CORTEX: A COMPILER FOR RECURSIVE DEEP LEARNING MODELS

Pratik Fegade [1] **Tianqi Chen** [1 2] **Phillip B. Gibbons** [1] **Todd C. Mowry** [1]

ABSTRACT

Optimizing deep learning models is generally performed in two steps: (i) high-level graph optimizations such as kernel fusion and (ii) low level kernel optimizations such as those found in vendor libraries. This approach often leaves significant performance on the table, especially for the case of *recursive* deep learning models. In this paper, we present CORTEX, a compiler-based approach to generate highly-efficient code for recursive models for low latency inference. Our compiler approach and low reliance on vendor libraries enables us to perform end-to-end optimizations, leading to up to 14X lower inference latencies over past work, across different backends.

1 INTRODUCTION

Deep learning models are increasingly being used in production as part of applications such as personal assistants, self-driving cars (Maqueda et al., 2018; Bojarski et al., 2016) and chatbots (Yan et al., 2016; Li et al., 2016). These applications place strict requirements on the inference latency of the models. Therefore, a wide variety of hardware substrates, including CPUs (Zhang et al., 2018), GPUs (Nvidia AI, 2019) and specialized accelerators (Jouppi et al., 2017), are being used in production for low latency inference.

Reducing inference latency is especially hard for models with recursive and other dynamic control flow. Such models have been proposed to handle data in fields like natural language and image processing. Textual data, represented as parse trees, can be fed to models such as TreeLSTM (Tai et al., 2015) and MV-RNN (Socher et al., 2012b). Hierarchical and spatial relations in images can be learned by modeling them as trees (Socher et al., 2012a) or graphs (Shuai et al., 2015). These recursive models are often extensions of models designed for sequential data such as LSTM (Hochreiter & Schmidhuber, 1997) and GRU (Cho et al., 2014). A simple recursive model is illustrated in Fig. 1.[1] We use this model as a running example throughout the text.

Past work on recursive and dynamic models such as DyNet (Neubig et al., 2017a;b), Cavs (Xu et al., 2018) and PyTorch (Paszke et al., 2019) has relied on hardware-specific, highly-optimized vendor libraries such as cuDNN (Chetlur et al., 2014) for Nvidia GPUs and MKL (Intel, 2020a) for Intel CPUs. The use of vendor libraries allows these frameworks to offer a generic interface to users, while employing specialized and high-performance kernel implementations in the runtime, and to effectively utilize the wide array of backends that need to be targeted.

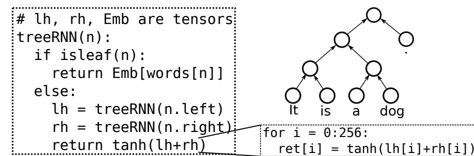


Figure 1. A simple recursive model. The text 'It is a dog.' is parsed into the parse tree which is then fed to the model.

Vendor libraries, however, have disadvantages in terms of *model coverage* and *development effort*. As these libraries are highly optimized, implementing them is a very intensive process. They, therefore, contain implementations only for the most commonly used models and kernels. For example, cuDNN contains implementations for the LSTM and GRU models, but not for the less commonly used TreeLSTM and MV-RNN models.

Moreover, each kernel in a vendor library is optimized *in isolation*. This often precludes optimizations such as *kernel fusion* (combining multiple kernel calls into a single call) that have proven quite beneficial (Microsoft, 2020). *Model persistence* (persisting any model parameters that are reused in every iteration of a recursive or iterative model in fast on-chip memory) is another important optimization (Zhang et al., 2018; Holmes et al., 2019; Diamos et al., 2016). But exploiting such reuse is difficult when using vendor libraries, especially on accelerators such as GPUs with manually managed caches (Liu et al., 2019; Vasilache et al., 2018; Chen et al., 2018a). These difficulties also hold for frameworks such as Nimble (Shen et al., 2020), which relies on auto-tuned implementations of individual kernels.

In this work, instead of relying on vendor libraries or

[1]Carnegie Mellon University, Pittsburgh, USA [2]OctoML. Correspondence to: Pratik Fegade <ppf@cs.cmu.edu>.

Proceedings of the 4th MLSys Conference, San Jose, CA, USA, 2021. Copyright 2021 by the author(s).

[1]This is a simplified model used here for illustrative purposes. Our evaluation is performed on actual models.

auto-tuned kernels, we propose a compiler-based approach, which enables us to perform optimizations such kernel fusion and model persistence. While there is past work that compiles common feed forward models, applying this approach to *recursive* models has the following challenges:

C.1 Effective representation of recursive control flow: Fig. 1 illustrates that recursive models contain dynamic control flow, along with regular numerical (tensor) code. Such models require an intermediate representation (IR) that is amenable to compiler optimizations and code generation over tensor computations with recursive control flow.

C.2 Optimizing recursive control flow: Achieving low latency inference for recursive models requires effective ways to execute the control flow without hindering optimizations such as kernel fusion.

C.3 Static optimizations: Dynamic models are generally optimized at *runtime* by constructing a dataflow graph that unrolls all recursion and makes optimizations such as *dynamic batching* easier (Neubig et al., 2017a; Looks et al., 2017). Such optimizations have to be performed *statically* in a compiler-based approach.

With these challenges in mind, we present CORTEX[2], a compiler framework enabling users to express iterative and recursive models and to generate efficient code across different backends (CPUs and GPUs). To overcome challenge **C.1**, we observe that the control flow in recursive models often depends solely on the input data structure. This insight, along with a few others discussed in §2, enables us to lower the recursive computation into an efficient loop-based one (illustrated in Fig. 2). To overcome **C.2** and **C.3**, we employ scheduling primitives to perform optimizations such as *specialization* and *dynamic batching* (Neubig et al., 2017b; Looks et al., 2017), along with compile-time optimizations such as *computation hoisting*.

CORTEX's compiler-based approach enables it to optimize model computations in an end-to-end manner, without having to treat operators as black-box function calls, as is the case when using vendor libraries. This enables extensive kernel fusion (§7.3) while avoiding some overheads associated with the dynamic batching optimization (§7.2). As part of CORTEX's design, we extend a tensor compiler (Ragan-Kelley et al., 2013; Chen et al., 2018a; Baghdadi et al., 2019; Kjolstad et al., 2017). This enables us to reuse past work on tensor compilers in the context of recursive models. It also opens the door to the use of the extensive work on auto-scheduling (Mullapudi et al., 2016; Adams et al., 2019; Chen et al., 2018b; Zheng et al., 2020) for optimizing these models. Table 1 provides a qualitative comparison of CORTEX with related work on recursive models.

[2]**CO**mpiler for **R**ecursive **T**ensor **EX**ecution

Table 1. Comparison between CORTEX and related work on recursive models (Cavs, DyNet, Nimble and PyTorch).

Framework	Kernel Fusion	Vendor Libraries	Dynamic Batching	Model Persistence
Cavs	Partial	Y	Y	N
DyNet	N	Y	Y	N
Nimble	Partial	N	N	N
PyTorch	N	Y	N	N
CORTEX	Y	N	Y	Y

In short, this paper makes the following contributions:

1. We design CORTEX, a compiler-based framework that enables end-to-end optimization and efficient code generation for low latency inference of recursive deep learning models.
2. As part of the design, we broaden the abstractions provided by tensor compilers and propose new scheduling primitives and optimizations for recursive models.
3. We prototype the proposed framework, evaluate it against state-of-the-art recursive deep learning frameworks (Xu et al., 2018; Neubig et al., 2017a; Paszke et al., 2019) and report significant performance gains (up to 14X) on Nvidia GPUs and Intel and ARM CPUs.

2 OVERVIEW

Recursive deep learning models generally traverse recursive data structures while performing tensor computations. Efficiently executing such models is challenging because their dynamic control flow often precludes common optimizations such as kernel fusion. In CORTEX, we observe that the control flow in recursive models often satisfies certain properties, allowing us to lower it to loop-based iterative control flow efficiently. In particular, we note that a lot of recursive models have the following properties:

P.1 All control flow depends on the connectivity of the data structure, and not on dynamically computed data.

P.2 All recursive calls can be made before performing any tensor computation.

P.3 Recursive calls to the children of a data structure node are independent of each other: the arguments to one call do not depend on the results of a previous call.

Property **P.1** implies that all control flow in the model is encapsulated in the input data structure. Property **P.2** means that computation can start at the leaves of the data structure, moving up towards the roots. Property **P.3** allows us to process sibling nodes in parallel. Taken together, these properties make it possible to generate efficient loop-based code for these recursive model computations.

We now look at CORTEX's compilation and runtime workflows (illustrated in Fig. 2) that make use of these insights. Compilation starts with the recursive model computation ① expressed in the Recursive API (RA). The user can also

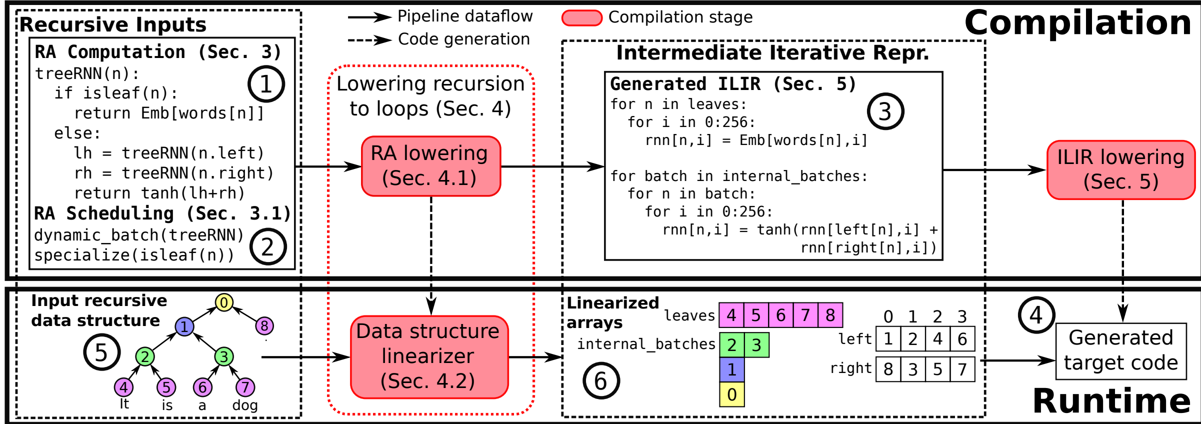


Figure 2. Overview of the CORTEX compilation and runtime pipeline.

specify some scheduling primitives ② at this stage to control how the recursive computation is lowered. The compiler then generates Irregular Loop IR (ILIR) ③ corresponding to the input computation, according to the scheduling primitives provided by the user. The ILIR is an extension of the IR used by tensor compilers, designed to support additional features such as indirect memory accesses and variable loop bounds. It is purely loop-based and data structure agnostic. The RA lowering phase thus lowers all recursive control flow into loops and all data structure accesses to potentially indirect memory accesses at this stage. Loop optimizations such unrolling, tiling, etc., as performed in tensor compilers, can be performed here, after which target-specific code ④ is generated as part of ILIR lowering.

