ACROBAT: OPTIMIZING AUTO-BATCHING OF DYNAMIC DEEP LEARNING AT COMPILE TIME

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

ABSTRACT

Dynamic control flow is an important technique often used to design expressive and efficient deep learning computations for applications such as text parsing, machine translation, exiting early out of deep models and so on. The control flow divergence resulting from dynamic control flow makes batching, an important optimization enabling high throughput and hardware utilization, difficult to perform manually. In this paper, we present ACROBAT, a framework that enables efficient automatic batching for dynamic deep learning computations by performing hybrid static+dynamic compiler optimizations and end-to-end tensor code generation. ACROBAT performs up to 8.5× better than DyNet, a state-of-the-art framework for automatic batching, on an Nvidia GeForce GPU.

.

1 INTRODUCTION

Deep Learning (DL) has come to play an increasing role in a wide range of applications in the recent years. As their applications have become more and more complex, DL models themselves have increased in size and complexity. For inference serving as well as for training, these models place extreme demands on DL systems and hardware today.

An important source of complexity in DL computations is the use of dynamic control flow as part of execution. Unlike a static feed-forward model computation, the execution of a computation with dynamic control flow, or a *dynamic computation* can differ across different inputs to the model. Among other applications, this property has been used effectively to (1) model structured data such as parse trees (Socher et al., 2013a; 2012) and images (Shuai et al., 2015), (2) perform better quality machine translations and text parsing by employing beam search (Wiseman & Rush, 2016; Koehn, 2004; Buckman et al., 2016), and (3) exit early out of convolutional (Kaya & Dumitras, 2018; Teerapittayanon et al., 2017) and transformer (Xin et al., 2020; Elbayad et al., 2019) models for reduced inference latency. The adaptability afforded by dynamic control flow is thus useful in a variety of situations.

Batching is an important optimization that improves the throughput and hardware utilization during training and inference of a DL model. While straightforward for static DL computations, the presence of control flow divergence in dynamic computations makes manual batching difficult and error-prone. Thus, there has been significant past effort on performing automatic batching, or *auto-batching*, for dynamic DL computations. In order to handle the lack of execution knowledge of a dynamic computation during compilation, past works usually either (1) heavily rely on dynamic analyses, enabling them to handle general dynamic control flow (Neubig et al., 2017b; Looks et al., 2017), or (2) are specialized for specific control flow patterns or models, thus relying more on static analyses (Xu et al., 2018; Fegade et al., 2021). The former frameworks often *incur high execution overheads* caused by dynamic analysis, while the latter ones *lack the generality* to support the wide range of existing and future control flow patterns in DL computations.

Further, past work often *heavily relies on vendor libraries* such as cuDNN (Chetlur et al., 2014) and oneAPI (Intel, 2022). However, as implementing vendor libraries is an intensive process, they usually only implement commonly used, standard tensor operators. Further, as these kernels are optimized in isolation, without any contextual knowledge about the larger application they are used in, important optimizations such as kernel fusion can no longer be performed.

In order to overcome these limitations of past work, we propose ACROBAT[1], an auto-batching framework for dynamic DL computations which relies on novel *hybrid static+dynamic optimizations* and *end-to-end tensor kernel compilation*. Our main insight in designing ACROBAT is that despite the lack of perfect execution knowledge during compilation for dynamic models, the compiler can

[1]Carnegie Mellon University, Pittsburgh, USA [2]OctoAI. Correspondence to: Pratik Fegade <pratikfegade@gmail.com>.

Proceedings of the 5^{th} MLSys Conference, Santa Clara, CA, USA, 2024.

[1]**A**utomated **C**ompiler and **R**untime-**o**ptimized **Bat**ching

Table 1. Comparison between ACROBAT and other solutions for auto-batching dynamic DL computations. Purely static or dynamic approaches can be overly conservative, or have high overheads respectively, unlike ACROBAT's hybrid analysis.

Framework	PyTorch	DyNet	Cortex	TFFold	ACROBAT
Auto-batch support	No	Yes	Yes	Yes	Yes
Auto-batch analysis	-	Dyn. only	Static only	Dyn. only	Hybrid
Vendor library use	High	High	None	High	None
Generality	High	High	Low	Mid	High
User impl. effort	Low	Low	High	Low	Low
Performance	Low	Low	High	Low	High

often perform static analysis and optimizations to aid the dynamic analysis. This reduces execution overheads while effectively exploiting parallelism in the input computation. ACROBAT relies on traditional compiler techniques such as context-sensitivity (Aho et al., 2007) and taint analysis as well as on minimal user annotations to enable such static analysis. Further, ACROBAT's end-to-end tensor kernel generation enables it to automatically generate kernels optimized and specialized to the larger computation again using static analysis to identify and exploit data reuse opportunities (as we see in §5). ACROBAT's generality allows one to express a wide variety of control flow patterns, ranging from simple conditional statements to complex recursive computations using a simple high-level language. Table 1 provides a qualitative comparison of ACROBAT with related work.

In short, this paper makes the following contributions:

1. We survey and characterize the dynamic control flow structures found in different DL computations.
2. Employing novel hybrid static+dynamic optimizations and automated end-to-end kernel code generation, we design ACROBAT, an auto-batching framework for dynamic computations. This design allows us to reduce execution overheads and to generate efficient tensor kernels that effectively exploit data reuse opportunities. In developing these optimizations, we heavily rely on traditional compilation techniques.
3. We prototype ACROBAT, evaluate it against state-of-the-art deep learning frameworks (Xu et al., 2018; Neubig et al., 2017a; Paszke et al., 2019) and report significant performance gains on Nvidia GPUs.

2 BACKGROUND

2.1 Dynamic Control Flow in DL computations

In this section, we take a look at the different kinds of control flow dynamism present in various DL computations in the context of the auto-batching problem. This will inform how we design a system to exploit parallelism across tensor operators in the batched execution of dynamic DL computation.

Note that given a computation involving control flow, there are often multiple ways to implement it. We consider the most natural way to implement a given computation. For example, a top-down tree traversal can be implemented as a breadth-first traversal (BFS) or a depth-first traversal (DFS). While a BFS traversal can be more efficient, the DFS-based traversal is more natural to implement. The discussion below is also summarized in Table 2.

Table 2. Control flow properties found in DL computations. Legend: ITE: iterative control flow, REC: recursive control flow, TDC: model exhibits tensor-dependent control flow (where control flow decisions are predicated on values on intermediate tensors), IP: computation exhibits high instance parallelism, ICF: model inference exhibits control flow, TCF: model training exhibits control flow.

Deep Learning Computations	ITE	REC	TDC	IP	ICF	TCF
RNN (Rumelhart et al., 1986), LSTM (Hochreiter & Schmidhuber, 1997), GRU (Cho et al., 2014), GraphRNN (You et al., 2018)	✓				✓	✓
Speculative decoding for transformers (Leviathan et al., 2023)	✓		✓	✓	✓	
DIORA (Drozdov et al., 2019), Chinese Segmentation (Chen et al., 2015)	✓			✓	✓	✓
DAG-RNN (Shuai et al., 2015), TreeLSTM (Socher et al., 2013a), MV-RNN (Socher et al., 2012)		✓		✓	✓	✓
StackLSTM (Dyer et al., 2015)	✓		✓		✓	✓
Beam search (Wiseman & Rush, 2016) with LSTM	✓		✓	✓	✓	
Mixture-of-experts (Shazeer et al., 2017; Ma et al., 2018; Fedus et al., 2021)			✓		✓	✓
Early exit models (Kaya & Dumitras, 2018; Teerapittayanon et al., 2017; Elbayad et al., 2019)			✓		✓	
Tree-to-tree NN (Chen et al., 2018b), Doubly Recurrent NN (Alvarez-Melis & Jaakkola, 2017)		✓	✓	✓	✓	✓
R-CNN (Girshick et al., 2013), Fast R-CNN (Girshick, 2015)	✓		✓	✓	✓	✓

Control Flow Surrounding Static Sub-Graphs: We observe that for most DL computations exhibiting control flow dynamism, the dynamic control flow *surrounds* tensor computations. Consider the simple sequential RNN model implemented by the `@rnn` function shown in Listing 1. Here, we see that the sequential control flow surrounds an RNN cell on lines 5 and 6, which is a static sub-graph of tensor computations with no intervening control flow.

Tensor-Dependent Control Flow: Control flow decisions often depend on the values of intermediate tensors in DL computations. Examples of such models and computations include beam search in machine translation, StackLSTMs (Dyer et al., 2015), Tree-to-Tree neural networks (T2TNN) (Chen et al., 2018b), models with early exits (Kaya & Dumitras, 2018; Teerapittayanon et al., 2017; Xin et al., 2020; Elbayad et al., 2019) and Mixture-of-Experts (Shazeer et al., 2017; Ma et al., 2018; Fedus et al., 2021). Meanwhile, in models such as TreeLSTM (Socher et al., 2013a), DAG-RNN, sequential RNNs and their variants, control flow only depends on the inputs and not on intermediate tensors.

Repetitive Control Flow: We say that a model exhibits repetitive control flow if it can be expressed as an iterative or recursive computation. This includes iterative models such

as RNNs and their variants (LSTM and GRU (Cho et al., 2014) for example) and StackLSTMs, and recursive models such as TreeLSTM, Tree-to-Tree neural networks and DAG-RNNs (Shuai et al., 2015). On the other hand, Mixture-of-Experts and early exit models do not exhibit repetitive control flow. Such models contain conditional execution in an otherwise static feed-forward network. Repetitive control flow can often also be nested. The GraphRNN model, for example, executes two RNNs, one nested inside the other. Similarly, the DRNN model, which is used for top-down recursive tree generation, involves iterative generation of children for a given tree node.

