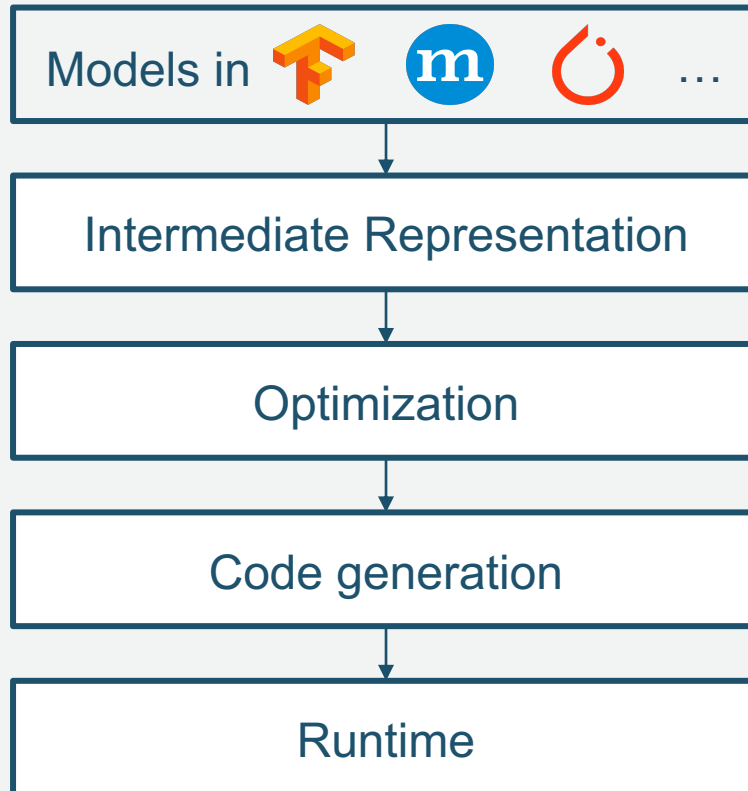
Jacques Pienaar Google River Riddle Google Tatiana Shpeisman Google Nicolas Vasilache Google

Oleksandr Zinenko Google

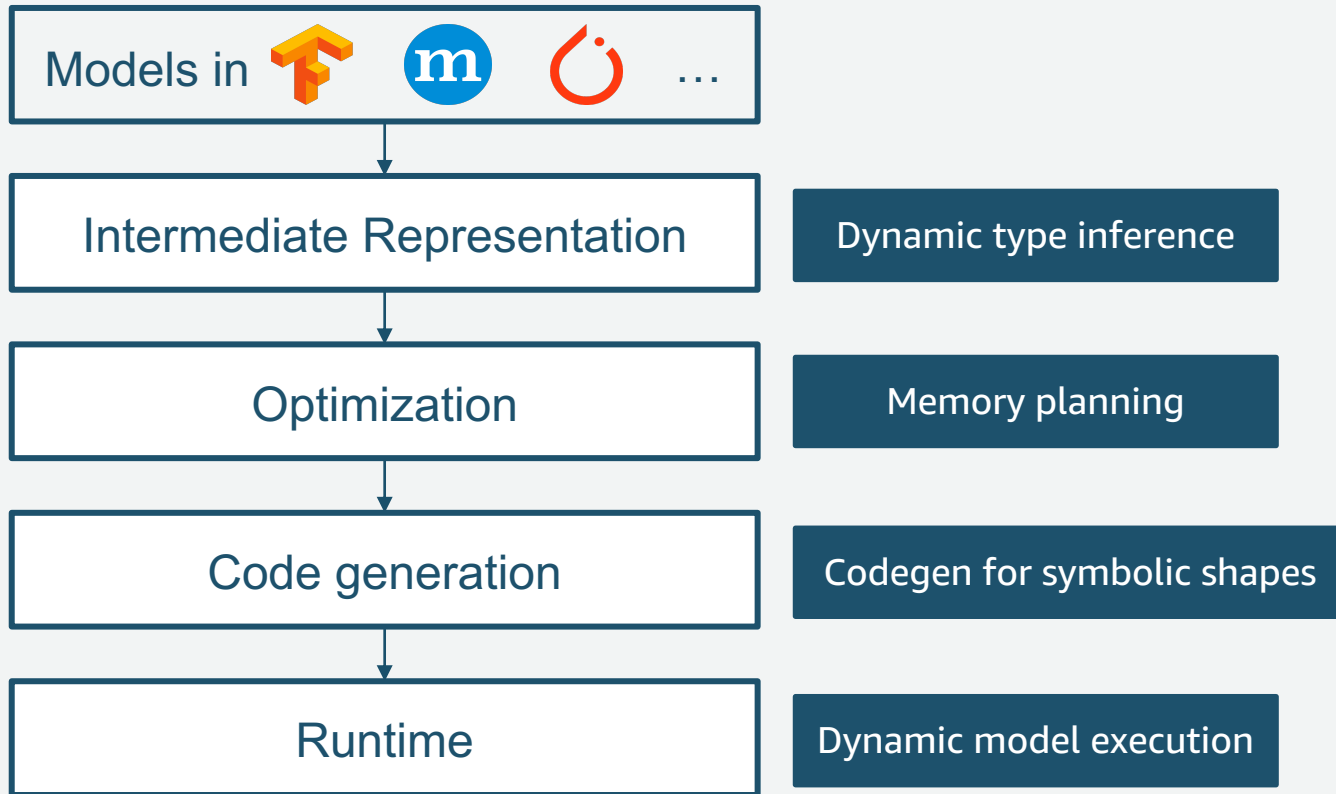
But none of them fully support dynamic models...