The runtime workflow mirrors the lowering during compilation. We start with pointer linked recursive data structures ⑤ such as sequences, trees or directed acyclic graphs (DAGs), which are then lowered to arrays ⑥, or in other words *linearized*, by the data structure linearizer. Such linearization makes it possible for the generated iterative code to traverse the data structures. The linearizer must ensure that the data dependences between the nodes of the data structure are satisfied as it performs this lowering. Note that the linearization stage does not involve any tensor computations. This is because property **P.1** allows us to separate out the recursive control flow from the tensor computation. We therefore perform linearization on the host CPU.

We now discuss each of the aforementioned compilation and execution stages below.

3 RECURSIVE API (RA)

CORTEX needs to have an end-to-end view of the model computation in order to perform optimizations such as kernel fusion. Accordingly, the input program needs to contain enough information about the tensor operations performed in the model to enable scheduling when it is lowered to the

```
1  ################## Model computation ##################
2  # H: Hidden and embedding size
3  # V: Vocabulary size
4  # N: Total number of nodes in the input data structure(s)
5  Tensor Emb = input_tensor((V,H))
6  Tensor words = input_tensor((N))
7
8  # A placeholder that represents results of recursive calls
9  Tensor rnn_ph = placeholder((N,H))
10 # Base case definition
11 Tensor leaf_case =
12   compute((N,H), lambda n,i: Emb[words[n],i])
13 # Recursive body definition
14 Tensor lh = compute((N,H), lambda n,i: rnn_ph[n.left,i])
15 Tensor rh = compute((N,H), lambda n,i: rnn_ph[n.right,i])
16 Tensor recursive_case =
17   compute((N,H), lambda n,i: tanh(lh[n, i]+rh[n, i]))
18 # Conditional check for the base case
19 Tensor body = if_then_else((N,H), lambda n,i: (isleaf(n),
20                            leaf_case, recursive_case))
21 # Finally, create the recursion
22 Tensor rnn = recursion_op(rnn_ph, body)
23
24 ############### RA scheduling primitives ###############
25 dynamic_batch(rnn)
26 specialize_if_else(body)
```

Listing 1. Simplified implementation of the model in Fig. 1 in RA.

ILIR. Therefore, the RA models an input computation as a DAG of operators where each operator is specified as a loop nest. This is seen in Listing 1, which shows the simplified model from Fig. 1 expressed in the RA. Along with the RA computation, the user also needs to provide basic information about the input data structure such as the maximum number of children per node, and the kind of the data structure (sequence, tree or DAG). This information is used during compilation, and can be easily verified at runtime.

3.1 Recursion Scheduling Primitives

When lowering the recursive computation to loops, we need to ensure that the data dependences between the data structure nodes are satisfied. As these dependences generally specify only a partial ordering on the nodes, we have significant freedom when scheduling the computations. Different schedules may afford different degrees of parallelism, or

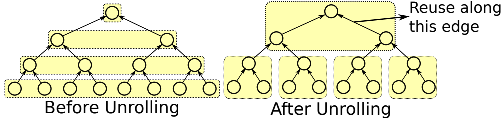


Figure 3. Change in execution schedule due to unrolling

allow for data reuse. Lines 25 and 26 specify scheduling primitives in Listing 1. We propose the following scheduling primitives to exploit these opportunities:

Dynamic Batching: Dynamic batching (Neubig et al., 2017b; Gao et al., 2018; Looks et al., 2017; Xu et al., 2018) involves batching operators on-the-fly to exploit parallelism in a batch in models with dynamic control flow. As control flow in the models we study depends only on the input data structure (property **P.1**), we perform dynamic batching during linearization. With dynamic batching, nodes in a tree are processed top-to-bottom as shown in ⑥ in Fig. 2.

Specialization: Recursive computations tend to have frequent conditional checks to check for the base condition. These checks can hinder optimizations such as computation hoisting and constant propagation (§4.3) and have execution overheads of their own. Thus, we allow the user to specialize the program for the two branches of a conditional check. Listing 2 shows the generated ILIR for our simple recursive model. Note how it has separate loop nests for the computation of leaves and internal nodes as the leaf check was asked to be specialized (on line 26 in Listing 1).

Unrolling: Unrolling recursion changes the order in which nodes are processed (as illustrated in Fig 3), moving a node's computation closer in time to its children's computation. This allows reuse of the children's hidden state via fast on-chip caches, as opposed to the slower off-chip memory. In Fig 3 (right), for example, reuse can be exploited along every edge within a recursive call (yellow box in the figure). Unrolling also creates opportunities for kernel fusion as we can then fuse operators across the children's computations.

Recursive Refactoring: Kernel fusion is harder to perform across recursive call boundaries. In such cases, recursive refactoring can be used to change the recursion backedge. Consider the computation on the left in Fig. 4. $\mathtt{A}_1$, $\mathtt{A}_2$ and `B` represent tensor operators such that there is a dependence from $\mathtt{A}_1$ to $\mathtt{A}_2$. In this case, the recursive backedge goes from B/$\mathtt{A}_2$ to $\mathtt{A}_1$. Fusing kernels in $\mathtt{A}_1$`(n)` and $\mathtt{A}_2$`(n.left)` or $\mathtt{A}_2$`(n.right)` would be hard as the kernels lie across a recursive call boundary. Refactoring changes this boundary (the backedge now goes from $\mathtt{A}_1$ to $\mathtt{A}_2$). Thus, $\mathtt{A}_1$`(n)`, $\mathtt{A}_2$`(n.left)` and $\mathtt{A}_2$`(n.right)` now lie in the same call and can easily be fused.

Note that unrolling and recursive refactoring can lead to repeated and redundant computations for DAGs as nodes can have multiple parents. Thus, we currently support these optimizations only for trees and sequences.

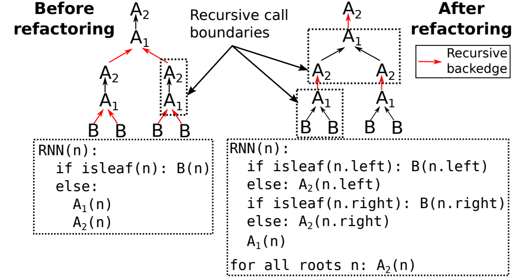


Figure 4. Recursive refactoring changes recursion backedge

4 LOWERING RECURSION TO LOOPS

4.1 RA Lowering

The lowering from the RA to the ILIR is, in essence, a lowering from recursion to iteration. Accordingly, we make all the temporary tensors explicit during the lowering. For instance, in the ILIR for our running example in Listing 2, the tensors `lh` and `rh` are explicitly created. We also materialize the tensor `rnn`, which stores the result of the computation. Each of the three tensors store data for each recursive call, which in this case amounts to each tree node.

```
1  for n_idx = 0:leaf_batch_size:
2    node = leaf_batch[n_idx]
3    for i = 0:256:
4      rnn[node,i] = Emb[words[node],i]
5
6  for b_idx = 0:num_internal_batches:
7    for n_idx = 0:batch_sizes[b_idx]:
8      node = internal_batches[b_idx,n_idx]
9      for i = 0:256:
10       lh[node,i] = rnn[left[node],i]
11     for i = 0:256:
12       rh[node,i] = rnn[right[node],i]
13     for i = 0:256:
14       rnn[node,i] = tanh(lh[node,i] + rh[node,i])
```

Listing 2. ILIR generated for the model in Fig. 1

The scheduling primitives of recursive refactoring and unrolling are handled by appropriately transforming the input RA computation before the lowering. Specialized branches are handled by generating two versions of the computation, each specialized for one target of the branch. The data structure linearizer partitions nodes for such specialized branches and the ILIR employs the correct version of the computation for the respective node partition. The lowering phase generates the appropriate loop nest that iterates over the output of the data structure linearizer. By default, the ILIR iterates over the nodes, but if the user specifies dynamic batching, the ILIR iterates over batches of nodes (as in Listing 2).

4.2 Data Structure Linearization

At runtime, the data structure linearizer traverses the input linked structure and lays it out as arrays for the lowered loop-based computation to iterate upon. The pseudocode for the linearizer for our running example is shown below.

```
1  leaf_batch, internal_batches = [], [[]]
2  left, right = [], []
3
4  def linearizer(n):
5    if isleaf(n): leaf_batch.append(node)
6    else:
7      linearizer(n.left)
8      linearizer(n.right)
9      left[n], right[n] = n.left, n.right
10     internal_batches[node.height].append(node)
11
12 leaf_batch_size = len(leaf_batch)
13 batch_sizes = [len(b) for b in internal_batches]
14 num_internal_batches = len(internal_batches)
```

The data structure linearizer is generated during RA lowering. In the absence of specialization and dynamic batching, the linearizer essentially has to traverse the data structure as the input program does, while keeping track of the order of nodes encountered. This ordering over the nodes would satisfy data dependences and can be used during the tensor computations. Thus, in this simple case, the data structure linearizer is essentially the input program, stripped of all tensor computation. For conditional checks marked for specialization, the linearizer will separately collect nodes that follow each of the two branches of the check. For dynamic batching, we emit code to traverse the data structure and identify batches of nodes that can be processed in parallel.