The presence of recursive, as opposed to iterative control flow, can often complicate static analysis as parallelism is more easily exploited with the latter. We see in §4.2 how exploiting parallelism across recursive calls at runtime, for example, can require the multiple concurrent execution contexts, similar to the fork-join parallelism paradigm (McCool et al., 2012).

Control-Flow in Training and Inference: We see, in Table 2, that the computation for a lot of the models involve dynamic control flow during both training as well as inference. This is however, not the case for models with early exits, where during training, we often wish to train all the exit branches rather than evaluating one, as is the case during inference. Further, search procedures such as beam search are often used only during inference and hence the underlying model may not exhibit dynamism during training (unless the model computation itself involves dynamism, as in the case of RNN models, for example).

Control Flow Parallelism: Dynamic control flow can lead to parallelism in a DL computation. Such a computation may exhibit (1) *Batch Parallelism* that exists across different input instances in the mini-batch, and/or (2) *Instance Parallelism* which refers to the parallelism that arises due to dynamic control flow dependences, such as recursive parallelism. The amount of such parallelism differs widely across computations. Recursive models, often (though not always) have significant parallelism across different recursive calls. Correspondingly, iterative computations may contain loops that can be executed concurrently. An example is the call to the `@map` function call in the RNN implementation in Listing 1.

2.2 Dynamic Batching

ACROBAT builds upon dynamic batching (Looks et al., 2017; Neubig et al., 2017b), a prior technique to perform auto-batching in the presence of dynamic control flow. Given a mini-batch of input instances, dynamic batching involves lazily executing the model computation for each input instance while building dataflow graphs (DFGs) of tensor operators for each instance in the background. The execution of these DFGs is triggered when the value of a particular tensor is requested (when the model contains tensor-dependent control flow, for example). During this execution, the runtime can identify batching opportunities within the DFGs and launch batched kernels appropriately.

3 ACROBAT: OVERVIEW AND API

Control flow dynamism necessitates reliance on potentially expensive runtime analysis for auto-batching. In ACROBAT, we observe that aggressive static analysis often provides sufficient information to reduce the overheads of such analyses. Such analyses further enable us to generate specialized and more efficient tensor kernels in an end-to-end manner.

```
def @rnn(inps, state, bias, i_wt, h_wt) {
  match(inps) {
    Nil => Nil,
    Cons(inp, tail) => {
      let inp_linear = bias + nn.dense(inp, i_wt);
      let new_state = sigmoid(inp_linear + nn.dense(state, h_w));
      Cons(new_state, @rnn(tail, new_state, bias, i_w, h_w))
    }}}

def @main(rnn_bias: Tensor[(1, 256)], rnn_i_wt: Tensor[(256, 256)],
          rnn_h_wt: Tensor[(256, 256)], rnn_init: Tensor[(1, 256)],
          c_wt: Tensor[(16, 512)], cbias: Tensor[(1, 16)],
          inps: List[Tensor[(1, 256)]]) {
  (* Recursive computation stage (program phase 1) *)
  let rnn_res =
    @rnn(inps, rnn_init, rnn_bias, rnn_i_wt, rnn_h_wt);
  (* Output transformations stage (program phase 2) *)
  @map(fn(p: Tensor[(1, 256)]) {
         nn.relu(cbias + nn.dense(p, c_wt))
       }, rnn_res) }
```

Listing 1. A simple RNN model expressed in a functional language (here, Relay (Roesch et al., 2019) is used for illustration) as an input to ACROBAT.

We will now look at ACROBAT's compilation and execution workflows (illustrated in Fig. 1) that make use of the above insights. ACROBAT has been designed to take an unbatched DL computation expressed in a simple Turing-complete functional language as an input. This enables ACROBAT users to easily express models with dynamic control flow, such as the ones discussed in §2.1. For example, Listing 1 illustrates a simple RNN model which ACROBAT can take as an input.

Given an input computation ①, compilation in ACROBAT begins with batched kernel generation ②. Here, ACROBAT performs novel static analysis (§5.1) to identify data reuse opportunities and accordingly generates batched kernels ③ implementing the tensor operators used in the input program. Further, gather operator fusion (§5.2) enables us to generate specialized kernels that minimize data movement. These unoptimized kernels are then optimized by an auto-scheduler ④. Once optimized, target code ⑩ such as CUDA C++ can be generated for the batched kernels. Concurrently, the input program is further optimized and compiled ⑤ in an ahead-of-time (AOT) fashion to generate C++ code ⑦. As part of this compilation, ACROBAT generates code to (1) enable low overhead scheduling via our

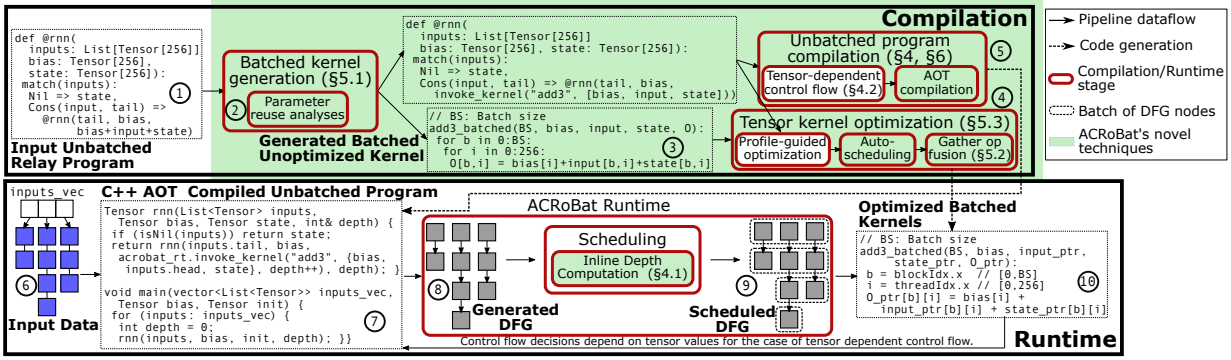

Figure 1. Overview of ACROBAT's workflow. Fig. 7 in the appendix shows a corresponding overview of DyNet, a prior fully dynamic approach. Note how ACROBAT performs significant novel analysis and code generation at compile-time to reduce runtime overheads.

inline depth computation approach, and (2) automatically enable concurrent execution in the presence of tensor dependent control flow (§4.2).

At runtime, ACROBAT lazily executes the AOT compiled input program ⑦ on a mini-batch of inputs ⑥, and constructs DFGs ⑧. The ACROBAT runtime library will then schedule these DFGs (using inline depth computation as mentioned above) ⑨, while looking for batching opportunities. Then, it will invoke the optimized batched kernels ⑩ for each identified batch of DFG nodes. If the input program exhibits tensor dependent control flow, the execution cycles back to the AOT compiled program which will execute further and create more DFGs.

We will now take a look at ACROBAT's hybrid optimizations in §4 and its tensor kernel generation in §5.

4 Hybrid Static+Dynamic Optimizations

Dynamic control flow often precludes static program transformations. Therefore, ACROBAT takes a hybrid approach whereby it exploits static program knowledge by either (1) providing hints to the dynamic analysis (§4.1), or (2) generating code that affords the dynamic analysis greater freedom in exploiting parallelism (§4.2). Further, static analysis also enables us to perform optimizations such as kernel fusion, which is important for high performance (§7.4). Below, we provide more details regarding our hybrid analysis.

4.1 Inline Depth Computation

As past work (Fegade et al., 2021) has noted, prior fully dynamic approaches incur significant scheduling overheads. For instance, as we will show in Table 5, DyNet's scheduling overheads dominate the time spent in tensor computations for the TreeLSTM model. Instead, as described below, ACROBAT devises a scheme to perform scheduling as it constructs the DFGs, thereby lowering scheduling overheads greatly (§7).

A DFG scheduling algorithm has two goals:

G.1 Correctness: Scheduling tasks such that dependences between the tasks are respected.

G.2 Performance: Identifying and exploiting parallelism.

Given a DFG(s), we can satisfy both these goals by executing DFG nodes (each of which represents one tensor operator) in the increasing order of their topological depth[2], such that nodes at the same depth are executed concurrently (Neubig et al., 2017a; Looks et al., 2017). We make the following two observations in order to compute these depths during DFG construction:

O.1 The order in which the unbatched program invokes the tensor operators, i.e. the order in which nodes are added to the DFGs, is a valid dependency order.

O.2 Information about instance parallelism (for example, recursive parallelism in the TreeLSTM model as seen in Table 2) is often available during compilation.

```
1  List<Tensor> rnn(List<Tensor> inps, Tensor state, Tensor bias,
2                   Tensor i_wt, Tensor h_wt, int& depth) {
3    if (inps == ListNil()) return ListNil();
4    auto inp_linear = AcrobatRT.InvokeKernel("bias_dense",
5                        0, {bias, i_wt, inps.head});
6    auto new_state = AcrobatRT.InvokeKernel("sigmoid_add_dense",
7                          depth++, {inp_linear, h_wt, state});
8    return ListCons(new_state, rnn(inps.tail, state, bias, i_wt,
9                                   h_wt, depth)); }
10
11 vector<Tensor> main(Tensor rnn_bias, Tensor rnn_i_wt,
12                     Tensor rnn_h_wt, Tensor rnn_init, Tensor c_wt,
13                     Tensor cbias, vector<List<Tensor>> inps_vec) {
14   vector<Tensor> res;
15   for (auto inps: inps_vec) {
16     int depth = 0;
17     /* Recursive computation stage (program phase 1) */
18     auto rnn_res = rnn(inps, rnn_init, rnn_bias, rnn_i_wt,
19                        rnn_h_wt, depth);
20     /* Output transformations stage (program phase 2) */
21     depth++;
22     res.push_back(map([&](Tensor p) { AcrobatRT.InvokeKernel(
23       "relu_bias_dense", depth, {cbias, c_wt, p}); }, rnn_res)); }
24   return res; }
```

Listing 2. AOT compiled output for the RNN model in Listing 1, with inline depth computation code highlighted.