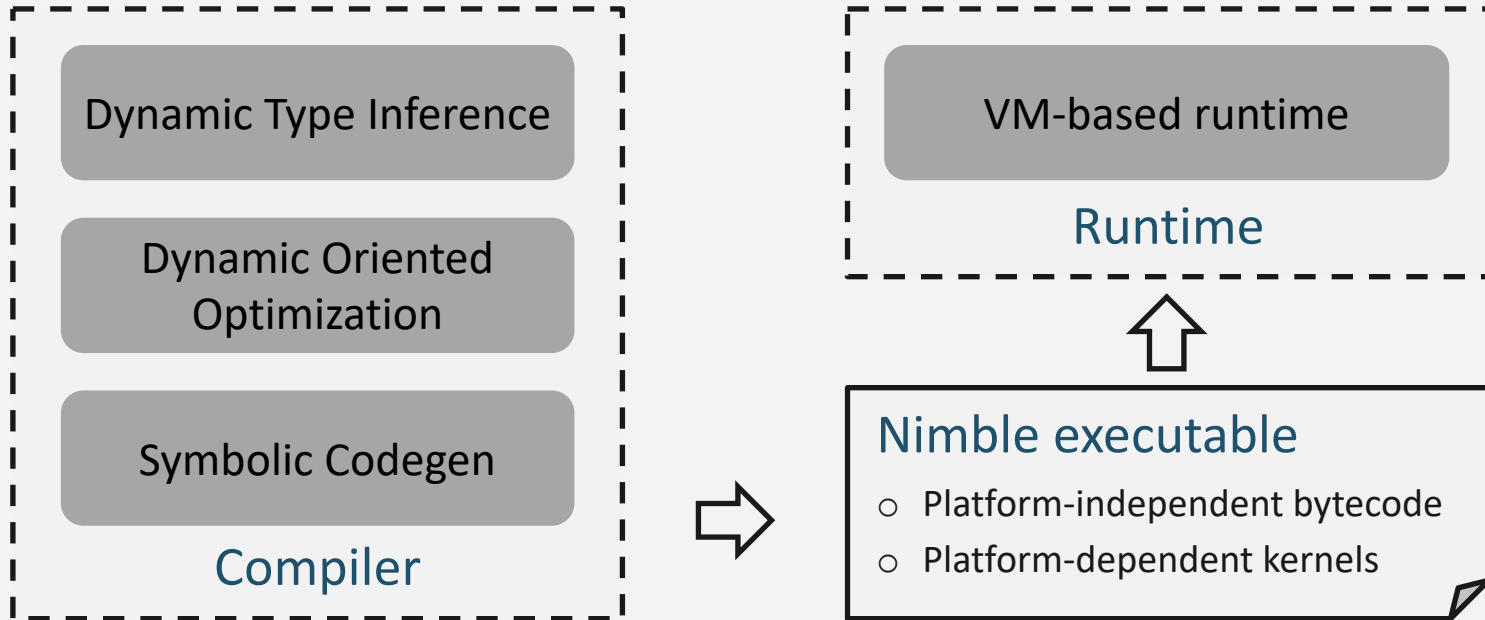
Challenges to support dynamic models



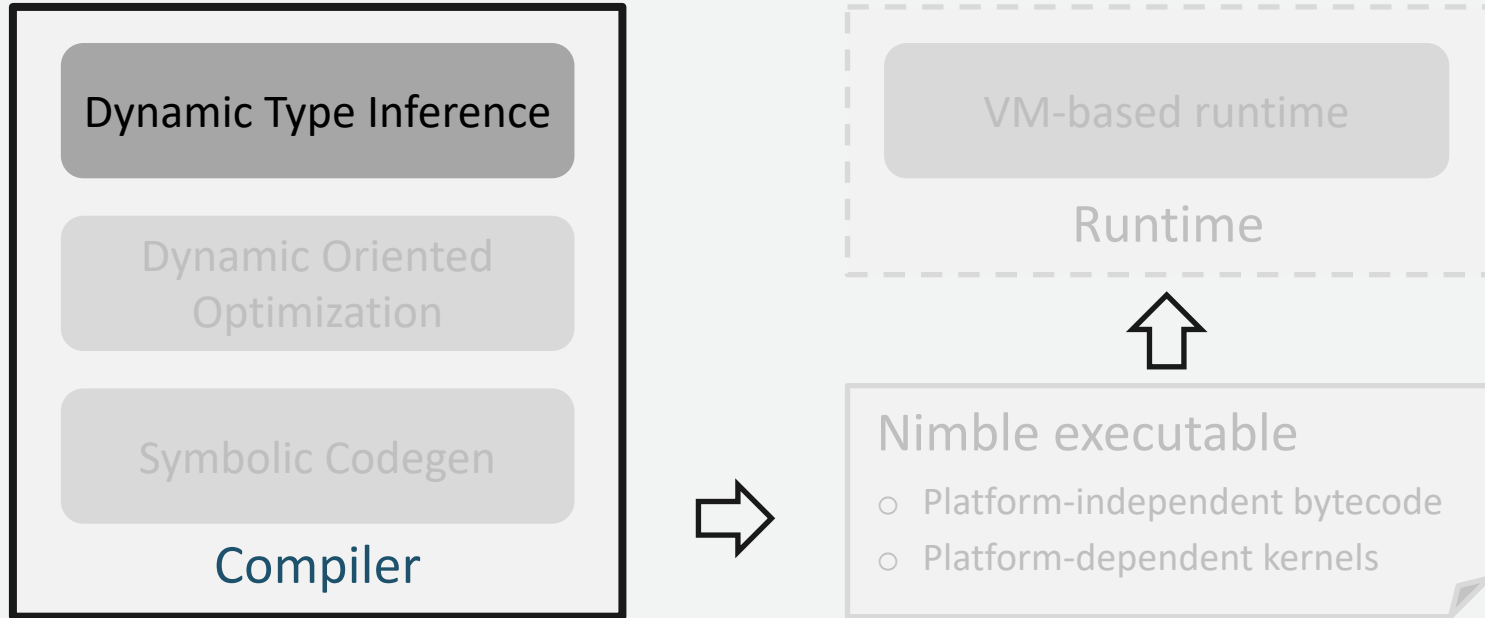
Challenges to support dynamic models



Nimble: compile and execute dynamic models



Nimble: compile and execute dynamic models



Any: typing dynamic dimension

Any: an unknown dimension at compilation time

Define a tensor type:

Tensor<(Any, 3, 32, 32), fp32>

Any in operator type relation

Describe the type relation between inputs and outputs

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

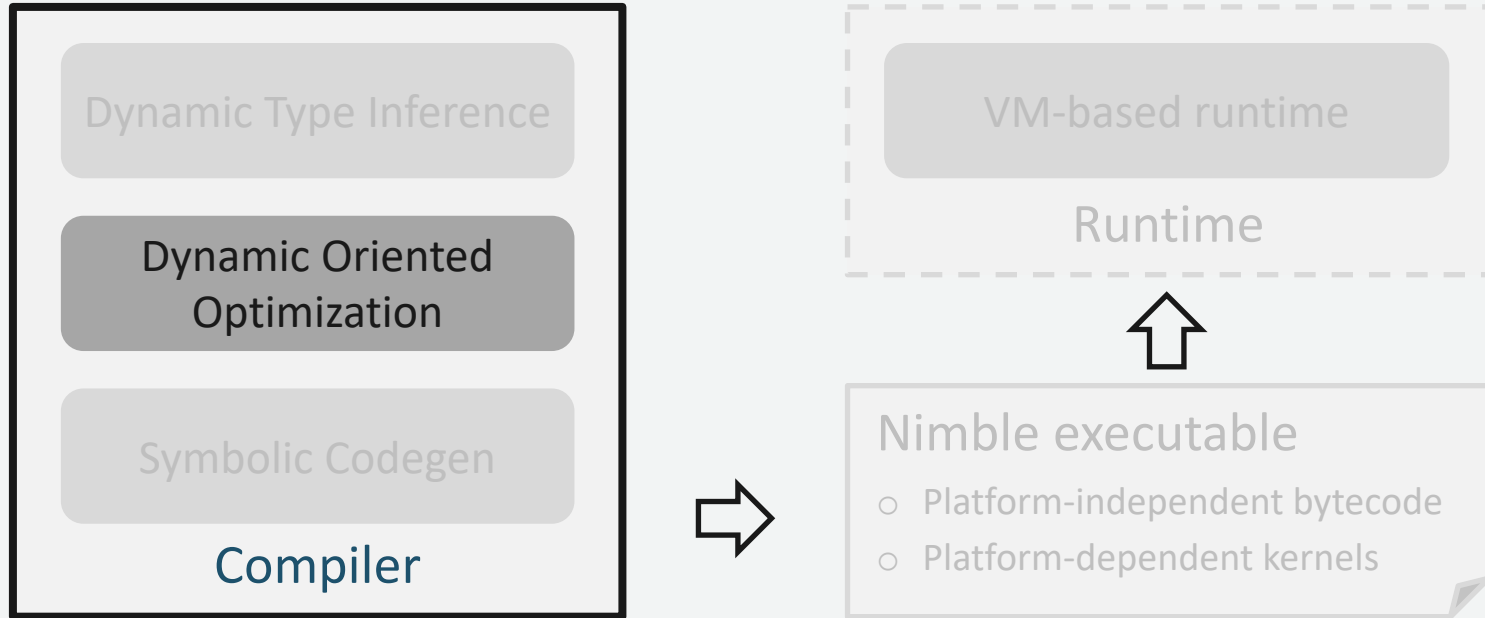
```
broadcast: fn(Tensor<(Any, Any), fp32>,  
              Tensor<( 1, 8), fp32>)  
           -> Tensor<(Any, 8), fp32>
```

Valid only when Any = 1 or 8

How to infer the shape at runtime?

- Instrument *shape functions* in the program
 - Calculate the output shape
 - Perform the type checking
- Advantages of shape function:
 - Low overhead at runtime
 - Treat as regular ops and apply optimization
 - Generate shape functions for fused ops

Nimble: compile and execute dynamic models



Problem in memory planning

Existing deep learning compilers don't encode memory allocation in IRs

- Memory planning coupled with runtime
- Complicated under heterogeneous execution
- Don't support dynamic memory allocation

Approach for memory planning

Explicitly manifest the memory allocation in the program

- Perform optimization such as liveness analysis, device placement
- No runtime modification and negligible runtime overhead

Introduce new IR nodes

- `invoke_mut`
- `alloc_storage`
- `alloc_tensor`
- `kill`

Example 1: Manifest the memory allocation (static shape)

```
fn main(t1, t2: Tensor<10>) -> Tensor<10> {  
  
    add(t1, t2)  
  
}
```


Example 1: Manifest the memory allocation (static shape)

```
fn main(t1, t2: Tensor<10>) -> Tensor<10> {  
    let buf = alloc_storage(size=40);  
    let out = alloc_tensor(buf, offset=0, shape=(10), dtype=f32);  
    add(t1, t2)  
}
```

1. Explicit allocate output buffer

Example 1: Manifest the memory allocation (static shape)

```
fn main(t1, t2: Tensor<10>) -> Tensor<10> {  
    let buf = alloc_storage(size=40);  
    let out = alloc_tensor(buf, offset=0, shape=(10), dtype=f32);  
    invoke_mut(add, (t1, t2), (out));  
    out  
}
```

2. Update the kernel call with explicit output buffer

Example 2: Manifest the memory allocation (dynamic shape)

```
fn (x: Tensor<?, 2>, y: Tensor<1, 2>)->Tensor<?, 2> {
```

```
    concat((x, y))
```

```
}
```

Example 2: Manifest the memory allocation (dynamic shape)

```
fn (x: Tensor<?, 2>, y: Tensor<1, 2>)->Tensor<?, 2> {  
  let xshape = shape_of(x);  
  let yshape = shape_of(y);  
  
  concat((x, y))  
}
```