4.3 Computation Hoisting and Constant Propagation

Recursive and iterative models often use an initial value for the base case. If this initial value is same for all leaves, the same computation is redundantly performed for all leaves. When lowering to the ILIR, such computation is hoisted out of the recursion. We also specially optimize the case when the initial value is the zero tensor.

5 IRREGULAR LOOPS IR (ILIR)

We have briefly mentioned that the ILIR is an extension of the program representation used by tensor compilers. Accordingly, computation and optimizations are specified separately in the ILIR. The computation is expressed as a DAG of operators, each of which produce a tensor by consuming previously-produced or input tensors. Optimizations such as loop tiling, loop unrolling, vectorization, etc. can be performed with the help of scheduling primitives.

The ILIR is generated when the recursive RA computation is lowered. As the ILIR is loop-based and data structure agnostic, this lowering gives rise to indirect memory accesses and loops with variable loop bounds. Note how, in Listing 2, the variable `node` used to index the tensor `rnn` in the loop on line 1 is a non-affine function of the loop variable `n_idx`. Furthermore, the loop on line 7, which iterates over a batch of nodes, has a variable bound, as batches can be of different lengths. In order to support these features, we extend a tensor compiler with (1) non-affine index expressions, (2) loops with variable bounds, and (3) conditional operators. We describe these modifications in further detail below.

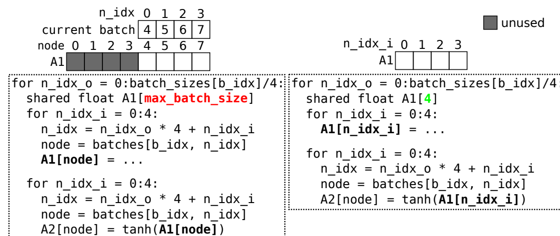


Figure 5. Dense indexing for intermediate tensors

5.1 Indirect Memory Accesses

We represent non-affine index expressions arising as part of indirect memory accesses as uninterpreted functions of loop variables (Strout et al., 2018). Indirect memory accesses necessitate further changes, which are described next.

Bounds Inference: During compilation, a tensor compiler infers loop bounds for all operators in the input program. For each operator *op* producing a tensor t, the compiler first computes what regions of t are required for its consumers. This quantity is then translated to the loop bounds for *op*. In a traditional tensor compiler, this is straightforward as there is a one-to-one correspondence between the loops of an operator and the corresponding tensor dimensions. This is not, however, the case with ILIR, as is apparent in Listing 2. Tensors `lh`, `rh` and `rnn` have two dimensions each, but the generated ILIR has three loops for each of their corresponding operators. Therefore, we require that the ILIR explicitly specify the relationship between tensor dimensions and the loops in the corresponding operator's loop nest. This is discussed further in §A.2 in the appendix.

Tensor Data Layouts: Data layouts of intermediate tensors often need to be changed to allow for an efficient use of the memory subsystem. To enable such optimizations, the ILIR exposes data layout primitives, which allow tensor dimensions to be split, reordered and fused, similar to the corresponding loop transformations.

When an intermediate tensor is stored in a scratchpad memory, as `A1` is Fig. 5, indexing it with non-affine expressions leads to a sparsely filled tensor. Such a sparsely filled tensor occupies excess memory, which is problematic as scratchpad memory space is often at a premium. This is seen on the left size of Fig. 5 where half of `A1` is unused. In such a case, we can index the tensor by the loop iteration space instead as seen on the right side of Fig. 5. Note how we now need to allocate a much smaller tensor in the scratchpad memory. This transformation also reduces indexing costs by turning indirect memory accesses into affine accesses. It is exposed as a scheduling primitive as well.

5.2 Conditional Operator

To lower conditional checks such as the `isleaf` check in our model, we add a conditional operator to the ILIR. It takes two sub-graphs and a conditional check as inputs and is lowered to an `if` statement. A conditional operator would have been generated in the ILIR for our running example if the user had *not* specialized the leaf check.

More details regarding ILIR lowering as well as a few minor optimizations we do therein can be found in the appendix.

6 IMPLEMENTATION

For the purposes of evaluation, we prototype the CORTEX pipeline for the common case. In this section, we talk about a few implementation details regarding the same.

RA Lowering: As part of RA lowering, we have implemented support for dynamic batching and specialization, for the common case of leaf checks.

ILIR Lowering: We extend TVM (Chen et al., 2018a) v0.6, a deep learning framework and a tensor compiler. Our current prototype implementation does not perform auto-scheduling on the generated ILIR. Therefore, the model implementations used for evaluation were based on manually-defined schedules. We then performed auto-tuning via grid search to search the space of certain schedule parameters. Prior work on auto-scheduling is complementary to our techniques, and could readily be applied to the prototype.

Data Structure Linearizers: We implemented data structure linearizers (one each for trees and DAGs) for our evaluation. We use a numbering scheme, described in §B of the appendix, for data structures nodes that generally reduces the costs of leaf checks and iterating over batches.

7 EVALUATION

We now evaluate CORTEX against Cavs, DyNet and PyTorch. Cavs and DyNet are both open source, state-of-the-art frameworks for recursive neural networks, and have been shown to be faster than generic frameworks like PyTorch and TensorFlow (Neubig et al., 2017b; Xu et al., 2018). PyTorch is included for reference. We evaluate these systems on Intel and ARM CPUs and on Nvidia GPUs.

7.1 Experimental Setup

Models and Schedules: We primarily use the models and datasets listed in Table 2. The TreeGRU model is similar to the TreeLSTM model, except that it uses the GRU RNN cell. The TreeLSTM and TreeGRU models were scheduled similarly to the sequential LSTM and GRU schedules proposed in GRNN (Holmes et al., 2019). In the CORTEX and PyTorch implementations for TreeLSTM, TreeGRU and

Table 2. Models and datasets used in our evaluation

Model	Short name	Dataset used
Benchmarking model used in (Looks et al., 2017)	TreeFC	Perfect binary trees (height 7)
Recursive portion of DAG-RNN (Shuai et al., 2015)	DAG-RNN	Synthetic DAGs (size 10x10)
Child-sum TreeGRU	TreeGRU	Stanford sentiment treebank (Socher et al., 2013)
Child-sum TreeLSTM (Tai et al., 2015)	TreeLSTM	Stanford sentiment treebank
MV-RNN (Socher et al., 2012b)	MV-RNN	Stanford sentiment treebank

DAG-RNN, the matrix-vector multiplications involving the inputs were performed at the beginning of the execution by a call to a matrix multiplication kernel as in GRNN. DyNet's dynamic batching algorithm generally performs this optimization automatically and we found that doing so manually resulted in higher inference latencies, so we report the automatic numbers. Unless otherwise noted, inference latencies do not include data transfer times.

For each model, we perform measurements for two batch sizes (1 and 10) and two hidden sizes (256 and 512 for TreeFC, DAG-RNN, TreeGRU and TreeLSTM and 64 and 128 for MV-RNN). The smaller and larger hidden sizes are henceforth referred to as h_s and h_l respectively.

Experimental Environment: We use the three environments listed in Table 3 for the evaluation. We use cuBLAS, Intel MKL and OpenBLAS for all BLAS needs on the GPU, Intel and ARM backends respectively. DyNet also uses the Eigen library. We compare against PyTorch 1.6.0, DyNet's commit `32c71acd` (Aug. 2020) and Cavs' commit `35bcc031` (Sept. 2020).

7.2 Overall Performance

We compare CORTEX's performance with that of PyTorch and DyNet for the five models in Table 2 across the three backends. The open-source implementation of Cavs that we evaluate against has a few limitations—it does not fully support CPU backends, or DAG-based models. It does not implement the lazy batching optimization as described in the Cavs paper. It does not perform specialization nor does it provide the user flexibility to perform the optimization manually. In order to present a fair comparison with Cavs, we therefore use the TreeFC, TreeGRU and TreeLSTM models on the GPU backend, with specialization disabled in CORTEX and do not include the input matrix-vector multiplications in both Cavs and CORTEX. We were also unable to get the streaming and fusion optimizations in Cavs working for the TreeFC and TreeGRU models.

We first look at PyTorch. Speedups over PyTorch implementations for the GPU and Intel backends and for hidden size h_s are shown in Fig. 6. PyTorch does not perform automatic dynamic batching or kernel fusion. Due to the lack of batching, it cannot exploit parallelism across data structure nodes leading to poor performance. The lack of

Table 3. Experimental environment

Hardware	Software[1]	Short name
Nvidia Tesla V100 GPU (Google Cloud n1-standard-4 instance)	CUDA 10.2, cuDNN 8.0, Eigen 3.3.7	GPU
8 core, 16 thread Intel CascadeLake CPU (Google Cloud n2-standard-16 instance)	Intel MKL (v2020.0.1), Eigen (commit `527210`)	Intel
8 core ARM Graviton2 CPU (AWS c6g.2xlarge instance)	Eigen (commit `527210`), OpenBLAS (commit `5c6c2cd4`)	ARM

[1] All cloud instances ran Ubuntu 18.04.