Based on these observations, we set the depth of an oper-

[2] If $\mathcal{P}(n)$ denotes all the set of all producers of all the tensors a node n consumes, then its depth d_n is given by $d_n = 1 + \max_{p \in \mathcal{P}(n)} d_p$ if $\mathcal{P}(n) \neq \varnothing$ and 0 otherwise.

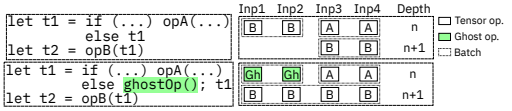

Figure 3. Ghost operators can enable better batching.

ator to be equal to its position in the dependency ordering induced by the execution of the unbatched program, thus meeting goal **G.1**. Then, we rely on observation **O.2** above in order to discover and exploit opportunities for parallelism by using the following techniques:

Instance Parallelism: We note that instance parallelism often stems from recursion or the use of the functional `@map` function on a list of independent items (observation **O.2**). We ensure that such concurrent operators are assigned the same depth during the execution of the unbatched program. We rely on simple user annotations to obtain information about recursive parallelism[3]. Fig. 2 shows an example where the two recursive calls to `funA` are annotated as concurrent. Note also that past work auto-parallelization (Hogen et al., 1992; Aleen & Clark, 2009) could potentially be used in lieu of such annotations. Listing 2 shows the AOT compiled code generated for the RNN model in Listing 1. We see, on line 23, how all invocations of the `relu_bias_dense` kernel inside the `@map` function are assigned the same depth.

```
funA() {
 concurrent {
  funA(); funA();
 }
 funC();
}
```

Figure 2. Concurrent call annotation.

Combating Eagerness of Depth Scheduling: As noted in past work (Neubig et al., 2017b), a depth-based scheduling scheme, like the one ACROBAT uses, can often be too eager in executing tensor operators, leading to a sub-optimal amount of exploited parallelism. Past work has relied on *agenda-based scheduling* (Neubig et al., 2017b), a more expensive scheduling scheme, as an alternative to the depth-based scheme to alleviate this problem. ACROBAT instead relies on compile-time analysis, as described below.

Ghost Operations: In the presence of conditional if statements, eager batching leads to sub-optimal batching as illustrated in the upper panes of Fig. 3. We see that eager batching leads to a sub-optimal batching schedule as the instances of operation B for inputs Inp1 and Inp2 are batched eagerly and more importantly separately from the instances of operation B for inputs Inp3 and Inp4. In such situations, ACROBAT can statically insert *ghost operations* to essentially delay the scheduling and execution of certain operators, as shown in the lower panes of the figure. Note that ghost operations merely affect ACROBAT's scheduling behavior and the are ignored during tensor kernel execution.

[3]Users can mark a set of function calls as concurrent in the input code. Of the seven models we evaluate in §7, four required one such annotation each, while the rest did not require any.

Program Phases: On the other hand, when repetitive (recursive or iterative) control flow is present, we rely on *program phases* (Sherwood et al., 2003) to combat the aforementioned sub-optimality of the scheduling. Given knowledge of such program phases, ACROBAT waits to schedule and execute operators in a phase until operators in all previous phases have been scheduled and executed. We find that considering individual semantic stages of the input DL computation as individual phases is a good heuristic for dividing the computation into phases. ACROBAT also provides a way for users to override this heuristic by manually annotate program phases, though in our evaluation, we did not need such annotations. We provide more details and explanations about program phases and ghost operations in §A.3 of the appendix.

Further, ACROBAT is also able to statically hoist operators, which we describe in more detail in §A.1 of the appendix. As an example, in Listing 2, the invocation of the kernel `bias_dense` on line 5 is assigned a statically computed depth of 0, which during runtime, effectively hoists the kernel invocation out of the recursion.

4.2 Tensor Dependent Control Flow

ACROBAT executes the unbatched program lazily to create DFGs for each input instance in the batch. In the absence of tensor dependent control flow, we can first execute the unbatched program for each instance sequentially and trigger the batching and execution of all the DFGs at once. In the presence of tensor dependent control flow, however, such sequential execution would not allow us to exploit any batch parallelism as we would be required to trigger the execution at control flow decisions that depend on the value of intermediate tensors. While prior work places the burden of restructuring input computations to alleviate this issue on the user, ACROBAT automatically generates code to execute the unbatched program for each input instance concurrently by using *fibers*[4]. This way, the unbatched programs can be executed for each instance to a point where none can progress without triggering the evaluation of the DFG. At this point, the evaluation can be performed, and the concurrent executions resumed after as illustrated in Fig. 4. Correspondingly, in order to exploit instance parallelism in the presence of tensor dependent control flow, ACROBAT launches concurrent fibers, similar to the fork-join model of parallelism (McCool et al., 2012). ACROBAT thus combines the static knowledge of parallelism with dynamic concurrent execution as part of its hybrid analysis to effectively exploit parallelism in the presence of tensor dependent control flow.

[4]Fibers (Boost, 2022) allow multiple execution stacks to be cooperatively scheduled on a single process.

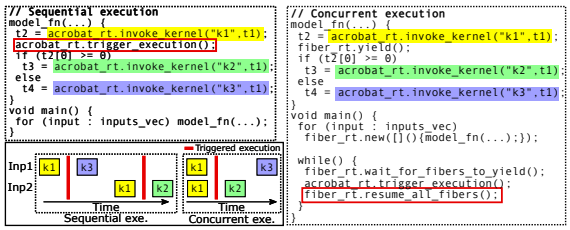

Figure 4. Concurrent execution of the unbatched program in the presence of tensor-dependent control flow.

Table 3. Models and datasets used in the evaluation.

Model	Description	Dataset
TreeLSTM	TreeLSTM	Stanford sentiment treebank (Socher et al., 2013b)
MV-RNN	MV-RNN	Stanford sentiment treebank
BiRNN	Bidirectional RNNs	XNLI (Conneau et al., 2018)
NestedRNN	An RNN loop nested inside a GRU loop	GRU/RNN loops iterate for a random number of iterations in [20, 40].
DRNN	Doubly recurrent neural networks for top-down tree generation	Randomly generated tensors.
Berxit	Early exit for BERT inference (Xin et al., 2021). All layers share weights.	Sequence length 128.
StackRNN	StackLSTM parser with LSTM cells replaced by RNN cells.	XNLI

5 END-TO-END TENSOR KERNEL GENERATION

As we alluded to above, ACROBAT enables end-to-end, uniform and automatic tensor kernel code generation by avoiding the use of vendor libraries. This allows ACROBAT to support a larger set of operators without additional compiler development effort. More details about ACROBAT's tensor kernel generation are provided below.

5.1 Exploiting Parameter Reuse

Given the input unbatched computation, ACROBAT needs to generate batched kernels implementing the tensor operators used in the computation. Generating these kernels is not straightforward because some input tensors (often model parameters) might be shared across calls to the operator. For example, across multiple calls to the element-wise addition operator `add3` used in the input computation ① in Fig. 1, the `bias` argument will be shared (as it is a model parameter) and hence should be reused across all values of the arguments `input` and `state`. This can be seen in the corresponding batched kernel (③ and ⑩) in Fig. 1.

A completely dynamic approach to auto-batching, such as the one used in DyNet, is unable to accurately identify such parameter reuse, and instead relies on heuristics, which can be brittle, leading to sub-optimal performance (§7.3). On the other hand, ACROBAT uses a 1-context sensitive[5] taint analysis to identify such shared arguments to tensor operators. The use of static analysis here allows ACROBAT to obtain accurate knowledge about the parameter reuse patterns.

Beyond the analysis described above, ACROBAT further explores opportunities for data reuse by employing code duplication and horizontal fusion as described in §B.1.

[5]Context sensitivity is a static analysis technique that allows the compiler to reason about a function in the different contexts it may be called under leading to increased analysis precision. For the DL computations we worked with, we found that a 1-context sensitive analysis was sufficient. Deeper contexts might be useful, however, for more complex computations.

5.2 Fusing Memory Gather Operations

As ACROBAT identifies batching opportunities across the DFGs dynamically, the input tensors to all DFG nodes in a batch may not be laid out contiguously in the accelerator's memory. In this scenario, prior work performs a memory gather before operating on the tensors (by invoking vendor library kernels), leading to significant data movement (§7.4). Instead, ACROBAT generates specialized batched kernels to directly operate on tensors scattered in memory, in effect fusing the expensive gather operation with the batched kernel. The generated batched kernel ⑩ in Fig. 1 illustrates this. This fusion can lead to a significant performance improvement as seen in §7.

6 IMPLEMENTATION DETAILS

Our prototype of ACROBAT is built upon TVM (Chen et al., 2018a) v0.9.dev0, a DL framework and a tensor compiler. It thus accepts as input computations expressed in Relay. Our prototype, ACROBAT also performs the grain size coarsening optimization (Zha et al., 2019; Xu et al., 2018; Fegade et al., 2021; Gao et al., 2018; Silfa et al., 2020), which is discussed more in §A.2 of the appendix.

As demonstrated in §7.2, we find that using an interpreted virtual machine (VM) for executing the unbatched programs can incur significant VM overheads in the presence of control flow dynamism. Therefore, ACROBAT compiles the input computation to C++ in an AOT fashion (as discussed in the appendix in §C). Further, as TVM does not support training, we evaluate ACROBAT for (batched) inference of DL computations. Other implementation details, including those on ACROBAT's use of TVM's auto-scheduler, can be found in the appendix in §C.

7 EVALUATION

We now evaluate ACROBAT against Cortex and DyNet on an Nvidia GPU. Cortex and DyNet are both state-of-the-art auto-batching frameworks for DL computations exhibiting recursive and general unrestricted control flow respectively. They have been shown to be faster than generic frameworks

Table 4. Relay VM vs. ACROBAT's AOT compilation: Inference latencies in ms.

