# Auto-batching Techniques for Dynamic Deep Learning Computations

**Pratik Pramod Fegade**

CMU-CS-22-152

November 2022

School of Computer Science
Computer Science Department
Carnegie Mellon University
Pittsburgh, PA

**Thesis Committee:**

Todd C. Mowry, Co-Chair
Phillip B. Gibbons, Co-Chair
Tianqi Chen, Co-Chair
Graham Neubig
Saman Amarasinghe, Massachusetts Institute of Technology

*Submitted in partial fulfillment of the requirements for the degree of Doctor of Philosophy.*



This research was supported by grants from the National Science Foundation, Oracle and IBM and the Parallel Data Lab (PDL) Consortium at CMU (Amazon, Google, Hewlett Packard Enterprise, Hitachi, Ltd., IBM Research, Intel Corporation, Meta, Microsoft Research, NetApp, Inc., Oracle Corporation, Pure Storage, Salesforce, Samsung Semiconductor Inc., Seagate Technology, Two Sigma and Western Digital).

The views and conclusions contained in this document are those of the author and should not be interpreted as representing the official policies, either expressed or implied, of NSF or any sponsoring institution.

**Keywords:** deep learning, compilers, dynamism, auto-batching, tensor compilers, ragged tensors, control flow

# Abstract

Deep learning has increasingly begun to be used across a wide range of computing applications. Dynamism—the property where the execution of a computation differs in some way across different inputs—has been shown to be an important tool in enabling deep learning models to effectively adapt to and model the varying structure of input data in various domains, thereby achieving high accuracy. On the other hand, dynamism often makes batching, an important performance optimization for deep learning computations, difficult to apply. This thesis presents techniques to enable efficient auto-batching—automatically enabling batched execution for a computation—for dynamic deep learning computations. Specifically, we consider two kinds of dynamism commonly exhibited by deep learning computations—control flow dynamism, where the model computation involves control flow structures such as conditional statements, loops and recursion, and shape dynamism, where the model computation involves computation with tensors of different shapes across different input data.

Past work has proposed a variety of approaches towards tackling the auto-batching problem in the presence of dynamism. However, we note that past work is characterized by significant *fragmentation* from a compilation and execution point of view. Techniques often target individual components of the compilation and runtime stack without taking a holistic view of the entire stack, and hence the entire computation into account. For instance, tensor kernels are often optimized in isolation, without knowledge of the larger surrounding computation, while auto-batching techniques often primarily rely either on compile-time program transformations, or on runtime analyses, rather than an end-to-end approach.

Taking these limitations of past work into account, the techniques in this thesis explicitly attempt to remove the fragmentation present in today's deep learning stacks to enable efficient auto-batching. Specifically, we rely on two insights (1) hybrid static+dynamic analysis to exploit all the available parallelism while keeping the runtime overheads to a minimum and (2) allowing the flow of information across the compilation and execution of tensor operators and the surrounding computation. These insights enable us to obtain significant gains over past work. For instance, Cortex, which is a compiler specialized for recursive deep learning computations achieves up to 14× faster inference over past work, while ACRoBat, an auto-batching framework that can handle unrestricted control flow is up to 8.5× faster. On the other hand, CoRa, a tensor compiler we designed for efficient batch execution in the presence of shape dynamism performs on-par with highly hand-optimized implementations of the transformer model.

# Acknowledgments

In the six and half years that I have spent at CMU, I have interacted with a lot of people, and my PhD would be all the more incomplete without them.

To start of, I would like to thank my advisors Prof. Todd Mowry, Prof. Phil Gibbons and Prof. Tianqi Chen. They have played a big role in directing my research, while giving me the space necessary to disagree, make my own mistakes and learn from them. Six years ago, I started with Todd working on the project Chris Fallin was leading (to develop compiler techniques to enable automated program comprehension at a semantic level) in collaboration with Phil. Working with Todd, Phil and Chris helped me gain a foothold in the PhD program. In the next three years, I added on Phil as an advisor and carved out a direction for myself to work on. In the fourth year of my PhD, when this direction did not seem promising anymore, Todd and Phil, helped me successfully pivot to deep learning. This was also when they introduced me to Tianqi (along with CK Luk, Olatunji Ruwase, Mangpo Phothilimthana and Cliff Young), whom I went on to collaborate for all the work in this thesis, eventually adding him on as an advisor.

Beyond technical advice and direction, Todd, Phil and Tianqi have always been a source of support and guidance. I am immensely thankful to them for this and sorry for the frustrated rants I often subjected them to during the course of the PhD. With their high-level guidance, technical and otherwise as well as their attention to detail, they have shaped this thesis and my PhD to an immeseasurable extent.

I would also like to thank my thesis committee members Prof. Graham Neubig and Prof. Saman Amarasinghe, for providing useful feedback and different perspectives on my work as well as this thesis. Their comments during my thesis proposal helped me come up with a concrete plan for this thesis, and the confidence to stick with it.

The work in this thesis has also greatly benefited from discussions, insights and feedback for a variety of people including Tunji, Randy Huang, Prof. Zhihao Jia, Prof. Andy Pavlo, Prof. Emma Strubbel, CK, Mangpo and Cliff.

I would also like to thank my collaborators in the industry. I spent the summer of 2019 at Oracle Labs working with Christian Wimmer. Despite my lack of experience working in the industry, Christian was extremely patient and always available to answer questions—from difficulties understanding the GraalVM source code to how the build system worked. I would also like to thank Harshad Kasture, Mario Wolczko and Matthias Brantner for their advice and insights into my work and into the best food on the Oracle campus. I would also like to thank Mark Stoodley and Vijay Sundaresan of IBM Canada for their guidance and advice on our work on microservice compilation.

I would also like to thank the members of the Parallel Data Lab (PDL). The annual PDL Visit Day and Retreat[1] have always been the highlight of the year. The feedback I received from the industry visitors helped me better understand the big picture and orient my work accordingly. As a part of preparations for the retreat, Greg Ganger, Dave O'Hallaron, Charlie McGuffey and other PDL members have helped improve the presentation skills of countless PDL members, including mine. I am grateful to Karen, for organizing the PDL events, Joan for making sure our posters looked professional, and for being incredibly patient with my last minute poster changes, and Jason, Mitch, Chad and Bill for making sure the PDL events went smoothly and without any hitches. During the first few years of my PhD, the CIC 4th Floor Reading Group[2] provided me with an opportunity to explore the areas of computer architecture and computer systems beyond the corner of the compiler world I had made my home in, while in the later years of my PhD, the Catalyst research group has also been a source of fruitful research discussions and feedback.

The speaking and writing skills requirements went a long way towards helping me improve my soft skills. I am grateful to Tianqi, Prof. Nathan Beckmann, Prof. Anupam Gupta, and fellow PhD students Ryan Kavanagh and Jonathan Laurent for serving on my speaking skills committee and giving useful feedback. I'm also thankful to Prof. Jan Hoffmann, Prof. Jonathan Aldrich and Ankush Das for serving on my writing skills committee.

This journey would not have been possible without the extremely random discussions, camaraderie and support of the wonderful friends I made at CMU and in Pittsburgh.

In Chris, Dominic Chen and Jin Kyu Kim, I found brilliant researchers, amazing mentors and enviable friends. Chris has been heavily involved in my research as well as my growth during for the entire duration of my PhD. His ability to zero in on the issue at the core of any problem and his incisive comments have helped me improve the work in this thesis in ways hard to quantify. Dominic started working with Phil in the second year of my PhD, and since then, the three of us have met weekly to discuss all things research and beyond. These meetings have kept me sane throughput and I appreciate the fact that Chris and Dominic made the meeting work even after graduating from their respective PhDs. Jin made me feel welcome when I moved in GHC 6219, and has been a source of constant support, advice and friendship ever since. Our frequent trips to Seoul Mart, Two Sisters and Ramen Bar[3] were a source of great joy and went a long way towards making Pittsburgh feel like home.

Logan Brooks and Yuanhao Wei, along with Jin, formed the GHC 6219 family. With them, I have spent countless hours chatting about the best restaurants in Pittsburgh and the travails of a PhD in the same breath. They truly made the journey bearable and the segfaults easier to debug. Yong Kiam Tan joined us in 2019 and tolerated and participated in our often irrelevant discussions making them all the more lively and enjoyable.

I am also immensely grateful to my house-mates and neighbors I met during my stay in Pittsburgh. I arrived in the US with Devdeep, with whom I shared a room or a wall during my undergraduate

[1] I'm very glad the PDL Retreat returned for an in-person iteration in 2022 after a COVID-19-induced hiatus.

[2] The reading group was an unfortunate casualty of the COVID-19 pandemic

[3] I will be eternally grateful to Jin for introducing me to Pho and Ramen!

studies. In a completely foreign place, Devdeep's company has always made me feel at home. I will always cherish our lazy movie nights and Devdeep's unrelenting but ultimately unsuccessful efforts to get me interested in video games (and cars, and flying, and Visual Studio Code[4] and using more than one monitor, and …). With Akshay, Prerana, Ashwin, Aravind and Kalyani, we started our journeys at CMU and in Pittsburgh together. In Squirrel Hill, where I would go on to spend the majority of my time in Pittsburgh in a creaky but cozy home, I met Shreyanshi, Shivani, Meghna, Sheona, Maahin, Rohit, Pragya and Noori. As we were all stuck in the confines of our bedrooms (which doubled as classrooms and offices) during the COVID-19 pandemic, their company kept me sane. The late night walks on Murray Avenue with Shivani and the puzzles we solved[5] will always be fondly remembered. Shreyanshi's ability to heavily hype the mundane and the sushi dinners we enjoyed, Meghna and Maahin's delicious food, Rohit's crazy stories, the wide ranging conversations (from Noori and the cat's[6] antics to pediatrics) we had with Pragya and Noori's canine companionship provided much-needed fuel for the PhD.

Thanks also to my friends at CMU and beyond. I'm thankful for Ankur's advice, support and our long ranting sessions on the PhD, politics, love and beyond. Saloni's crazy doctor stories provided a window beyond the silo of computer science. Ananya Joshi and KA always provided a supportive and empathetic space for me to vent. Jovina Vaswani's poems and cheerfulness, and May Li's movie and TV show recommendations[7] kept me going in the later slow parts of the PhD. My weekly calls with Anant helped me find support and reminded me of the existence of life after the PhD. Aishwarya has always been there to hear me rant, celebrate my milestones and give me pep talks when I am down.

I would also like to thank my therapists Kym, Paul and Alex for helping me navigate the ups and downs of the journey and grow as a person along the way.

All of this would not be possible without my family. Mom, Dad and my sister Anuja have been my pillars, standing by me, supporting me and admonishing me when I strayed. I am immensely and forever grateful to them for their unwavering patience, love and advice as I sometimes got too engrossed in the PhD and my life here. In the slow last few months of the PhD, Mom flew to Pittsburgh, keeping me sane with her delicious food and cheerful company. I'm very thankful for that. Back home in India, Anuja and Dad held out for the few months Mom was here, and that must have been hard!

Finally, I would also like to thank Tazza d'Oro[8], Dobra Tea Cafe, Turkish Kebab House, Allegro Hearth Bakery and Sichuan Gourmet for keeping me well-fed and well-sugared and for providing a change from my terrible cooking.

[4] I'm faithful to emacs.

[5] Actually, it was just one very large puzzle which we never really completed.

[6] The cat for some reason never had a name!

[7] Dark and Comrade Detective are pretty cool shows!

[8] For the umpteenth time, I invoke COVID-19 here to say how they have amazing sandwiches and pastries and the fact that La Prima replaced them in the GHC cafe after the pandemic was just one more reason I got too lazy to never really go back to campus for work.

# Contents

# List of Figures

# List of Tables

# 1 Introduction

Since AlexNet [67]'s extraordinary success in the ImageNet Large Scale Visual Recognition Challenge in 2012, the field of deep learning has grown exponentially, expanding beyond the traditional domains of image and natural language processing and being incorporated in some way or the other in most domains of computing. Deep learning models, which are the central abstractions used in deep learning, therefore have been designed to operate on a wide variety of data such as spatial images, temporal signals, relational data, molecular graphs, and so on. Beyond the ability to support these disparate kinds of data, deep learning also needs to be amenable to highly efficient execution due to its importance to the modern computating landscape today.

In deep learning, data are often modeled using tensors, which are multi-dimensional arrays, most often containing real numbers. As a consequence, deep learning computations usually involve computations on tensors, expressed in terms of common tensor operators such as linear transformations, non-linear activations and so on. Most tensor operators can be expressed as nested loop computations over the multi-dimensional input tensors to produce an output tensor. Figure 1.1 shows the structure of some common deep learning models and the tensor operators they employ as part of their computation.

## 1.1 Dynamism in Deep Learning Computations

In order to effectively model and perform learning on data, deep learning models and computations in general[1] are often designed to mirror the structure of the input data. For instance, image processing models often rely on the 2-dimensional convolution operator, while sequence models such as recurrent neural networks (RNNs) [108] iterate over the tokens of an input sequence in order. Such a design allows better modeling of the underlying data distribution and therefore leads to better deep learning outcomes.

*Execution Dynamism*, or *dynamism* for short, is an important technique that deep learning reseachers often rely on in order to design such expressive models. Informally, we say that a deep learning computation exhibits dynamism if the execution of the computation may differ in some way across multiple inputs. For example, the RNN model executes as many iterations as there are tokens in the input sequence. Therefore, its execution (specifically the number of iterations it executes) differs across multiple input sequences of varying lengths and it is therefore referred to as a *dynamic* model. On the other hand,

[1] In this thesis, we use the term deep learning computations to include specific deep learning models, as well as other supplementary computations used in deep learning such as beam search decoding [131], or larger deep learning applications composed of multiple deep learning models.