1. Extract the shape from tensors

Example 2: Manifest the memory allocation (dynamic shape)

```
fn (x: Tensor<?,2>, y: Tensor<1,2>)->Tensor<?,2> {  
  let xshape = shape_of(x);  
  let yshape = shape_of(y);  
  
  invoke_shape_func(concat, (xshape, yshape), (oshape), ...);  
  
  concat((x, y))  
}
```

2. Compute the output shape using shape function

Example 2: Manifest the memory allocation (dynamic shape)

```
fn (x: Tensor<?,2>, y: Tensor<1,2>)->Tensor<?,2> {  
  let xshape = shape_of(x);  
  let yshape = shape_of(y);  
  
  invoke_shape_func(concat, (xshape, yshape), (oshape), ...);  
  let buf1 = alloc_storage(size=oshape);  
  let out = alloc_tensor(buf1, oshape, ...);  
  concat((x, y))  
  
}
```

3. Allocate the output buffer using the calculated output shape

Example 2: Manifest the memory allocation (dynamic shape)

```
fn (x: Tensor<?,2>, y: Tensor<1,2>)->Tensor<?,2> {  
  let xshape = shape_of(x);  
  let yshape = shape_of(y);  
  
  invoke_shape_func(concat, (xshape, yshape), (oshape), ...);  
  let buf1 = alloc_storage(size=oshape);  
  let out = alloc_tensor(buf1, oshape, ...);  
  invoke_mut(concat, (x, y), (out));  
  out  
}
```

4. Update the kernel call with explicit output buffers

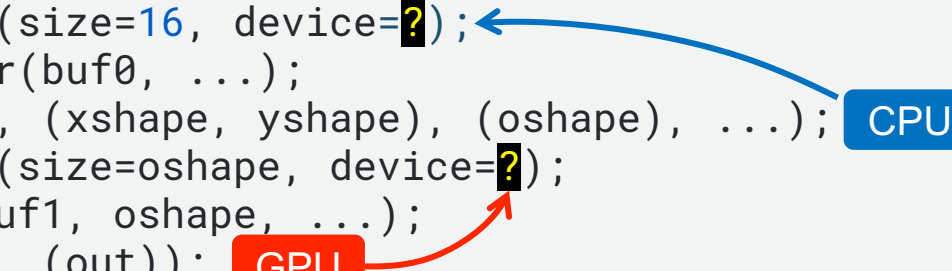
Example 2: Manifest the memory allocation (dynamic shape)

```
fn (x: Tensor<?,2>, y: Tensor<1,2>)->Tensor<?,2> {  
  let xshape = shape_of(x);  
  let yshape = shape_of(y);  
  let buf0 = alloc_storage(size=16);  
  let oshape = alloc_tensor(buf0, ...);  
  invoke_shape_func(concat, (xshape, yshape), (oshape), ...);  
  let buf1 = alloc_storage(size=oshape);  
  let out = alloc_tensor(buf1, oshape, ...);  
  invoke_mut(concat, (x, y), (out));  
  out  
}
```