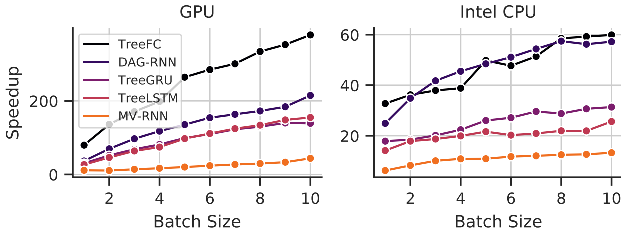


Figure 6. Speedup over PyTorch for hidden size h_s

Table 4. Cavs vs. CORTEX: Inference latencies (Cavs/CORTEX) in ms and speedups on GPU

Hidden Size	Batch Size	TreeFC Time	TreeFC Speedup	TreeGRU Time	TreeGRU Speedup	TreeLSTM Time	TreeLSTM Speedup
h_s	1	0.97/**0.09**	10.24	1.95/**0.15**	12.94	2.54/**0.22**	11.38
h_s	10	3.74/**0.27**	14.06	3.28/**0.27**	12.18	4.01/**0.44**	9.05
h_l	1	1.22/**0.16**	7.41	2.01/**0.2**	10.22	2.56/**0.28**	9.04
h_l	10	5.8/**0.69**	8.46	3.66/**0.61**	5.96	4.43/**0.91**	4.88

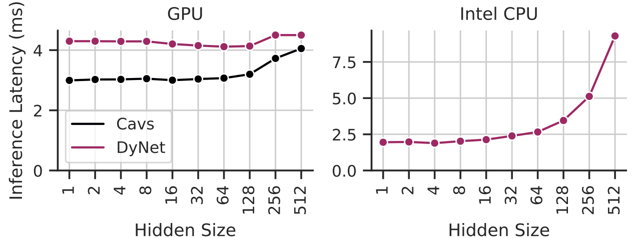


Figure 7. Inference latency vs. hidden size for the recursive portion of TreeLSTM for batch size 10.

batching and kernel fusion also means that PyTorch cannot exploit data reuse across batch elements, as well as between multiple kernel calls. As such reuse opportunities grow with increasing batch size, the performance gap between PyTorch and CORTEX widens. Further, as batch sizes increase, other overheads such as kernel invocation overheads also increase for PyTorch (as PyTorch needs to invoke more kernels), but not for CORTEX due its extensive kernel fusion, as we discuss later. CORTEX performs better on the GPU backend because it can effectively utilize the higher available parallelism on the GPU due to dynamic batching and the scratchpad memories due to aggressive kernel fusion.

We now compare the inference latencies of CORTEX with Cavs and DyNet, shown in Tables 4 and 5, respectively. CORTEX latencies are up to 14X lower due to a number of reasons. As compared to CORTEX, Cavs and DyNet incur significant overheads unrelated to tensor computations. This can be seen in Fig. 7, which plots inference latency as a function of hidden size for the TreeLSTM model[3] for batch size 10 for Cavs and DyNet on the GPU and Intel backends. At low hidden sizes, the inference latencies are quite high and are mainly comprised of overheads. As the overheads are relatively higher for the GPU backend, we explore those below. Apart from kernel call overheads, the discussion of the other overheads applies to the CPU backends too.

Table 6 lists the time spent in some runtime components for DyNet, Cavs, and CORTEX, for the same model configuration as above on the GPU backend. DyNet and Cavs implement generalized runtime algorithms, which cause overheads in dynamic batching and graph construction. At runtime, DyNet constructs a dataflow graph of tensor operators and performs dynamic batching on the same. As compared to Cavs and CORTEX, which deal with graphs corresponding to the input data structures, DyNet therefore must handle a much larger graph. Cavs' 'think-like-a-vertex' approach also has non-trivial overheads as compared to CORTEX, which is specialized for recursive data structures. CORTEX's dynamic batching overheads are limited to linearization, before tensor computations are executed.

[3] We use only the recursive part of the TreeLSTM model, without the input matrix-vector multiplications.

As Cavs and DyNet rely on vendor libraries, they need to ensure that inputs to batched kernel calls are contiguous in memory. The resulting checks and memory copy operations have significant overheads (Xu et al., 2018), both on the CPU and the GPU ('Mem. mgmt. time' in Table 6). As CORTEX manages the entire compilation process, it is free from such contiguity restrictions.

CORTEX performs aggressive kernel fusion (illustrated in Fig. 8 and explored more in §C of the appendix using the roofline model (Williams et al., 2009)), which has the dual effect of generating faster GPU code (seen in the 'GPU computation time' column in Table 6) as well as lowering CUDA kernel call overheads. As seen in Table 6, both DyNet and Cavs execute a high number of kernel calls, which cause non-trivial overheads as CUDA kernels calls are expensive (Lustig & Martonosi, 2013; Nvidia, 2021). The high number of kernel and memory copy calls also contributes to a high amount of CPU time spent in the CUDA API as seen in the column 'CPU CUDA API time'.

To our knowledge, there are no hand-optimized recursive model implementations available. Therefore, we compare CORTEX with GRNN's hand-optimized GPU implementations of the sequential LSTM and GRU models. These implementations use a lock-free CUDA global barrier implementation (Xiao & Feng, 2010), which is faster than the lock-based one (Xiao & Feng, 2010) used by CORTEX. For a fair comparison, we also compare against a version of the

Table 5. DyNet vs. CORTEX: Inference latencies (DyNet/CORTEX) in ms and speedups across different backends

Backend	Hidden Size	Batch Size	TreeFC		DAG-RNN		TreeGRU		TreeLSTM		MV-RNN	
			Time	Speedup	Time	Speedup	Time	Speedup	Time	Speedup	Time	Speedup
GPU	h_s	1	0.41/**0.08**	5.13	1.79/**0.22**	8.15	1.41/**0.18**	7.69	1.84/**0.24**	7.73	0.8/**0.34**	2.38
GPU	h_s	10	1.54/**0.17**	9.26	3.83/**0.39**	9.81	4.72/**0.35**	13.51	5.28/**0.39**	13.59	3.46/**0.78**	4.42
GPU	h_l	1	0.4/**0.12**	3.31	1.78/**0.26**	6.85	1.41/**0.25**	5.66	1.78/**0.29**	6.12	0.87/**0.39**	2.24
GPU	h_l	10	1.48/**0.37**	3.97	3.77/**0.54**	6.92	4.63/**0.75**	6.17	5.1/**0.7**	7.32	3.47/**1.11**	3.14
Intel	h_s	1	0.42/**0.12**	3.46	1.12/**0.19**	5.81	0.98/**0.18**	5.42	1.15/**0.23**	5.06	0.43/**0.29**	1.51
Intel	h_s	10	3.41/**0.64**	5.29	6.07/**0.89**	6.79	4.09/**0.89**	4.58	5.59/**1.02**	5.5	4.68/**1.22**	3.83
Intel	h_l	1	0.93/**0.42**	2.22	2.21/**0.6**	3.66	2.45/**0.58**	4.19	2.95/**0.54**	5.42	1.68/**1.08**	1.55
Intel	h_l	10	8.03/**2.3**	3.49	11.57/**2.27**	5.09	8.63/**2.97**	2.91	12.36/**3.02**	4.09	21.2/**7.3**	2.9
ARM	h_s	1	1.35/**0.21**	6.57	3.48/**0.38**	9.23	2.57/**0.3**	8.49	2.15/**0.39**	5.46	0.52/**0.4**	1.32
ARM	h_s	10	5.27/**1.58**	3.32	11.08/**2.52**	4.4	9.59/**1.81**	5.3	10.59/**2.58**	4.1	5.36/**2.61**	2.05
ARM	h_l	1	3.24/**0.79**	4.11	14.39/**1.55**	9.31	8.74/**0.99**	8.8	6.11/**1.35**	4.54	1.96/**1.95**	1.01
ARM	h_l	10	10.58/**6.54**	1.62	26.84/**8.67**	3.1	21.42/**6.08**	3.52	20.11/**8.86**	2.27	**15.35**/16.8	0.91

Table 6. Time spent (ms) in various activities[1] for DyNet, Cavs, and CORTEX for TreeLSTM on the GPU backend for batch size 10 and hidden size 256.

Framework	Dyn. batch/ Graph const.	Mem. mgmt. time (CPU/GPU)	GPU computation time	#Kernel calls[2]	CPU CUDA API time[3]	Exe. time[4]
DyNet	1.21/1.82	1.46/1.03	1.71	389	12.28	17.381
Cavs	0.4/-	0.85/1.16	0.71	122	9.56	11.57
CORTEX	0.01/-	-/-	0.32	1	0.35	0.35

[1] The timings reported correspond to multiple runs, and were obtained using a combination of manual instrumentation and profiling using `nvprof`.
[2] Does not include memory copy kernels.
[3] Includes all kernel calls as well as calls to `cudaMemcpy` and `cudaMemcpyAsync`.
[4] DyNet and Cavs normally execute CUDA kernels asynchronously. For the purposes of profiling (i.e., this table only), these calls were made synchronous, which leads to slower execution. Shown are execution times under `nvprof` profiling, provided as a reference.

GRNN implementations which use the lock-based implementation. We find that CORTEX-generated code performs competitively as compared to these hand-optimized implementations (Fig. 9). Notably, CORTEX can generalize these optimizations for recursive models.

7.3 Benefits of Optimizations

We now look at CORTEX's different optimizations and their relative benefits. Fig. 10a shows inference latencies for different models (on GPU for hidden size 256) as we progressively perform optimizations. Kernel fusion provides significant benefits for all models. Fusion benefits GPUs more as GPUs have manually managed caches, which kernels optimized in isolation cannot exploit. Complex models such as TreeLSTM that provide more fusion opportunities benefit more. Specialization enables computation hoisting and constant propagation (§4.3), which dramatically reduce computation in tree-based models as trees have a larger proportion of leaves. For DAG-RNN, which performs computations on DAGs, specialization does not lead to any speedup as expected. Finally, model persistence leads to non-negligible improvements by reducing accesses to the GPU global memory. We discuss some optimization tradeoffs involving register pressure in §D in the appendix.

7.4 Other Scheduling Primitives

We now turn to the scheduling primitives of unrolling and recursive refactoring.

Unrolling: We evaluate unrolling on the TreeLSTM model on the GPU backend and a hidden size of 256. In this case, after unrolling, the cost of a barrier cannot be amortized across all nodes in a batch, as illustrated in Fig. 11. This leads to slower inference (Fig.10b) despite the increased data reuse and kernel fusion (§3.1). We then evaluate unrolling on the simpler TreeRNN model, which is an extension of sequential RNNs for trees. When scheduling this model implementation, we perform the computation for one node in one GPU thread block, thus avoiding additional global barriers when unrolled. Therefore, unrolling leads to a drop in the inference latency for this model.

Recursive Refactoring: We evaluate recursive refactoring on the TreeGRU model. In this case, refactoring enables us to reduce the number of global barriers as in the GRNN GRU implementation (Holmes et al., 2019). However, we find that in the case of TreeGRU, this does not give us significant speedups (Fig. 10c). To explore further, we simplify the TreeGRU model (referred to as SimpleTreeGRU[4]) and apply the same optimization again. For the case of this simplified TreeGRU model, refactoring reduces the inference latency by about 25%. We also use recursive refactoring in the sequential GRU model implementation discussed above.