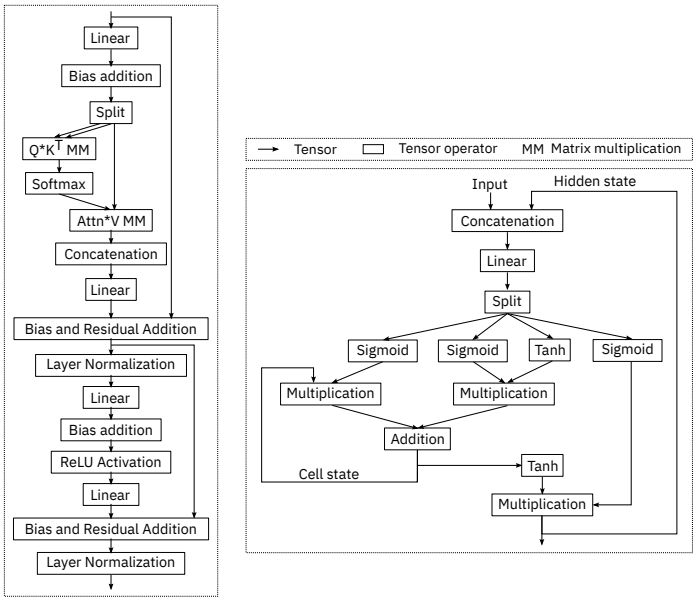
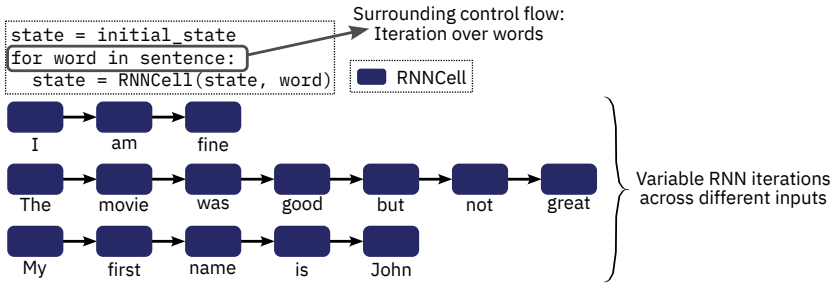
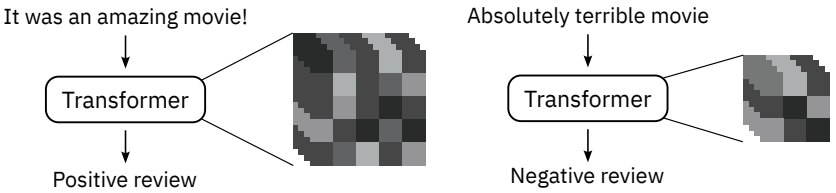



Figure 1.1: Two example deep learning models: An encoder layer of a transformer [137] and a Long Short Term Memory (LSTM) [49] cell.




(a) Control flow dynamism in RNNs.




*Longer input sentences lead to larger attention matrices, while shorter ones lead to smaller attention matrices*

(b) Shape dynamism in transformers.

Figure 1.2: Illustration of control flow and shape dynamism in deep learning computations.

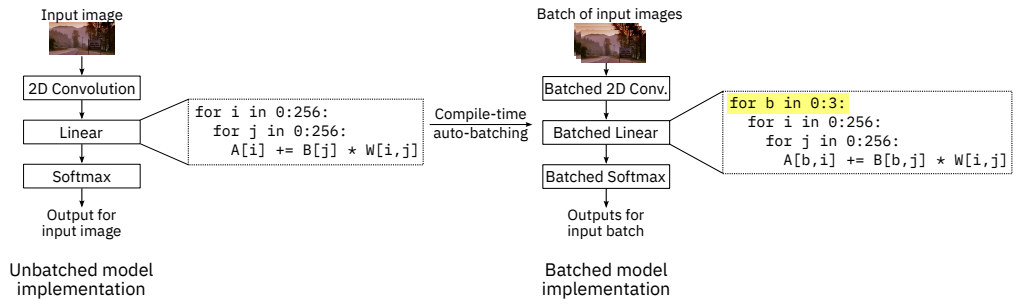



Figure 1.3: Auto-batching for static deep learning computations.

AlexNet executes the same set of operations in the same order for all input images, and therefore does not exhibit dynamism. Such models are referred to as *static* models. We will further explore dynamism in different deep learning computations in Chapter 2.

In this thesis, we focus our attention on deep learning computations that exhibit either *control flow dynamism* or *shape dynamism*. A computation is said to exhibit control flow dynamism when different inputs may follow different control flow paths during the execution. RNNs exhibit control flow dynamism. Specifically they exploit iterative control flow in order to process temporal sequences. On the other hand, a computation is said to exhibit shape dynamism when the execution of the computations involves tensors of varying shapes across different inputs. A common example is the transformer model, where depending on the length of the input sequence, the intermediate tensors have different shapes. We illustrate these different kinds of dynamism in Figures 1.2a and 1.2b.

## 1.2 Batching as a Performance Optimization in Deep Learning

In general, batching the execution of multiple computational tasks instead of executing them separately often leads to significant performance improvements by, for instance, amortizing execution overheads over the entire batch of tasks. Similarly, in deep learning computations, batching the execution for various data inputs is often found to improve throughput and underlying hardware utilization. As we saw above, deep learning computations are usually composed of various tensor operators. Executing a computation in a batched fashion therefore boils down to executing each tensor operator involved in the computation over an entire batch of inputs, to produce a batch of outputs. This is illustrated for a static model in Figure 1.3.

Due to the fact that batching is primarily a performance optimization, deep learning practitioners often prefer designing and implementing *unbatched implementations* of deep learning models. An unbatched implementation of a computation is an implementation which performs execution over one data input at a time. Therefore, it is preferable for a deep learning framework to automatically enable batched execution of an unbatched implementation of a computation. In this thesis, we refer to this process, illustrated in Figure 1.4, as *auto-batching*.

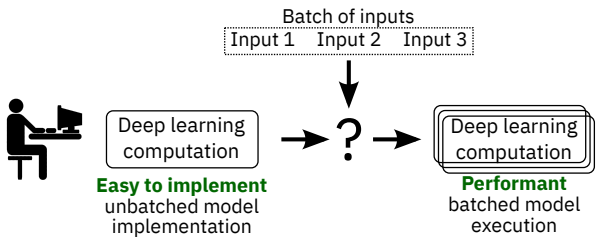



Figure 1.4: The auto-batching problem.

A deep learning framework can perform auto-batching either during compilation, or during execution. When performed during compilation, the framework statically transforms the input unbatched computation to emit a batched implementation of the same computation, which is then executed. On the other hand, when performed during execution, the deep learning framework lazily executes the unbatched computation for all the data inputs in the batch and then, at runtime, identifies opportunities to execute different tensor operators in a batched fashion. We will look at this process, referred to as *dynamic batching*, in more detail in Chapter 2.

## 1.3 The Problem: Auto-batching in the Presence of Dynamism

As we saw above, static computations execute the same sequence of tensor operators for all inputs over tensors of known shapes. It is therefore straightforward to perform static auto-batching for such computations by essentially replacing every tensor operator involved in the computation by the corresponding batched operator. Note how the batched linear operator in Figure 1.3 has an additional loop corresponding to the additional batch dimension[2]. Modern deep learning frameworks such TensorFlow and PyTorch can in fact perform auto-batching for such computations during compilation. On the other hand, the compiler lacks perfect execution knowledge about the unbatched input computation in the presence of dynamism. This makes efficient compile-time auto-batching difficult, as past work has shown.

Let us take the example of the RNN model to illustrate this further. The execution of this model over a batch of input sequences of varying lengths is shown in Figure 1.5. Due to the variation in the number of iterations executed for each input sequence, the simple auto-batching scheme we saw above would not work. At best, we would have to pad the input sequences so that they are all of the same length as shown in the bottom part of the figure. This would, however, lead to wasted computation on the added padding data, thereby leading to sub-optimal performance.

On the other hand, when attempting to execute the transformer model, which exhibits shape dynamism, in a batched fashion on a batch of input sequences of varying lengths, one has to perform operations on *ragged tensors* [129] as shown in Figure 1.6. Ragged tensors are tensors the slices of one or more inner dimensions of which have variable sizes. Executing tensor operators on such ragged ten-

[2]Note also how the model parameter `W` is shared across multiple iterations of this batch loop. This data reuse is one reason for the performance benefits obtained due to batching.

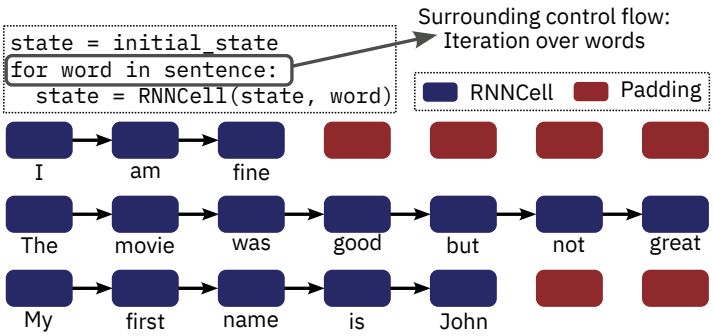



Figure 1.5: Simple auto-batching when applied to RNNs leads to wasted computation.

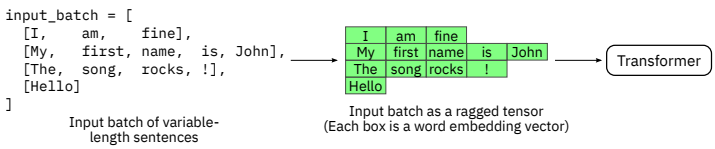



Figure 1.6: Batching in the presence of shape dynamism leads to computations on ragged tensors.

sors is not straightforward. As with the RNN example above, one can pad the tensor so it is essentially transformed into a dense tensor. As we shall see in Chapter 5, here too, such padding can lead to high performance penalties.

## 1.4 Fragmentation in Past Solutions

There has been significant work that addresses the problem of auto-batching for dynamic deep learning computations. Below, we give a high-level overview and discuss some downsides of this body of work. We further explore this in Chapter 2.

### 1.4.1 Fragmentation Between the Compiler and the Runtime

We saw above how our simple static auto-batching technique does not lead to efficient batched execution for models that exhibit control flow dynamism. Past work has explored other program transformations, often akin to those used to solve the related auto-vectorization problem, for this purpose. Due to the compile-time nature of these techniques, they are often very conservative in the kinds of transformations they can perform, or too restrictive in the kinds of control flow patterns they support. For example, Jax's `vmap` and `pmap` primitives, which implement static auto-batching, do not support control flow structures such as recursion. On the other hand, past work has also explored runtime techniques and analyses to perform auto-batching, on the lines of the dynamic batching approach outlined above. This includes frameworks such as DyNet and TensorFlow Fold. The heavy reliance on dynamic analysis in such approaches, however, can lead to high execution overheads and therefore low end-to-end performance.

In general, we note that past work for auto-batching in the presence of control flow dynamism has tended to either rely primarily on static or on dynamic analyses. As we show in the later chapters, this fragmentation between the compilation and the runtime stages of the deep learning workflow can lead to inefficiencies in the execution of the computations.

### 1.4.2 Fragmentation Between Tensor Kernels and the Surrounding Computations

Traditionally, deep learning workflows have optimized and compiled the individual tensor operators involved in a computation separately from the surrounding computation which dictates how the tensor operators are invoked. This has often manifested in the use of tensor kernels implemented in libraries provided by hardware vendors (or *vendor libraries*, for short) such as cuDNN for Nvidia GPUs, Intel oneDNN for Intel CPUs and AOCL for AMD CPUs. More recently, the use of tensor compilers for automatically generating and optimizing tensor operators has also increased. Even in these cases, the kernels are often optimized in isolation from the surrounding computation. In general, this fragmentation between the handling of tensor kernels and the surrouding computation can also lead to sub-optimal end-to-end performance.

In the case of shape dynamism, specifically, this can lead to an impedance mismatch between the dynamism exhibited by the computation and the operations provided by vendor libraries or tensor compilers. The latter two most commonly implement operations on dense or sparse tensors, either of which, when used for executing operations on ragged tensors, can cause significant performance degradation.

## 1.5 Thesis Statement and Contributions

As we saw above, past work for auto-batching in the presence of shape and control flow dynamism suffers from fragmentation along two axes—across the compiler and the runtime, and across the compilation and execution of tensor kernels and the larger surrounding computation. Accordingly, in this thesis, we propose and provide evidence for the following hypothesis:

> *Efficient and performant auto-batching for deep learning computations exhibiting control flow and/or shape dynamism can be achieved via (1) the use of hybrid static and dynamic program analyses, rather than either of them individually, and (2) allowing the flow of information across the compilation and execution of tensor operators and the surrounding computation.*

Concretely, we design and evaluate the following three compiler-based frameworks to support this hypothesis (the structure of the thesis is also illustrated in Figure 1.7.):

### Cortex

Cortex (Chapter 3) is a compiler we design to accelerate inference for recursive deep learning computations. By specializing for the control flow structure of execution of such recursive computations,

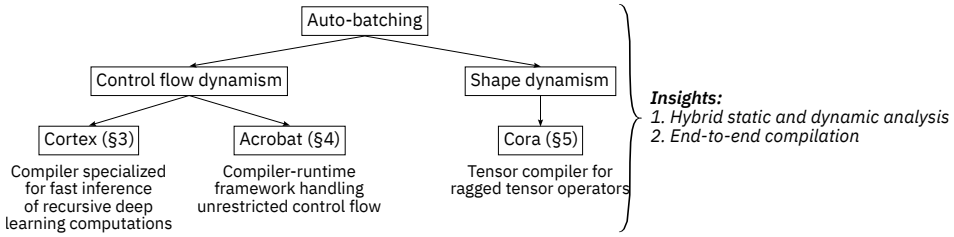



Figure 1.7: Structure of this thesis.

Cortex intelligently splits the input recursive computation into a lightweight recursive data structure traversal which can be executed on the host CPU and iterative tensor computations which can be executed on the deep learning accelerator. This reliance on a lightweight dynamic analysis in the form of the data structure traversal, the code for which can be generated automatically by the compiler, can be seen as an example of the aforementioned hybrid static and dynamic analysis. Further, Cortex performs an end-to-end compilation of both the surrounding control flow and the tensor operators as part of the same pipeline. This design allows Cortex to enable highly efficient execution of recursive computations, performing up to 14× faster than DyNet [84, 85] which implements a completely runtime auto-batching technique. Chapter 3 describes Cortex in further detail, while Appendix A provides other supplementary material on the same.

### ACRoBat

Cortex provides encouraging evidence for our aforementioned twin insights of hybrid analysis and end-to-end compilation. Next, we describe ACRoBat, a compiler and runtime framework to enable auto-batching in the presence of unrestricted control flow dynamism. In the absence of intimate knowledge about the execution of the input computation, ACRoBat relies on dynamic analysis much more heavily as compared to Cortex. However, ACRoBat also employs aggressive compile-time analysis and program transformations to reduce the execution overheads of its dynamic analysis. Further, as compared to Cortex, ACRoBat automates the optimization of tensor kernels, while taking the context of the larger surrounding context into account. ACRoBat's design of a holistic compilation and runtime stack enables it to be up to 8.5× faster than DyNet, and only at most two times as slow as Cortex, while supporting a much wider range of computations in a more user-friendly manner. Chapter 4 describes ACRoBat in further detail.

### CoRa

Moving on to efficient batched execution in the presence of shape dynamism, Chapter 5 (in conjunction with Appendix B) describes our design of CoRa, a compiler that enables one to generate efficient implementations of ragged tensor operators on CPUs and GPUs. CoRa is built on the insight that after some cheap runtime pre-computations (another example of hybrid static and dynamic analysis), one can generate highly efficient ragged operator implementations, in a manner similar to current dense

tensor compilers. Further, CoRa's use of partial padding enable it to enjoy most of the benefits of full paddding, without the accompanying performance penalties. Overall, such a design enables a CoRa-generated implementation of the transformer model to perform on-par with FasterTransformer [88], a highly hand-optimized implementation of the same.

