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.

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.

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

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.

for i = 0:256:
  ret[i] = tanh(lh[i]+rh[i])
auto-tuned kernels, we propose a compiler-based approach, which enables us to perform optimizations such as 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.

\[\text{Table 1. Comparison between Cortex and related work on recursive models (Cavs, DyNet, Nimble and PyTorch).}\]

<table>
<thead>
<tr>
<th>Framework</th>
<th>Kernel Fusion</th>
<th>Vendor Libraries</th>
<th>Dynamic Batchng</th>
<th>Persistence</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cavs</td>
<td>Partial</td>
<td>Y</td>
<td>Y</td>
<td>N</td>
</tr>
<tr>
<td>DyNet</td>
<td>N</td>
<td>Y</td>
<td>Y</td>
<td>N</td>
</tr>
<tr>
<td>Nimble</td>
<td>Partial</td>
<td>N</td>
<td>N</td>
<td>N</td>
</tr>
<tr>
<td>PyTorch</td>
<td>N</td>
<td>Y</td>
<td>N</td>
<td>N</td>
</tr>
<tr>
<td>Cortex</td>
<td>Y</td>
<td>N</td>
<td>Y</td>
<td>Y</td>
</tr>
</tbody>
</table>

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:

\[\text{P.1} \quad \text{All control flow depends on the connectivity of the data structure, and not on dynamically computed data.}\]

\[\text{P.2} \quad \text{All recursive calls can be made before performing any tensor computation.}\]

\[\text{P.3} \quad \text{Recursive calls to the children of a data structure node are independent of each other: the arguments to one call do not depend on the results of a previous call.}\]

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 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 as 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

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
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 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. A₁, A₂ and B represent tensor operators such that there is a dependence from A₁ to A₂. In this case, the recursive backedge goes from B_A2 to A₁. Fusing kernels in A₁ (n) and A₂ (n.left) or A₂ (n.right) would be hard as the kernels lie across a recursive call boundary. Refactoring changes this boundary (the backedge now goes from A₁ to A₂). Thus, A₁ (n), A₂ (n.left) and A₂ (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.

### 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.

```plaintext
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 for b_idx = 0:num_internal_batches:
6 for n_idx = 0:batch_sizes[b_idx]:
7 node = internal_batches[b_idx,n_idx]
8 for i = 0:256:
9 lh[node,i] = rnn[left[node],i]
10 for i = 0:256:
11 rh[node,i] = rnn[right[node],i]
12 for i = 0:256:
13 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.
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 dependencies 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.

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 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>
<thead>
<tr>
<th>Models and Schedules used in our evaluation</th>
<th>Model</th>
<th>Short name</th>
<th>Dataset used</th>
</tr>
</thead>
<tbody>
<tr>
<td>Benchmarking model used in (Looks et al., 2017)</td>
<td>TreeFC</td>
<td>Perfect binary trees (height 7)</td>
<td></td>
</tr>
</tbody>
</table>
| Recursive portion of DAG-RNN (Smaia et al., 2015) | DAG-RNN | Synthetic DAGs (size 10x10)
| Child-sum TreeFC | TreeFC | Stanford sentiment treebank (Socher et al., 2013) |
| Child-sum TreeLSTM (Tat et al., 2015) | TreeLSTM | Stanford sentiment treebank |
| MV-RNN (Socher et al., 2012b) | MV-RNN | Stanford sentiment treebank |
At low hidden sizes, the inference latencies are quite high. Table 6 lists the time spent in some runtime components. 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.

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 \(^1\) 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.

\(^1\) We use only the recursive part of the TreeLSTM model, without the input matrix-vector multiplications.
We now turn to the scheduling primitives of unrolling and recursive refactoring.

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 fusion benefits (Fig. 10c). To explore further, we simplify the TreeGRU 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) 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.

### 7.4 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, 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 \( h = (1 - z) * h' \).
We now compare the memory consumption of \( C \) (Fig. 8). This reduces its memory consumption. Further, DyNet requires extra scratch space to ensure contiguous memory allocation, however, DyNet’s memory consumption is higher than \( C \) because during the forward inference pass, DyNet and Cavs do not free the memory used by these intermediate tensors. Therefore, their memory consumption is quite high as compared to \( C \) 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.

<table>
<thead>
<tr>
<th>Batch Size</th>
<th>TreeLSTM/TreeGRU/MV-RNN</th>
<th>DAG-RNN</th>
<th>TreeFC</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>9.64</td>
<td>8.2</td>
<td>3.04</td>
</tr>
<tr>
<td>10</td>
<td>1</td>
<td>95.14</td>
<td>30.36</td>
</tr>
</tbody>
</table>

\( \mu s \) for different models.  

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. CORTEX specializes for the case of recursive data structures, the linearization overheads are quite low.

## 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.
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. Escaping 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, Sea- gate, 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.
Cortex: A Compiler for Recursive Deep Learning Models

References


CORTEX: A Compiler for Recursive Deep Learning Models


Neubig, G., Dyer, C., Goldberg, Y., Matthews, A., Ammar, W., Anastasopoulos, A., Ballesteros, M., Chiang, D.,


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. 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).

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.

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.

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.

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
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 $\text{rnn}$ 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 $\text{sigmoid}$ 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 — $\text{batch_begin}$ and $\text{batch_length}$, which store the starting node and the length, respectively, of every batch. Thus, node $n$ is in batch $i$ if $\text{batch_begin}[i] \leq n < \text{batch_begin}[i] + \text{batch_length}[i]$. 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 ($O$) 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 ($F$) in the model, which remains constant across the three frameworks, and the total number of bytes ($B$) 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 $H = h_x$, we obtain:  

\[
F = B \times N \times (4 \times H \times H + H) + H \times H
\]

\[
B_{\text{CORTEX}} = 4 \times (2 \times H \times H + H) + B \times N \times (2 \times H + H + H)
\]

\[
B_{\text{DyNet}} = 4 \times (\log_2(N) \times (2 \times H \times H + H) + B \times N \times (2 \times H + H + H + H))
\]

\[
B_{\text{PyTorch}} = 4 \times (B \times N \times (2 \times H \times H + H) + B \times N \times (2 \times H + H + H + H))
\]
\[ O_{\text{CORTEX}} = \frac{F}{E_{\text{CORTEX}}} \approx \frac{B \times N_0}{3 \times B + 2} \]
\[ O_{\text{DyNet}} = \frac{F}{E_{\text{DyNet}}} \approx \frac{B \times N_0}{5 \times B + 8 \times \log_2(N_0)} \]
\[ O_{\text{PyTorch}} = \frac{F}{E_{\text{PyTorch}}} \approx 0.5 \]

As can be seen, \( O_{\text{CORTEX}} > O_{\text{DyNet}} > O_{\text{PyTorch}} \), 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.

\section*{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 (Sakhdhnagool et al., 2019) can potentially be applied in our context to alleviate this issue.