Hidden Size	Batch Size	TreeLSTM		MV-RNN		BiRNN	
		VM	AOT	VM	AOT	VM	AOT
small	8	30.68	2.66	4.0	0.55	29.88	2.23
small	64	28.94	9.47	3.91	1.63	28.88	5.47
large	8	31.64	3.85	4.34	1.06	32.04	4.82
large	64	29.49	15.9	4.36	4.6	30.43	13.72

like PyTorch and TensorFlow (Neubig et al., 2017a;b; Fegade et al., 2021). We also compare ACROBAT's performance with that of PyTorch (§**??**).

7.1 Experimental Setup

Models: We use the models listed in Table 3 for the evaluation. For each model, we look at two model sizes—small and large. For the MV-RNN model, we use hidden sizes 64 and 128 for the small and large model sizes, while for the Berxit model, the small model uses the same hyper-parameters as the $\text{BERT}_{\text{BASE}}$ model (Devlin et al., 2018), while the large model uses the same hyper-parameters as the $\text{BERT}_{\text{LARGE}}$ model (Devlin et al., 2018), except that we use 18 layers instead of 24 in this case. For the remaining models, the small and the large model sizes use hidden sizes of 256 and 512 respectively.

Experimental Environment: We run our experiments on a Linux workstation with an AMD Ryzen Threadripper 3970X CPU (64 logical cores with 2-way hyperthreading) and an Nvidia RTX 3070 GPU. The machine runs Ubuntu 20.04, CUDA 11.1 and cuDNN 8.0.5. We compare against DyNet's commit 3e1b48c7 (March 2022) which uses the Eigen library (v3.3.90).

7.2 Benefits of AOT Compilation

We first look at the benefits of AOT compilation (§6). The performance of the TreeLSTM, MV-RNN and BiRNN models[6] when executed using the Relay VM and ACROBAT's AOT compiler (with the grain size coarsening, gather operator fusion and program phase optimizations turned on) is shown in Table 4. We see that overheads significantly slow down the execution (by up to 13.45×) as compared to the AOT compiled native code for these models. Therefore, for the rest of this section, we evaluate ACROBAT's performance with AOT compilation turned on.

7.3 Overall Performance

In this section, we compare ACROBAT's performance with that of PyTorch, DyNet and Cortex.

[6]ACROBAT's prototype implementation does not currently support the execution of the remaining models in Table 3 using the Relay VM.

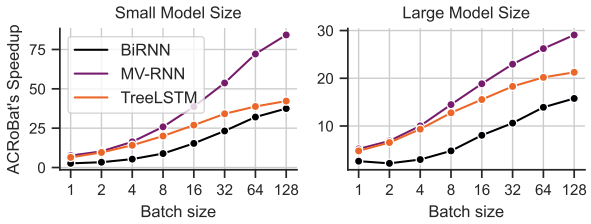

Figure 5. Speedups obtained over PyTorch for the TreeLSTM, MV-RNN and BiRNN models.

Performance Comparison with PyTorch
Fig. 5 compares ACROBAT's performance with that of PyTorch (v1.9.0a0+gitf096245) for the TreeLSTM, MV-RNN and BiRNN models[7]. PyTorch does not perform auto-batching and is therefore unable to exploit any available instance or batch parallelism in the evaluated computations. Further, ACROBAT's kernel fusion and other static optimizations also increase its performance relative to PyTorch. The speedups are higher for the small model size as compared to the larger model sizes because the relative importance of exploiting instance and batch parallelism is lower for the large model size due to the increased parallelism in individual tensor operators. ACROBAT's relatively worse performance on the BiRNN model as compared to the other two can be attributed to the absence of instance parallelism in BiRNN leading to a lower amount of parallelism that ACROBAT can exploit. Similarly, due to TreeLSTM exhibiting a higher amount of static and tensor parallelism as compared to MV-RNN, the relative importance of exploiting instance and batch parallelism is lower, leading to performance lower than that of MV-RNN.

Performance Comparison with DyNet
We now compare ACROBAT's performance with that of DyNet. As mentioned in §6, TVM does not support the training of DL models. Therefore, due to lack of access to trained model parameters, we use pseudo-randomness to emulate tensor dependent control flow in the NestedRNN, DRNN, Berxit and StackRNN models as part of our evaluation. We ensure that the pseudo-randomness is uniform across the ACROBAT and DyNet implementations by using pre-determined random seeds for a fair comparison. An exception is the DRNN model when inline depth computation is performed. In this case, ACROBAT exploits DRNN's recursive instance parallelism using fibers (§4.2) leading to a change in the random control flow decisions taken. We account for this by presenting the mean execution time across 50 different random seeds.

The execution latencies for DyNet and ACROBAT are

[7]We use TorchScript only for the BiRNN model as it does not currently support recursive data types (PyTorch Community, 2020), such as the parse trees the TreeLSTM and MV-RNN models operate on.

Table 5. DyNet vs. ACROBAT: Inference latencies (DyNet/ACROBAT) in ms and speedups. The DyNet implementation of the Berxit model was killed due to out-of-memory errors for a batch size of 64.

Hidden Size	Batch Size	TreeLSTM		MV-RNN		BiRNN		NestedRNN		DRNN		Berxit		StackRNN	
		Time	Speedup	Time	Speedup	Time	Speedup	Time	Speedup	Time	Speedup	Time	Speedup	Time	Speedup
small	8	4.31/**1.48**	2.93	2.11/**0.54**	3.96	3.13/**2.16**	1.45	**29.38**/31.01	0.95	6.7/**1.74**	3.87	63.54/**38.49**	1.66	47.78/**22.69**	2.11
small	64	26.18/**5.81**	4.51	12.45/**1.48**	8.47	12.04/**4.86**	2.49	84.55/**65.73**	1.29	25.3/**5.24**	4.84	-/204.54	-	213.98/**39.06**	5.48
large	8	4.58/**2.4**	1.92	2.27/**1.04**	2.19	**3.95**/4.43	0.9	46.03/**35.61**	1.3	8.44/**2.45**	3.45	113.18/**64.49**	1.76	64.67/**43.75**	1.48
large	64	26.53/**11.44**	2.33	13.89/**4.46**	3.13	**12.11**/13.11	0.93	**94.97**/100.17	0.95	26.5/**9.99**	2.66	-/335.3	-	230.74/**86.82**	2.66

Table 6. Time spent (ms) in various activities[1] for DyNet and ACROBAT for batch size 64.

Activity	TreeLSTM, small		BiRNN, large	
	DyNet	ACROBAT	DyNet	ACROBAT
DFG construction	8.8	1.5	4.5	1.0
Scheduling	9.7	0.4	3.3	0.4
Memory copy time	3.1	0.1	2.3	0.2
GPU kernel time[2]	6.1	4.0	6.6	11.2
#Kernel calls	1653	183	580	380
CUDA API time[3]	16.5	3.9	12.0	11.1

[1] The timings reported correspond to multiple runs, and were obtained using manual instrumentation and profiling using Nvidia Nsight Systems. Due to profiling overheads, the execution times may not match the ones in Table 5.
[2] Includes memory copy kernels.
[3] Includes calls `cudaMemcpy`, `cudaMemcpyAsync` and all kernels.

Table 7. Model execution times in ms after the improvements described in §7.3 were made for the TreeLSTM, MV-RNN and DRNN models. DN, DN++ and AB stand for DyNet, DyNet with improvements and ACROBAT respectively.

Model Size	Batch Size	TreeLSTM			MV-RNN			DRNN		
		DN	DN++	AB	DN	DN++	AB	DN	DN++	AB
small	8	4.31	3.8	1.48	2.11	1.05	0.54	6.7	3.29	1.74
small	64	26.18	22.69	5.81	12.45	3.15	1.48	25.3	18.51	5.24
large	8	4.58	4.14	2.4	2.27	1.83	1.04	8.44	3.82	2.45
large	64	26.53	24.09	11.44	13.89	10.47	4.46	26.5	18.86	9.99

shown in Table 5[8]. ACROBAT performs better than DyNet in most cases due to a number of reasons. Table 6 lists the time spent by the frameworks for different runtime activities for the TreeLSTM model. We see that ACROBAT's optimizations such as static kernel fusion and grain size coarsening reduce the number of tensor kernels invoked, thereby significantly reducing DFG construction and scheduling overheads. Further, inline depth computation allows ACROBAT to exploit available parallelism with lower overheads. Optimizations such as static kernel fusion and gather operator fusion enable ACROBAT to launch fewer GPU kernels, further reducing the time spent in the CUDA API. We look at the benefits of each of ACROBAT's optimizations in more detail in §7.4.

While, overall, ACROBAT performs 2.3× better than DyNet across all model configurations, DyNet performs slightly better than ACROBAT for some configurations of the BiRNN and NestedRNN models. For the former, Table 6 shows that while ACROBAT incurs lower runtime overheads for DFG construction, scheduling and memory transfer, it spends a higher amount of time in kernel execution compared to DyNet. We believe that better tensor kernel optimizations can help reduce this performance gap.

Beyond the above reasons, ACROBAT performs better on specific benchmarks for the reasons discussed next.

Accurate parameter reuse inference and automated batched kernel generation: As mentioned in §5.1, ACROBAT's use of static analysis for inferring parameter reuse allows it to have accurate knowledge to statically generate the appropriate batched kernels. On the other hand, DyNet's heuristic-based approach is unable to batch instances of certain operators, forcing sequential unbatched execution which leads to low performance. For instance, DyNet heuristically batches multiple instances of the matrix multiplication operator only when the first argument of all the instances is the same tensor. This usually works as the first argument is often a model parameter, usually as part of a linear transformation. Our DyNet implementation of the MV-RNN model, however, multiplies two intermediate tensor activations together, as a result of which DyNet is unable to batch instances of this operator, forcing sequential unbatched execution. When we modify DyNet's heuristic for matrix multiplication, its performance improves significantly as shown in Table 7.