In addition to the chapters and appendices mentioned above, Chapter 2 explores in more detail the kinds of control flow and shape dynamism exhibited by deep learning computations and also surveys and describes more deeply the past work undertaken in order to enable auto-batching in the presence of dynamism.

# 2 Background

This chapter provides the necessary background upon which the techniques of this thesis are built. First, in §2.1, we explore in more detail the dynamism found in various deep learning computations. §2.2 then gives a high-level overview of what a common deep learning compilation workflow looks like, in the context of the auto-batching problem, while §2.3 explores some of the past work that has been done towards enabling efficient auto-batching in the presence of dynamism.

## 2.1 Dynamism in Deep Learning

In Chapter 1, we saw how in recent years, highly expressive deep learning models have often involved dynamism. Below, we will look at how different deep learning computations exhibit control flow and shape dynamism as part of their execution.

### 2.1.1 Dynamic Control Flow in Deep Learning Computations

As discussed before, a computation is said to exhibit control flow dynamism when different inputs to the computation may follow different control flow paths during execution. This behaviour is most often achieved via the use of control flow constructs such as iteration, recursion, conditional statements, and so on[1]. Such computations also often rely on irregular data structures and perform manipulations on them. For example, as discussed below, a lot of recursive computations work on data structures such as trees and graphs, while the StackLSTM [33] model includes, as part of its execution, an entire shift-reduce parser and its corresponding stacks. Thus, there is a wide variety of dynamic control flow patterns found in deep learning computations. Below, we explore some control flow structures or patterns commonly encountered in deep learning computations[2]. This discussion is also summarized in Table 2.1.

#### Control Flow Surrounding Static Blocks

We observe that for most ML computations exhibiting control flow dynamism, the dynamic control flow *surrounds* tensor computations. Consider the simple sequential RNN model implemented by the

[1] Recall the RNN model computation we saw in Chapter 1, which relied on iterative control flow.

[2] Given such 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 maybe more efficient, the DFS-based traversal is more natural to implement.

`@rnn` function shown in Listing 2.1. Here, we see that the sequential control flow surrounds an RNN cell on lines 5 and 6, which is a sub-graph of tensor computations with no intervening control flow. We refer to such a block of tensor computations as a *static block*.

```
1  def @rnn(inputs, state, bias, iweight, hweight) {
2    match(inputs) {
3      Nil => Nil,
4      Cons(input, tail) => {
5        let input_linear = bias + nn.dense(input, iweight);
6        let new_state = sigmoid(input_linear + nn.dense(state, hweight));
7        Cons(new_state, @rnn(tail, new_state, bias, iweight, hweight))
8      }
9    }
10 }
11
12 def @main(rnn_bias: Tensor[(1, 256)], rnn_iweight: Tensor[(256, 256)],
13           rnn_hweight: Tensor[(256, 256)], rnn_init: Tensor[(1, 256)],
14           cweight: Tensor[(16, 512)], cbias: Tensor[(1, 16)],
15           inputs: List[Tensor[(1, 256)]]) {
16   let _ = db.set_phase(0);
17   let rnn_res =
18     @rnn(inputs, rnn_init, rnn_bias, rnn_iweight, rnn_hweight);
19   let _ = db.set_phase(1);
20   @map(fn(p: Tensor[(1, 256)]) {
21         nn.relu(cbias + nn.dense(p, cweight))
22       }, rnn_res)
23 }
```

Listing 2.1: A simple RNN model expressed in Relay [106] as an input to ACRoBat (discussed further in Chapter 4).

### Tensor-Dependent Control Flow

Control flow decisions often depend on the values of intermediate tensors in ML computations. Examples of models and computations exhibiting such tensor-dependent control flow include beam search in machine translation, StackLSTMs, Tree-to-Tree neural networks (T2TNN) [17], models with early exits [34, 62, 127, 144], Mixture-of-Experts [35, 77, 112] and other ML computations such as the No U-Turn Sampler (NUTS) [50]. Meanwhile, in models such as TreeLSTM [125], DAG-RNN [115], sequential RNNs and their variants, control flow only depends on the inputs and not on intermediate tensors. Such models are said to exhibit tensor-independent control flow.

### 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 [18] for example) and StackLSTMs, and recursive models such as TreeLSTM, Tree-to-Tree neural networks and DAG-RNNs. 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 [149] model, for example,

executes two RNNs, one nested inside the other. Similarly, the DRNN [6] 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 §4.2.3 how, in ACRoBat, exploiting parallelism across recursive calls at runtime, for example, can require multiple concurrent execution contexts, similar to the fork-join parallelism paradigm [78].

#### Control-Flow in Training and Inference

We see, in Table 2.1, 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 exists, 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).

#### Irregular Data Structures

Dynamic control flow often involves the use of irregular data structures. Models such as the TreeLSTM and MV-RNN, for example, involve traversals over parse trees of textual data. Similarly, DAG-RNN models an image as a directed acyclic graph (DAG) in order to perform image segmentation. Fast WaveNet [91], an efficient way to perform sequence generation using the WaveNet [133] model, involves the use of queues. Similarly, as mentioned above, StackLSTM and its variants involve parsing the input text via a shift-reduce parser, thereby necessitating the use of stacks and buffers during the model execution.

#### Control Flow Parallelism

Dynamic control flow can lead to parallelism in deep learning computations. 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 implementation of the RNN model in Listing 2.1.

In the context of this work, we identify the following three sources of parallelism in a dynamic deep learning computation.

1. *Batch Parallelism:* This is parallelism that exists across the different input instances in the mini-batch.
2. *Instance Parallelism:* This refers to the aforementioned control flow parallelism which exists within the execution of a single input instance.
3. *Static Parallelism:* This is parallelism that may exist across tensor operators within one static block. For example, in a Long Short Term Memory (LSTM) cell [49], one can compute the four gates concurrently.

Table 2.1: Control flow properties found in deep learning computations. Legend: ITE: iterative, REC: recursive, TDC: tensor dependent control flow, IFP: instance parallelism, ICF: inference exhibits control flow, TCF: training exhibits control flow, IDS: irregular data structures.

| Deep Learning Computations | ITE | REC | TDC | IFP | ICF | TCF | IDS |
|---|---|---|---|---|---|---|---|
| Sequential RNN [108], LSTM [49], GRU [18], GraphRNN [149] | ✓ | | | | ✓ | ✓ | |
| DIORA [32], Chinese Segmentation [16] | ✓ | | | ✓ | ✓ | ✓ | |
| DAG-RNN [115], LatticeRNN [68], TreeLSTM [125], MV-RNN [120] | | ✓ | | ✓ | ✓ | ✓ | ✓ |
| StackLSTM [33] | ✓ | | ✓ | | ✓ | ✓ | ✓ |
| Beam search [131] with LSTM | ✓ | | ✓ | ✓ | ✓ | | |
| Mixture-of-experts [35, 77, 112] | | | ✓ | | ✓ | ✓ | |
| Early exit models [34, 62, 127, 144] | | | ✓ | | ✓ | | |
| No U-Turn Sampler [50] | | ✓ | ✓ | | ✓ | ✓ | |
| Tree-to-tree NN [17], Doubly Recurrent NN [6] | | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ |
| R-CNN [45], Fast R-CNN [44] | ✓ | | ✓ | ✓ | ✓ | ✓ | |
| Fast WaveNet [44] | ✓ | | ✓ | | ✓ | | ✓ |

We note that this list of characteristics is not exhaustive. Moreover, the examples above demonstrate that the properties listed are often independent of each other and hence the same computation can exhibit multiple properties simultaneously. For example, the StackLSTM model is an iterative model which exhibits tensor-dependent control flow and low control flow parallelism.

### 2.1.2 Shape Dynamism in Deep Learning Computations

As we briefly mentioned in Chapter 1, a model is said to exhibit shape dynamism when the execution of the model involves tensors with shapes that differ across multiple data inputs. We also saw that the transformer-based models such as BERT [27], GPT [11, 100, 101] and so on, which form the base of the largest NLP models today, exhibit such shape dynamism. Graph neural networks [110], when designed to process multiple graphs, potentially with a varying number of nodes or edges, can also exhibit shape dynamism [38]. Similarly, image processing models [48] which are agnostic to the size of the input image can process multiple images of the different sizes together, leading to varying shapes of intermediate tensors.

We also saw that performing batched execution of models that exhibit shape dynamism requires that we rely on the abstraction of a ragged tensor, which is tensor the slices of one or more dimensions of which are of variable lengths. Ragged tensors are similar to sparse tensors. Unlike sparse tensors, however, data in a slice of a ragged tensor are tightly packed. Further, when a sparse tensor is stored using a compressed format, the tensor elements which are not explicitly stored are valid and are assumed to be equal to some value, most commonly zeros. On the other hand, the data absent in a ragged tensor are truly invalid and therefore do not correspond to any implicit value.

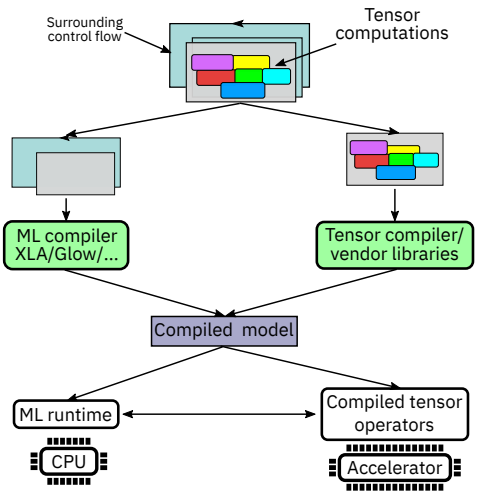



Figure 2.1: Illustration of a typical deep learning compilation workflow.

## 2.2 Deep Learning Compilation Workflow

In this section, we give a brief overview of how a deep learning compilation and execution workflow is usually designed, specifically in the context of dynamism. We saw that a deep learning computation can be thought of as a set of tensor operators, the execution of which is orchestrated by surrounding control flow. Most deep learning workflows are structured around these two aspects of the computation, though the line between them can often be blurry. Below, we describe each of these aspects in more detail. A simple workflow is also illustrated in Figure 2.1.

### 2.2.1 High-level Orchestration

This part of the deep learning workflow performs optimizations at the level of multiple tensor operators, and how they are invoked. It is here that optimizations such as kernel fusion [157] or tensor layout selection are performed. Auto-batching is also performed at this granularity of tensor operators.

Traditionally, at this stage deep learning computations are often represented as task graphs consisting of multiple tensor operators. We refer to such graphs as dataflow graphs (DFGs). Such a representation allows compiler developers to conveniently express optimizations as sub-graph rewrites [60]. Deep learning compilers such as XLA [126] and Glow [107] as well as inference engines such as TensorRT [89] and OpenVINO [55] which optimize model execution for inference, for instance, are primarily designed around the DFG abstraction. On the other hand, such a representation is not very amenable to expressing dynamic control flow. As a result, as dynamic control flow started being incorporated in an increasingly large number of models, the community has worked towards extending the DFG representations or towards developing other representations that are more amenable to the expression and optimization of control flow dynamism. For example, the DFG representation used by TensorFlow supports common control flow operators such as while loops and simple conditional statements [150]. TensorFlow also supports distributed execution for computations with such control

flow operators. Jeong et al. further propose extensions to the DFG representation to support recursive control flow in [58]. More recently, compiler representations such as Relay [106] (along with the improvements to the Relay stack proposed by Nimble [113]), TorchScript [97] which is used in PyTorch [92] and the MLIR [70] framework forego the restrictive DFG representation to instead rely on rich traditional representations allowing one to express, compile and optimize computations with arbitrary control flow.

Supporting dynamic control flow also leads to other optimization challenges. The use of dynamically typed languages such as Python further excacerbates this problem. Due to the use of DFGs as a traditional compiler representation, a lot of deep learning optimizations have been developed for this representation. This has lead to a large body of work that tries to extract, either statically or dynamically, DFGs given a computation expressed in a more expressive language/intermediate representation. For example, Janus [59], Terra [63] and TorchDynamo [23] solve the problem of enabling static graph optimizations in computations expressed using high-level languages such as Python. They speculatively extract and optimize traces of tensor operators in a manner similar to traditional just-in-time compilation. PyTorch's LazyTensor [124] also creates DFGs at runtime that can be optimized to accelerate dynamic computations.

### 2.2.2 Tensor Operators

This stage concerns with generating/developing efficient implementations of individual tensor operators. These implementations can be hand-written and provided as part of vendor libraries such as Nvidia's cuDNN for Nvidia GPUs and Intel's oneDNN for Intel CPUs and GPUs. Or they may be automatically generated by *tensor compilers*. A tensor compiler refers to a domain specific language and its accompanying compilation stack which allows users to express computations on tensors and generate efficient implementations for these computations for a variety of hardware targets. Common examples includes Halide [103], TVM [13], Tensor Comprehensions [135] and Tiramisu [7] for dense tensor computations and Taco [65] and Comet [81, 130] for sparse tensor computations. Tensor compilers separate the definition of the computation from its implementation, allowing a user to explore multiple implementations quickly to optimize the computation. The user optimizes the computation usually by specifying transformations over loop nests such as loop fusion, loop tiling, vectorization and so on. This is ilustrated for a simple tensor computation in Figure 2.2.

The ability of a tensor compiler to separate the definition and the implementation of the computation also allows the use of automated techniques [1, 14, 80, 117, 156] to search over various loop nest transformations in order to reduce or even eliminate the optimization effort on the part of the user. This technique is broadly referred to as *auto-scheduling*.

Work on tensor compilation as well as auto-scheduling initially considered only tensor operators operating on tensors with statically known shapes. This work has also now been started to be extended to support dynamic tensor shapes [136, 155, 158].

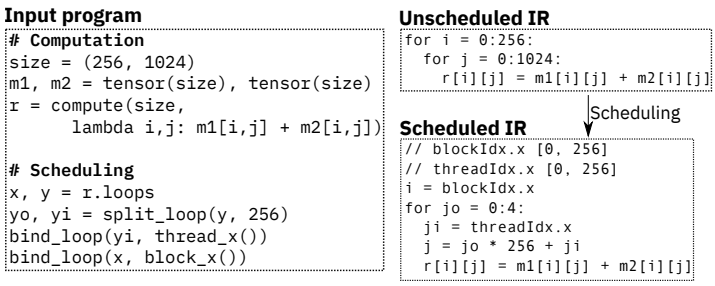



Figure 2.2: Using a tensor compiler to express and optimize elementwise matrix addition.

