# NIMBLE: EFFICIENTLY COMPILING DYNAMIC NEURAL NETWORKS FOR MODEL INFERENCE

**Haichen Shen** [*1] **Jared Roesch** [*2] **Zhi Chen** [1] **Wei Chen** [1] **Yong Wu** [1]
**Mu Li** [1] **Vin Sharma** [1] **Zachary Tatlock** [3] **Yida Wang** [1]

## ABSTRACT

Modern deep neural networks increasingly make use of features such as control flow, dynamic data structures, and dynamic tensor shapes. Existing deep learning systems focus on optimizing and executing static neural networks which assume a pre-determined model architecture and input data shapes—assumptions that are violated by dynamic neural networks. Therefore, executing dynamic models with deep learning systems is currently both inflexible and sub-optimal, if not impossible. Optimizing dynamic neural networks is more challenging than static neural networks; optimizations must consider all possible execution paths and tensor shapes. This paper proposes *Nimble*, a high-performance and flexible system to optimize, compile, and execute dynamic neural networks on multiple platforms. *Nimble* handles model dynamism by introducing a dynamic type system, a set of dynamism-oriented optimizations, and a light-weight virtual machine runtime. Our evaluation demonstrates that *Nimble* outperforms existing solutions for dynamic neural networks by up to 20× on hardware platforms including Intel CPUs, ARM CPUs, and Nvidia GPUs.

## 1 INTRODUCTION

As deep learning-based applications have become ubiquitous, so have systems for optimizing, executing, and deploying such applications. A number of systems research projects focus on enhancing the performance of a subset of pre-trained models produced by deep learning (DL) researchers on various platforms (Dahl et al., 2012; Han et al., 2016; Johnson et al., 2016; Liu et al., 2019; Wang et al., 2019). Specifically, these models represented as static data flow graphs where the sizes of each input and output (i.e., tensors or $n$-dimensional arrays) are known a priori, ensuring the execution path remains unchanged on every invocation. We refer to models with this static nature as *static models*. Continued advances in neural networks, especially those in natural language processing, have introduced new dynamism in models, such as control flow (Hochreiter & Schmidhuber, 1997; Sutskever et al., 2014), dynamic data structures (Tai et al., 2015; Liang et al., 2016), and dynamic shapes (Devlin et al., 2018). We refer to models exhibiting these behaviors as *dynamic models*.

As dynamic models mature and continue to move from research to production, it calls for an efficient and cross-platform inference system. This poses new challenges for deep learning practitioners, as dynamic models introduce input-dependent graph topology, breaking existing system assumptions and invalidating optimizations designed for purely static data flow graphs. However, no existing solutions fulfill these requirements.

Many existing approaches to dynamic model optimization apply or extend existing deep learning frameworks (Xu et al., 2018; Gao et al., 2018; Yu et al., 2018; Jeong et al., 2018; 2019; Neubig et al., 2017; Looks et al., 2017). However, deep learning frameworks optimized for training can be limiting in model inference settings due to their rich feature set. In order to realize these features frameworks are often monolithic, large, and non-portable. Moreover, approaches which inherit from frameworks rely on third-party kernel libraries such as OpenBLAS (Zhang et al., 2014), cuDNN (Chetlur et al., 2014), and oneDNN (Intel, 2020) to achieve competitive performance. These libraries expose a fixed set of operators for the corresponding hardware, compromising the portability of dynamic models which require a large number of operators with varying data types and shapes. Designing a new interface independent of existing frameworks provides a clean programming model but often at the cost of performance, due to dynamic interpretation of the model (Neubig et al., 2017).

An alternative approach that has generated significant interest in both academia and industry is the end-to-end optimization of neural networks using deep learning compilers, such as XLA (XLA Team, 2017), Glow (Rotem et al., 2018), TVM (Chen et al., 2018a), and MLIR (Lattner et al., 2021). Deep learning compilers differ from traditional deep learning frameworks by separating execution into a compi-

[*]Equal contribution [1]Amazon Web Services [2]OctoML [3]University of Washington. Correspondence to: Haichen Shen <shaichen@amazon.com>.

*Proceedings of the $4^{th}$ MLSys Conference*, San Jose, CA, USA, 2021. 

lation, and a runtime phase. The compilation phase enables whole-model optimization at the graph level, and workload specific kernel code-generation for multiple hardware platforms, while the runtime phase executes the compiled module.

However, deep learning compilers have been primarily restricted to static models due to lack of support for dynamism. Specifically, in order to compile and execute the dynamic models, a system requires an intermediate representation (IR) which can statically represent dynamic constructs, a code generator which can generate kernels for dynamically varying data shapes, and a runtime to handle the dynamic execution and kernel dispatch accordingly. Further, dynamic-specific optimizations, such as dynamic memory planning, the process of statically optimizing dynamic allocations, are necessary to achieve desirable performance. None of these features exist in the current deep learning compilers.

To this end, we present *Nimble*, a high-performance and portable system for compiling, optimizing, and executing dynamic neural networks on multiple platforms. To the best of our knowledge, this is the first attempt to systematically handle dynamic models from a compiler perspective. First, we introduce type system extensions to handle data with unknown dimension, which is common in dynamic models, by performing type checking and inference for shapes with *Any*. Second, we devise several optimizations specific to dynamic models, including dynamic shape-aware code generation, memory planning, and device placement. Third, we propose a virtual machine (VM)-based runtime, which decouples the platform-independent controlling logic and platform-dependent kernel implementation, to be portable, light-weight, and most importantly, able to execute dynamic models. Evaluation on LSTM (Hochreiter & Schmidhuber, 1997), Tree-LSTM (Tai et al., 2015) and BERT (Devlin et al., 2018) shows that *Nimble* lowers the latency by 1.05× to 19.9× compared to the best solution whichever on mainstream hardware platforms both in the cloud (Intel CPUs and Nvidia GPUs) and at the edge (ARM CPUs).

In summary, this paper makes the following three core contributions:

- Proposes and builds an end-to-end system for efficient dynamic model inference across multiple hardware platforms, including an empirical study to benchmark the results;
- Devises several compilation and optimization techniques, including a dynamic type system, a memory planning pass, a heterogeneous device placement mechanism to place computation and data, and a symbolic kernel code generation and shape-based dispatch algorithm;
- Designs and implements tensor based abstract machine with a platform-independent instruction set to efficiently and flexibly execute dynamic models across platforms.

The rest of the paper is organized as follows. Section 2 reviews the limitation of existing deep learning compilers and gives the overview of *Nimble*. Section 3 presents the design and implementation of the compilation flow of *Nimble*, followed by VM-based runtime in Section 4. Section 5 provides the evaluation results using various models on different hardware platforms. Section 6 covers related work, and Section 7 concludes the paper.

## 2 Challenges and Our Approach

### 2.1 Limitation of Deep Learning Compilers

As aforementioned, existing solutions to dynamic models either rely on or extend deep learning frameworks. These solutions bring significant challenges in portability and cross-platform support due to the gigantic codebase and the vendor library dependency. Deep learning compilers provide an alternative approach as being portable across platforms with minimal memory footprint by virtue of being light-weight and dependency-free.

However, current deep learning compilers are not able to process dynamic models due to missing the following dynamism-specific features.