Further, as described in §5, ACROBAT's end-to-end kernel generation leads to a broader coverage over tensor operators for which batching is supported as compared to approaches such as DyNet which rely on vendor libraries. As a result, DyNet does not support batching for certain operators, again leading to sequential execution and low performance. Specifically, DyNet does not support batched execution for the argmax operator, which the StackRNN model uses in order to determine the next parser action in every iteration based on the result of the embedded RNN cell. Similarly, the element-wise multiplication operator, used in the DRNN model, is executed in an unbatched manner when broadcasting needs to be performed. On the other hand, ACROBAT automatically generates optimized batched implementations of these tensor operators. We also find that DyNet is unable to batch calls to the operator that constructs constant tensors. We use this operator to initialize the hidden states of tree leaves in the TreeLSTM model. ACROBAT, on the other hand, statically recognizes that a constant tensor can be reused and thereby only creates the tensor once. The performance of the TreeLSTM model improves when we exploit this reuse manually in DyNet, as Table 7 shows.

[8] We consider the best of the two scheduling schemes DyNet implements (Neubig et al., 2017b) for each model configuration.

Table 8. Cortex vs. ACROBAT: Inference latencies in ms. Note that unlike ACROBAT, Cortex is limited to recursive computations, and does not support the other models in Table 3. Further, Cortex places a high development burden on its users by relying on manual kernel optimization.

Hidden Size	Batch Size	TreeLSTM		MV-RNN		BiRNN	
		Cortex	ACROBAT	Cortex	ACROBAT	Cortex	ACROBAT
small	8	**0.79**	1.48	1.14	**0.54**	**1.28**	2.16
small	64	**3.62**	5.81	6.92	**1.48**	**3.48**	4.86
large	8	**1.84**	2.4	5.3	**1.04**	**2.47**	4.43
large	64	**10.23**	11.44	41.15	**4.46**	**10.74**	13.11

Automated code generation for handling tensor dependent control flow: The DRNN model constructs a tree from an input vector representation in a top-down recursive manner. It exhibits both tensor-dependent control flow as well as instance parallelism (multiple subtrees can be generated concurrently). We saw how ACROBAT can automatically exploit instance parallelism in the presence of tensor-dependent control flow with the use of fibers in §4.2. On the other hand, DyNet is unable to exploit this parallelism and therefore ACROBAT's performance on this model is significantly better than that of DyNet. Table 7 also shows the performance improvement obtained in DyNet for the DRNN model when the instance parallelism exhibited by the model computation is manually exploited as detailed above.

Performance Comparison with Cortex
Table 8 compares the performance of ACROBAT with that of Cortex for the TreeLSTM, MV-RNN and the BiRNN models. Note that this is not an apples-to-apples comparison because, Cortex, being specialized for recursive computations, does not support general control flow (as is present in the other models in Table 3) unlike ACROBAT as mentioned in Table 1. Further, Cortex places a high development burden on users who are required to manually optimize and tune their models for specific hardware, unlike ACROBAT's automatic kernel generation[9]. Similarly, while ACROBAT can automatically hoist the input linear transformations out of the recursive computation in the TreeLSTM and BiRNN models (as described in §A.1), they need to be manually hoisted and offloaded to cuBLAS in the case of Cortex.

Being highly specialized for recursive computations, Cortex is able to exploit aggressive kernel fusion, model persistence and incur low kernel call overheads, thus performing up to 1.87× better than ACROBAT for the TreeLSTM and BiRNN models. However, note that Cortex performs much worse than ACROBAT on the MV-RNN model. This is because Cortex's restrictive API necessitates additional copies of the embedding vectors for the leaves of the input parse trees, which ACROBAT can avoid due to its more flexible interface. Overall, ACROBAT delivers performance comparable to that of Cortex, while supporting a much wider range of DL computations with much lesser developer effort.

[9]For example, implementing the MV-RNN model in Cortex requires 325 LoC in Python, as compared to the 79 LoC of Relay and 108 LoC of Python in ACROBAT.

7.4 Benefits of Optimizations

We now evaluate the relative benefits of the different optimizations ACROBAT performs. Fig. 6 shows the execution times for the models in Table 3 (at a batch size of 64) as we progressively perform optimizations. Standard kernel fusion (i.e. kernel fusion not including gather operator fusion as discussed in §5.2) provides significant benefits for all models[10]. Grain size coarsening and inline depth computation, both of which reduce scheduling overheads, are most beneficial for models with a relatively high amount of control flow such as TreeLSTM and MV-RNN. Further, in the case of the DRNN model, inline depth computation also enables ACROBAT to exploit the instance parallelism inherent in the computation (§4.2) leading to lower execution time. The BiRNN model involves per-token output linear operators as in token classification. Here, program phases allow ACROBAT to batch all these operators together as described in §4.1. The StackRNN model executes different tensor operators depending on the current parser action, which involves a conditional statement. Ghost operators therefore enable more optimal exploitation of parallelism leading to better performance.

Gather operator fusion is advantageous for some benchmarks and but not others. Such fusion leads to indirect memory accesses which can cause a slowdown in the kernel execution. While ACROBAT does hoist such loads out of loops when appropriate, this is not always possible depending on the schedule generated by the auto-scheduler. Further, gather operator fusion leads to a slowdown mostly in models with iterative execution and little instance parallelism. As in DyNet, when gather operator fusion is turned off, ACROBAT perform the explicit memory gather only when the input tensors are not already contiguous in memory. This is more likely to the case in such iterative models, thus blunting the advantages of gather operator fusion. Also, in models such as Berxit, the relatively high tensor computation cost of a coarsened static block further reduces any benefits gather operator fusion might provide.

Overall, models with a relatively lower amount of control flow or a higher amount of tensor computations such as Berxit or NestedRNN or models with the large size benefit less from optimizations that reduce scheduling overheads.

[10]The kernels used in the implementations with and without standard kernel fusion were auto-scheduled for the same number of auto-scheduler iterations.

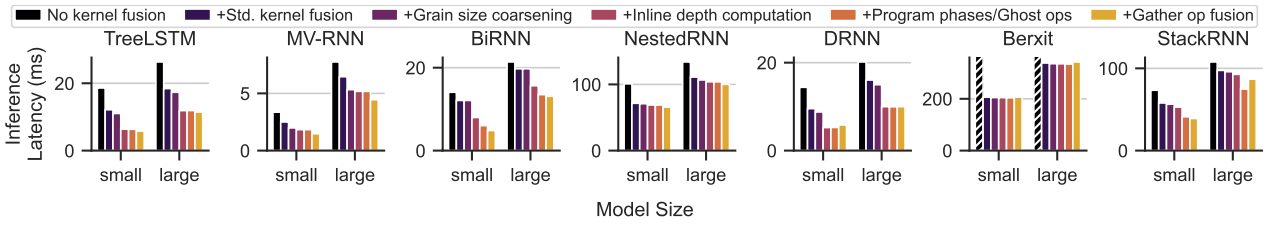

Figure 6. Benefits of different optimizations. The unfused executions of Berxit were killed due to out-of-memory errors.

8 Related Work

Auto-Batching for Dynamic Control Flow: There has been significant work on auto-batching techniques for dynamic computations. Beyond dynamic batching (which is used in various forms in DyNet, TensorFlow Fold, Cavs, Cortex and in ByteTransformer (Zhai et al., 2023) specifically for transformer models), static program transformations (Bradbury & Fu, 2018; Agarwal, 2019; Agarwal & Ganichev, 2019; Frostig et al., 2018; Radul et al., 2020) have also been explored for auto-batching. Such techniques are often unable to fully exploit all the available parallelism in the program as noted in (Radul et al., 2020). ACROBAT builds on these past techniques and effectively uses both static as well as dynamic analysis thus achieving lower runtime overheads while exploiting all the available parallelism. Online batching approaches for low latency RNN inference such as BatchMaker (Gao et al., 2018) and E-BATCH (Silfa et al., 2020) are complementary to ACROBAT. (Qiao & Taura, 2019) proposes improvements to the dynamic batching technique for back propagation, while ED-Batch (Chen et al., 2023) proposes efficient approaches to scheduling and memory planning for the dynamic batching. These can be further improved with ACROBAT's hybrid optimizations. Further, while grain size coarsening has been explored in past work, we use it statically in the context of general purpose auto-batching framework.

Optimizing Dynamic DL Computations: Beyond auto-batching, there is a large body of work on optimizing the execution of dynamic DL computations. Past work (Jeong et al., 2019; Kim et al., 2021; Suhan et al., 2021) has explored the lazy creation of DFGs that can be optimized to accelerate dynamic models. There has also been work (Durvasula et al., 2024; Zheng et al., 2023) at better scheduling and low-overhead execution tensor kernels to optimize for the dynamic execution patterns of dynamic DL computations. Further SoD^2 (Niu et al., 2024) develops techniques for optimizing dynamic computations including those with dynamic shapes. These techniques, which do not perform batching, are complementary to ACROBAT's techniques. While ACROBAT builds upon TVM, our techniques can be implemented in other commonly used compiler frameworks with expressive representations (PyTorch, 2020; Lattner et al., 2020) in a straightforward manner.

The gather operator fusion optimization is similar to the gather and scatter fusion (CUTLASS, 2022) performed for sparse GEMM in the CUTLASS library though we perform this optimization automatically as part of compilation. As mentioned in §C.1, ACROBAT borrows some techniques from DietCode for efficient code generation. DietCode's techniques are complementary to ours and it can be fully integrated into ACROBAT for better kernel performance.