5. Manifest memory allocation for shape functions

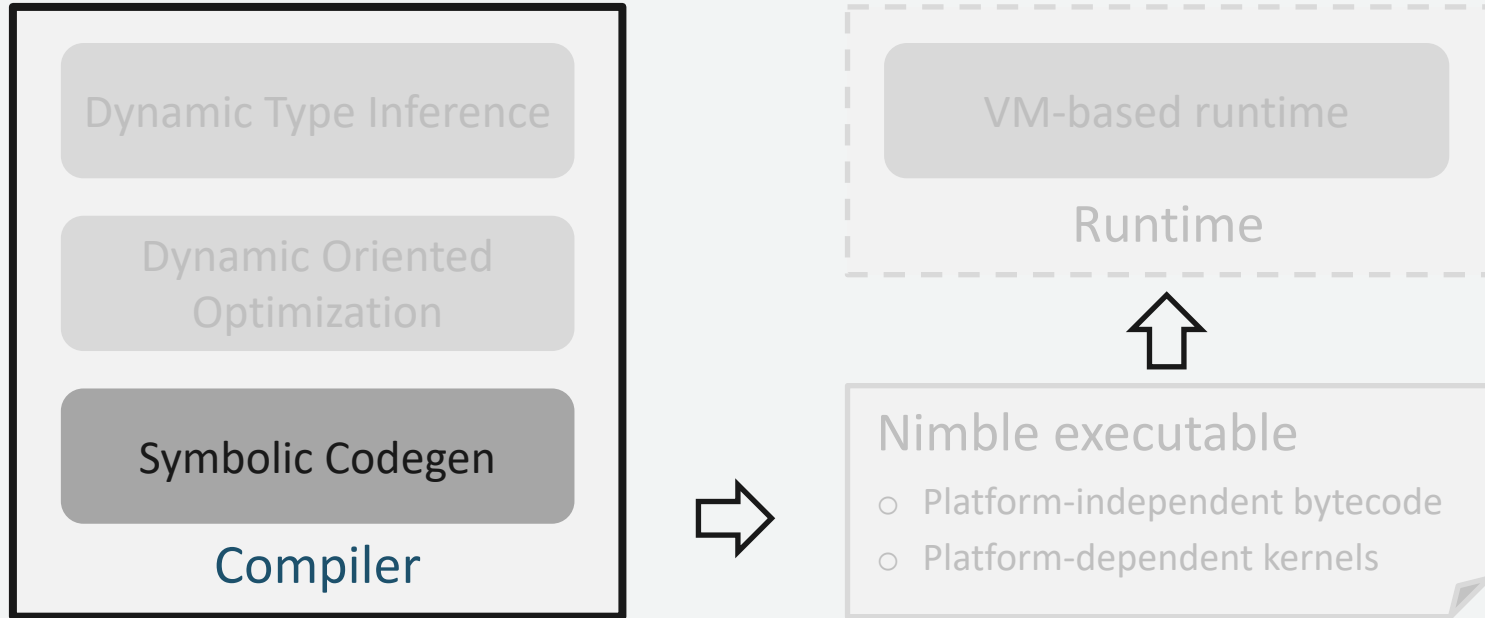
Which device to place each buffer?

```
fn (x: Tensor<?,2>, y: Tensor<1,2>)->Tensor<?,2> {  
  let xshape = shape_of(x);  
  let yshape = shape_of(y);  
  let buf0 = alloc_storage(size=16, device=?);  
  let oshape = alloc_tensor(buf0, ...);  
  invoke_shape_func(concat, (xshape, yshape), (oshape), ...);  
  let buf1 = alloc_storage(size=oshape, device=?);  
  let out = alloc_tensor(buf1, oshape, ...);  
  invoke_op(concat, (x, y), (out));  
  out  
}
```



Use constraints and union-find algorithm

Nimble: compile and execute dynamic models



Challenges to symbolic code generation

Symbolic-shaped kernels perform worse than static-shaped kernels.

How to tune kernels with symbolic shapes?

Challenges to symbolic code generation

Symbolic-shaped kernels perform worse than static-shaped kernels.

- Loop tiling + parallelism → boundary check in the loop body

How to tune kernels with symbolic shapes?

Challenges to symbolic code generation

Symbolic-shaped kernels perform worse than static-shaped kernels.