- **An IR for representing dynamism.** Performing data type and shape inference on static models is straightforward as they are known during declaration and remain unchanged during runtime. However, the shape of an input tensor may vary wildly across different input samples in a dynamic model. The emergence of control flow constructs further complicates this problem as different execution paths can emit substantially different data. A fully static IR, hence, is inadequate to cope with the dynamic characteristics of these models.
- **A set of dynamic-oriented optimizations.** Existing deep learning compilers, e.g. TVM (Chen et al., 2018a) and Glow (Rotem et al., 2018), expect static input for each optimization. The memory spaces of each tensor are pre-allocated and their live cycles are determined using a dedicated optimization pass. They also ensure the homogeneous execution of the entire model because all kernels are executed on the same device. However, these optimizations may completely break when dynamism appears, in which different execution paths possibly require different amounts of memory with undetermined sizes before runtime. Therefore, certain IR nodes may be introduced to help runtime type inference and memory allocation. The operations in these nodes are intrinsically more CPU friendly, which would lead to the serious performance problem if not placed correctly.
- **A symbolic kernel code generator.** Code generation (codegen) is responsible for generating high-performance executable kernels for operators. Recent research (Chen et al., 2018a;b; Zheng et al., 2020b; Adams et al., 2019; Zheng et al., 2020a) has achieved impressive results in

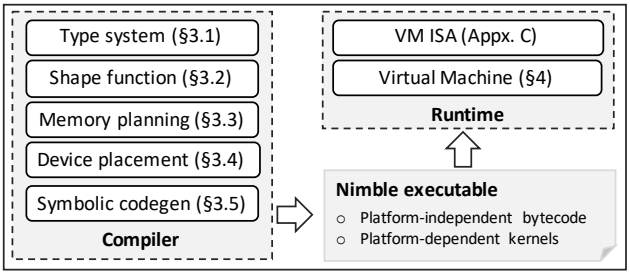


Figure 1: *Nimble* overview. *Nimble* consists of a compiler that handles model with dynamism and a runtime that executes dynamic models in multiple platforms.

kernel performance with static shapes on multiple backends. Nonetheless, challenges in codegen with symbolic shapes remain unexplored. After applying the same set of loop optimization, kernels generated with symbolic shapes could still perform bad if the loop boundary is not handled properly. Meanwhile, kernel tuning under symbolic shape settings becomes more challenging as the search space grows exponentially.

- **A light-weight and cross-platform runtime.** For efficiency purpose, the runtime of static models could be simply designed as a sequential executor that traverses the input data flow graph in the topological order to invoke operators one by one. However, the execution path of dynamic models may only be known at runtime and the kernels for certain operators must be dispatched according to the data shape determined at runtime, making a simple graph-traversing runtime insufficient.

### 2.2 Our Approach

To address these challenges, this paper presents *Nimble*, a high-performance and flexible system for compiling and optimizing dynamic models for multiple platforms. In general, the design goals of *Nimble* are:

1. **Supporting dynamic models.** *Nimble* targets models with all types of dynamism, including control flow, dynamic data structures and varied data shapes.
2. **Being portable and light-weight.** The module that *Nimble* produces should be executable across a number of platforms on the cloud (high-end CPUs and GPUs) and at the edge (low-power CPUs and GPUs). The runtime should be light enough to run on devices with minimal compute power and memory capacity.
3. **Enabling high performance.** *Nimble* should be performant in the context of dynamism across platforms.

Figure 1 shows the system architecture of *Nimble* that we propose to achieve the aforementioned design goals. It is a system consisting of two major components, namely a compiler and a runtime. *Nimble* takes a model in the format of mainstream deep learning frameworks, converts it into a unified intermediate representation (*IR*), then optimizes and compiles the IR into an executable that contains both platform-agnostic bytecode and platform-dependent kernel code, and finally loads the executable to execute in the VM-based runtime. The bytecode is executed by *Nimble*'s runtime interpreter, which is shareable across various platforms. This design effectively enables us to only maintain one version of the execution logic, but focus more on the performance critical operator kernels. The kernels are optimized for a specific hardware platform to achieve high performance.

To effectively support dynamic models without performance degradation for static models, we introduce various analysis and optimization techniques in *Nimble*'s compiler. First, a set of IR extensions are devised to represent dynamic shapes (*Any* shape) and dynamic allocations for static optimization of dynamic program behaviors (Section 3.1). Second, shape functions are attached to operators to compute the output shapes dynamically and perform type checking at runtime (Section 3.2). Third, a memory planning optimization is employed to reduce the amount of memory consumed (Section 3.3). Fourth, a heterogeneous device placement mechanism is designed to place IR nodes on "the-best" device to reduce expensive cross-device data transferring and synchronization (Section 3.4). Finally, the compiler features a code generator that is capable of specializing the codegen of certain likely shapes (Section 3.5). Once the executable with dynamic behavior is compiled, the VM-based runtime can load and interpret it with intelligent dynamic kernel dispatching (Section 4). We detail the design and implementation of each of these features in the subsequent sections.

## 3 Compiler Support for Dynamism

A key challenge preventing existing deep learning compilers from handling dynamism is the lack of a uniform and dynamic representation. For example, optimizations and runtime of existing IR, e.g., TVM, always assume the presence of static shape information. These assumptions introduce quite a few challenges for optimization to dynamism.

In order to handle dynamism, we design a set of IR extensions which expose the essential semantics required to optimize dynamic programs. The approach is implemented in *Nimble* on top of the Apache TVM (version 0.6) deep learning compiler infrastructure (Chen et al., 2018a) to leverage its frontend converters from various DL frameworks to its IR. TVM's frontends alleviate the need to frontend specific details enabling our work to focus on contributions such as IR extensions and optimizations. To use *Nimble*, one only needs to feed it with a pre-trained model, perform compilation and then inference. Furthermore, the lessons here are applicable to other compiler efforts.

This section describes how we transform standard TVM programs into our dynamic dialect to easily apply static optimizations to dynamic programs, much as we do in the traditional compiler optimization. Particularly, we detail

three key components required to compile dynamic models.

- An extended type system which enables static tracking of dynamic shapes.
- A series of optimization passes that make dynamic output shapes, allocation, and device placement explicit.
- A set of codegen techniques for producing code of kernels with dynamic input and output shapes.

### 3.1 Typing

Deep learning compilers use type systems to represent, check and infer the data types and shapes of tensors. In some frameworks and compilers this is separated into two steps, shape inference and data type inference. TVM performs both simultaneously and refers to them as type inference (Roesch et al., 2019), a terminology we will use throughout the section.

A *Tensor Type* is designated by an $n$-dimensional shape (defined as a tuple of integers describing the tensor's dimensions) and a data type (e.g. `float32` or `int64`). Current deep learning IRs only support codegen when all dimensions of a tensor's shape are known at compile-time, i.e., static shapes are mandatory for type inference and checking.

In the context of dynamic models, many data shapes can only be determined at runtime. Therefore, the previous assumption of static data shapes does not hold. In order to support dynamic data shapes, *Nimble* introduces a special dimension called `Any` to represent statically unknown dimensions. For example, we can represent a tensor type as `Tensor[(1, 10, Any), float32]`, where the size of the third dimension in this tensor is unknown while the other two dimensions have concrete values. This concept has been introduced in other frameworks. Janus (Jeong et al., 2019) uses similar denotation to represent a dynamic dimension but only for type unification, while *Nimble* extends type inference to handle `Any` as described next.

**Operator Type Relation** A type relation describes the relationship between types of operator inputs and outputs. The type system of TVM relies on these type relations [1] to infer and bidirectionally propagate type and shape relationships between inputs and outputs of operators across the whole network.

The type relation must be generalized to properly handle dynamic shapes. For example, a program which grows a tensor on each loop iteration (a case existing in the decoder of many NLP models) is both impossible to type and compile without proper type system support. With the introduction of `Any`, we are able to improve the existing type relations to support dynamic models.

There are two dynamic type relation cases in particular. First, operators with dynamic output shapes depending on the input data, such as `arange`[2] and `unique`[3], will use `Any` to describe those shapes. Second, when input shapes contain `Any` dimension, the type relation needs to propagate `Any` correctly to the output types and relax typing constraints that hold in the static cases when necessary. For example, the rules for broadcast[4] type relation between two dimensions with `Any` are defined as follows:

$$\begin{aligned} \text{broadcast\_rel}(\texttt{Any}, 1) &\rightarrow \texttt{Any} \\ \text{broadcast\_rel}(\texttt{Any}, d) &\rightarrow d \quad (d > 1) \\ \text{broadcast\_rel}(\texttt{Any}, \texttt{Any}) &\rightarrow \texttt{Any} \end{aligned}$$

Note that due to the presence of dynamic shapes, these type relation rules can no longer rule out all type errors at compile-time. For example, for the second rule shown above, when `Any` is neither 1 nor $d$ at runtime, it then violates the broadcast type constraints. To address this, we take the gradual typing (Siek & Taha, 2006) approach and leave certain type checking at runtime after `Any` is instantiated by a concrete value (see Section 3.2 for more details). One could eliminate these errors using a more advanced type system, but at increased complexity.