7.5 Data Structure Linearization Overheads

The data structure linearizer (§4.2) lowers input data structures to arrays on the host CPU, performing dynamic batching if necessary. The table below lists linearization times (in

[4] Instead of $h = z * h_{t-1} + (1 - z) * h'$, where h' is the result of a linear transform, the h-gate in SimpleTreeGRU is computed as $h = (1 - z) * h'$.

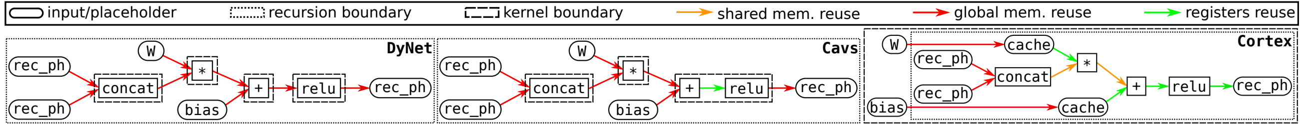


Figure 8. Kernel fusion and model persistence in CORTEX: CORTEX is able to exploit fast on-chip memory (registers and shared memory) better than DyNet and Cavs. This reduces accesses to the slow off-chip global memory. Note also how CORTEX persists the model parameters (`W` and `bias`) and reuses the cached versions every iteration.

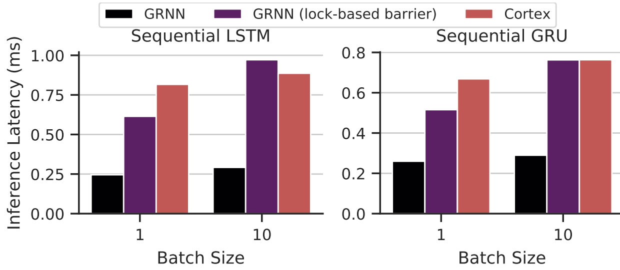


Figure 9. CORTEX vs. hand-optimized GRNN code for sequence length 100 and hidden and input sizes 256.

μs) for different models.[5] We find that on the GPU backend for batch size 10 and hidden size h_s, linearization overheads, as a percentage of total runtime, range from 1.2% (for MV-RNN) to 24.4% (for DAG-RNN). Note that the linearization time is independent of the hidden size as no tensor computations are performed at this stage. As CORTEX specializes for the case of recursive data structures, the linearization overheads are quite low.

Batch Size	TreeLSTM/TreeGRU/MV-RNN	DAG-RNN	TreeFC
1	1.31	8.2	3.04
10	9.64	95.14	30.36

7.6 Memory Usage

We now compare the memory consumption of CORTEX with PyTorch, DyNet and Cavs. The peak GPU memory consumption for different models for batch size 10 and hidden size h_s is shown in Fig. 12. PyTorch uses the least amount of memory as it does not perform dynamic batching. DyNet and Cavs are designed for both deep learning training and inference. As gradient computations during training require the values of intermediate operations computed during the forward pass, DyNet and Cavs do not free the memory used by these intermediate tensors. Therefore, their memory consumption is quite high as compared to CORTEX, which is designed for inference. We also compare against a version of DyNet (shown as 'DyNet (inference)' in Fig. 12) modified to simulate the deallocation of a tensor when it is no longer needed in the forward inference pass. Despite this deallocation, however, DyNet's memory consumption is higher than CORTEX's. CORTEX materializes fewer intermediate tensors to the GPU's global memory due to kernel fusion (Fig. 8). This reduces its memory consumption. Further, DyNet requires extra scratch space to ensure contiguous inputs to vendor library calls, as discussed previously.

[5]Models using the same dataset are grouped together.

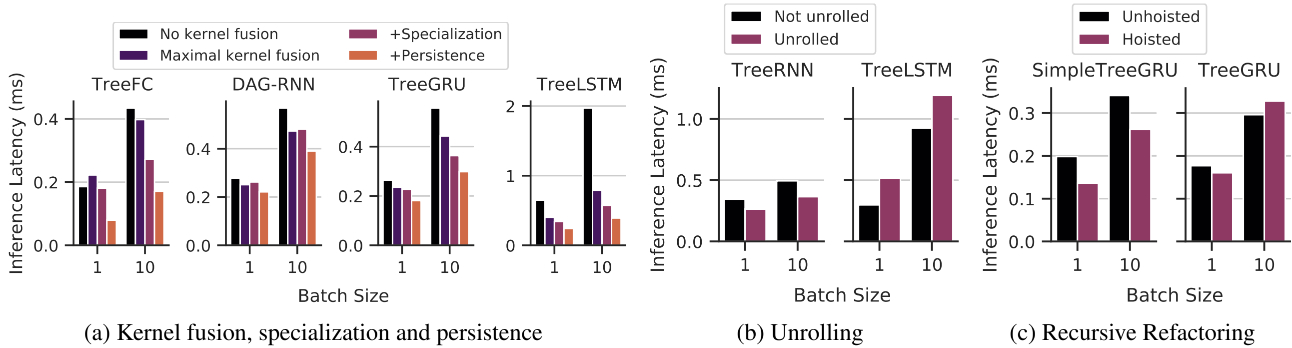
8 RELATED WORK

Compilers for Machine Learning: Tensor compilers such as TVM (Chen et al., 2018a), Halide (Ragan-Kelley et al., 2013), Tiramisu (Baghdadi et al., 2019), Tensor Comprehensions (Vasilache et al., 2018) and Taco (Kjolstad et al., 2017) have been well studied. There are similarities between sparse tensor computations, as supported in Taco, and the ILIR, which lead to similar implementation techniques. For example, the idea of dense layouts for intermediate tensors (§5.1) is similar to the concept of workspaces for Taco introduced in (Kjolstad et al., 2019). More generally, however, CORTEX extends the abstractions provided by tensor compilers to support recursive computations and develops specialized optimizations for the same.

Deep learning compilers such as XLA (Team, 2017) and Glow (Rotem et al., 2018) optimize static feed forward models and can perform partial kernel fusion and code generation. Further, in (Radul et al., 2020), the authors develop techniques to efficiently lower recursion into iterative control flow while performing dynamic batching for the XLA toolchain. Inference engines such as TensorRT (NVIDIA, 2020) and OpenVINO (Intel, 2020b) optimize model execution for inference. The techniques we develop in this paper could be used as a low-level backend for these deep learning compilers and optimizers. MLIR (Lattner et al., 2020) provides infrastructure to build deep learning compilers, and CORTEX could potentially be built using MLIR.

Optimizing Dynamic Neural Networks: There is a large body of work aimed at optimizing recursive and more generally, dynamic neural networks.

Variants of dynamic batching have been used in frameworks such as DyNet, Cavs, BatchMaker (Gao et al., 2018), TensorFlow Fold (Looks et al., 2017) and Matchbox (Bradbury & Fu, 2018). Unlike these, CORTEX performs dynamic batching before any tensor computations. Model persistence was first proposed by Persistent RNNs (Diamos et al., 2016), subsequently used in GRNN (Holmes et al., 2019) and VPPS (Khorasani et al., 2018) and adapted for CPUs in DeepCPU (Zhang et al., 2018). CORTEX is able to extend the these optimizations to recursive models and formalize them as transformation primitives in the compiler.



(a) Kernel fusion, specialization and persistence (b) Unrolling (c) Recursive Refactoring

Figure 10. Benefits of different optimizations on the GPU backend for hidden size 256.

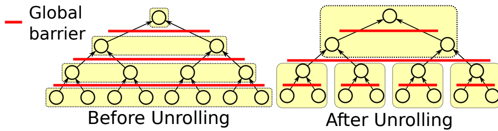


Figure 11. Unrolling TreeLSTM leads to additional barriers.

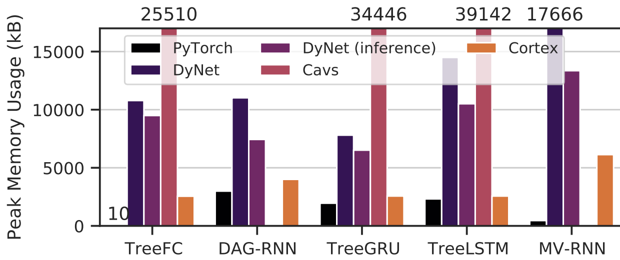


Figure 12. Peak GPU memory consumption in kilobytes, for batch size 10 and hidden size h_s.

Nimble (Shen et al., 2020) adapts deep learning compiler technology for better supporting dynamic models. Janus (Jeong et al., 2019) speculatively creates dataflow graphs that can be optimized to accelerate dynamic models. Similar to DyNet, this leads to overheads at runtime. In (Jeong et al., 2018), the authors extend TensorFlow's static dataflow graph with recursion. Further, while CORTEX currently focuses on acyclic data structures, the ILIR infrastructure could also be used to support deep learning on more graphs, as is supported by DGL (Wang et al., 2019).

As we saw in §3, CORTEX provides a lower level of programming abstraction as compared to the frameworks mentioned above. We believe that CORTEX could be potentially used as a backend for these frameworks, which would alleviate the disadvantages of using vendor libraries discussed in §1.

Sparse Polyhedral Framework: The Sparse Polyhedral Framework (SPF) (Strout et al., 2018; Mohammadi et al., 2019; Nandy et al., 2018) extends the polyhedral model for the case of sparse tensor computations. CORTEX borrows from these works techniques such as the use of uninterpreted functions to represent indirect memory accesses. The data structure linearizer in CORTEX can be viewed as an instance of the inspector-executor technique (Agrawal et al., 1995). Using this technique to lower data structures has also been proposed in (van der Spek et al., 2010).

