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 TensorFlow and PyTorch, with potentially no source code changes. The results are improvements in speed and memory usage: most inference is at least 2x faster than on CPU.

TVM: An Automated End-to-End Optimizing Compiler

TVM is an open source compiler for machine learning. It supports multiple backends, including CPU, GPU, and more.

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, David Leventhal, Richard Norton, and Sathya Sivasubramaniam

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

Chris Lattner*, Mehdi Amini, Uday Bondhugula, Albert Cohen, Andy Davis, Jacques Pienaar, River Riddle, Tatiana Shepsman, Nicolas Vasilevskiy, and Oleksandr Zinenko

But none of them fully support dynamic models…
Challenges to support dynamic models

- Models in $\text{T}$, $\text{m}$, $\text{o}$, ...
- Intermediate Representation
- Optimization
- Code generation
- Runtime
Challenges to support dynamic models

1. Models in Intermediate Representation
2. Intermediate Representation Optimization
3. Code generation
4. Runtime

- Dynamic type inference
- Memory planning
- Codegen for symbolic shapes
- Dynamic model execution
Nimble: compile and execute dynamic models

Compiler
- Dynamic Type Inference
- Dynamic Oriented Optimization
- Symbolic Codegen

Nimble executable
- Platform-independent bytecode
- Platform-dependent kernels

Runtime
- VM-based runtime
Nimble: compile and execute dynamic models

Compiler

- Dynamic Type Inference
- Dynamic Oriented Optimization
- Symbolic Codegen

Runtime

VM-based runtime

Nimble executable
- Platform-independent bytecode
- Platform-dependent kernels

Platform-independent bytecode
Platform-dependent kernels
Any: typing dynamic dimension

Any: an unknown dimension at compilation time

Define a tensor type:

\[
\text{Tensor<}(\text{Any}, 3, 32, 32), \text{ fp32}>)
\]
Any in operator type relation

Describe the type relation between inputs and outputs

```plaintext
range: 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

Compiler
- Dynamic Type Inference
- Dynamic Oriented Optimization
- Symbolic Codegen

Runtime
- VM-based runtime
- Nimble executable
  - Platform-independent bytecode
  - Platform-dependent kernels
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)

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

    add(t1, t2)
}
```
Example 1: Manifest the memory allocation (static shape)

```rust
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)

```rust
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)

```plaintext
fn (x: Tensor<?, 2>, y: Tensor<1, 2>) -> Tensor<?, 2> {
    concat((x, y))
}
```
Example 2: Manifest the memory allocation (dynamic shape)

```rust
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)

```rust
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)

```rust
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)

```rust
fn (x: Tensor<?, 2>, y: Tensor<1, 2>)\rightarrow 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)

```rust
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?

```rust
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=CPU);
    let oshape = alloc_tensor(buf0, ...);
    invoke_shape_func(concat, (xshape, yshape), (oshape), ...);
    let buf1 = alloc_storage(size=oshape, device=GPU);
    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

Compiler

- Dynamic Type Inference
- Dynamic Oriented Optimization
- Symbolic Codegen

Platform-independent bytecode
Platform-dependent kernels

Runtime

VM-based runtime

Nimble executable
- Platform-independent bytecode
- Platform-dependent kernels
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

Compiler

Dynamic Type Inference
Dynamic Oriented Optimization
Symbolic Codegen

Nimble executable
- Platform-independent bytecode
- Platform-dependent kernels

Runtime

VM-based runtime
Nimble executable

Trained model -> Dynamic IR -> Optimization -> Executable -> VM Executor

VM Object (hardware independent):
- Bytecode:
  - VM Func 0
  - VM Func 1
  - ...
  - VM Func N
- Data:
  - Const 0
  - Const 1
  - ...
  - Const K

Kernel lib (hardware dependent):
- Kernel 0
- Kernel 1
- ...
- Kernel M
## Tensor-oriented CISC-style VM ISA

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

<table>
<thead>
<tr>
<th>Unit: us/token</th>
<th>1 layer</th>
<th>2 layers</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Intel</td>
<td>Nvidia</td>
</tr>
<tr>
<td>Nimble</td>
<td>47.8</td>
<td>54.6</td>
</tr>
<tr>
<td>PyTorch</td>
<td>2.2x</td>
<td>1.5x</td>
</tr>
<tr>
<td>DyNet</td>
<td>19.6x</td>
<td>1.3x</td>
</tr>
<tr>
<td>MXNet</td>
<td>4.5x</td>
<td>2.5x</td>
</tr>
<tr>
<td>TensorFlow</td>
<td>6.3x</td>
<td>5.6x</td>
</tr>
</tbody>
</table>
## Evaluation: Tree-LSTM and BERT

<table>
<thead>
<tr>
<th></th>
<th>Intel</th>
<th>ARM</th>
</tr>
</thead>
<tbody>
<tr>
<td>Nimble</td>
<td>40.3</td>
<td>86.3</td>
</tr>
<tr>
<td>PyTorch</td>
<td>17.4x</td>
<td>19.9x</td>
</tr>
<tr>
<td>DyNet</td>
<td>2.4x</td>
<td>3.6x</td>
</tr>
<tr>
<td>TF Fold</td>
<td>5.2x</td>
<td>-</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>Intel</th>
<th>Nvidia</th>
<th>ARM</th>
</tr>
</thead>
<tbody>
<tr>
<td>Nimble</td>
<td>307.0</td>
<td>95.2</td>
<td>2862.6</td>
</tr>
<tr>
<td>PyTorch</td>
<td>1.6x</td>
<td>2.3x</td>
<td>4.1x</td>
</tr>
<tr>
<td>MXNet</td>
<td>1.5x</td>
<td>1.6x</td>
<td>3.0x</td>
</tr>
<tr>
<td>TensorFlow</td>
<td>2.5x</td>
<td>1.3x</td>
<td>1.05x</td>
</tr>
</tbody>
</table>

- **Tree-LSTM latency (us/token)**
- **BERT latency (us/token)**
Nimble overhead compared to static compiler

<table>
<thead>
<tr>
<th>Device</th>
<th>TVM lat. (ms)</th>
<th>Nimble lat. (ms)</th>
<th>Kernel lat. (ms)</th>
<th>Others (ms)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Intel</td>
<td>19.4</td>
<td>24.3</td>
<td>21.1</td>
<td>3.26</td>
</tr>
<tr>
<td>ARM</td>
<td>223.5</td>
<td>237.4</td>
<td>228.6</td>
<td>8.82</td>
</tr>
<tr>
<td>Nvidia</td>
<td>5.6</td>
<td>5.9</td>
<td>5.6</td>
<td>0.26</td>
</tr>
</tbody>
</table>

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