**Type Inference** One caveat of the `Any` dimension is that unknown dimensions will propagate during type inference, reducing chances for shape specialization. To limit the loss of precision introduced by using `Any` dimensions, we introduce *sub-shaping* to the type system. Much like sub-typing used in popular programming languages (Liskov & Wing, 1994; Amadio & Cardelli, 1993), our type system extension enables values with concrete dimensions to be valid sub-types of tensor types with dynamic dimensions. For example a `Tensor[(128, 128), f32]` is a valid sub-type of `Tensor[(any, 128), f32]`. Furthermore we use program analysis to detect and remove unnecessary dynamism, by communicating extra shape information which can be used in downstream compilation. We do this in two critical spots: first we refine any false dynamic dimensions with static dimensions using a secondary shape analysis; and second, in code generation we use a single variable dimension for equivalent dynamic dimensions.

### 3.2 Shape Function

The introduction of `Any` dimension invalidates the pre-allocation mechanism adopted in the existing deep learning compiler. Instead, we now have to track the amount of memory required to be allocated in parallel to computing. Furthermore, static type checking cannot eliminate all type errors at compile-time due to dynamic tensor shapes. Consequently, we define a *shape function* to compute the output

[1] Relay (Roesch et al., 2019) includes more details of the type relations with static cases.

[2] `arange` generates a range of values in a (start, stop, step) interval the arrays output size is a function of input arguments.

[3] `unique` selects the unique elements of a tensor.

[4] `https://docs.scipy.org/doc/numpy/user/basics.broadcasting.html`

shape for storage allocation and verify the type relation in accord with the semantics of every operator. The shape function is similar in structure to the type relations described in Section 3.1 but are present at runtime instead of compile-time. It enables compiling and embedding the computation of output shapes into the program.

According to the characteristics of the operator, we divide the shape functions in three different modes: data independent, data dependent, and upper bound. *Data independent* shape functions are used for operators in which the output shape only depends on the shapes of inputs such as normal 2-D convolution. *Data dependent* shape functions require the concrete input values to compute the output shapes. For example, the output shape of `arange` depends on the value of start, stop, and step. In addition, there are certain operators such as Non Maximum Suppression (`nms`) where the complexity of computing the output shapes is on par with the complexity of executing the operator itself. In order to avoid the redundant computation, we use an *upper bound* shape function to quickly estimate an upper bound shape for the output. We also require such operators to return the output shape along with output value, so as to use the real shape to slice the output tensors into precise output shape and layout.

It is worth noting that in the presence of dynamic shape functions, operator fusion needs to be specially taken care of. Operator fusion, which combines *basic operators* into a *composite operator*, is a critical technique for performance optimization as it reduces unnecessary memory copies and improves the cache locality. The compiler can easily connect the shape functions of basic operators to form the shape function for a composite operator when all shape functions are data independent. However, a basic operator with a data dependent or upper bound shape function cannot be fused to other operators, i.e., taking the outputs of other operators as its inputs to fuse together, as the shape function requires to access to the intermediate result within a composite operator. As a result, we explicitly define the fusion policy to prevent this from happening.

### 3.3 Memory Planning

Many deep learning compilers use a form of static memory planning that coalesces memory into contiguous chunks and minimizes allocations. For devices such as GPUs these optimizations are essential for reducing memory fragmentation and ensuring allocation does not hamper kernel performance. Existing deep learning compiler IRs hide memory allocation behind a functional interface, where each operator implicitly allocates their output storage. Then before execution, the system performs static-memory planning on the data flow graph enabling efficient pre-allocation of the required output buffers. Due to this "out-of-band" nature of memory allocation, it is challenging to customize, modify, or compose memory optimizations with other passes. For example, if one needs to adjust memory allocation for heterogeneous execution, modifications to the runtime system are required. TVM's graph runtime is one such example of static memory planning. Due to the coarse-grained memory semantics of deep learning models, it is essential that memory optimizations occur at a suitably high-level of abstraction, unlike traditional compilers. Existing work provides no clear path to performing static optimization on dynamic memory allocation.

In order to perform what we refer to as "dynamic memory planning" we have extended TVM to transform its IR with implicit memory allocations to one with explicit buffer allocation and manipulation. The key to this transformation is an inter-procedural change of calling convention, with each operator now taking its outputs explicitly. The transformation makes it possible to track and optimize dynamic memory allocations in the IR. In order to perform this optimization we have introduced four new IR constructs, (a) `invoke_mut(op, inputs, outputs)` which takes outputs as mutable in-out arguments, (b) `alloc_storage(size, alignment, device)` which allocates a region of memory of a particular size, (c) `alloc_tensor(storage, offset, shape, dtype, attrs)` which allocates a tensor at a particular storage offset with a shape and data type, and (d) `kill(tensor)` which frees a tensor before its reference count becomes zero due to exiting the frame.

We illustrate how this works with an example of transforming a single statically shaped operation such as broadcasting addition. Note that in the below code examples `Tensor<d1, ..., dn>` is shorthand for a tensor of shape `(d1, ..., dn)` containing floating point values.

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

Here we only must allocate a single buffer, that is, the return buffer for the addition operation.

```
fn main() -> Tensor<10> {
  let t1 = ...; let t2 = ...;
  let buf1 = alloc_storage(40,64,cpu);
  let out1 = alloc_tensor(buf1,0,(10),f32);
  invoke_mut(add, (t1, t2), (out1));
  out1
}
```

The above transformation replaces the operator invocation `add` to code which first allocates an output tensor from backing storage at offset zero, and call to `invoke_mut`. For the sake of space, we present a more complex example in the Appendix A, which illustrates how to handle memory allocation when operators have dynamic shaped inputs. The key insight is to internalize a notion of memory allocation into the IR, enabling static optimization of both static and dynamic allocations in the presence of control and dynamic

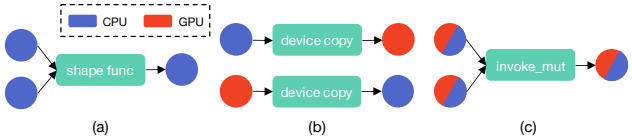


Figure 2: Some heterogeneous device placement rules. (a) The inputs and outputs of shape functions are placed on CPU. (b) `device_copy` changes the device of output accordingly. (c) The device of all arguments to `invoke_mut` must be the same.

shapes. Now that all allocations are explicit in the IR, we can provide analogous optimizations in the static case on dynamic programs, for example we have implemented a storage coalescing pass to group storage into a larger region which allows the multiplexing of multiple tensor allocations to a single piece of storage. Futher optimization like liveness analysis and graph coloring algorithm can be applied to the program to reuse the storages.

### 3.4 Heterogeneous Device Placement

As discussed in Section 3.2, shape functions are executed at runtime to calculate the output shape of an operator. These functions should execute on the CPU as their outputs are used to compute the size of allocated memory. In the case of heterogeneous execution (i.e., CPU and GPU), it is essential to carefully place the execution of IR nodes to proper devices. Otherwise, considerable overhead from data transfers and device synchronization will occur if the inputs to shape functions and kernels need to be copied from or to GPU. To minimize the performance penalty, we analyze the program to place sub-expressions on the most suitable devices.

We introduce a unification based analysis for computing the correct device placement and allocation based on the previous scheduling of the compute kernels. The goal of our device analysis is to assign each IR node properly to minimize the number of cross-device copies. We introduce a concept of `DeviceDomain` to represent the domain of a device, including source and destination. Each expression in the IR defaults to the empty domain, meaning no constraint on its device placement. In addition, a new IR construct, `device_copy`, is introduced to facilitate the heterogeneous execution of the *Nimble* runtime. It represents a data transfer between different devices and is inserted when a cross-device data copy is mandatory. Our analysis is formulated as a set of device placement rules which describes how device constraints flow, and then we use unification, a technique common in type inference and compilers, to compute the precise device placement. Figure 2 depicts some highlights of the rules to assign and propagate the device types: (a) both inputs and outputs of shape function are assigned to CPU, (b) the output of `device_copy` is assigned to the device that is copied to, (c) all arguments to `invoke_mut` should have the same device domain. The full set of rules can be found in the Appendix B.