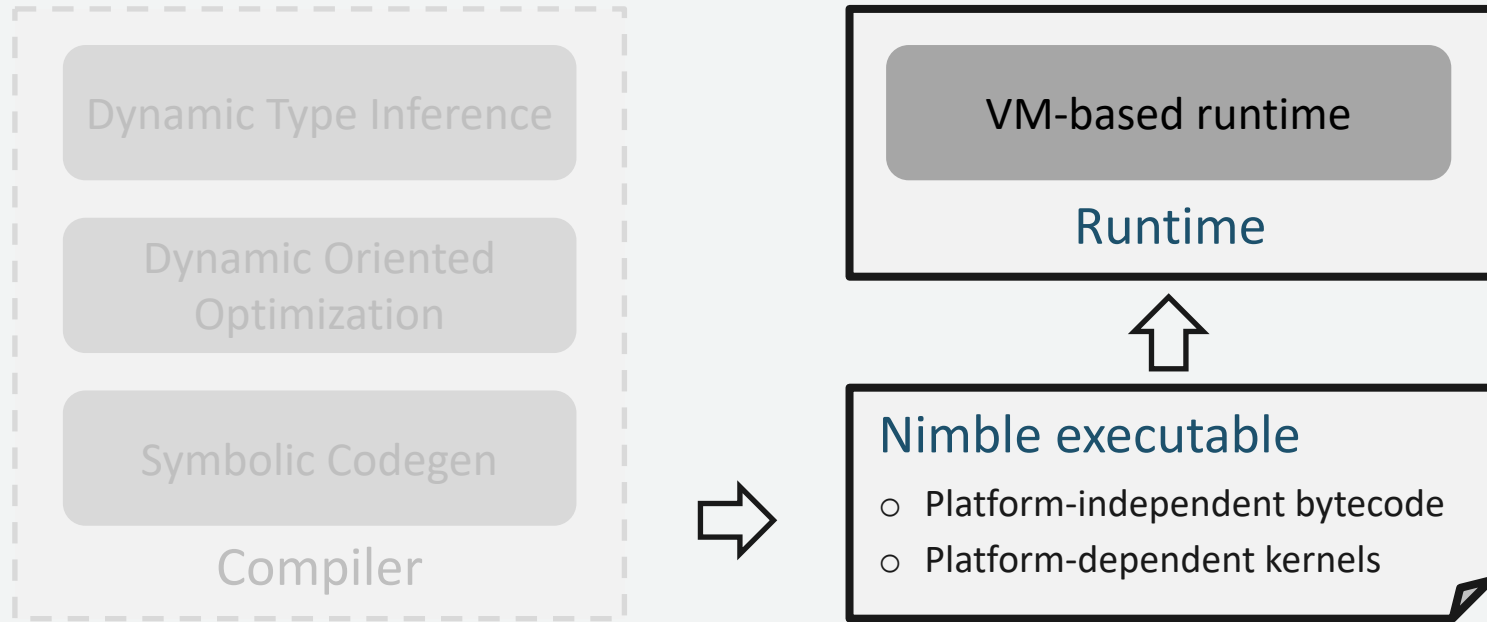
- Loop tiling + parallelism → boundary check in the loop body
 - ❑ Generate multiple kernels based on the tiling factor
 - ❑ Use symbolic simplifier to remove the boundary check
 - ❑ Dispatch to a corresponding kernel at runtime

How to tune kernels with symbolic shapes?