## 2.3 Past Work

In this section, we will look at the previous efforts towards enabling auto-batching first for control flow dynamism and then for shape dynamism.

### 2.3.1 Auto-Batching for Control Flow Dynamism

As we discussed before, there has been significant work towards enabling efficient auto-batching in the presence of control flow dynamism. We can divide this past work into two categories—approaches tailored to specific control flow patterns or models, and approaches that have been designed to handle a large class of computations with dynamic control flow. We look at both of these in turn below.

#### Specialized Approaches

RNNs are an important class of NLP models. As a result, multiple efforts have targeted efficient auto-batching for RNNs and their variants. These include approaches such as BatchMaker [42] and E-Batch [116]. Both of these are designed to enable fast RNN inference in production. They batch and schedule the RNN model at the granularity of one RNN cell and can refill the batch with queued requests when the execution of one request is finished, thereby enabling low inference latencies. Similar techniques have also been explored for batching beam search as described in [147].

StackLSTM-based models are often used in order to perform text parsing as a input step to enable other downstream tasks which operate on the parsed representations (often in the form of a tree, referred to as a *parse tree*). Approaches such as SPINN [9], Batched Recurrent Neural Network Grammars (RNNGs) [87] and Parallelizable StackLSTM [29] have explored how this class of model computations can be batched to increase efficiency. Given the narrow focus of these approaches, they are often able to develop techniques tailored to the computation under consideration. For example, the SPINN approach proposes a single model (which is amenable to efficient batched execution) in order to perform both text parsing as well as the downstream task of text encoding.

Beyond specific models, approaches such as Cavs [146] have attempted to enable efficient auto-batching computations that can be expressed as a graph traversal with a fixed per-node computation. Computations that can be expressed this way include RNNs and their variants, TreeRNNs and their variants as well as other models outside natural language processing such as DAG-RNN, which has

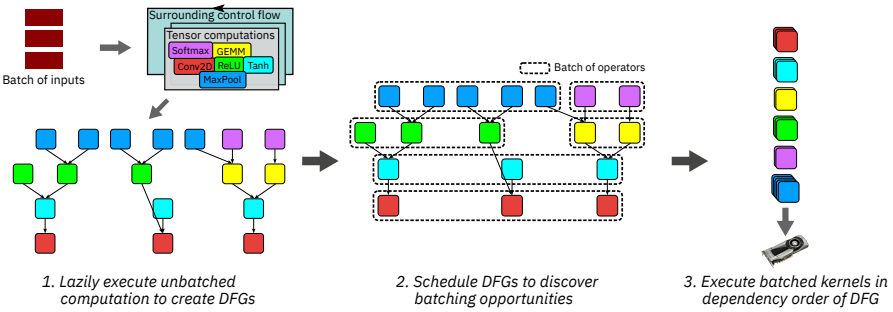



Figure 2.3: Dynamic batching: a fully dynamic auto-batching technique for control flow dynamism.

been proposed for image segmentation. This approach therefore separates out the dynamic parts of the computation (as embodied in the structure of the input graph) from the static parts (as embodied in the fixed per-node function). The static parts can therefore be compiled and optimized statically while batching can be performed at the granularity of the graph nodes during runtime.

### Approaches with a Broader Scope

Past work on auto-batching that aims to handle a broad class of computations with control flow can range from fully static approaches to fully dynamic ones.

Static approaches include MatchBox [10], Jax's `vmap` and `pmap` program transformations [40], TensorFlow's `pfor` primitive [2, 3] as well as the local static auto-batching technique described in [102]. These propose program transformations to generate a batched implementation of a computation given an unbatched implementation of the same. Transformations such as these are generally inspired by the single-program multiple-data (SPMD) programming model used by Intel ISPC [93] and (under the name SIMT) NVIDIA CUDA [86]. Due to the static nature of these approaches, they incur little to no runtime overheads. On the other hand, they are often unable to exploit all the parallelism present in the computation, as Radul et al. describe [102]. Further, such approaches can also be limited by the kinds of control flow they can support. For example, TensorFlow's `pfor` primitive only supports the control flow operators allowed in TensorFlow's dataflow graph format, thereby disallowing recursion, for instance. Similar restrictions are also present in Jax [72] and MatchBox.

On the other end of the spectrum, dynamic batching [75, 85] has been proposed as an entirely dynamic approach to auto-batching. It involves lazily executing the model computation while building a DFG of tensor operations in the background. Given a mini-batch of input instances, such DFGs can be generated for each of the input instance in the mini-batch. The execution of these graphs is triggered when the value of a particular tensor is requested (at the end of one training loop iteration, for example, or when the model contains tensor-dependent control flow). When the execution is triggered this way, the runtime can identify batching opportunities within the DFGs (one corresponding to each input instance in the mini-batch) and then launch kernels appropriately. This process is illustrated in Figure 2.3. Due to the time spent in generating multiple dataflow graphs and scheduling them, dynamic

batching often incurs high execution overheads. On the other hand, due to the availabilty of perfect execution knowledge, it allows one to exploit all the available parallelism in the computation.

Further improvements and variations on this basic approach have been proposed. As described above, dynamic batching creates and schedules the DFGs at the granularity of individual tensor operators. Just in time batching [151] explores the trade-offs involved in coarsening the granularity of the DFGs to static blocks of various sizes, and proposes an approach to choose the correct granularity given an input computation. In [99], Qiao and Taura propose that scheduling and batching be carried out independently for the backward pass of a computation during training, as opposed to using the schedules generated during the forward pass. They notice that in a lot of the computations with control flow dynamism, the backward pass exhibits more parallelism as compared to the forward pass, which remains unexploited unless scheduling is performed again during the backward pass to discover it. In [102], Radul et al. propose program counter auto-batching, which can be thought of as an eager variation of dynamic batching. This allows one to skip the graph construction and scheduling steps, thereby reducing execution overheads. However, due to the inability of the algorithm to have visibility into all the tensor operators executed by the computation due to its eager nature, it is unable to exploit control flow parallelism within the execution of a single data input.

#### Use of Vendor Libraries

A lot of the approaches described above, both specialized or otherwise, heavily rely on vendor libraries in order to effectively target the wide range of deep learning hardware. 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. Further, as we alluded to in Chapter 1, the use of vendor libraries precludes the optimization of tensor kernels in the context of the larger surrounding computation.

### 2.3.2 Batching for Shape Dynamism

Enabling batched execution for models exhibiting shape dynamism requires the ability to efficiently perform operations on ragged tensors, as discussed in §2.1.2. Below we discuss some of the ways one can use past work for this purpose.

#### Reusing Dense or Sparse Infrastructure

Kernels for both dense as well as sparse tensor computations can be used to execute ragged tensor operators. Ragged tensors can be appropriately padded to make them dense, as is shown in Figure 2.4. Then, one can use kernels for dense tensor computations, either those provided by vendor libraries or those generated by tensor compilers, for executing ragged operators. Further, one can also model ragged tensors as sparse tensors and use the corresponding sparse tensor infrastructure. As we will see

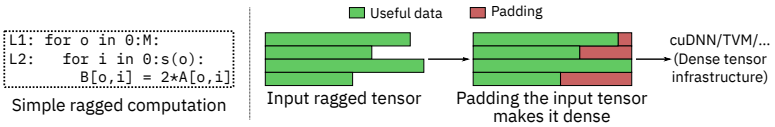



Figure 2.4: Padding ragged tensors to enable use of dense tensor infrastructure.

in Chapter 5, however, both of these approaches can have non-trivially large performance penalties. Computation on padded data in the former approach is wasted and can be significant. On the other hand, while ragged tensors can be modeled as sparse tensors, they are much denser and have densely packed data, unlike sparse tensors. This impedance mismatch can also preclude the most optimal execution for ragged tensor computations.

Hand-optimized Implementations of Ragged Operators

Due to the sub-optimal performance obtained when using dense or sparse tensor infrastructure for ragged tensor operations, there have also been some efforts in developing hand-optimized implementations of commonly used operators and models for ragged tensors. For example, both Intel's MKL and Nvidia's CUTLASS libraries provide implementations of batched GEMM operators where each GEMM has variable dimensions (we refer to this batched GEMM operation as the *vgemm* operation). Similarly, Li et al. propose techniques [74] to more efficiently perform batching and tiling for such batched GEMM operations. Efficient implementations of the vgemm operator also are part of the MAGMA [83] and the MKL [54] libraries. Beyond individual operators, ByteDance's EffectiveTransformer [12] provides an implementation of the transformer model without padding. These optimizations have also been incorporated into Nvidia's FasterTransformer [88].

We therefore see that past work often fails to exploit optimization opportunities across multiple parts of the compilation pipeline as well as between the compiler and the runtime. In the next three chapters, we describe how one can overcome such fragmentation, to enable performant batched execution of dynamic deep learning computations.

# 3 Cortex: Compiler-Based Auto-Batching for Recursive Deep Learning Models

We saw in Chapter 2 that dynamic and irregular control flow is an important source of dynamism in deep learning computations. Before tackling the problem of efficient auto-batching for unrestricted control flow patterns, examples of which we saw in Chapter 2, we first only consider recursive tensor-independent control flow in this chapter. This chapter presents the design and evaluation of Cortex (**Co**mpiler for **R**ecursive **T**ensor **Ex**ecution), a compiler we designed to solve auto-batching for this specialized case[1]. While this is restrictive, RNNs and their variants, which are an important sequence model, can be easily represented as recursive computations with no tensor-dependent control flow and hence can be handled using the techniques presented in this chapter along with other models we will look at. Later, in chapter Chapter 4, we will build upon the insights obtained by designing Cortex, to build ACRoBat, a compiler-based framework for the broader case of general control flow.

As we discussed in Chapter 2, past approaches to auto-batching suffer from (one or more of) the following disadvantages:

1. Extensive use of vendor libraries leading to tensor kernels being optimized in isolation. This further leads to the inability to perform optimizations such as kernel fusion and model persistence. The latter can lead to significantly degraded performance as we see later in this chapter. Further, the high development cost associated with vendor libraries leads to low model coverage.
2. Purely static techniques leading to conservative program transformations and low exploited parallelism.
3. Fully dynamic analyses leading to high execution overheads.

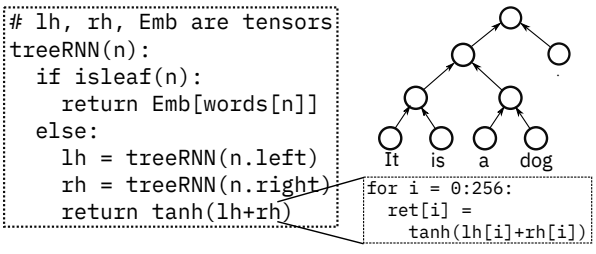



Figure 3.1: A simple recursive model. The text 'It is a dog.' is parsed into the parse tree which is then fed to the model.

[1] A paper [36] describing Cortex was published at MLSys 2022.

Table 3.1: Comparison between Cortex and related work on recursive models (Cavs, DyNet, Nimble and PyTorch).

| Framework | Kernel Fusion | Vendor Libraries | Auto-Batching | Model Persistence |
|---|---|---|---|---|
| Cavs | Partial | Y | Y | N |
| DyNet | N | Y | Y | N |
| Nimble | Partial | N | N | N |
| PyTorch | N | Y | N | N |
| Cortex | Y | N | Y | Y |

In order to overcome these disadvantages, Cortex takes a fully compiler-based approach, which enables us to perform compile-time optimizations such kernel fusion and model persistence by taking the surrouding computation into account. 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:** Figure 3.1 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:** Low latency inference for recursive models necessitates 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 which unrolls all recursion and makes optimizations such as *auto-batching* easier [75, 84]. Such optimizations have to be performed *statically* in a compiler-based approach.

To overcome **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 §3.1, enables us to lower the recursive computation into an efficient loop-based one (illustrated in Figure 3.2). To overcome **C.2** and **C.3**, we employ scheduling primitives to perform optimizations such as *specialization* and *auto-batching* [75, 85], 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 (§3.6.3) while avoiding some overheads associated with auto-batching (§3.6.2). As part of Cortex's design, we extend the TVM tensor compiler [13], but the techniques can be used with other tensor compilers [7, 13, 103]. 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 [1, 14, 80, 156] for optimizing these models. Table 3.1 provides a qualitative comparison of Cortex with related work on recursive models.

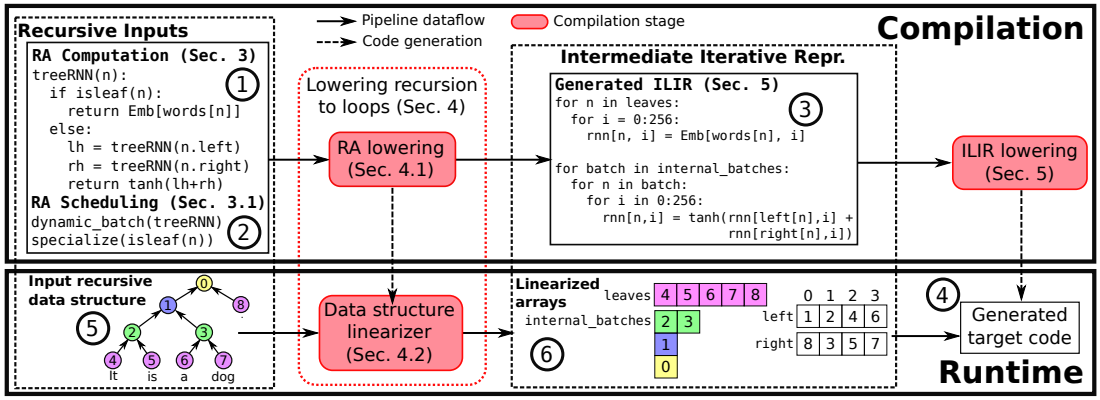



Figure 3.2: Overview of the Cortex compilation and runtime pipeline.

## 3.1 Overview

Efficiently executing computations with dynamic control flow is challenging because the control flow often precludes common optimizations such as kernel fusion. We note that the computations Cortex is designed for, however, have the following properties which enable efficient batched execution:

**P.1** These computations often traverse an input recursive data structure, and all control flow depends on the connectivity of this data structure. The latter essentially means the computation does not involve tensor-dependent control flow.

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

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

Property **P.1** implies that all control flow in the model is encapsulated in the input data structure. Property **P.2** entails that computation starts 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 Figure 3.2) that make use of these insights. Compilation starts with the recursive model computation ① expressed in the Recursive API (RA). The user can also specify some scheduling primitives ② at this stage to control how the recursive computation is lowered. The compiler then generates Irregular Loop IR (ILIR) ③ corresponding to the input computation, according to the scheduling primitives provided by the user. The ILIR is an extension of the IR used by tensor compilers, designed to support additional features such as indirect memory accesses and variable loop bounds. It is purely loop-based and data structure agnostic. The RA lowering phase thus lowers all recursive control flow into loops and all data structure accesses to potentially indirect memory accesses at this stage. Loop optimizations such unrolling, tiling, etc., as performed in tensor compilers, can be performed here, after which target-specific code ④ is generated as part of ILIR lowering.