Based on these rules, we use a union-find data structure to bidirectionally propagate and unify the device placement of each IR node. We introduce two operations, `union(s, t)` and `find(s)`, to achieve `DeviceDomain` unification throughout the entire program. `union(s,t)` unions the equivalence device domains of `s` and `t` into one equivalence domain when the device types match. `find(s)` returns the representative of the device domain that `s` belongs to. These two operations are applied until all IR nodes are annotated. If there is no constraint to a device domain, we assign the compilation target (i.e., GPU) to it in favor of better kernel performance. The result of the heterogeneous device placement composes with memory planning and shape function insertion resulting in correctly placed allocations.

### 3.5 Symbolic Codegen

Deep learning compilers (Chen et al., 2018a; Ragan-Kelley et al., 2013) have demonstrated competitive performance compared to manually tuned kernels on multiple platforms. Recent trends apply machine learning based search to further reduce or eliminate complex manual performance tuning using either template based (Chen et al., 2018b; Zheng et al., 2020b) or search based (Adams et al., 2019; Zheng et al., 2020a) approaches. However, existing work which focuses on tuning in the presence of static shapes falls short with symbolic or dynamic shapes. There are two inherent challenges with regard to codegen of symbolic shapes.

- How to achieve the same performance of kernels generated with symbolic shapes as that with static shapes when applying the same schedule?
- How to extend the machine learning based approach to tune kernels with symbolic shapes?

Loop parallelism and loop tiling are common optimization techniques that exploit multi-core capabilities by achieving data access patterns which are memory hierarchy aware for both CPUs and GPUs. However, the combination of these techniques lead to complex loop boundary conditions. In many static cases, it is possible to prove these conditions always hold, and thus eliminate checks which hamper further optimizations such as unrolling. While straightforward to handle with static shapes, it becomes a non-trivial challenge when performing symbolic codegen. If not carefully handled, the boundary condition checks will stay, leading to poor performance.

To address this issue, we generate multiple kernels according to the residues modulo of the tiling factor and then dispatch based on the actual shape at runtime. For example, suppose a symbolic dimension $x$ is tiled by a factor of 8, we then duplicate the generated kernel for 8 times, and replace the symbolic var $x$ by $8k+r$ in each copy, where $k = \lfloor x/8 \rfloor$ and $r \in [0..7]$. By applying this technique in conjunction with an enhanced symbolic expression simplification pass, we can eliminate most boundary checks to achieve performance

that is nearly identical to kernels compiled with a single static shape. Lastly, we automatically generate a dispatch function that invokes the corresponding kernel based on the residue. In addition, the dispatch function can be extended to invoke either compiler generated kernels or *third party library* whichever is faster from the profiling results. The increased kernel size is relatively small compared to the overall deep learning models. In case where resources are extremely limited, we can either generate fewer number of kernels than the tiling factor or reduce the tiling factor to find an acceptable trade-off between code size and performance.

A known issue to machine learning based tuning is that it may take a long time (usually hours) to find the best schedule for a single kernel. When it comes to symbolic shapes, the tuning time may be exponentially longer if we naively tune for every possible shape. In this paper, we extend the template based tuning approach for symbolic shapes to make tuning time tractable. The template based tuning approach takes a human-defined code template and a search space, and searches the best configuration within the search space by using machine learning algorithms. We observe that a good configuration for one shape usually performs well on other shapes. Based on this observation, we devise the following mechanism to tune the kernel for symbolic shapes.

1. First replace the symbolic dimensions by a large enough value (e.g., 64) so that the search space can cover most possibilities, and run the tuning algorithm on the static shape for a sufficient number of iterations.
2. Pick top $k$ configurations, apply them to a selection of other shapes, and evaluate their performance.
3. Pick the configuration that performs best on average among shapes previously evaluated.

Empirically, we found that $k = 100$ covers most of the best configurations for other shapes. Current popular dynamic models usually only require kernels with one symbolic variable. As a result, we choose the values of power of two up to 256 in the cross evaluation of other shapes. If there is more than one symbolic variable, a more sophisticated selection approach might be required to limit the evaluation time of step 2. We leave this to the future work. Further, if the workload distribution is known, we can adjust the weighting of known shapes in step 3.

## 4 VIRTUAL MACHINE

The conventional runtime of existing deep learning compilers which naively executes a model operator by operator in topological order does not work for executing the compiled modules of dynamic models. A more intelligent and powerful execute engine is required to handle the control flow execution logic, and dispatch different kernels accordingly. In order to achieve these goals and be portable to different platforms, we design and implement a virtual machine (VM)-based runtime.

In *Nimble*, we compile a dynamic model into a *VM executable* that contains platform-independent bytecode and platform-dependent kernel code, which can be later loaded and executed. The bytecode consists of a series of instructions that predicate the order of kernel invocation and control flow execution logic. This design compliments conventional runtime's capability for executing highly optimized kernels but not directly handling orchestration between kernels.

The design of the VM instruction set is motivated by the simple observation that kernel execution dominates neural network execution time. It is quite different from traditional language virtual machines, which contain many instructions that perform little work, leading to a profile where the cost of each instruction executed matters. Our ISA is composed of CISC-style instructions in which each instruction corresponds to a primitive IR expression on tensors, such as allocation and kernel invocation, which in turn may correspond to executing multiple "low-level" operations. For example, `LoadConst idx, $reg` is capable of multiple addressing modes as it first reads the index `idx` and then loads the data from a constant pool to the destination register `$reg`. A complete list of instruction set can be found in the Appendix C. We naturally select a register-based virtual machine design (Davis et al., 2003) for a compact bytecode, which is easy for users to read and modify. We provide the abstraction of an infinite set of virtual registers as it significantly simplifies optimizations and allocation (similar to SSA) and minimizes conceptual barriers to rapid prototyping and modification.

Instructions are represented using a traditional tagged union containing the op-code and the data payload. This representation enables both efficient serialization and instruction decoding and dispatch. *Nimble* uses variable-length instruction format due to the inclusion of variable sized operands such as data shapes in the instructions.

After we have generated an executable from the compiling phase, we can create an interpreter to load it. When execution begins, the interpreter runs a dispatch loop which checks the op-code and executes the appropriate logic, then repeats. As our instructions are coarse-grained (i.e., they can be viewed as super-instructions), the number of branches generated by the dispatch-loop is lower than traditional programming language VMs, adding negligible overhead compared to ahead of time compilation.

**Discussion** An alternative solution to the runtime is ahead of time compilation to eliminate dispatch overhead. But due to the granularity of the operations, dispatch time makes up a very small portion in the execution. More importantly, our VM provides flexibility traditionally attributed to virtual machines and a clear compiler/runtime split. We see the potential of VM to be integrated as a runtime module into

a larger system. For example, VM can provide resource isolation where multiple inference instances share the same hardware in the cloud. Furthermore, a Quality of Service (QoS)-aware system, e.g., (Kang et al., 2018; Yachir et al., 2009), could leverage VM to suspend the current model execution for a higher priority or time-critical model. Last, thanks to the simplicity of the VM design, one can verify the implementation of VM for security and privacy purposes.

# 5 EVALUATION

This section evaluates the performance of *Nimble* on dynamic models against existing state-of-the-art solutions, as well as its optimization implication. Specifically, the section seeks to answer the following questions:

- What is the overall performance of *Nimble* for dynamic models compared against state-of-the-art alternatives on various hardware platforms?
- How much overhead does *Nimble* VM introduce for handling dynamism at runtime?
- How effective are the proposed optimization techniques, such as memory planning and symbolic codegen?

## 5.1 Experiment setup

All experiments were conducted on Amazon EC2 instances. We evaluated *Nimble* on three hardware platforms: Intel Skylake CPUs (c5.9xlarge, 18 physical cores, hereinafter called *Intel CPU*), Nvidia Tesla T4 GPUs (g4dn.4xlarge, 1 card, 2,560 CUDA cores, hereinafter called *Nvidia GPU*), and ARM Cortex A72 (a1.4xlarge, 16 physical cores, hereinafter called *ARM CPU*). Although all tests are done on the cloud, our results of ARM CPU are portable to the edge devices, e.g. Raspberry Pi, due to the same architecture.