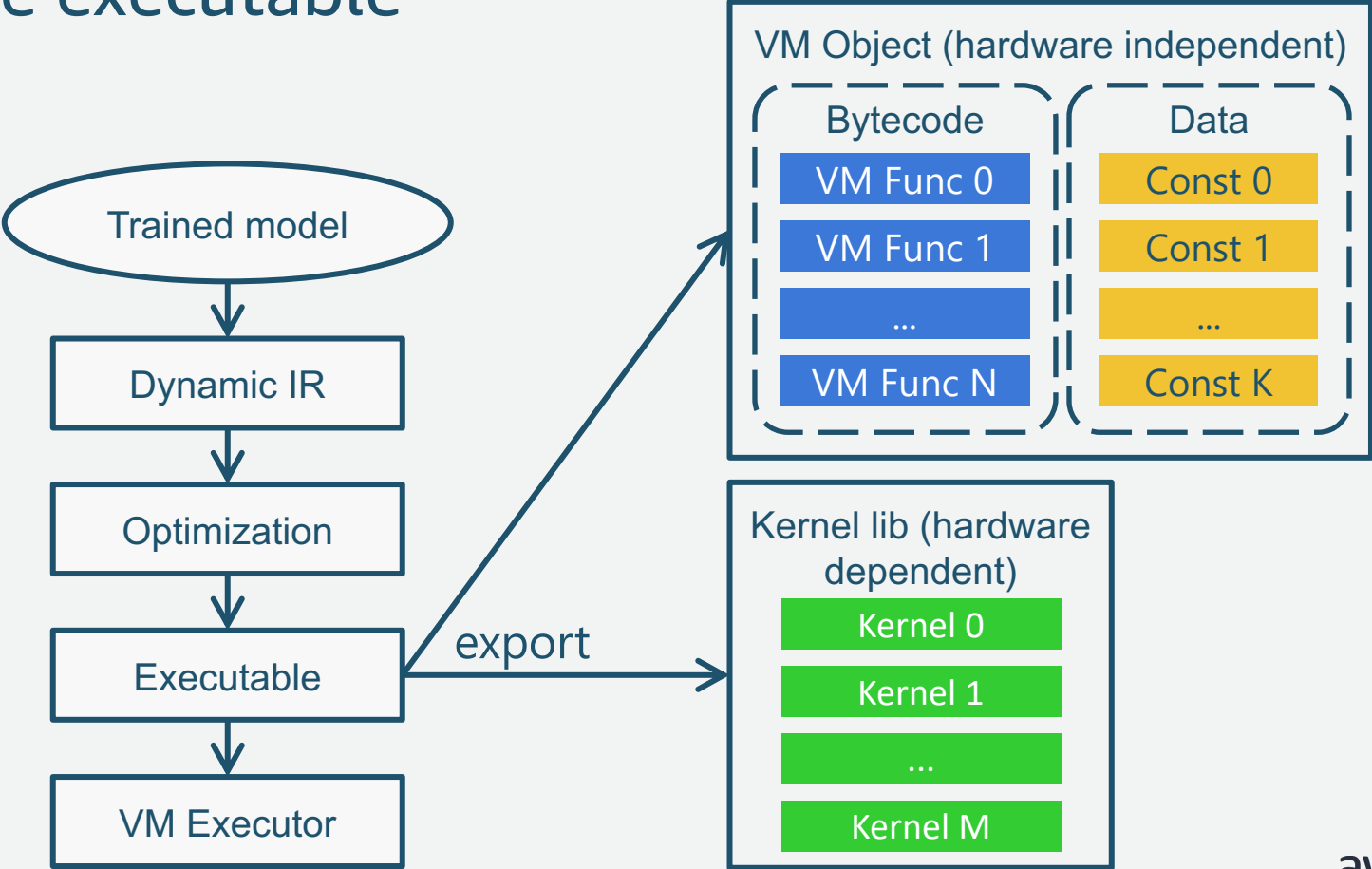
Tuning for symbolic shape

1. Tune the kernel after replacing the symbolic dims by a large value (e.g., 64, 128)
2. Pick top k configurations, and evaluate the performance on other shapes
3. Pick the configuration that performs best on average among shapes previously evaluated

Nimble: compile and execute dynamic models



Nimble executable



Tensor-oriented CISC-style VM ISA

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.
DeviceCopy	Copies a chunk of data from one device to another.



Evaluation

What is the overall performance?

How much overhead does *Nimble* introduce for handling dynamism?

How effective are the proposed optimization techniques?

Evaluation

What is the overall performance?

How much overhead does *Nimble* introduce for handling dynamism?

How effective are the proposed optimization techniques?



Evaluation Setup

Models

- LSTM (control flow), Tree-LSTM (dynamic data structure), BERT (dynamic input shapes)

Dataset

- MRPC for LSTM and BERT, Stanford Sentiment Treebank for Tree-LSTM

EC2 instances

- c5.9xlarge (Intel CPU), g4dn.4xlarge (Nvidia GPU), a1.4xlarge (ARM CPU)