9 CONCLUSION

In this paper, we presented CORTEX, a compiler for optimizing recursive deep learning models for fast inference. Eschewing vendor libraries, CORTEX's approach enables aggressive kernel fusion and end-to-end optimizations from the recursive control flow down to the tensor algebra computations. This allows CORTEX to achieve significantly lower inference latencies. Past work on machine learning compilers (Roesch et al., 2019; Shen et al., 2020; Yu et al., 2018; Wei et al.) as well as on deep learning (Tai et al., 2015; Shazeer et al., 2017; Elbayad et al., 2019) suggests that supporting efficient execution of various kinds of dynamism in ML models is very desirable. CORTEX demonstrates that a fruitful way of doing this is to exploit past work on general-purpose compilation, such as the inspector-executor technique or the sparse polyhedral framework. We believe it is also important to expand the scope of the highly specialized ML frameworks and techniques used today (without compromising their ability to optimize static feed-forward models), as we do in the case of the ILIR, for example. In the future, we hope (i) to apply these insights to develop similar techniques for training and serving models with potentially non-recursive dynamic control flow and (ii) to integrate CORTEX into higher level programming abstractions.

ACKNOWLEDGMENTS

This work was supported in part by grants from the National Science Foundation and Oracle, by a VMware University Research Fund Award, and by the Parallel Data Lab (PDL) Consortium (Alibaba, Amazon, Datrium, Facebook, Google, Hewlett-Packard Enterprise, Hitachi, IBM, Intel, Microsoft, NetApp, Oracle, Salesforce, Samsung, Seagate, and TwoSigma). We would like to thank Chris Fallin, Dominic Chen, Hao Zhang, Graham Neubig and Olatunji Ruwase for their suggestions and feedback on our work.

REFERENCES

Adams, A., Ma, K., Anderson, L., Baghdadi, R., Li, T.-M., Gharbi, M., Steiner, B., Johnson, S., Fatahalian, K., Durand, F., and Ragan-Kelley, J. Learning to optimize Halide with tree search and random programs. *ACM Trans. Graph.*, 38(4), July 2019. ISSN 0730-0301. doi: 10.1145/3306346.3322967. URL https://doi.org/10.1145/3306346.3322967.

Agrawal, G., Sussman, A., and Saltz, J. Integrated runtime and compile-time approach for parallelizing structured and block structured applications. *Parallel and Distributed Systems, IEEE Transactions on*, 6:747 – 754, 08 1995. doi: 10.1109/71.395403.

Baghdadi, R., Ray, J., Romdhane, M. B., Del Sozzo, E., Akkas, A., Zhang, Y., Suriana, P., Kamil, S., and Amarasinghe, S. Tiramisu: A polyhedral compiler for expressing fast and portable code. In *Proceedings of the 2019 IEEE/ACM International Symposium on Code Generation and Optimization*, CGO 2019, pp. 193–205. IEEE Press, 2019. ISBN 9781728114361.

Bojarski, M., Del Testa, D., Dworakowski, D., Firner, B., Flepp, B., Goyal, P., Jackel, L. D., Monfort, M., Muller, U., Zhang, J., et al. End to end learning for self-driving cars. *arXiv preprint arXiv:1604.07316*, 2016.

Bradbury, J. and Fu, C. Automatic batching as a compiler pass in PyTorch. In *Workshop on Systems for ML*, 2018.

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, October 2018a. USENIX Association. ISBN 978-1-939133-08-3. URL https://www.usenix.org/conference/osdi18/presentation/chen.

Chen, T., Zheng, L., Yan, E. Q., Jiang, Z., Moreau, T., Ceze, L., Guestrin, C., and Krishnamurthy, A. Learning to optimize tensor programs. *CoRR*, abs/1805.08166, 2018b. URL http://arxiv.org/abs/1805.08166.

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.

Cho, K., van Merrienboer, B., Gülçehre, Ç., Bougares, F., Schwenk, H., and Bengio, Y. Learning phrase representations using RNN encoder-decoder for statistical machine translation. *CoRR*, abs/1406.1078, 2014. URL http://arxiv.org/abs/1406.1078.

De Moura, L. and Bjørner, N. Z3: An efficient SMT solver. In *International conference on Tools and Algorithms for the Construction and Analysis of Systems*, pp. 337–340. Springer, 2008.

Diamos, G., Sengupta, S., Catanzaro, B., Chrzanowski, M., Coates, A., Elsen, E., Engel, J., Hannun, A., and Satheesh, S. Persistent RNNs: Stashing recurrent weights on-chip. In *Proceedings of the 33rd International Conference on International Conference on Machine Learning - Volume 48*, ICML'16, pp. 2024–2033. JMLR.org, 2016.

Elbayad, M., Gu, J., Grave, E., and Auli, M. Depth-adaptive transformer. *CoRR*, abs/1910.10073, 2019. URL http://arxiv.org/abs/1910.10073.

Gao, P., Yu, L., Wu, Y., and Li, J. Low latency RNN inference with cellular batching. In *Proceedings of the Thirteenth EuroSys Conference*, EuroSys '18, New York, NY, USA, 2018. Association for Computing Machinery. ISBN 9781450355841. doi: 10.1145/3190508.3190541. URL https://doi.org/10.1145/3190508.3190541.

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 https://doi.org/10.1162/neco.1997.9.8.1735.

Holmes, C., Mawhirter, D., He, Y., Yan, F., and Wu, B. GRNN: Low-latency and scalable RNN inference on GPUs. In *Proceedings of the Fourteenth EuroSys Conference 2019*, EuroSys '19, New York, NY, USA, 2019. Association for Computing Machinery. ISBN 9781450362818. doi: 10.1145/3302424.3303949. URL https://doi.org/10.1145/3302424.3303949.

Intel. Intel math kernel library, 2020a. URL https://software.intel.com/content/www/us/en/develop/tools/math-kernel-library.html. Last accessed July 18, 2020.

Intel. OpenVINO toolkit, 2020b. URL https://docs.openvinotoolkit.org/. Last accessed Oct 06, 2020.

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*, EuroSys '18, New York, NY, USA, 2018. Association for Computing Machinery. ISBN 9781450355841. doi: 10.1145/3190508.3190530. URL https://doi.org/10.1145/3190508.3190530.

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, Boston, MA, February 2019. USENIX Association. ISBN 978-1-931971-49-2. URL https://www.usenix.org/conference/nsdi19/presentation/jeong.

Jouppi, N. P., Young, C., Patil, N., Patterson, D. A., Agrawal, G., Bajwa, R., Bates, S., Bhatia, S., Boden, N., Borchers, A., Boyle, R., Cantin, P., Chao, C., Clark, C., Coriell, J., Daley, M., Dau, M., Dean, J., Gelb, B., Ghaemmaghami, T. V., Gottipati, R., Gulland, W., Hagmann, R., Ho, R. C., Hogberg, D., Hu, J., Hundt, R., Hurt, D., Ibarz, J., Jaffey, A., Jaworski, A., Kaplan, A., Khaitan, H., Koch, A., Kumar, N., Lacy, S., Laudon, J., Law, J., Le, D., Leary, C., Liu, Z., Lucke, K., Lundin, A., MacKean, G., Maggiore, A., Mahony, M., Miller, K., Nagarajan, R., Narayanaswami, R., Ni, R., Nix, K., Norrie, T., Omernick, M., Penukonda, N., Phelps, A., Ross, J., Salek, A., Samadiani, E., Severn, C., Sizikov, G., Snelham, M., Souter, J., Steinberg, D., Swing, A., Tan, M., Thorson, G., Tian, B., Toma, H., Tuttle, E., Vasudevan, V., Walter, R., Wang, W., Wilcox, E., and Yoon, D. H. In-datacenter performance analysis of a tensor processing unit. *CoRR*, abs/1704.04760, 2017. URL http://arxiv.org/abs/1704.04760.

Khorasani, F., Esfeden, H. A., Abu-Ghazaleh, N., and Sarkar, V. In-register parameter caching for dynamic neural nets with virtual persistent processor specialization. In *Proceedings of the 51st Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO-51, pp. 377–389. IEEE Press, 2018. ISBN 9781538662403. doi: 10.1109/MICRO.2018.00038. URL https://doi.org/10.1109/MICRO.2018.00038.

Kjolstad, F., Kamil, S., Chou, S., Lugato, D., and Amarasinghe, S. The tensor algebra compiler. *Proc. ACM Program. Lang.*, 1(OOPSLA):77:1–77:29, October 2017. ISSN 2475-1421. doi: 10.1145/3133901. URL http://doi.acm.org/10.1145/3133901.

Kjolstad, F., Ahrens, P., Kamil, S., and Amarasinghe, S. Tensor algebra compilation with workspaces. pp. 180–192, 2019. URL http://dl.acm.org/citation.cfm?id=3314872.3314894.

Lattner, C., Amini, M., Bondhugula, U., Cohen, A., Davis, A., Pienaar, J., Riddle, R., Shpeisman, T., Vasilache, N., and Zinenko, O. MLIR: A compiler infrastructure for the end of Moore's law, 2020. URL https://arxiv.org/abs/2002.11054.

Li, J., Monroe, W., Ritter, A., Galley, M., Gao, J., and Jurafsky, D. Deep reinforcement learning for dialogue generation. *arXiv preprint arXiv:1606.01541*, 2016.

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, Renton, WA, July 2019. USENIX Association. ISBN 978-1-939133-03-8. URL https://www.usenix.org/conference/atc19/presentation/liu-yizhi.

Looks, M., Herreshoff, M., Hutchins, D., and Norvig, P. Deep learning with dynamic computation graphs. *CoRR*, abs/1702.02181, 2017. URL http://arxiv.org/abs/1702.02181.

Lustig, D. and Martonosi, M. Reducing GPU offload latency via fine-grained CPU-GPU synchronization. In *2013 IEEE 19th International Symposium on High Performance Computer Architecture (HPCA)*, pp. 354–365. IEEE, 2013.

Maqueda, A. I., Loquercio, A., Gallego, G., García, N., and Scaramuzza, D. Event-based vision meets deep learning on steering prediction for self-driving cars. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, June 2018.

Microsoft. Microsoft deepspeed achieves the fastest bert training time, 2020. URL https://www.deepspeed.ai/news/2020/05/27/fastest-bert-training.html. Last accessed Sept 15, 2020.