To study the efficiency of *Nimble* in handling dynamic models, we compared it with mainstream deep learning frameworks, including TensorFlow (v1.15), MXNet (v1.6), PyTorch (v1.5) [5], DyNet (v2.1), as well as dynamic-specific systems TensorFlow Fold based on TensorFlow v1.0. We were unable to compare *Nimble* with Cavs (Xu et al., 2018), JANUS (Jeong et al., 2019), or Jeong et al.(Jeong et al., 2018) as none of them is open-source. No public deep learning compiler has claimed support for dynamic models.

Three popular models that represent different classes of dynamism were chosen in this experiment, viz. LSTM (Hochreiter & Schmidhuber, 1997) (dynamic control flow), Tree-LSTM (Tai et al., 2015) (dynamic data structure), and BERT (Devlin et al., 2018) (dynamic data shape). The input size / hidden size used in the LSTM and Tree-LSTM model are 300/512 and 300/150, respectively. We used BERT base implementation. For LSTM and BERT, we used Microsoft Research's Paraphrase Corpus (MRPC) (Dolan et al., 2005) with variable input lengths as our input dataset. For Tree-LSTM, we used the Stanford Sentiment Treebank (SST) (Socher et al., 2013) with various tree structures as the input dataset.

[5] We use PyTorch v1.4 on ARM CPU because PyTorch v1.5 fails to build on ARM instance.

## 5.2 Overall performance

We compared the overall performance of *Nimble* against baselines for each dynamic model. *Nimble* successfully accomplished inference for all models on all platforms. However, not all baseline systems could perform inference for these models. For instance, Tree-LSTM only runs on PyTorch, DyNet, and TensorFlow Fold as other frameworks cannot handle dynamic data structures. TensorFlow Fold was not designed to process BERT hence no result was obtainable. We cannot find a BERT implementation on DyNet. Finally, the model inference of Tree-LSTM on Nvidia GPU was omitted as it's hard to saturate GPU compute capability due to excessive control flows and small kernel sizes, making GPU less favorable deployment targets.

The baseline systems all make use of third-party kernel libraries to achieve high-performance by leveraging the heavily hand-optimized operators, which is handicapped when an operator is not supported on a specific target. However, *Nimble* has the ability to select either the self-compiled kernels or the ones provided by third-party library based on which one maximizes performance. It uses dynamic dispatch logic to invoke the selected kernels using platform-independent bytecode at runtime.

First, the latency result comparison is shown in Table 1. *Nimble* consistently outperforms the baseline on both 1- and 2-layer cases, with $2.2\times$, $1.3\times$, and $3.2\times$ faster than the best alternative on Intel CPU, Nvidia GPU, and ARM CPU, respectively. We implemented the LSTM model using a for-loop control flow in all systems for fair comparison. PyTorch has an alternative implementation for LSTM that unrolls the LSTM cells along the sequence length and dynamically batches the matrix multiplication for the inputs. Note that this optimization is orthogonal to the control flow handling and can be implemented in *Nimble*. DyNet is significantly slow on CPU because it only utilizes a single core. We observe that latency on Nvidia GPU is higher than Intel CPU with *Nimble*. This is because the size of LSTM model is relative small so that it cannot fully utilize the massive parallelism in the GPU. The significant performance improvement of *Nimble* comes from two aspects: (a) utilizing the deep learning compiler for better kernel fusion and implementation, (b) encoding the control flow into platform-independent instructions with minimal overhead.

Next, we inspected the performance of model inference on Tree-LSTM as exhibited in Table 2. The table shows that *Nimble* runs substantially faster than the baselines. On PyTorch, the performance speedups are $17.4\times$ on Intel CPU and $19.8\times$ on ARM CPU as PyTorch uses Python to handle

| Unit: $\mu$s/token | 1 layer Intel | 1 layer NV | 1 layer ARM | 2 layers Intel | 2 layers NV | 2 layers ARM |
|---|---|---|---|---|---|---|
| *Nimble* | **47.8** | **54.6** | **182.2** | **97.2** | **107.4** | **686.4** |
| PT | 103.6 | 80.6 | 2735.9 | 224.0 | 158.1 | 5862.2 |
| DY | 936.4 | 68.8 | 5701.6 | 2350.8 | 140.7 | 12811.3 |
| MX | 212.9 | 135.7 | 3695.9 | 401.7 | 223.8 | 7768.0 |
| TF | 301.4 | 304.7 | 978.3 | 687.3 | 406.9 | 2192.8 |

Table 1: LSTM model inference latency of *Nimble*, PyTorch (PT), DyNet (DY), MXNet (MX), and TensorFlow (TF) on Intel CPU, Nvidia (NV) GPU, and ARM CPU.

| Unit: $\mu$s/token | Intel | ARM |
|---|---|---|
| *Nimble* | **40.3** | **86.3** |
| PyTorch | 701.6 | 1717.1 |
| DyNet | 98.2 | 312.9 |
| TF Fold | 209.9 | – |

Table 2: Tree-LSTM model inference latency on Intel CPU and ARM CPU. TensorFlow Fold was not built successfully on ARM CPU.

the tree data structure. DyNet performs much better than PyTorch as it handles the control flow execution carefully in C++. However, *Nimble* still outperforms it as DyNet does not optimize computation kernels for CPUs. TensorFlow Fold is $5.2\times$ slower than *Nimble* on Intel CPU because it has to re-compile upon every input.

Last, Table 3 summarizes the performance comparison of BERT. The results indicate that *Nimble* outstrips the baselines for all frameworks on all platforms in the experiment. The reduction in latency compared to the best alternative is $1.5\times$, $1.3\times$, and $1.05\times$ on Intel CPU, Nvidia GPU, and ARM CPU, respectively. The reasons are two-fold: (a) similar to frameworks, *Nimble* is also able to use the well-tuned third-party libraries on Intel CPU (MKL) and Nvidia GPU (cuDNN); (b) *Nimble* can leverage more thorough operator fusion brought by the deep learning compiler.

In sum, the evaluation results demonstrate that *Nimble* produces more portable performance for all dynamic models on different platforms. Instead, the performance of frameworks is more platform dependent and varies from model to model.

### 5.3 Microbenchmark

This subsection analyzes the performance gain of *Nimble* by using BERT as the microbenchmark. Three studies will be conducted to examine (a) the overhead introduced by the VM, (b) the advantage of the proposed memory planning pass, and (c) the performance discrepancy between symbolic and static codegen. Other models share similar observations.

**Overhead in handling dynamism** In order to understand the overhead that *Nimble* spends to take care of dynamism, we compared it to TVM where static sequence length and TVM static runtime is used to execute BERT. Table 4 details the performance difference between *Nimble* and TVM.

| Unit: $\mu$s/token | Intel | Nvidia | ARM |
|---|---|---|---|
| *Nimble* | **307.0** | **95.2** | **2862.6** |
| PyTorch | 479.5 | 220.4 | 11851.2 |
| MXNet | 455.8 | 152.9 | 8628.0 |
| TensorFlow | 768.7 | 125.2 | 2995.4 |

Table 3: BERT model inference latency on Intel CPU, Nvidia GPU, and ARM CPU.

| Device | TVM lat. (ms) | *Nimble* lat. (ms) | kernel lat. (ms) | others (ms) |
|---|---|---|---|---|
| Intel | 19.38 | 24.32 | 21.06 | 3.26 |
| ARM | 223.50 | 237.41 | 228.59 | 8.82 |
| Nvidia | 5.58 | 5.86 | 5.60 | 0.26 |

Table 4: BERT model latency (sequence length 128) using TVM and *Nimble* on different hardware. *kernel latency* shows the time of kernel execution in *Nimble*, and *others* shows the extra latency introduced by other instructions.

TVM is 5% to 25% faster than *Nimble* on static shapes, though the absolute latency difference is small. The overhead comes from two aspects: (a) kernels generated with symbolic shapes cause extra overhead in the index computation; (b) other instructions in the VM are required to handle the dynamic execution, such as shape functions, dynamic memory allocation, instruction dispatch, etc. On Nvidia GPU, most of the bytecode latency is overlapped with the GPU execution thanks to the heterogeneous device placement (Section 3.4), and therefore the overhead of other instructions is negligible.