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

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

## 3.2 Recursive API (RA)

```
##################### Model computation ####################
# H: Hidden and embedding size
# V: Vocabulary size
# N: Total number of nodes in the input data structure(s)
Tensor Emb = input_tensor((V, H))
Tensor words = input_tensor((N,))

# A placeholder that represents results of recursive calls
Tensor rnn_ph = placeholder((N, H))
# Base case definition
Tensor leaf_case =
  compute((N, H), lambda n, i: Emb[words[n], i])
# Recursive body definition
Tensor lh = compute((N, H), lambda n, i: rnn_ph[n.left, i])
Tensor rh = compute((N, H), lambda n, i: rnn_ph[n.right, i])
Tensor recursive_case =
  compute((N, H), lambda n, i: tanh(lh[n, i] + rh[n, i]))
# Conditional check for the base case
Tensor body = if_then_else((N, H), lambda n, i: isleaf(n),
                           leaf_case, recursive_case)
# Finally, create the recursion
Tensor rnn = recursion_op(rnn_ph, body)

################ RA scheduling primitives ################
auto_batch(rnn)
specialize_if_else(body)
```

Listing 3.1: Model in Figure 3.1 as expressed in the 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 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 3.1 which shows the simplified model from Figure 3.1 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

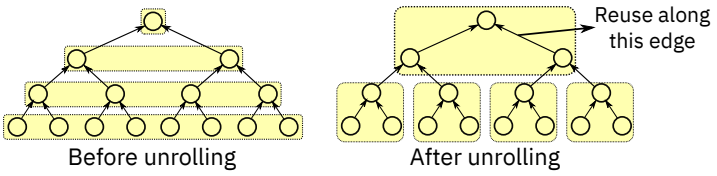



Figure 3.3: Change in execution schedule due to unrolling.

structure (sequence, tree or DAG). This information is used during compilation, and can be easily verified at runtime.

### 3.2.1 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 3.1. We propose the following scheduling primitives to exploit these opportunities:

#### Auto-Batching

Since control flow in the models we look at depends only on the input data structure (property **P.1**), we perform auto-batching during linearization. The execution order of nodes of a tree with auto-batching is illustrated top-to-bottom in ⑥.

#### Specialization

Recursive computations tend to have frequent conditional checks to check for the base condition. Such checks can hinder optimizations such as computation hoisting and constant propagation (§3.3.3), while having execution overheads of their own. We, therefore, allow the user to generate specialized versions of the program for the two branches of a conditional check. Listing 3.2 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 user specified that the leaf check be specialized (line 26 in Listing 3.1).

#### Unrolling

Unrolling recursion changes the order in which nodes are processed (as illustrated in Figure 3.3), 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 Figure 3.3 (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.

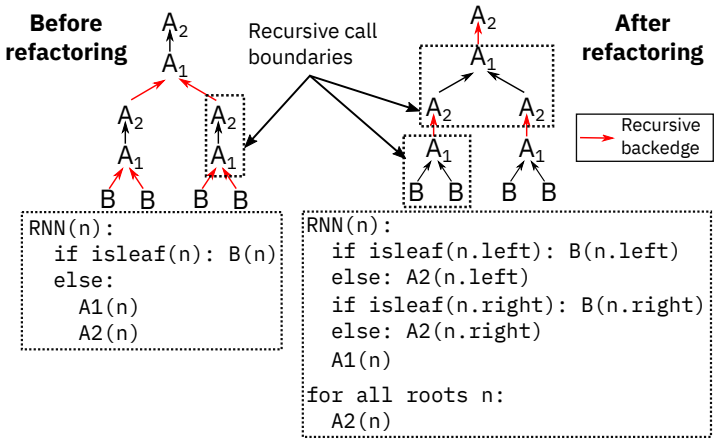



Figure 3.4: Recursive refactoring can be used to change the position of the recursion backedge with respect to the computation.

### Recursive Refactoring

Kernel fusion is harder to perform across recursive call boundaries. In such cases, recursive refactoring can be used to change the recursive backedge. Consider the computation on the left in Figure 3.4. `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.

## 3.3 Lowering Recursion to Loops

### 3.3.1 RA Lowering

The lowering from RA to ILIR is, in essence, a lowering from recursion to iteration. Just as we need to make the stack explicit in such a lowering in general purpose programs, we need to make explicit all the temporary tensors when lowering to ILIR. Note how, in the ILIR for our running example in Listing 3.2, the tensors `lh` and `rh` are explicitly created. We also materialize the tensor `rnn`, which stores the result of the computation. Each of the tensors `lh`, `rh` and `rnn` store data for each recursive call, which in this case amounts to each tree node.

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

```
for n_idx = 0:leaf_batch_size:
  node = leaf_batch[n_idx]
  for i = 0:256:
    rnn[node, i] = Emb[words[node], i]

for b_idx = 0:num_internal_batches:
  for n_idx = 0:batch_sizes[b_idx]:
    node = internal_batches[b_idx, n_idx]
    for i = 0:256:
      lh[node, i] = rnn[left[node], i]
    for i = 0:256:
      rh[node, i] = rnn[right[node], i]
    for i = 0:256:
      rnn[node, i] = tanh(lh[node, i] + rh[node, i])
```

Listing 3.2: ILIR generated for the model in Figure 3.1.

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 requires auto-batching to be performed, the ILIR iterates over batches of nodes (as in Listing 3.2).

### 3.3.2 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.

```
leaf_batch = []
internal_batches = [[]]
left, right = [], []

def linearizer(n):
  if isleaf(n):
    leaf_batch.add(node)
  else:
    linearizer(n.left)
    linearizer(n.right)
    left[n], right[n] = n.left, n.right
    internal_batches[node.height].add(node)

leaf_batch_size = len(leaf_batch)
batch_sizes = [len(b) for b in internal_batches]
num_internal_batches = len(internal_batches)
```

The data structure linearizer is generated during RA lowering. In the absence of specialization and auto-batching, the linearizer essentially has to traverse the data structure as the input program does, while keeping track of the order of nodes encountered. This ordering over the nodes would satisfy data dependences and can be used during the tensor computations. Thus, in this simple case, the data

structure linearizer is essentially the input program, stripped of all tensor computation. For conditional checks marked for specialization, the linearizer will separately collect nodes that follow each of the two branches of the check. For auto-batching, we emit code to traverse the data structure and identify batches of nodes that can be processed in parallel.

This linearization step, the code for which is generated by the compiler, therefore, is key in enabling Cortex to lower the input recursive control flow into efficient iterative code. Thus, it can be seen as an instantiation of the hybrid static and dynamic analysis we discussed in Chapter 1.

### 3.3.3 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.

## 3.4 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 input or previously-produced 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 3.2, 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 to support (1) non-affine index expressions, (2) loops with variable bounds, and (3) conditional operators. We describe these modifications in further detail below.

### 3.4.1 Indirect Memory Accesses

We represent non-affine index expressions arising as part of indirect memory accesses as uninterpreted functions of loop variables [123][2]. Indirect memory accesses necessitate further changes, which are described next.

[2]An uninterpreted function symbolically represents a function from its parameters to an output. The opaque nature of the abstraction and the lack of any implicit assumptions makes it convenient to explicitly specify relevant properties for different uninterpreted functions during compilation. We also use this abstraction later in Chapter 5 to represent ragged loop nests and transformations on the same.

### Bounds Inference

As a pass during code generation, a tensor compiler infers loop bounds for all operators in the input program usually proceeding from the outputs of the operator graph towards the inputs. For each operator *op* producing a tensor $t$, the pass 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 translation is straightforward as there is a one-to-one correspondence between the loops of an operator and the corresponding tensor dimensions. This does not, however, hold in our case, as is apparent in the ILIR in Listing 3.2. Tensors `lh`, `rh` and `rnn` have two dimensions each, but the generated ILIR has three loops for each of their corresponding operators. As a result, we require that the ILIR explicitly specify the relationship between tensor dimensions and the loops in the corresponding operator's loop nest. This is achieved 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 3.3 which shows the same ILIR as in Listing 3.2 but with the named dimensions annotated as comments[3]. 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`.

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

Listing 3.3: ILIR generated for the model in Figure 3.1.

### Tensor Data Layouts

Data layouts of intermediate tensors often need to be changed to allow for an efficient use of the memory subsystem. This can involve ensuring non-conflicting accesses to the GPU shared memory, or

[3]We do not cover the case of optimizations such as loop splitting which give rise to additional loops here for brevity. Similarly, operators involving reduction are not covered here.

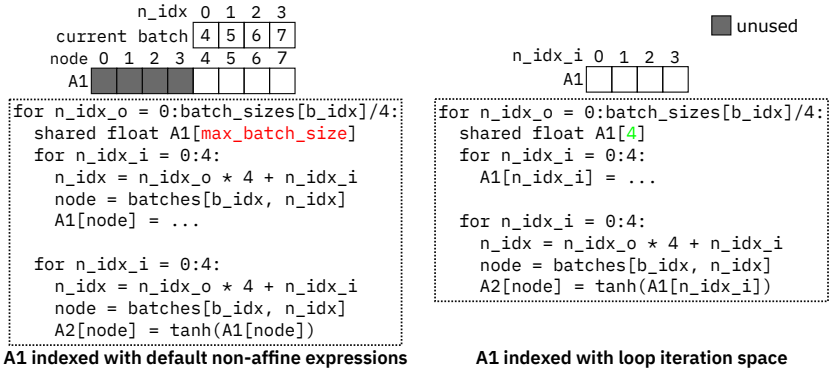



Figure 3.5: Dense indexing for intermediate tensors.

sequential accesses to memory on a CPU and so on. To facilitate 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 Figure 3.5, 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 Figure 3.5 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 Figure 3.5. 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.

### 3.4.2 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 an input and lowers down 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 Appendix A.

## 3.5 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 auto-batching and specialization, for the common case of leaf checks.

#### ILIR Lowering

We extend TVM [13] 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. When lowering the data structures to arrays, we number the nodes such that nodes in a batch (for auto-batching) are numbered consecutively and higher than their parents. We also ensure that all leaf nodes are numbered higher than all internal nodes. This scheme generally reduces the costs of leaf checks and iterating over batches.

## 3.6 Evaluation

We now evaluate Cortex against Cavs, DyNet and PyTorch. Cavs is an open source, state-of-the-art framework for recursive neural networks, while DyNet implements the dynamic batching technique we discussed in Chapter 2. Both have been shown to be faster than generic frameworks like PyTorch and TensorFlow [85, 146]. Evaluation with PyTorch is included for reference as it is more popular than Cavs and DyNet. We evaluate these systems on Intel and ARM CPUs and on Nvidia GPUs.

### 3.6.1 Experimental Setup

#### Models and Schedules

We primarily use the models and datasets listed in Table 3.2. The TreeGRU model is similar to the TreeLSTM model, except that it uses the GRU RNN cell. The TreeLSTM and TreeGRU models were scheduled similar to the sequential LSTM and GRU schedules proposed in GRNN [51]. In the Cortex and PyTorch implementations for TreeLSTM, TreeGRU and DAGRNN, 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 lead to 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 model sizes—small (which entails a hidden size of 256 for TreeFC, DAGRNN, TreeGRU and TreeLSTM and a hidden size of 64 for MVRNN) and large (which entails a hidden size of 512 for TreeFC, DAGRNN, TreeGRU and TreeLSTM and a hidden size of 128 for MVRNN).

Table 3.2: Models and datasets used for evaluating Cortex.

| Model | Short name | Dataset used |
|---|---|---|
| Benchmarking model used in [75] | TreeFC | Perfect binary trees (height 7) |
| Recursive portion of DAG-RNN [115] | DAGRNN | Synthetic DAGs (size 10x10) |
| Child-sum TreeGRU | TreeGRU | Stanford sentiment treebank [121] |
| Child-sum TreeLSTM [125] | TreeLSTM | Stanford sentiment treebank |
| MV-RNN [120] | MVRNN | Stanford sentiment treebank |

Table 3.3: Experimental environments used for evaluating Cortex.

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

[1] All cloud instances ran Ubuntu 18.04.

#### Experimental Environment

We use the three environments listed in Table 3.3 for our evaluation. We use cuBLAS, Intel MKL and OpenBLAS for the BLAS needs of Cortex as well as related work on the GPU, Intel and ARM backends respectively. DyNet also relies on the Eigen library. We compare against PyTorch 1.6.0, DyNet's commit `32c71acd` (Aug. 2020) and Cavs' commit `35bcc031` (Sept. 2020).

### 3.6.2 Overall Performance

We compare Cortex's performance with that of PyTorch and DyNet for the five models in Table 3.2 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 TreeGRU and TreeFC models.

We first look at PyTorch. As PyTorch does not perform auto-batching, its performance is quite poor. Speedups over PyTorch implementations for the GPU and Intel backends and for the small size are shown in Figure 3.6. Due to PyTorch's inability to perform batching, it cannot exploit parallelism across data structures nodes. As a result, it performs worse for larger batch sizes. It also performs worse for the TreeFC and DAGRNN models for the same reason. Their input data structures have a higher degree of available parallelism as compared to the remaining three models. Cortex performs better on

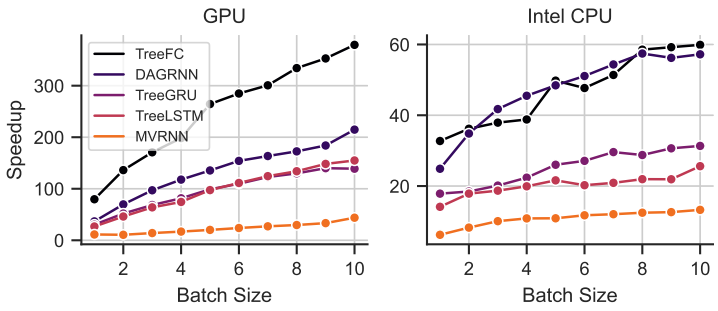



Figure 3.6: Cortex's Speedup over PyTorch for the small model sizes.

Table 3.4: DyNet vs. Cortex: Inference latencies (DyNet/Cortex) in $ms$ and speedups across different backends.

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

the GPU backend because it can effectively utilize the higher available parallelism on the GPU due to auto-batching and the scratchpad memories due to aggressive kernel fusion.