Traditional Compiler Techniques: ACROBAT uses compilation techniques for programs written in general-purpose languages. These include context-sensitivity (Aho et al., 2007), taint analysis which is extensively used for security purposes (Tripp et al., 2009; Huang et al., 2015), profile-guided optimization (Chen et al., 2006; Gupta et al., 2002) (as discussed in §C.1 of the appendix) and program phases, which have been used to adaptively optimize systems for different parts of a program for optimal performance (Huang et al., 2001; Barnes et al., 2002). ACROBAT's inline depth computation and DFG scheduling more generally are similar to work on static and dynamic instruction scheduling for pipelined and superscalar processors (Smith, 1989; Ponomarev et al., 2001; Fisher, 1981; Gibbons & Muchnick, 1986). However, ACROBAT applies these techniques in the context of a DL framework.

9 Conclusion

This paper presents ACROBAT, a compiler and runtime framework that performs auto-batching of dynamic DL computations. ACROBAT employs hybrid static+dynamic analysis to enable effective batching with low runtime overheads, and end-to-end code generation to generate highly optimized tensor kernels for efficient execution. While we evaluated these techniques only for the case of batched inference, we believe that they also apply to DL training. In the context of the rising importance of dynamism in DL computations, we believe that ACROBAT is an important step towards more collaborative relationships between various components of a DL framework such as the tensor compiler, the high-level language compiler and the runtime.

Acknowledgments

This work was supported in part by grants from the National Science Foundation (award CNS-2211882), Oracle, IBM, Qualcomm, DARPA (Real Time Machine Learning,

or RTML project) and by the Parallel Data Lab (PDL) Consortium (Amazon, Facebook, Google, Hewlett-Packard Enterprise, Hitachi, IBM, Intel, Microsoft, NetApp, Oracle, Pure Storage, Salesforce, Samsung, Seagate, TwoSigma and Western Digital). We would like to thank Saman Amarasinghe, Dominic Chen, Siyuan Chen, Stephen Chou, Chris Fallin, Graham Neubig, Olatunji Ruwase and the Catalyst Research Group at Carnegie Mellon University for their valuable suggestions and feedback on our work.

REFERENCES

Agarwal, A. Static automatic batching in TensorFlow. In Chaudhuri, K. and Salakhutdinov, R. (eds.), *Proceedings of the 36th International Conference on Machine Learning*, volume 97 of *Proceedings of Machine Learning Research*, pp. 92–101. PMLR, 09–15 Jun 2019. URL https://proceedings.mlr.press/v97/agarwal19a.html.

Agarwal, A. and Ganichev, I. Auto-vectorizing tensorflow graphs: Jacobians, auto-batching and beyond. *CoRR*, abs/1903.04243, 2019. URL http://arxiv.org/abs/1903.04243.

Aho, A. V., Lam, M. S., Sethi, R., and Ullman, J. D. *Compilers: principles, techniques, & tools*. Pearson Education India, 2007.

Aleen, F. and Clark, N. Commutativity analysis for software parallelization: Letting program transformations see the big picture. *SIGARCH Comput. Archit. News*, 37(1):241–252, mar 2009. ISSN 0163-5964. doi: 10.1145/2528521.1508273. URL https://doi.org/10.1145/2528521.1508273.

Alvarez-Melis, D. and Jaakkola, T. Tree-structured decoding with doubly-recurrent neural networks. In *ICLR*, 2017.

Barnes, R., Nystrom, E., Merten, M., and Hwu, W. Vacuum packing: extracting hardware-detected program phases for post-link optimization. In *35th Annual IEEE/ACM International Symposium on Microarchitecture, 2002. (MICRO-35). Proceedings.*, pp. 233–244, 2002. doi: 10.1109/MICRO.2002.1176253.

Boost. Boost.Fiber, 2022. URL https://www.boost.org/doc/libs/1_79_0/libs/fiber/doc/html/index.html. Last accessed July 1, 2022.

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

Buckman, J., Ballesteros, M., and Dyer, C. Transition-based dependency parsing with heuristic backtracking. In *Proceedings of the 2016 Conference on Empirical Methods in Natural Language Processing*, pp. 2313–2318, Austin, Texas, November 2016. Association for Computational Linguistics. URL https://aclweb.org/anthology/D16-1254.

Chen, S., Fegade, P., Chen, T., Gibbons, P. B., and Mowry, T. C. ED-batch: efficient automatic batching of dynamic neural networks via learned finite state machines. In *Proceedings of the 40th International Conference on Machine Learning*, ICML'23. JMLR.org, 2023.

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, W.-k., Bhansali, S., Chilimbi, T., Gao, X., and Chuang, W. Profile-guided proactive garbage collection for locality optimization. *SIGPLAN Not.*, 41(6):332–340, jun 2006. ISSN 0362-1340. doi: 10.1145/1133255.1134021. URL https://doi.org/10.1145/1133255.1134021.

Chen, X., Qiu, X., Zhu, C., and Huang, X. Gated recursive neural network for Chinese word segmentation. In *Proceedings of the 53rd Annual Meeting of the Association for Computational Linguistics and the 7th International Joint Conference on Natural Language Processing (Volume 1: Long Papers)*, pp. 1744–1753, Beijing, China, July 2015. Association for Computational Linguistics. doi: 10.3115/v1/P15-1168. URL https://aclanthology.org/P15-1168.

Chen, X., Liu, C., and Song, D. Tree-to-tree neural networks for program translation. *CoRR*, abs/1802.03691, 2018b. URL http://arxiv.org/abs/1802.03691.

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.

Conneau, A., Rinott, R., Lample, G., Williams, A., Bowman, S. R., Schwenk, H., and Stoyanov, V. XNLI: Evaluating cross-lingual sentence representations. In *Proceedings*

of the 2018 Conference on Empirical Methods in Natural Language Processing. Association for Computational Linguistics, 2018.

CUTLASS. Gather and Scatter Fusion, 2022. URL https://github.com/NVIDIA/cutlass/tree/master/examples/36_gather_scatter_fusion. Last accessed July 25, 2022.

Devlin, J., Chang, M., Lee, K., and Toutanova, K. BERT: pre-training of deep bidirectional transformers for language understanding. *CoRR*, abs/1810.04805, 2018. URL http://arxiv.org/abs/1810.04805.

Drozdov, A., Verga, P., Yadav, M., Iyyer, M., and McCallum, A. Unsupervised latent tree induction with deep inside-outside recursive autoencoders. In *North American Association for Computational Linguistics*, 2019.

Durvasula, S., Zhao, A., Kiguru, R., Guan, Y., Chen, Z., and Vijaykumar, N. Acs: Concurrent kernel execution on irregular, input-dependent computational graphs, 2024.

Dyer, C., Ballesteros, M., Ling, W., Matthews, A., and Smith, N. A. Transition-based dependency parsing with stack long short-term memory. *CoRR*, abs/1505.08075, 2015. URL http://arxiv.org/abs/1505.08075.

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

Fedus, W., Zoph, B., and Shazeer, N. Switch transformers: Scaling to trillion parameter models with simple and efficient sparsity. *CoRR*, abs/2101.03961, 2021. URL https://arxiv.org/abs/2101.03961.

Fegade, P., Chen, T., Gibbons, P., and Mowry, T. Cortex: A compiler for recursive deep learning models. In Smola, A., Dimakis, A., and Stoica, I. (eds.), *Proceedings of Machine Learning and Systems*, volume 3, pp. 38–54, 2021. URL https://proceedings.mlsys.org/paper/2021/file/182be0c5cdcd5072bb1864cdee4d3d6e-Paper.pdf.

Fisher. Trace scheduling: A technique for global microcode compaction. *IEEE Transactions on Computers*, C-30(7): 478–490, 1981. doi: 10.1109/TC.1981.1675827.

Frostig, R., Johnson, M., and Leary, C. Compiling machine learning programs via high-level tracing. 2018. URL https://mlsys.org/Conferences/doc/2018/146.pdf.

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.

Gibbons, P. B. and Muchnick, S. S. Efficient instruction scheduling for a pipelined architecture. In *Proceedings of the 1986 SIGPLAN symposium on Compiler construction*, pp. 11–16, 1986.

Girshick, R. B. Fast R-CNN. *CoRR*, abs/1504.08083, 2015. URL http://arxiv.org/abs/1504.08083.

Girshick, R. B., Donahue, J., Darrell, T., and Malik, J. Rich feature hierarchies for accurate object detection and semantic segmentation. *CoRR*, abs/1311.2524, 2013. URL http://arxiv.org/abs/1311.2524.

Gupta, R., Mehofer, E., and Zhang, Y. Profile guided compiler optimizations, 2002.

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

Hogen, G., Kindler, A., and Loogen, R. Automatic parallelization of lazy functional programs. In *Symposium Proceedings on 4th European Symposium on Programming*, ESOP'92, pp. 254–268, Berlin, Heidelberg, 1992. Springer-Verlag. ISBN 0387552537.

Huang, M., Renau, J., and Torrellas, J. Profile-based energy reduction in high-performance processors. In *4th Workshop on Feedback-Directed and Dynamic Optimization (FDDO-4)*, 2001.

Huang, W., Dong, Y., Milanova, A., and Dolby, J. Scalable and precise taint analysis for Android. In *Proceedings of the 2015 International Symposium on Software Testing and Analysis*, pp. 106–117, 2015.

Intel. Intel oneAPI Deep Neural Network Library, 2022. URL https://www.intel.com/content/www/us/en/developer/tools/oneapi/onednn.html#gs.4je6v8. Last accessed July 1, 2022.

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.

Kaya, Y. and Dumitras, T. How to stop off-the-shelf deep neural networks from overthinking. *CoRR*, abs/1810.07052, 2018. URL http://arxiv.org/abs/1810.07052.