**Memory planning** Section 3.3 proposed memory planning to coalesce memory allocation together and reuse the already allocated memory chunks. This pass reduces the number of buffer allocation by 47%, and the memory allocation latency is reduced by 75% on Intel CPU. We also compared the memory usage of *Nimble* with memory planning to TVM which statically analyze and pre-allocate memory on popular computer vision models such as ResNet (He et al., 2016), MobileNet (Howard et al., 2017), VGG (Simonyan & Zisserman, 2014) and SqueezeNet (Iandola et al., 2016). It turned out that *Nimble* uses up to 8% more memory footprint.

**Symbolic codegen** We selected 3 dense layers from BERT model and compared the performance between symbolic codegen and static codegen on ARM CPU. For symbolic codegen, we use `Any` as the sequence length during the compilation and evaluate the kernel with the sequence length 128 at runtime. For static codegen, we directly set the sequence length to 128 at compilation time. Figure 3 illustrates the relative latency of kernels generated with symbolic shapes to the baseline – kernel compiled with static shapes. The auto-tuning algorithm tiles the symbolic axis by 8 in all three kernels. We varied the number of generated kernels to be dispatched during the symbolic codegen from 8 (full dispatch) to 1 (no dispatch) as described in Section 3.5. We observe

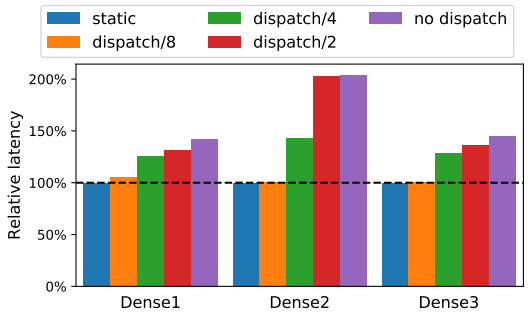


Figure 3: Relative latency of 3 dense operators using symbolic codegen and static codegen on ARM CPU. The latency of static-shaped kernels is used as the baseline. "dispatch/$k$" means that we generate $k$ symbolic kernels to be dispatched at runtime. "no dispatch" means that only one symbolic kernel is generated and therefore no dispatching is needed.

that symbolic codegen with full dispatch can achieve nearly identical performance to that for static codegen, while reducing the number of kernels hurts the performance. Similar trends are seen in dense operators with different shapes, other operators, and on other platforms.

## 6 Related Work

This section contrasts *Nimble* to existing solutions for executing dynamic neural networks.

**Deep learning frameworks** Some frameworks support dynamic control flow via the addition of primitives in their graph representations, such as *switch* and *merge* in TensorFlow (Yu et al., 2018) and *foreach*, *cond*, and *while_loop* in MXNet (Zheng, 2018). Indirect encodings of control flow require specialized data-flow runtimes which handle operations like switch, or hybrid runtimes which separate execution of the control and data planes. Both are heavily intrusive to the framework codebase. In addition, there have been many framework extensions to support different kinds of dynamism.TensorFlow Fold (Looks et al., 2017) conducts an analysis of the user's provided computation graph to identify dynamic operations that can be batched together. Once such operations are found, a batched TensorFlow graph is generated which provides shape specialized sub-graphs. Although this may provide speedup, it introduces large overhead as each path must be executed as a separated sub-computation graph, as well as limits further optimization. Jeong et al. (Jeong et al., 2018) and JANUS (Jeong et al., 2019) also extend TensorFlow to improve the performance for dynamic models. These extensions are framework specific and are not directly related to compilation techniques we employ. The use of speculative execution is complimentary to our techniques.

Dynet (Neubig et al., 2017) and PyTorch (Paszke et al., 2019) use host language features (i.e., Python's control flow) to dynamically unroll control flow to produce a static trace of a dynamic model. Tracing based approaches provide a flexible and friendly programming model at the cost of ahead of time optimization. Additionally, it requires the creation of a data flow graph for each trace, introducing overhead for control-flow constructs and limiting whole-program optimization, a challenge also faced in traditional tracing JIT compilation. JAX (Bradbury et al., 2018) also supports restricted forms of dynamic networks but its optimizations are fundamentally restricted by XLA, its underlying compiler. For example, dynamic value dependent control-flow is not supported in JAX's JIT mode. In contrast *Nimble* makes dynamic behaviors less costly to use without compromising performance for the static subset.

In addition, frameworks rely on third-party libraries (Chetlur et al., 2014; Intel, 2020) to implement operators with different data shapes, namely, they achieve good performance for models with dynamic shapes on a specific hardware platform only if the corresponding high-performance third-party library supports an operation. Therefore, frameworks, as well as the runtime systems derived from them for dynamic models (Xu et al., 2018; Gao et al., 2018), generally perform poorly on devices in the second tier of support such as ARM CPU, and on operator and shape combinations not found in popular benchmarks. In contrast *Nimble* works across all platforms and can generate performant code for new shapes and new devices on demand (Section 3.5).

**Deep learning compilers** Existing deep learning compilers, including XLA (XLA Team, 2017), TVM (Chen et al., 2018a), and Glow (Rotem et al., 2018), can compile deep learning models to run on multiple hardware platforms with accelerated performance. However, little work has been done on optimizing compilation for dynamic neural networks. MLIR (Lattner et al., 2021) is a promising direction and its IR supports dynamic shapes, but no dynamic optimizations or end to end performance have been reported yet. *Nimble*'s compilation and VM design is largely inspired by production compilers and VMs, such as LLVM (Lattner & Adve, 2004), GCC (gcc, 2019), and JVM (jvm, 2013) for general solutions to handle dynamic behaviors, such as control flow and variable-length input arrays.

## 7 Conclusion

This paper proposed *Nimble*, an end-to-end compiler and runtime solution to dynamic neural networks. *Nimble* is the first deep learning compiler that supports neural networks with dynamism, via a lightweight and portable VM-based runtime for executing compiled models on multiple platforms. Experimental results showed that *Nimble* efficiently executed popular dynamic models on multiple platforms with better performance and broader coverage compared to the state-of-the-art. Future work includes enabling dynamic model inference on emerging AI accelerators and high-performance training of dynamic models.

## REFERENCES


The java®virtual machine specification. https://docs.oracle.com/javase/specs/jvms/se7/html/index.html/, 2013.

Gcc, the gnu compiler collection. https://gcc.gnu.org/, 2019.

Adams, A., Ma, K., Anderson, L., Baghdadi, R., Li, T.-M., Gharbi, M., Steiner, B., Johnson, S., Fatahalian, K., Durand, F., et al. Learning to optimize halide with tree search and random programs. *ACM Transactions on Graphics (TOG)*, 38(4):1–12, 2019.

Amadio, R. M. and Cardelli, L. Subtyping recursive types. *ACM Trans. Program. Lang. Syst.*, 15(4):575631, September 1993.

Bradbury, J., Frostig, R., Hawkins, P., Johnson, M. J., Leary, C., Maclaurin, D., and Wanderman-Milne, S. JAX: composable transformations of Python+NumPy programs, 2018. URL http://github.com/google/jax.

Chen, T., Moreau, T., Jiang, Z., Zheng, L., Yan, E., Shen, H., Cowan, M., Wang, L., Hu, Y., Ceze, L., Guestrin, C., and Krishnamurthy, A. TVM: An automated end-to-end optimizing compiler for deep learning. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, pp. 578–594, Carlsbad, CA, 2018a.

Chen, T., Zheng, L., Yan, E., Jiang, Z., Moreau, T., Ceze, L., Guestrin, C., and Krishnamurthy, A. Learning to optimize tensor programs. In *Advances in Neural Information Processing Systems*, pp. 3389–3400, 2018b.

Chetlur, S., Woolley, C., Vandermersch, P., Cohen, J., Tran, J., Catanzaro, B., and Shelhamer, E. cudnn: Efficient primitives for deep learning. *CoRR*, abs/1410.0759, 2014. URL http://arxiv.org/abs/1410.0759.

Dahl, G. E., Yu, D., Deng, L., and Acero, A. Context-dependent pre-trained deep neural networks for large-vocabulary speech recognition. *IEEE Transactions on Audio, Speech, and Language Processing*, 20(1):30–42, 2012.