We now compare the inference latencies of Cortex with Cavs and DyNet, shown in Tables 3.5 and 3.4, respectively. Cortex latencies are much lower (up to 14X improvement) 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 Figure 3.7, which plots inference latency as a function of hidden size for the TreeLSTM model[4] 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 3.6 lists some runtime components of DyNet, Cavs, and Cortex, and the time they spend in each, for the same model configuration as above on the GPU backend. Both DyNet and Cavs implement generalized batching algorithms, which cause overheads in auto-batching and graph construction. At runtime, DyNet constructs a dataflow graph of tensor operators and performs dynamic batching on

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

Table 3.5: Cavs vs. Cortex: Inference latencies (Cavs/Cortex) in $ms$ and speedups on GPU

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

Table 3.6: Time spent ($ms$) in various activities[1] for DyNet, Cavs, and Cortex for TreeLSTM on the GPU backend for batch size 10 and hidden size 256.

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

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

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 adopts the 'think-like-a-vertex' approach which also has non-trivial overheads as compared to Cortex, which is specialized for recursive data structures. Cortex's auto-batching overheads are limited to linearization, before any tensor computations are executed.

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 [146], both on the CPU and the GPU ('Mem. mgmt. time' in Table 3.6). As Cortex manages the entire compilation process, it is free from such contiguity restrictions.

Cortex performs aggressive kernel fusion (illustrated in Figure 3.8), which has the dual effect of generating faster GPU code (seen in the 'GPU computation time' column in Table 3.6) as well as lowering CUDA kernel call overheads. As seen in Table 3.6, both DyNet and Cavs execute a high

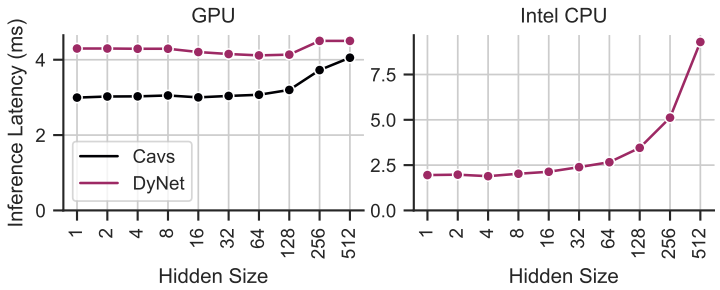



Figure 3.7: Inference latency vs. hidden size for the recursive portion of TreeLSTM for batch size 10.

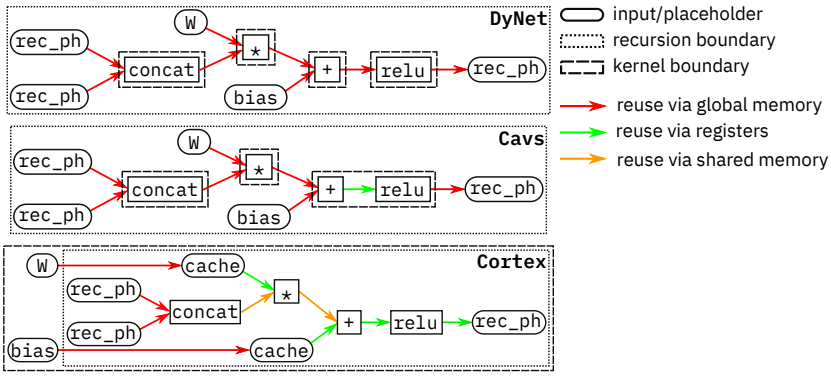



Figure 3.8: Kernel fusion and model persistence in Cortex: Cortex is able to exploit fast on-chip memory (registers and shared memory) better than DyNet and Cavs. This reduces accesses to the slow off-chip global memory. Note also how Cortex persists the model parameters (`W` and `bias`) and reuses the cached versions every iteration.

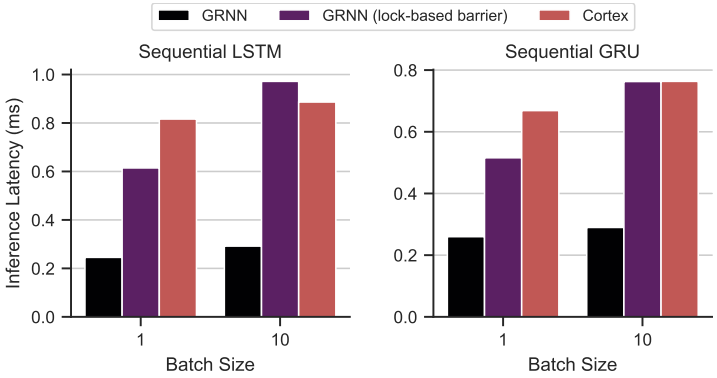



Figure 3.9: Cortex vs. hand-optimized GRNN code for sequence length 100 and hidden and input sizes 256.

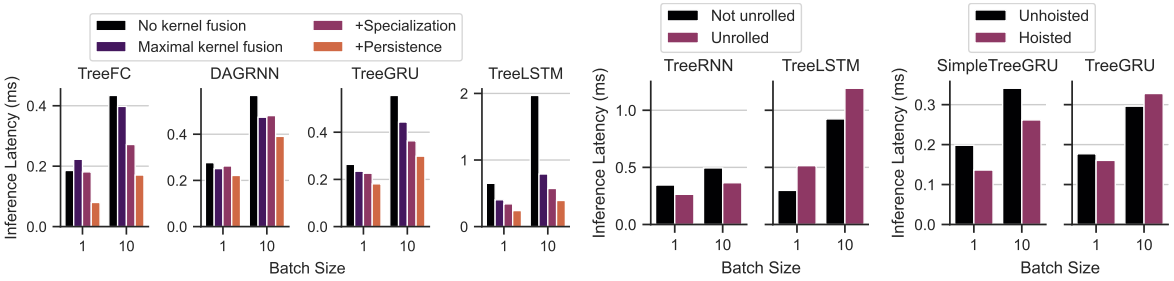
number of kernel calls, which, when taken together cause non-trivial overheads as CUDA kernels calls are expensive [76, 152]. The high number of kernel and memory copy calls also contributes to a high amount of CPU time spent in the CUDA API as seen in the column 'CPU CUDA API time'.

To our knowledge, there are no hand-optimized recursive model implementations available. Therefore, we compare Cortex with GRNN's hand-optimized GPU implementations of the sequential LSTM and GRU models. These implementations use a lock-free CUDA global barrier implementation [143], which is faster than the lock-based one [143] used by Cortex. For a fair comparison, we also compare against a version of the GRNN implementations which use the lock-based implementation. We find that Cortex-generated code performs competitively as compared to these hand-optimized implementations (Figure 3.9). Notably, Cortex can generalize these optimizations for recursive models.

### 3.6.3 Benefits of Optimizations

In this section, we look at different optimizations and their relative benefits in Cortex. Figure 3.10a shows the inference latency for different models (on the GPU for hidden size 256) as we progressively perform optimizations. Kernel fusion provides significant benefits for all models. The benefits are

pronounced on GPUs as GPUs have manually managed caches, which kernels optimized in isolation cannot exploit. Complex models such as TreeLSTM that provide more fusion opportunities benefit more. Specialization enables computation hoisting and constant propagation (3.3.3), which can dramatically reduce the amount of computation in tree-based models as trees have a larger proportion of leaves. For DAGRNN, which performs computations on DAGs, specialization does not lead to any speedup as expected. Finally, model persistence leads to non-negligible improvements in the inference latencies by reducing accesses to the GPU global memory. We discuss some optimization trade-offs involving register pressure in §A.5 in the appendix.




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

Figure 3.10: Benefits of different optimizations on the GPU backend for hidden size 256.

### 3.6.4 Other Scheduling Primitives

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

#### Unrolling

We evaluate unrolling on the TreeLSTM model for 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 Figure 3.11. This leads to higher inference latencies (Figure3.10b) despite the increased data reuse and kernel fusion (§3.2.1). We then evaluate unrolling on the simpler TreeRNN model, which is an extension of sequential RNNs for trees. When scheduling this model implementation, we perform the computation for one node in one GPU thread block, thus avoiding additional global barriers when unrolled. Therefore, unrolling leads to a drop in the inference latency for this model.

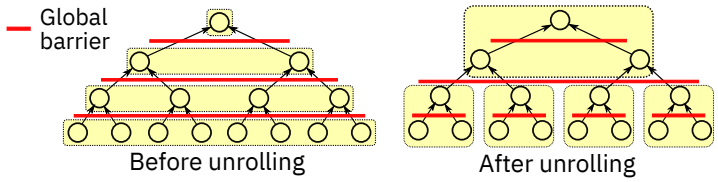



Figure 3.11: Unrolling TreeLSTM leads to additional barriers.

Table 3.7: Linearization overheads (in $\mu s$) in Cortex.

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

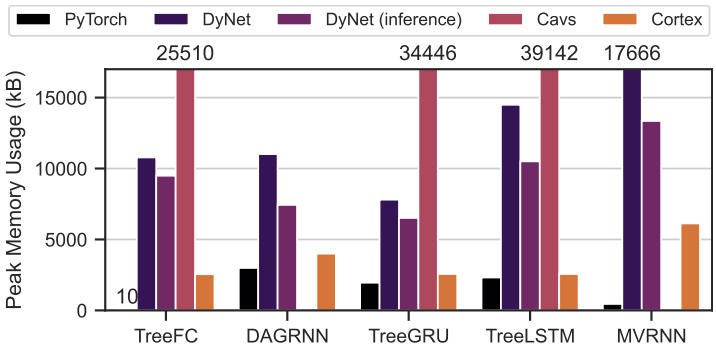



Figure 3.12: Peak GPU memory consumption in kilobytes, for batch size 10 and small model size.

Recursive Refactoring

We evaluate recursive refactoring on the TreeGRU model. In this case, refactoring allows us to reduce the number of global barriers as in the GRNN GRU implementation [51]. However, we find that in the case of TreeGRU, this does not give us significant speedups (Figure 3.10c). To explore further, we simplify the TreeGRU model (referred to as SimpleTreeGRU[5]) 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.

### 3.6.5 Data Structure Linearization Overheads

The data structure linearizer (§3.3.2) lowers input data structures to arrays and performs auto-batching if necessary, on the host CPU. Table 3.7 lists linearization times for different models (TreeLSTM, TreeGRU, and MVRNN are lumped together because they use the same input dataset). We find that on the GPU backend for batch size 10 and the small model size, linearization overheads, as a percentage of total runtime, range from 1.2% (for MVRNN) to 24.4% (for DAGRNN). 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.

### 3.6.6 Memory Usage

We now compare the memory consumption of Cortex with PyTorch, DyNet and Cavs. The peak GPU memory consumption for different models for batch size 10 and the small model size is shown in Figure 3.12. PyTorch uses the least amount of memory as it does not perform auto-batching. DyNet

[5]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 $h = (1 - z) * h'$.

and Cavs are designed for both deep learning training and inference. As gradient computations during training require the values of intermediate operations computed during the forward pass, DyNet and Cavs do not free the memory used by these intermediate tensors. Therefore, their memory consumption is quite high as compared to Cortex, which is designed for inference. We also compare against a version of DyNet (shown as 'DyNet (inference)' in Figure 3.12) modified to simulate the deallocation of a tensor when it is no longer needed in the forward inference pass. Despite this deallocation, however, DyNet's memory consumption is higher than Cortex's. Cortex materializes fewer intermediate tensors to the GPU's global memory due to kernel fusion (Figure 3.8). This reduces its memory consumption. Further, DyNet requires extra scratch space to ensure contiguous inputs to vendor library calls as discussed previously.

## 3.7 Related Work

### Compilers for Machine Learning

Tensor compilers (which we discussed in Chapter 2) such as TVM, Halide and Taco 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 (§3.4.1) is similar to the concept of workspaces for Taco introduced in [66]. On the other hand, as Cortex's aim is representing and compiling recursive deep learning models, while Taco mainly focuses on generating sparse kernels, there are significant differences as well. For example, we propose techniques, optimizations and scheduling primitives such as linearization, specialization and computation hoisting beneficial for recursive models. More generally, Cortex extends the abstractions provided by tensor compilers to support recursive computations and develops specialized optimizations for the same.

We saw in Chapter 2 that deep learning compilers such as XLA [126] and Glow [107] optimize static feed forward models and can perform partial kernel fusion and code generation. Further, in [102], the authors develop techniques to efficiently lower recursion into iterative control flow while performing auto-batching for the XLA toolchain. Inference engines such as TensorRT [89] and OpenVINO [55] optimize model execution for inference. The techniques we develop in this chapter could be used as a low-level backend for these deep learning compilers and optimizers. MLIR [70] provides infrastructure to build deep learning compilers and Cortex could potentially be built using MLIR.

### Optimizing Dynamic Neural Networks

Cortex is inspired by runtime approaches to auto-batching such as DyNet, TensorFlow Fold and more specialized approaches such as Cavs. Unlike these, Cortex performs auto-batching before any tensor computations. Beyond auto-batching, there is a large body of work aimed at optimizing recursive and more generally, dynamic neural networks. This includes, for instance, the approaches we outlined in

§2.2.1 of Chapter 2. The lazy or speculative approaches we discussed to handle control flow dynamism, can incur high execution overheads, similar to dynamic batching.

Model persistence was first proposed by Persistent RNNs [28], subsequently used in GRNN [51] and adapted for CPUs in DeepCPU [153]. In all of these works, it has been applied to the specific case of sequential RNNs (or their variants such as sequential LSTMs or GRUs), often with the use of hand-optimized or vendor library kernels. On the other hand, Cortex extends the optimizations proposed in these works to recursive models[6] and formalize them as transformation primitives in an end-to-end compilation workflow.

In general, Cortex provides a lower level of programming abstraction as compared to the deep learning frameworks we have discussed previously. Given this, we believe that Cortex could be potentially used as a backend for these frameworks, which would alleviate the disadvantages of using vendor libraries discussed in Chapter 1 and Chapter 2. For instance, deep learning on general graphs, as performed by graph neural networks, involves computations similar to the ones Cortex handles. Therefore, the ILIR infrastructure could also be used to express and optimize graph deep learning, potentially as a part of existing frameworks such as DGL [138].

Sparse Polyhedral Framework

The Sparse Polyhedral Framework (SPF) [79, 82, 123] extends the polyhedral model for the case of sparse tensor computations. Cortex borrows techniques such as the use of uninterpreted functions to represent indirect memory acceses from this body of work. The data structure linearizer in Cortex is an instance of the more general inspector-executor technique [4]. Using this technique to lower data structures has also been proposed in the past [134].

## 3.8 Chapter Summary

This chapter presented Cortex, a compiler for optimizing recursive deep learning computations with tensor-independent control flow for fast inference. For this specialized, but important class of computations, Cortex's approach eschews vendor libraries, thus allowing aggressive kernel fusion and end-to-end optimizations from the recursive control flow down to the tensor algebra computations, thus enabling highly efficient auto-batching. This allows Cortex to achieve up to 14× lower inference latencies. Cortex demonstrates that we can broaden the scope of deep learning computations that can be expressed and optimized using tensor compiler techniques. In the next chapter, we will look at how we can employ insights learnt from Cortex in designing an auto-batching framework for a much broader class of computations with dynamic control flow.