Mohammadi, M. S., Cheshmi, K., Dehnavi, M. M., Venkat, A., Yuki, T., and Strout, M. M. Extending index-array properties for data dependence analysis. In Hall, M. and Sundar, H. (eds.), *Languages and Compilers for Parallel Computing*, pp. 78–93, Cham, 2019. Springer International Publishing. ISBN 978-3-030-34627-0.

Mullapudi, R. T., Adams, A., Sharlet, D., Ragan-Kelley, J., and Fatahalian, K. Automatically scheduling halide image processing pipelines. *ACM Trans. Graph.*, 35 (4), July 2016. ISSN 0730-0301. doi: 10.1145/2897824.2925952. URL https://doi.org/10.1145/2897824.2925952.

Nandy, P., Hall, M., Davis, E. C., Olschanowsky, C., Mohammadi, M. S., He, W., and Strout, M. Abstractions for specifying sparse matrix data transformations. In *Proceedings of the Eighth International Workshop on Polyhedral Compilation Techniques*, 2018.

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*, 2017a. URL https://arxiv.org/abs/1701.03980.

Neubig, G., Goldberg, Y., and Dyer, C. On-the-fly operation batching in dynamic computation graphs. In *Proceedings of the 31st International Conference on Neural Information Processing Systems*, NIPS'17, pp. 3974–3984, Red Hook, NY, USA, 2017b. Curran Associates Inc. ISBN 9781510860964.

NVIDIA. NVIDIA TensorRT programmable inference accelerator, 2020. URL https://developer.nvidia.com/tensorrt. Last accessed July 18, 2020.

Nvidia. Cuda c++ programming guide, 2021. URL https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html. Last accessed March 04, 2021.

Nvidia AI, N. GPU inference on the rise, 2019. URL https://medium.com/@NvidiaAI/gpu-inference-on-the-rise-b415014019ec. Last accessed Oct 05, 2020.

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 Wallach, H., Larochelle, H., Beygelzimer, A., d'Alché-Buc, F., Fox, E., and Garnett, R. (eds.), *Advances in Neural Information Processing Systems 32*, pp. 8024–8035. Curran Associates, Inc., 2019. URL http://papers.neurips.cc/paper/9015-pytorch-an-imperative-style-high-performance-deep-learning-library.pdf.

Radul, A., Patton, B., Maclaurin, D., Hoffman, M., and A. Saurous, R. Automatically batching control-intensive programs for modern accelerators. In Dhillon, I., Papailiopoulos, D., and Sze, V. (eds.), *Proceedings of Machine Learning and Systems*, volume 2, pp. 390–399. 2020. URL https://proceedings.mlsys.org/paper/2020/file/140f6969d5213fd0ece03148e62e461e-Paper.pdf.

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. Association for Computing Machinery. ISBN 9781450320146. doi: 10.1145/2491956.2462176. URL https://doi.org/10.1145/2491956.2462176.

Rawat, P. S., Rastello, F., Sukumaran-Rajam, A., Pouchet, L.-N., Rountev, A., and Sadayappan, P. Register optimizations for stencils on GPUs. In *Proceedings of the 23rd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '18, pp. 168–182, New York, NY, USA, 2018. Association for Computing Machinery. ISBN 9781450349826. doi: 10.1145/3178487.3178500. URL https://doi.org/10.1145/3178487.3178500.

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. *CoRR*, abs/1904.08368, 2019. URL http://arxiv.org/abs/1904.08368.

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

Sakdhnagool, P., Sabne, A., and Eigenmann, R. RegDem: Increasing GPU performance via shared memory register spilling. *CoRR*, abs/1907.02894, 2019. URL http://arxiv.org/abs/1907.02894.

Shazeer, N., Mirhoseini, A., Maziarz, K., Davis, A., Le, Q. V., Hinton, G. E., and Dean, J. Outrageously large neural networks: The sparsely-gated mixture-of-experts layer. *CoRR*, abs/1701.06538, 2017. URL http://arxiv.org/abs/1701.06538.

Shen, H., Roesch, J., Chen, Z., Chen, W., Wu, Y., Li, M., Sharma, V., Tatlock, Z., and Wang, Y. Nimble: Efficiently compiling dynamic neural networks for model inference. *arXiv preprint arXiv:2006.03031*, 2020. URL https://arxiv.org/abs/2006.03031.

Shuai, B., Zuo, Z., Wang, G., and Wang, B. DAG-recurrent neural networks for scene labeling. *CoRR*, abs/1509.00552, 2015. URL http://arxiv.org/abs/1509.00552.

Socher, R., Huval, B., Bhat, B., Manning, C. D., and Ng, A. Y. Convolutional-recursive deep learning for 3d object classification. In *Proceedings of the 25th International Conference on Neural Information Processing Systems - Volume 1*, NIPS'12, pp. 656–664, Red Hook, NY, USA, 2012a. Curran Associates Inc.

Socher, R., Huval, B., Manning, C. D., and Ng, A. Y. Semantic compositionality through recursive matrix-vector spaces. In *Proceedings of the 2012 Joint Conference on Empirical Methods in Natural Language Processing and Computational Natural Language Learning*, EMNLP-CoNLL '12, pp. 1201–1211, USA, 2012b. Association for Computational Linguistics.

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.

Strout, M. M., Hall, M., and Olschanowsky, C. The sparse polyhedral framework: Composing compiler-generated inspector-executor code. *Proceedings of the IEEE*, 106 (11):1921–1934, 2018.

Tai, K. S., Socher, R., and Manning, C. D. Improved semantic representations from tree-structured long short-term memory networks. *arXiv preprint arXiv:1503.00075*, 2015.

Team, X. XLA - tensorflow, compiled, 2017. URL https://developers.googleblog.com/2017/03/xla-tensorflow-compiled.html. Last accessed Oct 04, 2020.

van der Spek, H. L. A., Holm, C. W. M., and Wijshoff, H. A. G. How to unleash array optimizations on code using recursive data structures. In *Proceedings of the 24th ACM International Conference on Supercomputing*, ICS '10, pp. 275–284, New York, NY, USA, 2010. Association for Computing Machinery. ISBN 9781450300186. doi: 10.1145/1810085.1810123. URL https://doi.org/10.1145/1810085.1810123.

Vasilache, N., Zinenko, O., Theodoridis, T., Goyal, P., DeVito, Z., Moses, W. S., Verdoolaege, S., Adams, A., and Cohen, A. Tensor comprehensions: Framework-agnostic high-performance machine learning abstractions. *arXiv preprint arXiv:1802.04730*, 2018.

Wang, M., Yu, L., Zheng, D., Gan, Q., Gai, Y., Ye, Z., Li, M., Zhou, J., Huang, Q., Ma, C., Huang, Z., Guo, Q., Zhang, H., Lin, H., Zhao, J., Li, J., Smola, A. J., and Zhang, Z. Deep graph library: Towards efficient and scalable deep learning on graphs. *CoRR*, abs/1909.01315, 2019. URL http://arxiv.org/abs/1909.01315.

Wei, J., Gibson, G., Vasudevan, V., and Xing, E. Dynamic scheduling for dynamic control flow in deep learning systems. URL http://www.cs.cmu.edu/~jinlianw/papers/dynamic_scheduling_nips18_sysml.pdf.

Williams, S., Waterman, A., and Patterson, D. Roofline: an insightful visual performance model for multicore architectures. *Communications of the ACM*, 52(4):65–76, 2009.

Xiao, S. and Feng, W. Inter-block GPU communication via fast barrier synchronization. In *2010 IEEE International Symposium on Parallel Distributed Processing (IPDPS)*, pp. 1–12, 2010.

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, Boston, MA, July 2018. USENIX Association. ISBN 978-1-939133-01-4. URL https://www.usenix.org/conference/atc18/presentation/xu-shizen.

Yan, R., Song, Y., and Wu, H. Learning to respond with deep neural networks for retrieval-based human-computer conversation system. In *Proceedings of the 39th International ACM SIGIR Conference on Research and Development in Information Retrieval*, SIGIR '16, pp. 55–64, New York, NY, USA, 2016. Association for Computing Machinery. ISBN 9781450340694. doi: 10.1145/2911451.2911542. URL https://doi.org/10.1145/2911451.2911542.

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. 1–15, 2018.

Zhang, M., Rajbhandari, S., Wang, W., and He, Y. DeepCPU: Serving RNN-based deep learning models 10x faster. In *2018 USENIX Annual Technical Conference (USENIX ATC 18)*, pp. 951–965, Boston, MA, July 2018. USENIX Association. ISBN 978-1-939133-01-4. URL https://www.usenix.org/conference/atc18/presentation/zhang-minjia.

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. USENIX Association, November 2020. ISBN 978-1-939133-19-9. URL https://www.usenix.org/conference/osdi20/presentation/zheng.

A ILIR LOWERING

A.1 Uninterpreted Functions

The ILIR extends a tensor compiler to support indirect memory accesses and variable loop bounds. During code generation, CORTEX therefore has to handle expressions involving such uninterpreted functions. In order to perform simplification over such expressions, for purposes such as proving if certain bound checks are redundant, we use the Z3 SMT solver (De Moura & Bjørner, 2008).

A.2 Bounds Inference

We briefly mentioned in §5.1 how in a traditional tensor compiler, there is a one-to-one relationship between the dimensions of a tensor and the loops in the corresponding operator's loop nest.[6] This can be seen in Fig. 13. In the figure, two loops, each corresponding to a dimension of the tensor r are generated in IR (shown in the generated code on the right).

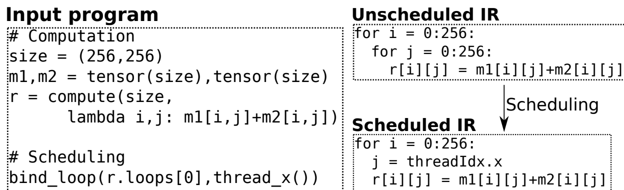


Figure 13. Element-wise matrix addition in a tensor compiler

We also saw how in the ILIR, this relationship needs to be explicitly specified. We do this by the way of *named dimensions*. Named dimensions are identifiers associated with tensor dimensions and loops, which allow us to explicitly specify and keep track of relationships between loops and tensor dimensions. Consider the ILIR in Listing 3, which shows the named dimensions annotated as comments. The dimensions of the tensor rnn are labeled with the named dimensions d_node and d_hidden. The tensor index dimension d_node corresponds to the two loop dimensions d_all_batches and d_batch.