Davis, B., Beatty, A., Casey, K., Gregg, D., and Waldron, J. The case for virtual register machines. In *Proceedings of the 2003 workshop on Interpreters, virtual machines and emulators*, pp. 41–49. ACM, 2003.

Devlin, J., Chang, M.-W., Lee, K., and Toutanova, K. Bert: Pre-training of deep bidirectional transformers for language understanding. *arXiv preprint arXiv:1810.04805*, 2018.

Dolan, B., Brockett, C., and Quirk, C. Microsoft research paraphrase corpus. *Retrieved March*, 29(2008):63, 2005.

Gao, P., Yu, L., Wu, Y., and Li, J. Low latency rnn inference with cellular batching. In *Proceedings of the Thirteenth EuroSys Conference*, pp. 31. ACM, 2018.

Han, S., Liu, X., Mao, H., Pu, J., Pedram, A., Horowitz, M. A., and Dally, W. J. EIE: Efficient inference engine on compressed deep neural network. In *Proceedings of the 43rd International Symposium on Computer Architecture*, ISCA 16, pp. 243254, 2016.

He, K., Zhang, X., Ren, S., and Sun, J. Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pp. 770–778, 2016.

Hochreiter, S. and Schmidhuber, J. Long short-term memory. *Neural Comput.*, 9(8):1735–1780, November 1997. ISSN 0899-7667. doi: 10.1162/neco.1997.9.8.1735. URL http://dx.doi.org/10.1162/neco.1997.9.8.1735.

Howard, A. G., Zhu, M., Chen, B., Kalenichenko, D., Wang, W., Weyand, T., Andreetto, M., and Adam, H. Mobilenets: Efficient convolutional neural networks for mobile vision applications. *arXiv preprint arXiv:1704.04861*, 2017.

Iandola, F. N., Han, S., Moskewicz, M. W., Ashraf, K., Dally, W. J., and Keutzer, K. Squeezenet: Alexnet-level accuracy with 50x fewer parameters and¡ 0.5 mb model size. *arXiv preprint arXiv:1602.07360*, 2016.

Intel. Intel math kernel library for deep learning networks, 2020. URL https://github.com/oneapi-src/oneDNN. [Online; accessed 3-Mar-2021].

Jeong, E., Jeong, J. S., Kim, S., Yu, G.-I., and Chun, B.-G. Improving the expressiveness of deep learning frameworks with recursion. In *Proceedings of the Thirteenth EuroSys Conference*, pp. 1–13, 2018.

Jeong, E., Cho, S., Yu, G.-I., Jeong, J. S., Shin, D.-J., and Chun, B.-G. JANUS: Fast and flexible deep learning via symbolic graph execution of imperative programs. In *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*, pp. 453–468, 2019.

Johnson, M. J., Duvenaud, D. K., Wiltschko, A., Adams, R. P., and Datta, S. R. Composing graphical models with neural networks for structured representations and fast inference. In *Advances in Neural Information Processing Systems 29*, pp. 2946–2954. 2016.

Kang, L., Zhao, W., Qi, B., and Banerjee, S. Augmenting self-driving with remote control: Challenges and directions. In *Proceedings of the 19th International Workshop*

*on Mobile Computing Systems & Applications*, pp. 1924, 2018.

Lattner, C. and Adve, V. Llvm: A compilation framework for lifelong program analysis & transformation. In *International Symposium on Code Generation and Optimization, 2004. CGO 2004.*, pp. 75–86. IEEE, 2004.

Lattner, C., Amini, M., Bondhugula, U., Cohen, A., Davis, A., Pienaar, J. A., Riddle, R., Shpeisman, T., Vasilache, N., and Zinenko, O. Mlir: Scaling compiler infrastructure for domain specific computation. In *Proceedings of the 19th ACM/IEEE International Symposium on Code Generation and Optimization*, 2021.

Liang, X., Shen, X., Feng, J., Lin, L., and Yan, S. Semantic object parsing with graph lstm. In *European Conference on Computer Vision*, pp. 125–143. Springer, 2016.

Liskov, B. H. and Wing, J. M. A behavioral notion of subtyping. *ACM Trans. Program. Lang. Syst.*, 16(6): 18111841, November 1994.

Liu, Y., Wang, Y., Yu, R., Li, M., Sharma, V., and Wang, Y. Optimizing CNN model inference on cpus. In *2019 USENIX Annual Technical Conference (USENIX ATC 19)*, pp. 1025–1040, 2019.

Looks, M., Herreshoff, M., Hutchins, D., and Norvig, P. Deep learning with dynamic computation graphs. *arXiv preprint arXiv:1702.02181*, 2017.

Neubig, G., Dyer, C., Goldberg, Y., Matthews, A., Ammar, W., Anastasopoulos, A., Ballesteros, M., Chiang, D., Clothiaux, D., Cohn, T., et al. Dynet: The dynamic neural network toolkit. *arXiv preprint arXiv:1701.03980*, 2017.

Paszke, A., Gross, S., Massa, F., Lerer, A., Bradbury, J., Chanan, G., Killeen, T., Lin, Z., Gimelshein, N., Antiga, L., Desmaison, A., Kopf, A., Yang, E., DeVito, Z., Raison, M., Tejani, A., Chilamkurthy, S., Steiner, B., Fang, L., Bai, J., and Chintala, S. PyTorch: An imperative style, high-performance deep learning library. In *Advances in Neural Information Processing Systems 32*, pp. 8024–8035. Curran Associates, Inc., 2019.

Ragan-Kelley, J., Barnes, C., Adams, A., Paris, S., Durand, F., and Amarasinghe, S. Halide: A language and compiler for optimizing parallelism, locality, and recomputation in image processing pipelines. In *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '13, pp. 519–530, New York, NY, USA, 2013. ACM. ISBN 978-1-4503-2014-6. doi: 10.1145/2491956.2462176. URL http://doi.acm.org/10.1145/2491956.2462176.

Roesch, J., Lyubomirsky, S., Kirisame, M., Pollock, J., Weber, L., Jiang, Z., Chen, T., Moreau, T., and Tatlock, Z. Relay: A high-level ir for deep learning. *arXiv preprint arXiv:1904.08368*, 2019.

Rotem, N., Fix, J., Abdulrasool, S., Deng, S., Dzhabarov, R., Hegeman, J., Levenstein, R., Maher, B., Nadathur, S., Olesen, J., Park, J., Rakhov, A., and Smelyanskiy, M. Glow: Graph lowering compiler techniques for neural networks. *CoRR*, abs/1805.00907, 2018. URL https://arxiv.org/abs/1805.00907.

Siek, J. G. and Taha, W. Gradual typing for functional languages. In *In Scheme and Functional Programming Workshop*, pp. 81–92, 2006.

Simonyan, K. and Zisserman, A. Very deep convolutional networks for large-scale image recognition. *arXiv preprint arXiv:1409.1556*, 2014.

Socher, R., Perelygin, A., Wu, J., Chuang, J., Manning, C. D., Ng, A. Y., and Potts, C. Recursive deep models for semantic compositionality over a sentiment treebank. In *Proceedings of the 2013 conference on empirical methods in natural language processing*, pp. 1631–1642, 2013.

Sutskever, I., Vinyals, O., and Le, Q. V. Sequence to sequence learning with neural networks. *CoRR*, abs/1409.3215, 2014. URL http://arxiv.org/abs/1409.3215.

Tai, K. S., Socher, R., and Manning, C. D. Improved semantic representations from tree-structured long short-term memory networks. *CoRR*, abs/1503.00075, 2015. URL http://arxiv.org/abs/1503.00075.

Wang, L., Chen, Z., Liu, Y., Wang, Y., Zheng, L., Li, M., and Wang, Y. A unified optimization approach for cnn model inference on integrated gpus. In *Proceedings of the 48th International Conference on Parallel Processing*, pp. 1–10, 2019.

XLA Team. Xla - tensorflow, compiled, March 2017. URL https://developers.googleblog.com/2017/03/xla-tensorflow-compiled.html.

Xu, S., Zhang, H., Neubig, G., Dai, W., Kim, J. K., Deng, Z., Ho, Q., Yang, G., and Xing, E. P. Cavs: An efficient runtime system for dynamic neural networks. In *2018 USENIX Annual Technical Conference (USENIX ATC 18)*, pp. 937–950, 2018.