[6] As mentioned in §3.6, we implement the TreeLSTM and TreeGRU models in Cortex similar to the GRNN implementations.

# 4 ACRoBat: Auto-Batching in the Presence of General Control Flow Dynamism

In Chapter 3, we saw how Cortex enables efficient auto-batching for recursive deep learning computations. However, as Chapter 2 described, there exists a wide variety in control flow patterns exhibited by deep learning computations beyond recursion. This chapter describes how we generalize the lessons learned from designing and evaluating Cortex to perform auto-batching for general unrestricted control flow.

When evaluating Cortex's performance, we discussed how Cortex's performance improvements can be attributed to the following:

1. Reduced execution overheads associated with dynamic approaches to auto-batching.
2. End-to-end compilation of the tensor operators with the surrounding control flow.

In this chapter, we discuss how we achieve these in the general case and describe ACRoBat (**A**utomated **C**ompiler and **R**untime-enabled **Bat**ching), a compiler and runtime-based framework which performs auto-batching for computations exhibiting general control flow.

As compared to the compilation of recursive computations in Cortex, the presence of general control flow often leads to a lack of execution knowledge during compilation. Our main insight in designing ACRoBat is that despite this, the compiler can often perform analysis and optimizations with the goal of aiding dynamic analysis and thereby reducing the execution overheads while effectively exploiting the parallelism in the input computation. Accordingly, ACRoBat employs novel *static+dynamic program analysis* to enable auto-batching with very low overheads. Further, ACRoBat's *end-to-end tensor kernel generation* enables it to automatically generate kernels optimized and specialized for the larger computation. Users of ACRoBat express their computations using an expressive high-level language. This generality allows one to express a wide variety of control flow patterns, ranging from simple conditional statements to complex recursive computations. Table 4.1 provides a qualitative comparison of ACRoBat with related work on auto-batching.

As part of its hybrid static+dynamic optimizations, ACRoBat relies heavily on static analysis techniques. In order to gain as much insight about the input computation as possible during compilation, ACRoBat employs traditional compiler techniques such as context-sensitivity and profile-guided optimizations, while also relying on minimal user annotations. ACRoBat also uses static analysis to identify and exploit data reuse opportunities when automatically generating and optimizing batched tensor kernels as we see in §4.3.

Table 4.1: Comparison between ACRoBat and other solutions for auto-batching dynamic deep learning 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 | - | Dynamic only | Static only | Dynamic only | Hybrid |
| Vendor libray use | High | High | None | High | None |
| Generality | High | High | Low | Mid | High |
| User implementation effort | Low | Low | High | Low | Low |
| Performance | Low | Low | High | Low | High |

## 4.1 Overview and API

The presence of control flow dynamism necessitates reliance on potentially expensive runtime analysis to perform auto-batching. In ACRoBat, we observe that while perfect knowledge is not possible, aggressive static analysis often provides sufficient information to help reduce the dynamic overheads of batching. Not only is this helpful for more efficiently handling the dynamic control flow, we find that we can also generate specialized and more efficient tensor kernels in an end-to-end manner.

We will now look at ACRoBat's compilation and execution workflows (illustrated in Figure 4.1) that make use of the above insights. ACRoBat takes as input an unbatched deep learning computation expressed in Relay [106] which is a simple but Turing-complete functional language developed for expressing deep learning computations. This enables ACRoBat users to express models with dynamic control flow, such as the ones discussed in Chapter 2, with relative ease. For example, Listing 2.1 illustrates a simple RNN model which ACRoBat can take as an input. ACRoBat also allows users to provide annotations which are used during compilation. We discuss these in the later sections of the chapter.

Given an input Relay computation ①, compilation in ACRoBat begins with batched kernel generation ②. Here, ACRoBat performs static analysis to identify data reuse opportunities and accordingly generates batched versions ③ of kernels implementing the tensor operators used in the input program. These unoptimized kernels are then optimized by the auto-scheduler module ④. This module performs profile-guided optimization as described in §4.3.3 to determine how to prioritize resources when auto-scheduling the kernels. Once optimized, target code ⑩ such as CUDA C++ can be generated for the batched kernels. Concurrently, the input Relay program is further optimized and compiled ⑤ in an ahead-of-time (AOT) fashion to generate C++ code ⑦.

During execution, ACRoBat lazily executes the AOT compiled input program ⑦ on a mini-batch of inputs ⑥, and constructs DFGs ⑧ by interfacing with the ACRoBat runtime library. Once these DFGs are constructed, the ACRoBat runtime library will schedule them ⑨, while looking for batching opportunities[1]. Then, it will invoke the generated and optimized batched kernels ⑩ for each batch

[1]In this work, we only consider parallelism across multiple calls to the same tensor operator. While multiple tensor operators can be executed in parallel as well, we leave that for future work.

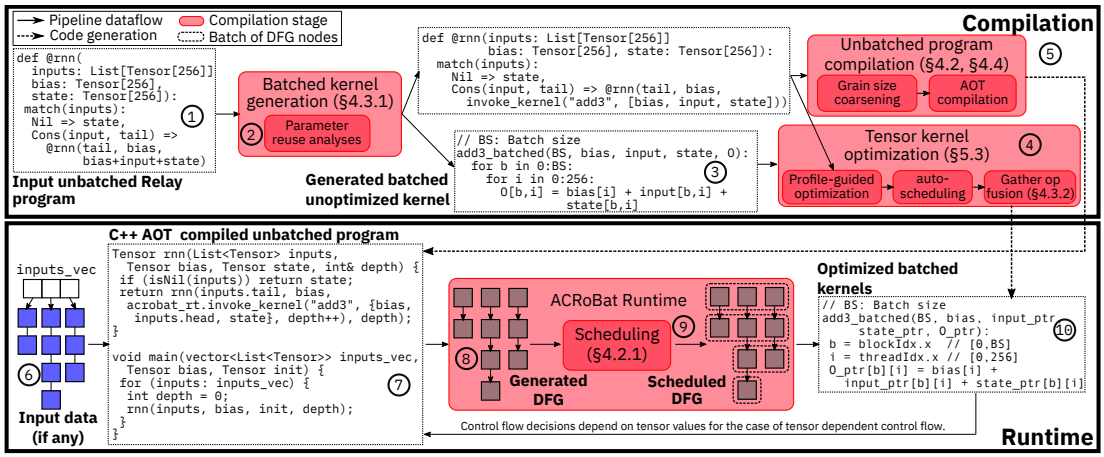



Figure 4.1: Overview of the ACRoBat compilation and runtime pipeline.

of DFG nodes that it identifies. Depending on whether the input program exhibits tensor dependent control flow, the execution can cycle back to the AOT compiled program which will execute further and create more DFGs.

Now, we will take a look at the different components of ACRoBat in more detail, starting with the hybrid static+ dynamic optimizations in §4.2, kernel generation and optimization in §4.3 and some implementation details in §4.4.

## 4.2 Hybrid Static+Dynamic Optimizations

Dynamic control flow often precludes static program transformations. Therefore, as we discussed above, ACRoBat takes a hybrid approach whereby it exploits static program knowledge by either (1) providing hints to the dynamic analysis (as in the case of the inline depth computation in §4.2.1), or (2) generating code that affords the dynamic analysis greater freedom in exploiting parallelism (for example, when it uses concurrent execution in presence of tensor dependent control flow as described in §4.2.3). Further, static analysis also allows us to perform optimizations such as kernel fusion, which is important for high performance (§4.5.4). Below, we provide more details regarding our hybrid analysis.

### 4.2.1 Inline Depth Computation

We saw that at runtime, ACRoBat schedules the execution of the tensor operators that form the nodes of the generated DFG. As part of this scheduling, ACRoBat identifies batching opportunities to exploit parallelism. Given one or more DFGs, each node is assigned a depth such that the depth of a node is larger than the depth of any of its producer nodes. This way, the dependency order of the nodes is obeyed when nodes are executed in the increasing order of their depths. Nodes at the same depth can be executed concurrently as they do not depend on each other. This allows the scheduler to exploit the aforementioned parallelism. A fully dynamic approach to assign the node depths, as used in TensorFlow Fold [75] and Dynet [85], performs a simple traversal of the generated DFGs. We now look at how

ACRoBat can effectively use information available during compilation in order to reduce the overheads of DFG scheduling.

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.

The depth assignment discussed above satisfies both of the above conditions. However, we note that a separate graph traversal after the DFGs have been constructed, as described above, is often unnecessary in order to perform this depth assignment. We make the following two observations:

**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 is often available during compilation.

```
List<Tensor> rnn(List<Tensor> inputs, Tensor state, Tensor bias,
                 Tensor iweight, Tensor hweight, int& depth) {
  if (inputs == ListNil())
    return ListNil();
  auto input_linear = AcrobatRT.InvokeKernel("bias_dense",
                        0, {bias, iweight, inputs.head});
  auto new_state = AcrobatRT.InvokeKernel("sigmoid_add_dense",
                       depth++, {input_linear, hweight, state});
  return ListCons(new_state, rnn(inputs.tail, state, bias, iweight,
                                 hweight, depth));
}

vector<Tensor> main(Tensor rnn_bias, Tensor rnn_iweight,
                    Tensor rnn_hweight, Tensor rnn_init, Tensor cweight,
                    Tensor cbias, vector<List<Tensor>> inputs_vec) {
  vector<Tensor> res;
  for (auto inputs: inputs_vec) {
    int depth = 0;
    AcrobatRT.SetPhase(0);
    auto rnn_res = rnn(inputs, rnn_init, rnn_bias,
                       rnn_iweight, rnn_hweight, depth);
    AcrobatRT.SetPhase(1);
    depth++;
    res.push_back(
      map([&](Tensor p) {
        AcrobatRT.InvokeKernel(
          "relu_bias_dense", depth, {cbias, cweight, p}); },
        rnn_res)
    );
  }
  return res;
}
```

Listing 4.1: AOT compiled output for the RNN model in Listing 2.1, with aspects pertaining to inline depth computation highlighted.

Based on these observations, we devise a scheme whereby we can assign correct depth values that respect the dependencies between tensor operators as well as exploit the available instance and batch parallelism as we execute the unbatched program for each input instance and construct the DFG. In this scheme, the depth of an operator is equal its position in the dependency ordering induced by

the execution of the unbatched program. This ensures that objective **G.1** is satisfied. Computing graph depths in this manner allows us to skip an explicit graph traversal for scheduling. Therefore, we exclusively rely on observation **O.2** above in order to discover and exploit opportunities for parallelism. Specifically, we use the following three 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 data items. Employing observation **O.2** above, we ensure, in our generated code, that such concurrent operations are assigned the same depths during the execution of the unbatched program for each input instance. We rely on user annotations to obtain information about recursive parallelism[2]. Listing 4.1 shows the AOT compiled code generated for the RNN model in Listing 2.1. We see, on line 27, how all the invocations of the `relu_bias_dense` kernel inside the `@map` function are assigned the same depth.

### Hoisting Independent Operations

Given a recursive computation, such as the `@rnn` function in Listing 2.1, 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 2.1 can be hoisted out of the recursion. In order to discover such operations that can be hoisted, we rely on a 1-context sensitive taint analysis[3]. As part of this analysis, we statically compute the depths of such operations. We see, in Listing 4.1, how the invocation of the kernel `bias_dense` on line 6 is assigned a statically computed depth of 0. During runtime, such operations are therefore 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 as part of the sequential dependency one at a time.

### Combating Eagerness of Depth Scheduling

A depth-based scheduling scheme, like the one ACRoBat uses, can often be too eager in executing tensor operators, which can leading to a sub-optimal amount of exploited parallelism. This has been well-documented in past work [85]. Depending on whether repetitive or merely conditional control flow is involved, we rely on the following techniques to allow for better performance.

*Program Phases:* For our RNN example in Listing 2.1, in order to exploit the most parallelism for the output operation on line 21, one should wait until all the RNN operations (i.e. the ones invoked in the `@rnn` function) for all the input instances have been executed. This way, all output operations corresponding to all words in all input instances can be batched and executed as one kernel invocation.

[2] ACRoBat allows users to mark any set of function calls as concurrent in the input Relay code.

[3] Context sensitivity [5] 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 static analysis precision. For the deep learning computations we worked with, we found that a 1-context sensitive analysis was sufficient. Deeper contexts might be useful, however, for more complex computations.

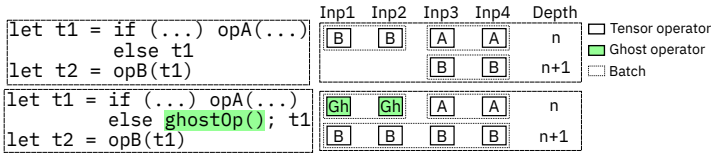



Figure 4.2: Ghost operations can enable better batching.

This would require that all these output operations across all input instances be assigned the same depth. However, this may not be the case as the length of each input sentence may, in general, vary. Therefore, in order to restrict the scope of the depth assignment, the user can provide program phase annotations (lines 16 and 19 in Listing 2.1). In essence, ACRoBat interprets such phase boundaries as boundaries for scheduling. ACRoBat, therefore, schedules and executes operations in one phase before moving on to the next. For our example, this way, we ensure that all the RNN functions are executed for all input instances before ACRoBat moves on to the output operations.

*Ghost Operations:* A similar situation can occur in the presence of a conditional if statement. This is illustrated in upper pane of Figure 4.2. 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 I3 and I4. In the lower pane, we insert a call to a ghost operation leading to an optimal schedule. ACRoBat can statically identify such cases and appropriately insert ghost operations as needed. Note that ACRoBat employs ghost operations merely to affect scheduling behavior and they are ignored during tensor kernel execution.

### 4.2.2 Grain Size Coarsening

Generally, scheduling is performed at the granularity of individual tensor operators i.e. each node in the DFG corresponds to one, potentially fused, tensor kernel call. As we discussed in Chapter 2, deep learning computations frequently contain larger static blocks embedded in the dynamic control flow (the LSTM cell in the case of TreeLSTM, for instance). Therefore, performing scheduling at the finer granularity of individual tensor operators is often unnecessary and leads to high scheduling overheads. ACRoBat, therefore, performs scheduling at the static block coarser granularity. As these blocks do not contain any control flow, coarsening the granularity this way does not lead to a loss of exploited parallelism but only reduces the size of the generated DFGs and hence the scheduling overheads. This optimization has been explored in past work [36, 42, 116, 146, 151] to some extent as well. The coarsening optimization is illustrated in Figure 4.3.