Compare to MXNet, PyTorch, DyNet, TensorFlow, TF Fold

Use batch = 1 for all cases



Evaluation: LSTM models

Unit: us/token	1 layer			2 layers		
	Intel	Nvidia	ARM	Intel	Nvidia	ARM
Nimble	47.8	54.6	182.2	97.2	107.4	686.4
PyTorch	2.2x	1.5x	15.0x	2.3x	1.5x	8.5x
DyNet	19.6x	1.3x	31.3x	24.2x	1.3x	18.7x
MXNet	4.5x	2.5x	20.3x	4.1x	2.1x	11.3x
TensorFlow	6.3x	5.6x	5.4x	7.1x	3.8x	3.2x

Evaluation: Tree-LSTM and BERT

	Intel	ARM
Nimble	40.3	86.3
PyTorch	17.4x	19.9x
DyNet	2.4x	3.6x
TF Fold	5.2x	-

Tree-LSTM latency (us/token)

	Intel	Nvidia	ARM
Nimble	307.0	95.2	2862.6
PyTorch	1.6x	2.3x	4.1x
MXNet	1.5x	1.6x	3.0x
TensorFlow	2.5x	1.3x	1.05x

BERT latency (us/token)



Nimble overhead compared to static compiler

Device	TVM lat. (ms)	Nimble lat. (ms)	Kernel lat. (ms)	Others (ms)
Intel	19.4	24.3	21.1	3.26
ARM	223.5	237.4	228.6	8.82
Nvidia	5.6	5.9	5.6	0.26

BERT with sequence length 128



Conclusion

- Nimble compiles and optimizes neural networks with dynamism
- We design and implement a lightweight and portable VM-based runtime
- Nimble lowers the latency by up to 30x compared to baseline on multiple hardware platforms.

Thank you