Yachir, A., Tari, K., Amirat, Y., Chibani, A., and Badache, N. Qos based framework for ubiquitous robotic services composition. In *2009 IEEE/RSJ International Conference on Intelligent Robots and Systems*, pp. 2019–2026, 2009.

Yu, Y., Abadi, M., Barham, P., Brevdo, E., Burrows, M., Davis, A., Dean, J., Ghemawat, S., Harley, T., Hawkins, P., et al. Dynamic control flow in large-scale machine learning. In *Proceedings of the Thirteenth EuroSys Conference*, pp. 18. ACM, 2018.

Zhang, X., Wang, Q., and Chothia, Z. Openblas. `http://xianyi.github.io/OpenBLAS`, 2014. [Online; accessed 13-May-2019].

Zheng, D. Optimize dynamic neural network models with control flow operators, 7 2018. URL `https://cwiki.apache.org/confluence/display/MXNET/Optimize+dynamic+neural+network+models+with+control+flow+operators`.

Zheng, L., Jia, C., Sun, M., Wu, Z., Yu, C. H., Haj-Ali, A., Wang, Y., Yang, J., Zhuo, D., Sen, K., Gonzalez, J. E., and Stoica, I. Ansor: Generating high-performance tensor programs for deep learning. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pp. 863–879, 2020a.

Zheng, S., Liang, Y., Wang, S., Chen, R., and Sheng, K. Flextensor: An automatic schedule exploration and optimization framework for tensor computation on heterogeneous system. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, pp. 859–873, 2020b.

# APPENDICES

## A MEMORY PLANNING EXAMPLE

In the case of operators with dynamic shaped inputs, we need to insert shape functions before the kernel invocation to compute the output shapes and allocate memory accordingly as detailed in Section 3.2. Our uniform treatment of shape functions as standard tensor expressions enables them to be fused and optimized like normal, but one challenge is that we must now manifest memory allocations in a fixed point until we allocate the outputs for both the compute and necessary shape functions. We illustrate this below how the explicit memory allocation transformation works with a single dynamic concatenation.

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

This is the same transformation as the simple example shown in Section 3.3 with the addition of carefully inserting invocations to the shape function to compute sizes of output buffers for the dynamically sized kernel.

```
1  fn (x: Tensor<?, 2>, y: Tensor<1, 2>)
2      ->Tensor<?, 2> {
3    let in_sh0 = shape_of(x);
4    let in_sh1 = shape_of(y);
5    let buf0 = alloc_storage(16, 64, ...);
6    let out_sh0 = alloc_tensor(buf0, ...);
7    invoke_shape_func(concat,
8        (in_sh0, in_sh1), (out_sh0,), ...);
9    let buf1 = alloc_storage(...);
10   let out0 = alloc_tensor(
11       buf1, out_sh0, ...);
12   invoke_mut(concat, (x, y), (out0));
13   out_0
14 }
```

After the transformation you may notice we have introduced a call to `invoke_shape_func` which invokes a shape function for a kernel (line 7). The shape function requires input shapes as arguments which further require us to invoke `shape_of` for both `%x` and `%y` (line 3-4). `shape_of` will be directly mapped to a VM instruction to retrieve the shape of a tensor at runtime. The transformation also inserts an additional storage and tensor allocation for the output of the shape function (line 5-6).

## B HETEROGENEOUS DEVICE PLACEMENT RULES

The full set of heterogeneous device placement rules are listed below:

- `shape_of`. Defaults to the CPU domain because we can access the shape of a tensor regardless of which device it is placed on.
- Shape functions. These IRs take the output of one or multiple `shape_of` and then derive the shape of an operation according to predefined type inference rules. The output of a shape function is used to compute the amount of memory that this operator requires at runtime, which only needs a few cheap scalar arithmetic computation. Therefore, the inputs and outputs would be better on a CPU domain as well.
- `device_copy`. The input and output of this IR are on different domains as it copies data from one domain to another. The device domains of the input and output are propagated in the opposite directions to other IR nodes that are reachable to/from the device copy node.
- Memory operations. The device domain of storage from `alloc_storage` is designated in the expression, and later is propagated to the device domain of the tensors allocated from this storage via `alloc_tensor`.
- `invoke_mut`. All arguments used in the `invoke_mut` must have the same device domain.
- Other common IR nodes. The device domain of other common IR nodes, e.g. variables, constants, operators, etc., can be directly propagated from the above nodes.

## C VM ISA

Table C.1 details the opcode and the functionality of each instruction. Recall that *Nimble* is designed to support neural networks with dynamic features, such as control flow and dynamic data structures etc., in a portable, high-performance, and light-weight manner. A set of instructions are proposed to fulfill this task. These instructions provide not only high-level information about the dynamic model behavior but also an architectural level interface for better orchestration and virtualization of the execution of control logic and optimized operator kernels. The current instruction set only contains 22 instructions for dynamic model inference. It largely reduces the dispatching overhead and simplifies bytecode serialization and deserialization. We categorize these instructions as follows:

- Register-to-Register Operations. Register-to-Register operations, e.g. `Move`, transfer data between different offset of the register file. Objects are reference counted, make use of copy-on-write and passed by reference ensuring register operations are cheap even if the size of underlying container is large.
- Memory Operations. Memory operations can allocate space for tensors, load constant tensors, and so on. Due to the design of our constant pool, weights (which are constant during inference) can remain in-memory

| Instruction | Description |
|---|---|
| Move | Moves data from one register to another. |
| Ret | Returns the object in the result register to the caller's register. |
| 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. |
| LoadConsti | Loads a constant immediate. |
| AllocStorage | Allocates a storage block on a specified device. |
| AllocTensor | Allocates a tensor object with a static shape from a storage. |
| AllocTensorReg | Allocates a tensor object given the shape in a register. |
| AllocADT | Allocates a data type using the entries from a register. |
| AllocClosure | Allocates a closure with a lowered virtual machine function. |
| FreeStorage | Free allocated memory back to memory manager. |
| FreeTensor | Release the memory occupied by a tensor back to the storage object. |
| Invoke | Invokes a function. |
| InvokeClosure | Invokes a closure. |
| InvokePacked | Invokes an optimized operator kernel. |
| GetField | Gets the value at a certain index from a VM object. |
| GetTag | Gets the tag of an Algebraic Data Types (ADT) constructor. |
| DeviceCopy | Copies a chunk of data from one device to another. |
| ShapeOf | Retrieves the shape of a tensor. |
| ReshapeTensor | Assigns a new shape to a tensor without altering its data. |
| Fatal | Raises fatal in the VM. |

Table C.1: The opcode and the description of *Nimble*'s instruction set.

with no specialized support. They can be referenced by the `LoadConst` instruction. `FreeStorage` and `FreeTensor` free the memory before the reference count becomes zero.

- Call Operations. Call operations are the most frequently executed instructions. The ISA has specialized call instructions for invoking a global function, a kernel primitive, and a closure. `InvokePacked` is the most performance-critical one. It is in charge of invoking the operator kernels that are optimized either by the underlying deep learning compiler or a third-party library. Kernel primitives are ahead-of-time compiled and can leverage both compiler-generated kernels and the third-party libraries using the dispatch function described in Section 3.5. On the other hand, both `Invoke` and `InvokeClosure` call into a global VM function where a closure object carries the captured registers.

- Control Flow Operations. Unconditional jump instructions, e.g. `Goto` and `Ret`, are used by both static and dynamic models to jump to a specific program point. Only dynamic models need conditional control operations, e.g. `If`, to determine the direction of branching. The interpreter updates the PC using the offset from either the true branch or false branch based on the conditional value.

- Miscellaneous Operations. To ease compiler optimizations (e.g. memory planning and device placement) and code generation, we offer the native support of several instructions in the VM, namely `ShapeOf`, `DeviceCopy`, and `ReshapeTensor`. These three instructions are used to directly manipulate runtime data, such as extracting the shape of a tensor, moving data between different devices, and transforming the shape of a tensor. With the help of them, we could preserve more coarse-grained IR at the frontend making optimization simpler.