Named dimensions also make the semantic meaning of loops and index expressions explicit. For example, the first dimension of the tensor rnn is labeled d_node and corresponds to the space of all nodes. It, therefore, does not make sense to index rnn by b_idx, the loop variable for the loop associated with d_all_batches.

[6]For brevity, we will not cover the case of optimizations such as loop splitting that give rise to additional loops. Similarly, operators involving reduction are not covered here.

```
1  # rnn[d_node, d_hidden]
2  L1: for n_idx = 0:leaf_batch_size:          # d_batch
3        node = leaf_batch[n_idx]
4  L2:   for i = 0:256:
5          rnn[node,i] = Emb[words[node],i]
6
7  L3: for b_idx = 0:num_internal_batches:     # d_all_batches
8  L4:   for n_idx = 0:batch_sizes[b_idx]:     # d_batch
9          node = internal_batches[b_idx,n_idx]
10 L5:     for i = 0:256:                      # d_hidden
11           lh[node,i] = rnn[left[node],i]
12 L6:     for i = 0:256:                      # d_hidden
13           rh[node,i] = rnn[right[node],i]
14 L7:     for i = 0:256:                      # d_hidden
15           rnn[node,i] = tanh(lh[node,i] + rh[node,i])
```

Listing 3. ILIR generated for the model in Fig. 1

A.3 Caching Tensors Indexed by Non-Affine Expressions

We saw in §5.1 how when an intermediate tensor is stored in scratchpad memory, it can be better to index it by the dense contiguous loop iteration space as opposed to the sparse index space of the original tensor. A similar situation arises when caching a tensor accessed by multiple non-affine index expressions. Assume, for example, if we wished to cache the tensor rnn in loop L4 in Listing 3, to be used when accessing rnn[left[node],i] and rnn[right[node],i]. We create a cached tensor with an additional dimension corresponding to the multiple non-affine index expressions, as shown in the listing below.

```
1   for b_idx = 0:num_internal_batches:
2     for n_idx = 0:batch_sizes[b_idx]:
3       node = internal_batches[b_idx,n_idx]
4       for i = 0:256:
5         # rnn_cache has an additional dimension
6         rnn_cache[b_idx,n_idx,i,0] = rnn[left[node],i]
7         rnn_cache[b_idx,n_idx,i,1] = rnn[right[node],i]
8
9   for b_idx = 0:num_internal_batches:
10    for n_idx = 0:batch_sizes[b_idx]:
11      node = internal_batches[b_idx,n_idx]
12      for i = 0:256:
13        rnn[node,i] = tanh(rnn_cache[b_idx,n_idx,i,0] +
14                           rnn_cache[b_idx,n_idx,i,1])
```

A.4 Barrier Insertion

We need to insert synchronization barriers and memory fences when threads read data written by other threads. This is true on CPUs as well as on accelerators such as GPUs. The barrier insertion pass in TVM does well on tensor programs that do not have loop-carried dependencies. Specifically, given a loop-carried dependence, the pass conservatively places barriers in the innermost loop, as opposed to placing it in the body of the loop that actually carries the dependence. This can lead to unnecessary barriers, leading to inflated runtimes.

As we iterate sequentially either over data structure nodes (when dynamic batching is not performed) or batches of nodes (when dynamic batching is performed), the data dependencies between a node and its children manifest as

$$\mathcal{F} = B \times N \times (\underbrace{4 \times H \times H}_{\text{Matrix-vector (MV) multiplication}} + \underbrace{H}_{\text{Bias computation}})$$

$$\mathcal{B}_{\text{CORTEX}} = 4 \times (\underbrace{2 \times H \times H + H}_{\substack{\text{Model params: Matrix and bias} \\ \text{(read once and cached)}}} + B \times N \times (\underbrace{2 \times H}_{\text{Read children hidden states}} + \underbrace{H}_{\text{Write back hidden state}}))$$

$$\mathcal{B}_{DyNet} = 4 \times (\underbrace{\log_2(N) \times (2 \times H \times H + H)}_{\substack{\text{Model params: Matrix and bias} \\ \text{(read for every dyn. batch)}}} + B \times N \times (\underbrace{2 \times H}_{\substack{\text{Read children} \\ \text{hidden states}}} + \underbrace{H}_{\substack{\text{Write back} \\ \text{MV results}}} + \underbrace{H}_{\text{Read MV result}} + \underbrace{H}_{\substack{\text{Write back} \\ \text{hidden state}}}))$$

$$\mathcal{B}_{PyTorch} = 4 \times (\underbrace{B \times N \times (2 \times H \times H + H)}_{\substack{\text{Model params: Matrix and bias} \\ \text{(read for every node)}}} + B \times N \times (\underbrace{2 \times H}_{\substack{\text{Read children} \\ \text{hidden states}}} + \underbrace{H}_{\substack{\text{Write back} \\ \text{MV results}}} + \underbrace{H}_{\text{Read MV result}} + \underbrace{H}_{\substack{\text{Write back} \\ \text{hidden state}}}))$$

Figure 14. The operational intensities for PyTorch, DyNet and CORTEX, for the TreeFC model. Here, N is the number of nodes in a tree, B is the batch size and H is the hidden size.

loop-carried dependencies in the generated ILIR code. This can be seen in the generated ILIR for the running example, in Listing 3. In the listing, the data written to tensor `rnn` in loops `L2` and `L7` is read by loops `L5` and `L6`. This dependence only exists across a node and its children. We are also guaranteed, by the properties described in §2 and the way the data structure linearizer works, that no node in a batch may be a child of any other node in the same batch. Thus, the dependence is carried by loop `L3`, and not by loop `L4`.

Given this dependence, we would need a barrier at the start of every iteration of loop `L3`. However, the conservative barrier insertion pass in TVM instead places a barrier in the body of loop `L4`. We therefore designed a modification to the pass to insert the barrier in the outer loop, which actually carries the dependence.

A.5 Other Optimizations during ILIR Lowering

Below, we discuss two minor optimizations and scheduling knobs we implemented.

Loop Peeling: The generated ILIR in CORTEX involves loops with variable loop bounds. Splitting such loops gives rise to bounds checks in the bodies of the loops. We employ loop peeling to ensure that such checks are only employed for the last few iterations of the loop.

Rational Approximations of Nonlinear Functions: We use rational approximations for the $tanh$ and $sigmoid$ functions, which makes exploiting SIMD instructions on CPUs easier.

B DATA STRUCTURE LINEARIZATION

In our data structure linearizers, when lowering a pointer linked data structure to arrays, we associate the nodes with integer identifiers. When doing so for the case of dynamic batching, we ensure that nodes in a batch are numbered consecutively and higher than their parents. This enables us to lower the batches into two arrays — `batch_begin` and `batch_length`, which store the starting node and the length, respectively, of every batch. Thus, node `n` is in batch `i` if `batch_begin[i] <=` `n` $<$ `batch_begin[i] + batch_length[i]`. This numbering scheme also ensures that all leaf nodes are numbered higher than all internal nodes. This reduces the cost of checking if a node is a leaf. When nodes are numbered in this way, a leaf check involves a single comparison as opposed to a memory load (to load the number of children of a node under question, for example) and a comparison in the case where the numbering were arbitrary. This scheme thus generally reduces the overheads of iterating over batches and performing leaf checks.

C ROOFLINE PERFORMANCE ANALYSIS FOR TREEFC MODEL

The roofline model (Williams et al., 2009) is a simple analytical performance model that can be used to quantify the amount of reuse exploited by a given computation kernel. As part of this model, the reuse exploited by a kernel is captured in the operational intensity ($\mathcal{O}$) of that kernel. This metric is computed as the amount of computation performed per byte transferred from the memory. Below, we analyze the PyTorch, DyNet and CORTEX implementations of the simple TreeFC model using the roofline model.

Let N be the number of nodes in a tree, B be the batch size and H be the hidden size. In Fig. 14, we compute the total number of floating point operations ($\mathcal{F}$) in the model, which remains constant across the three frameworks, and the total number of bytes ($\mathcal{B}$) read or written to the off-chip memory.

Assuming, $N, H \gg B \geq 1$ and $N \approx H = N_0$, which is the case for our evaluation of the TreeFC model when $H = h_s$, we obtain

$$\mathcal{O}_{\text{CORTEX}} = \frac{\mathcal{F}}{\mathcal{B}_{\text{CORTEX}}} \approx \frac{B \times N_0}{3 \times B + 2}$$
$$\mathcal{O}_{DyNet} = \frac{\mathcal{F}}{\mathcal{B}_{DyNet}} \approx \frac{B \times N_0}{5 \times B + 8 \times \log_2(N_0)}$$
$$\mathcal{O}_{PyTorch} = \frac{\mathcal{F}}{\mathcal{B}_{PyTorch}} \approx 0.5$$

As can be seen, $\mathcal{O}_{\text{CORTEX}} > \mathcal{O}_{DyNet} > \mathcal{O}_{PyTorch}$, suggesting that CORTEX generated kernels exploit more data reuse as compared to DyNet and PyTorch. One should note that this is a simple model that does not take into account other overheads associated with DyNet and PyTorch such as the kernel call and dynamic batching overheads discussed in §7.2.

D REGISTER PRESSURE IN CUDA

CORTEX-generated CUDA kernels are often large, due to optimizations such as aggressive kernel fusion, loop peeling, loop unrolling and recursive unrolling. Furthermore, model persistence uses GPU registers to persist model weights. These factors lead to high register pressure. We find that recursive unrolling precludes us from using persistence for the TreeLSTM and TreeRNN models discussed in §7.4. Similarly, we note that we cannot apply the loop peeling and model persistence optimizations in the case of the TreeLSTM model at the same time. In our schedules, we have explored this trade-off space and evaluated on the best performing schedule. We note that techniques developed in past work such as (Rawat et al., 2018) and (Sakdhnagool et al., 2019) can potentially be applied in our context to alleviate this issue.