Kim, T., Jeong, E., Kim, G.-W., Koo, Y., Kim, S., Yu, G., and Chun, B.-G. Terra: Imperative-symbolic co-execution of imperative deep learning programs. In Ranzato, M., Beygelzimer, A., Dauphin, Y., Liang, P., and Vaughan, J. W. (eds.), *Advances in Neural Information Processing Systems*, volume 34, pp. 1468–1480. Curran Associates, Inc., 2021. URL https://proceedings.neurips.cc/paper/2021/file/0b32f1a9efe5edf3dd2f38b0c0052bfe-Paper.pdf.

Koehn, P. Pharaoh: a beam search decoder for phrase-based statistical machine translation models. In *Conference of the Association for Machine Translation in the Americas*, pp. 115–124. Springer, 2004.

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.

Leviathan, Y., Kalman, M., and Matias, Y. Fast inference from transformers via speculative decoding, 2023.

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.

Ma, J., Zhao, Z., Yi, X., Chen, J., Hong, L., and Chi, E. H. Modeling task relationships in multi-task learning with multi-gate mixture-of-experts. In *Proceedings of the 24th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*, pp. 1930–1939, 2018.

McCool, M., Robison, A. D., and Reinders, J. Chapter 8 - fork–join. In McCool, M., Robison, A. D., and Reinders, J. (eds.), *Structured Parallel Programming*, pp. 209–251. Morgan Kaufmann, Boston, 2012. ISBN 978-0-12-415993-8. doi: https://doi.org/10.1016/B978-0-12-415993-8.00008-6. URL https://www.sciencedirect.com/science/article/pii/B9780124159938000086.

Neubig, G., Dyer, C., Goldberg, Y., Matthews, A., Ammar, W., Anastasopoulos, A., Ballesteros, M., Chiang, D., Clothiaux, D., Cohn, T., Duh, K., Faruqui, M., Gan, C., Garrette, D., Ji, Y., Kong, L., Kuncoro, A., Kumar, G., Malaviya, C., Michel, P., Oda, Y., Richardson, M., Saphra, N., Swayamdipta, S., and Yin, P. Dynet: The dynamic neural network toolkit, 2017a.

Neubig, G., Goldberg, Y., and Dyer, C. On-the-fly operation batching in dynamic computation graphs, 2017b.

Niu, W., Agrawal, G., and Ren, B. Sod2: Statically optimizing dynamic deep neural network. *arXiv preprint arXiv:2403.00176*, 2024.

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.

Ponomarev, D., Kucuk, G., and Ghose, K. Reducing power requirements of instruction scheduling through dynamic allocation of multiple datapath resources. In *Proceedings. 34th ACM/IEEE International Symposium on Microarchitecture. MICRO-34*, pp. 90–101, 2001. doi: 10.1109/MICRO.2001.991108.

PyTorch. TorchScript, 2020. URL https://pytorch.org/docs/stable/jit.html. Last accessed Sept 09, 2021.

PyTorch Community. Github issue number 42487: Support recursive data type in TorchScript, 2020. URL https://github.com/pytorch/pytorch/issues/42487. Last accessed July 25, 2022.

Qiao, Y. and Taura, K. An automatic operation batching strategy for the backward propagation of neural networks having dynamic computation graphs, 2019. URL https://openreview.net/forum?id=SkxXwo0qYm.

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.

Roesch, J., Lyubomirsky, S., Kirisame, M., Weber, L., Pollock, J., Vega, L., Jiang, Z., Chen, T., Moreau, T., and Tatlock, Z. Relay: A high-level compiler for deep learning, 2019. URL https://arxiv.org/abs/1904.08368.

Rumelhart, D. E., Hinton, G. E., and Williams, R. J. Learning representations by back-propagating errors. *nature*, 323(6088):533–536, 1986.

Schuster, M. and Paliwal, K. K. Bidirectional recurrent neural networks. *IEEE transactions on Signal Processing*, 45(11):2673–2681, 1997.

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.

Sherwood, T., Sair, S., and Calder, B. Phase tracking and prediction. *ACM SIGARCH Computer Architecture News*, 31(2):336–349, 2003.

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.

Silfa, F., Arnau, J., and González, A. E-BATCH: energy-efficient and high-throughput RNN batching. *CoRR*, abs/2009.10656, 2020. URL https://arxiv.org/abs/2009.10656.

Smith, J. Dynamic instruction scheduling and the astronautics zs-1. *Computer*, 22(7):21–35, 1989. doi: 10.1109/2.30730.

Socher, R., Huval, B., Manning, C. D., and Ng, A. Y. Semantic Compositionality Through Recursive Matrix-Vector Spaces. In *Proceedings of the 2012 Conference on Empirical Methods in Natural Language Processing (EMNLP)*, 2012.

Socher, R., Perelygin, A., Wu, J., Chuang, J., Manning, C. D., Ng, A., 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, Seattle, Washington, USA, October 2013a. Association for Computational Linguistics. URL https://www.aclweb.org/anthology/D13-1170.

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, 2013b.

Suhan, A., Libenzi, D., Zhang, A., Schuh, P., Saeta, B., Sohn, J. Y., and Shabalin, D. Lazytensor: combining eager execution with domain-specific compilers. *CoRR*, abs/2102.13267, 2021. URL https://arxiv.org/abs/2102.13267.

Teerapittayanon, S., McDanel, B., and Kung, H. T. Branchynet: Fast inference via early exiting from deep neural networks. *CoRR*, abs/1709.01686, 2017. URL http://arxiv.org/abs/1709.01686.

Tripp, O., Pistoia, M., Fink, S. J., Sridharan, M., and Weisman, O. Taj: Effective taint analysis of web applications. *SIGPLAN Not.*, 44(6):87–97, jun 2009. ISSN 0362-1340. doi: 10.1145/1543135.1542486. URL https://doi.org/10.1145/1543135.1542486.

Wiseman, S. and Rush, A. M. Sequence-to-sequence learning as beam-search optimization. *CoRR*, abs/1606.02960, 2016. URL http://arxiv.org/abs/1606.02960.

Xin, J., Tang, R., Lee, J., Yu, Y., and Lin, J. Deebert: Dynamic early exiting for accelerating BERT inference. *CoRR*, abs/2004.12993, 2020. URL https://arxiv.org/abs/2004.12993.

Xin, J., Tang, R., Yu, Y., and Lin, J. Berxit: Early exiting for bert with better fine-tuning and extension to regression. In *Proceedings of the 16th conference of the European chapter of the association for computational linguistics: Main Volume*, pp. 91–104, 2021.

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.

You, J., Ying, R., Ren, X., Hamilton, W. L., and Leskovec, J. Graphrnn: A deep generative model for graphs. *CoRR*, abs/1802.08773, 2018. URL http://arxiv.org/abs/1802.08773.

Zha, S., Jiang, Z., Lin, H., and Zhang, Z. Just-in-time dynamic-batching. *CoRR*, abs/1904.07421, 2019. URL http://arxiv.org/abs/1904.07421.

Zhai, Y., Jiang, C., Wang, L., Jia, X., Zhang, S., Chen, Z., Liu, X., and Zhu, Y. Bytetransformer: A high-performance transformer boosted for variable-length inputs, 2023.

Zheng, B., Jiang, Z., Yu, C. H., Shen, H., Fromm, J., Liu, Y., Wang, Y., Ceze, L., Chen, T., and Pekhimenko, G. Dietcode: Automatic optimization for dynamic tensor programs. In Marculescu, D., Chi, Y., and Wu, C. (eds.), *Proceedings of Machine Learning and Systems*, volume 4, pp. 848–863, 2022. URL https://proceedings.mlsys.org/paper/2022/file/fa7cdfad1a5aaf8370ebeda47a1ff1c3-Paper.pdf.

Zheng, B., Yu, C. H., Wang, J., Ding, Y., Liu, Y., Wang, Y., and Pekhimenko, G. Grape: Practical and efficient graphed execution for dynamic deep neural networks on gpus. In *Proceedings of the 56th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO '23, pp. 1364–1380, New York, NY, USA, 2023. Association for Computing Machinery. ISBN 9798400703294. doi: 10.1145/3613424.3614248. URL https://doi.org/10.1145/3613424.3614248.

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 MORE DETAILS ON HYBRID STATIC+DYNAMIC OPTIMIZATIONS

A.1 Operator Hoisting

Given a recursive computation, such as the `@rnn` function in Listing 1, often certain tensor operators are not part of the sequential dependency induced by the recursion. For example, the linear transformation of the input on line 5 in Listing 1 can be hoisted out of the recursion. Instead of relying on a runtime scheduling algoritm to identify this as is done in past work, ACROBAT statically discovers such operators that can be hoisted. We achieve this by relying on a 1-context sensitive taint analysis to statically compute depths of such operators. We see, in Listing 2, how the invocation of the kernel `bias_dense` on line 5 is assigned a statically computed depth of 0. During runtime, such operators are thus effectively hoisted out of the recursion. For the RNN example, this allows us to batch the linear transformations for all input word embeddings together rather than execute them one at a time.

A.2 Grain Size Coarsening

Generally, scheduling is performed at the granularity of individual tensor operators i.e. each node in the DFG corresponds to one tensor kernel call. We saw in §2.1, how DL computations frequently contain larger static sub-graphs embedded in the dynamic control flow. Therefore, ACROBAT performs scheduling at the coarser granularity of static sub-graphs, thus reducing scheduling overheads. As these blocks do not contain any control flow, coarsening the granularity this way does not lead to a loss of exploited parallelism. This optimization has also been explored in past work (Zha et al., 2019; Xu et al., 2018; Fegade et al., 2021; Gao et al., 2018; Silfa et al., 2020) and is illustrated in Fig. 8.

A.3 Combating Eagerness of Depth Scheduling

We saw in §4.1 how ACROBAT relies on ghost operations and program phases to combating eagerness of depth scheduling. Below, we provide more detailed explanation of the same.

Ghost Operators: In upper panes of Fig. 3, we see that eager batching leads to a sub-optimal batching schedule in the presence of a conditional statement. Specifically, the instances of operator B for inputs Inp1 and Inp2 are batched eagerly and, more importantly, separately from the instances of operator B for inputs Inp3 and Inp4. In the lower panes, we insert a call to a ghost operator leading to an optimal schedule. ACROBAT statically identifies such cases and insert ghost operators as needed. Note that ghost operators merely affect scheduling and are ignored during kernel execution.

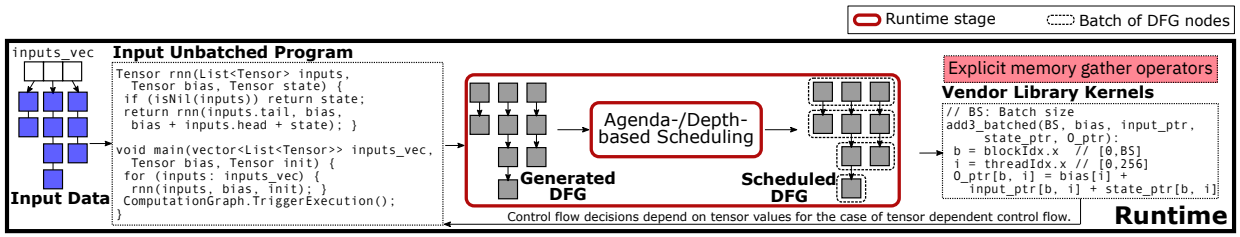

Figure 7. Overview of DyNet's runtime pipeline. Note the lack of any static, or compile-time analyses as well as how DyNet relies on explicit memory gather oprerations, leading to high data movement costs as we show in §7.

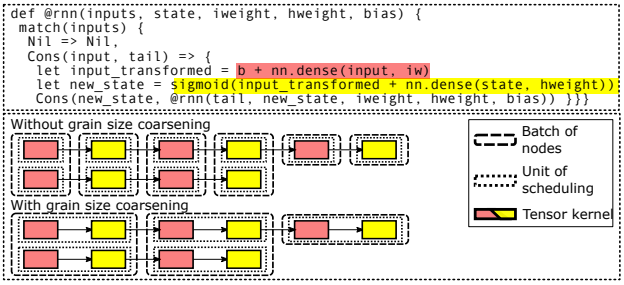

Figure 8. Grain size coarsening for the `@rnn` function in Listing 1.

Program Phases: For our RNN example in Listing 1, in order to exploit the most parallelism for the output operator on line 19, one should wait until all the operators invoked in the `@rnn` functionhave been executed for all the input instances. This way, all output operators corresponding to all words in all input instances can be executed as one batched kernel invocation. This would require that all these output operators be assigned the same depth. However, this may not be the case as the length of each input sentence may vary. Semantically, we can divide the RNN computation into two semantic stages—the initial recursive computations, and the following output transformations. Given such program phases, ACROBAT schedules and executes operators in one phase before moving on to the next. This way, ACROBAT ensures that all the RNN functions are executed for all input instances before moving on to the output operators.

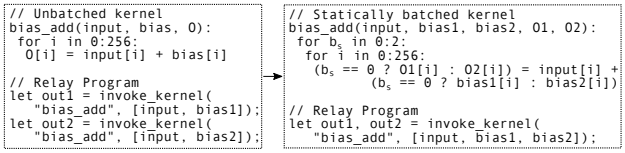

Figure 9. Horizontal fusion promotes parameter reuse.

B More Details on ACROBAT's Tensor Kernel Generation

B.1 Exploiting Data Reuse

Code Duplication for Better Data Reuse: Code reuse in the input program can often prohibit the parameter reuse mentioned above. Consider the following code listing, where, similar to the RNN model implemented in Listing 1, we implement a bidirectional RNN (BiRNN) (Schuster & Paliwal, 1997) computation. Here, we invoke the same `@rnn` function with different model parameters to implement the forward and backward RNNs. In this case, the tensor operators invoked by the `@rnn` function will not be statically determined to have any arguments constant across multiple calls, thereby precluding data reuse for the model parameters. In order to remedy this, before generating the batched kernels, ACROBAT recognizes such cases of data reuse (again using a context-sensitive taint analysis) and transitively duplicates the necessary functions to enable data reuse later when generating the batched kernels[11]. In the case of the BiRNN example, for instance, ACROBAT will transitively duplicate the `@rnn` function (including the tensor operators it invokes) and use a different copy of the `@rnn` function for each of the two forward and backward calls in the listing below.

```
1  (* Type annotations are omitted in the listing for simplicity. *)
2  def @main(f_rnn_bias, f_rnn_i_wt, f_rnn_h_wt, f_rnn_init,
3            b_rnn_bias, b_rnn_i_wt, b_rnn_h_wt, b_rnn_init,
4            inps_list) {
5    let rinps_list = @reverse_list(inps_list);
6    let forward_res = @rnn(inps_list, f_rnn_init,
7                           f_rnn_bias, f_rnn_i_wt, f_rnn_h_wt);
8    let backward_res = @rnn(rinps_list, b_rnn_init,
9                            b_rnn_bias, b_rnn_i_wt, b_rnn_h_wt);
10 }
```

Reuse Within Static Blocks: Given a tensor operator, the analysis discussed above takes into account parameters shared across calls made by different input instances in the mini-batch. This usually applies to model parameters as they are shared across multiple input instances. It is often the case, however, that multiple calls to the same tensor operator within the same static block share a parameter. For example, this is the case in the commonly used LSTM cell, where the computation of the four gates all involve concurrent linear transformations of the same input vector. In such cases, ACROBAT horizontally fuses such calls in order to exploit the parameter data reuse. This is illustrated in Fig. 9.

[11] Simply inlining the `@rnn` function will not work here as it is a recursive function.

C MORE IMPLEMENTATION DETAILS

C.1 Tensor Kernel Optimization

Below, we discuss how ACROBAT relies on TVM's auto-scheduler (Zheng et al., 2020) to automatically generate optimized implementations of batched versions of (potentially fused) tensor operators used in the input program.

Auto-scheduler Operator Priorities: Given a DL computation consisting of a number of tensor operators, the auto-scheduler prioritizes the optimization of tensor operators based on their relative estimated execution cost. Among other factors, this estimated cost is proportional to the number of times the operator is invoked during the execution of the input program. In order to accurately estimate this execution frequency for a given operator in the presence of control flow (such as repetitive or conditional control flow), ACROBAT relies on profile-guided optimization (PGO). When PGO is not possible, ACROBAT also provides a simple static analysis to heuristically perform this estimation based on how deeply nested an operator call is in the recursion.

Handling Variable Loop Extents: Due to the dynamic nature of ACROBAT's scheduling, the loop corresponding to the batch dimension in the generated unoptimized batched kernels has a variable extent (kernel ③ in Fig. 1, for example). In order to optimize these kernels, ACROBAT auto-schedules a corresponding kernel with a static loop extent for the batch dimension and automatically applies the generated schedule to the original kernel with the variable extent. Further, when generating code for loops with variable extents, we often have to insert conditional checks in order to avoid out of bounds accesses. We rely on the local padding and local partitioning techniques proposed in DietCode (Zheng et al., 2022) to eliminate these conditional checks when appropriate as they can be severely detrimental to performance

C.2 Ahead-of-time Compilation

We saw in §6 that ACROBAT compiles the input Relay computation to C++ in an ahead-of-time fashion. As part of this compilation, ACROBAT lowers all dynamic control flow as well as irregular data structures to native C++ control flow and classes. Relay handles scalars by modeling them as zero dimensional tensors. ACROBAT's AOT compiler lowers such zero-dimensional tensors and common arithmetic operators on them to native C++ scalars as well. We see, in §7.2, that this AOT compilation significantly reduces the execution overheads of dynamic control flow.

C.3 Other Details

As discussed in §6 of the main text, we prototype ACROBAT by extending TVM. We find that TVM's operator fusion pass is limited and is often unable to fuse memory copy operators such as tensor reshape, concatenation and transpositions. Therefore, in our implementations of the DL computations, we manually provide fusion hints to the compiler to force the fusion of such operators with their consumers. Further, our current prototype only supports the functional subset of Relay. Specifically, side-effects via mutable references are currently not supported. ACROBAT's runtime system has been heavily optimized to reduce runtime overheads. We use arena allocation (both on the CPU as well as on the GPU) and asynchronous execution on the GPU. We also batch memory transfer operations between the CPU and GPU when possible to reduce the CUDA API overheads.

Table 9. NestedRNN (small, batch size 8) execution times (without/with PGO), illustrating the benefits of using PGO invocation frequencies during auto-scheduling.

Auto-scheduler iters.	100	250	500	750	1000
Execution times (ms)	41.08/42.49	34.58/30.88	31.61/24.4	27.33/23.72	25.63/24.34

D SUPPLEMENTARY EVALUATION AND ADDITIONAL DETAILS

D.1 Benefit of PGO in Tensor Kernel Auto-Scheduling

We mentioned in §C.1 that ACROBAT uses invocation frequencies (obtained via PGO) to prioritize tensor operator optimization during auto-scheduling. In order to evaluate the benefit of this optimization, we look at the performance of NestedRNN with and without the optimization. This benchmark computation executes 30 iterations of the inner RNN loop per iteration of the outer GRU loop on an average. Therefore, the operators invoked in the RNN loop affect the performance of the benchmark much more than those invoked in the GRU loop. Table 9 shows the execution times of the benchmark with and without PGO for different iterations of the auto-scheduler[12] which shows how ACROBAT can better prioritize auto-scheduling for the RNN operators with PGO turned on.

[12] Due to the inherent randomness in the auto-scheduling process, the given execution times are averaged over 10 runs of the auto-scheduler each.