### 4.2.3 Tensor Dependent Control Flow

We saw that during execution, 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 input instance sequentially and trigger the batching and execution of all the DFGs at once. In the presence of tensor dependent control flow, however, we would be required

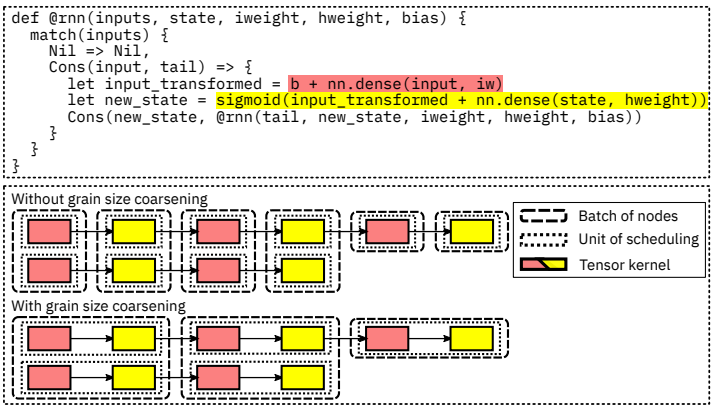



Figure 4.3: Grain size coarsening illustrated for the `@rnn` function shown in Listing 2.1.

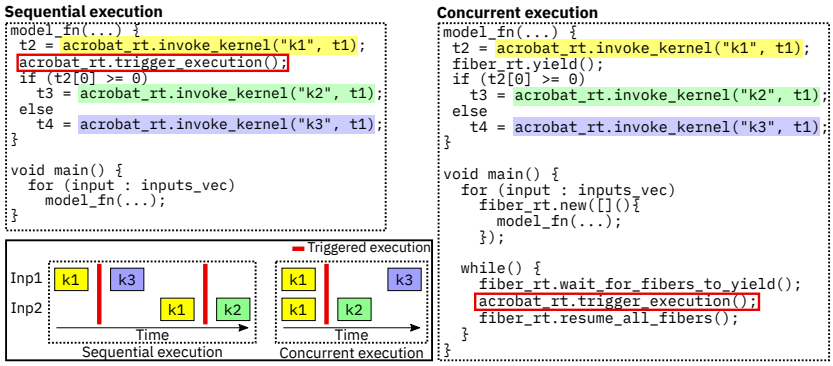



Figure 4.4: Concurrent execution of the unbatched program in the presence of tensor-dependent control flow.

to trigger the execution any time we encounter a control flow decision that depends on the value of an intermediate tensor. Therefore, when the unbatched programs are executed sequentially in the presence of tensor dependent control flow, we are not able to exploit any batch parallelism. Thus, in the presence of tensor dependent control flow, ACRoBat generates code to execute the unbatched program for each input instance concurrently by using *fibers* or userland threads [21]. This way, the unbatched program 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 for each instance resumed after as illustrated in Figure 4.4. Correspondingly, in order to exploit instance parallelism in the presence of tensor dependent control flow, ACRoBat launches concurrent strands of execution in newly spawned fibers, similar to the popular fork-join model of parallelism [78]. We therefore see that ACRoBat combines the static knowledge of batch and instance parallelism with dynamic concurrent execution as part of its hybrid analysis to effectively exploit parallelism in the presence of tensor dependent control flow.

## 4.3 End-to-end Tensor Kernel Generation

We saw above that as part of its compilation workflow, ACRoBat generates and optimizes batched kernels. This end-to-end code generation without reliance on external vendor libraries enables ACRoBat to uniformly handle all tensor operators used in the input program without additional compiler development effort. This enables ACRoBat to achieve a larger coverage of operators with batching supported. In contrast, using a vendor library would preclude an automated, uniform and general mapping from unbatched to batched tensor operators.

More details about ACRoBat's use of static analysis to extract information about the surrounding computation to generate efficient tensor kernels are provided below.

### 4.3.1 Exploiting Parameter Reuse

Given the input unbatched computation, ACRoBat needs to generate batched versions of the kernels implementing the tensor operators used in the computation. Generating a batched version of such a tensor operator is not as straightforward as adding a batch dimension to each input and output tensor as well as the computation loop nest. This is because some input tensors might be shared across calls to the operator. This is frequently the case for model parameters. For example, consider tensor operator `add3`, which implements the element-wise addition of three tensors used in the input computation ① in Figure 4.1. Across multiple calls to `add3`, 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 batched version of this operator (③ and ⑩) in Figure 4.1.

ACRoBat uses a 1-context sensitive taint analysis to identify such shared arguments to tensor operators. The use of static analysis for this purpose allows ACRoBat to obtain accurate knowledge about the parameter reuse patterns. On the other hand, without any knowledge about the usage patterns, inferring such tensor reuse dynamically with low overheads would involve the use of heuristics, which as we see in §4.5.3, can be brittle, leading to sub-optimal performance.

#### 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, in a manner similar to the RNN model implemented in Listing 2.1, we implement a bidirectional RNN (BiRNN) [111] 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[4]. In the case of the BiRNN example,

[4] Simply inlining the `@rnn` function will not work in this case it is recursive.

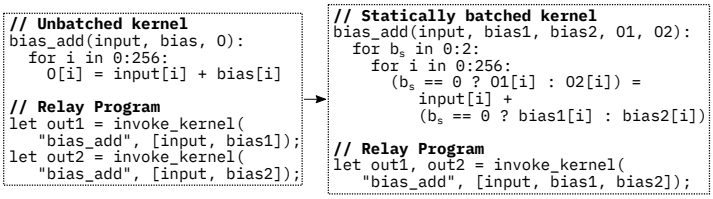



Figure 4.5: Horizontal fusion promotes parameter reuse.

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.

```
(* Type annotations are omitted in the listing for simplicity. *)
def @main(f_rnn_bias, f_rnn_iweight, f_rnn_hweight, f_rnn_init,
          b_rnn_bias, b_rnn_iweight, b_rnn_hweight, b_rnn_init,
          inputs_list) {
  let rinputs_list = @reverse_list(inputs_list);
  let forward_res = @rnn(inputs_list, f_rnn_init,
                         f_rnn_bias, f_rnn_iweight, f_rnn_hweight);
  let backward_res = @rnn(rinputs_list, b_rnn_init,
                          b_rnn_bias, b_rnn_iweight, b_rnn_hweight);
}
```

#### 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 operatorwithin 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 Figure 4.5.

### 4.3.2 Fusing Memory Gather Operations

During execution, ACRoBat identifies batching opportunities across the lazily generated DFGs and launches the appropriate batched kernel. Due to the dynamic nature of this process, the input tensors to all DFG nodes in a batch may not be laid out contiguously in the accelerator's memory. As a result, we need to perform a memory gather operation before being able to operate on the tensors. Performed naively, this can lead to a significant explosion in the data movement. Therefore, ACRoBat generates specialized batched kernels to directly operate on the input tensors scattered in memory, in effect fusing the expensive memory gather operation with the batched kernel. The generated batched kernel (10) in Figure 4.1 illustrates this. As we show in §4.5, ACRoBat's end-to-end generation of tensor kernels in this manner leads to a significant performance improvement.

### 4.3.3 Tensor Kernel Optimization

We saw above how ACRoBat generates unoptimized kernels for implementing batched versions of (potentially fused) tensor operators used in the input program. Then, we need to optimize the kernels and generate target code. Below, we provide some details on how we rely on the auto-scheduler [156] implemented as part of TVM in order to automatically optimize these kernels.

#### Auto-scheduler Operator Priorities

Given a deep learning computation consisting of a number of tensor operators, TVM's auto-scheduler prioritizes the optimization of tensor operators based on their relative execution cost. This priority is estimated as the product of the execution cost of the unoptimized kernel and the number of times the kernel is invoked during the execution of the input program. In the absence of dynamic control flow, the latter quantity is equal to the number of static calls to the kernel in the input program. In the presence of control flow (such as repetitive or conditional control flow), however, this can lead to incorrect priorities for the kernels, thereby resulting in suboptimal kernel performance. ACRoBat therefore relies on profile-guided optimization (PGO) to accurately estimate the relative importance of tensor operators. When PGO is not possible, ACRoBat also provides a simple static analysis to heuristically perform this estimation based on how deeply nested the call to an operator is in the recursion.

#### Dynamic Batch Size

Due to the dynamic nature of ACRoBat's scheduling, the loop corresponding to the batch dimension in the unoptimized batched kernels generated by ACRoBat has a variable loop extent (the outermost batch loop in kernel ③ in Figure 4.1, for example, has a variable extent BS). 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 loop extent.

When generating code for loops with variable loop extents, we often have to insert conditional check statements in order to avoid out of bounds accesses. Such conditional checks can be severely detrimental to performance. Therefore, we rely on the local padding and local partitioning techniques proposed in DietCode [155] to eliminate these conditional check statements when appropriate.

## 4.4 Other Optimization and Implementation Details

#### Ahead-of-time Compilation

We saw in §4.1 that ACRoBat takes a Relay program as input. As described in [113], the default Relay execution stack consists of a virtual machine (VM) which interprets a simple bytecode that Relay is compiled down to. While the overheads of the VM are insignificant for static straightline deep learning computations as the authors find in [113], we find that the overheads can be significant in the

presence of dynamic control flow and irregular data structures (§4.5.2). Therefore, ACRoBat instead 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 operations on them to native C++ scalars as well. We see, in §4.5.2, that this AOT compilation significantly reduces the execution overheads of dynamic control flow.

#### ACRoBat Runtime

We have optimized ACRoBat's runtime system to reduce 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.

#### Implementation Details

We prototype ACRoBat by extending TVM v0.9.dev0. TVM currently lacks support for training deep learning models. As a result, our prototype currently only supports the forward pass of an input model. However, ACRoBat's techniques as described above apply to both the forward as well as the backward passes. For the same reason, due to lack of access to trained model parameters, we use pseudo-randomness to emulate tensor dependent control flow in deep learning computations as part of our evaluation. Further, we find that TVM's operator fusion pass is limited and is often unable to fuse memory copy operations such as tensor reshape, concatenation and transpositions. Therefore, in our implementations of the deep learning 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[5].

## 4.5 Evaluation

We now evaluate ACRoBat against Cortex, DyNet and PyTorch on an Nvidia GPU.

### 4.5.1 Experimental Setup

#### Models

We use the models listed in Table 4.2 for the evaluation. For each model, we look at two model sizes—small and large. For the TreeLSTM, BiRNN, NestedRNN, DRNN, and StackRNN models, the small models size uses a hidden size of 256 and the large one uses a hidden size of 512. 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 [27], while the large model

[5] Side-effects via mutable references are currently not supported.

Table 4.2: Models and datasets used for evaluating ACRoBat.

| Model | Description | Dataset |
|---|---|---|
| TreeLSTM | TreeLSTM [125] model | Stanford sentiment treebank [121] |
| MVRNN | MV-RNN [120] model | Stanford sentiment treebank |
| BiRNN | Bidirectional RNNs [120] | XNLI [25] dataset |
| 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 [6] for top-down tree generation | Randomly generated tensors. |
| Berxit | Early exit for BERT inference [145]. All layers share weights. | Sequence length 128. |
| StackRNN | StackLSTM [33] parser with LSTM cells replaced by RNN cells. | XNLI |

Table 4.3: Relay VM vs. ACRoBat's AOT compilation: Forward pass latencies in $ms$.

| Hidden Size | Batch Size | TreeLSTM | | MVRNN | | BiRNN | |
|---|---|---|---|---|---|---|---|
| | | Relay VM | ACRoBat AOT | Relay VM | ACRoBat AOT | Relay VM | ACRoBat 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 |

uses the same hyper-parameters as the $\text{BERT}_{\text{LARGE}}$ model [27], except that we use 18 layers instead of 24 in this case.

#### Experimental Environment

We run our experiments on a Linux workstation with an AMD Ryzen Threadripper 3970X processor (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. DyNet also uses the Eigen library (v3.3.90). We compare against DyNet's commit `3e1b48c75` (March 2022) and PyTorch v1.9.0a0+gitf096245.

### 4.5.2 Benefits of AOT Compilation

We first look at the performance gains due to the AOT compilation described in §4.4. The performance of the TreeLSTM, MVRNN and BiRNN models[6] when executed using the Relay VM and ACRoBat's AOT compiler is shown in Table 4.3. All runs have the grain size coarsening, gather operator fusion and program phase optimizations turned on. As the table shows, VM overheads significantly slow down (up to 13.45× times) the execution 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 enabled.

[6] Our prototype implementation of ACRoBat does not currently support the execution of the remaining models in Table 4.2 using the Relay VM.

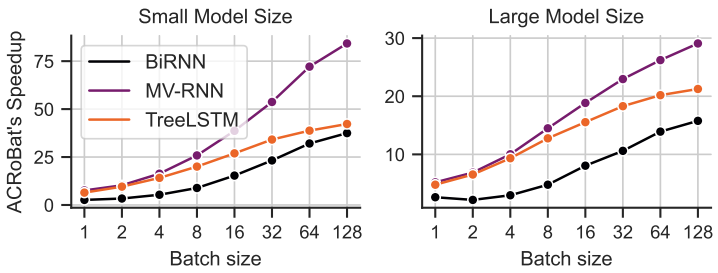



Figure 4.6: Speedups obtained over PyTorch for the TreeLSTM, MVRNN and BiRNN models.

### 4.5.3 Overall Performance

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

#### Performance Comparison with PyTorch

We first look at ACRoBat's performance as compared to that of PyTorch for the TreeLSTM, MVRNN and BiRNN models[7]. This data is shown in Figure 4.6. 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. We see that the speedups are higher for the small model size as compared to the larger model sizes. This is 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 MVRNN, the relative importance of exploiting instance and batch parallelism is lower, leading to performance lower than that of MVRNN.

#### Performance Comparison with DyNet

Next, we compare ACRoBat's performance with that of DyNet. As mentioned in §4.4, we simulate tensor dependent control flow in the NestedRNN, DRNN, Berxit and StackRNN models using pseudo-randomness. 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 (§4.2.3) 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.

[7]We were able to use TorchScript only for the BiRNN model as it does not currently support recursive data types [24], such as the parse trees used as inputs to the TreeLSTM and the MVRNN models.

Table 4.4: 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.

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

Table 4.5: 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 |
| Number of 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 Tables 4.4 and 4.6.
[2] Includes memory copy kernels.
[3] Includes all kernel calls as well as calls to `cudaMemcpy` and `cudaMemcpyAsync`.

The forward pass latencies for DyNet and ACRoBat are shown in Table 4.4[8]. ACRoBat performs better than DyNet in most cases due to a number of reasons. Table 4.5 lists the time spent by DyNet and ACRoBat in different runtime components for the TreeLSTM model. We see that ACRoBat spends a significantly lower amount of time in constructing and scheduling the DFG. ACRoBat's optimizations such as static kernel fusion and grain size coarsening reduce the size of the DFG by reducing the number of tensor kernels invoked, thereby reducing the construction and scheduling overheads. Further, ACRoBat's inline depth computation allows it to exploit the same amount of parallelism as DyNet's agenda based scheduling scheme [85] with much less execution 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 take a closer look at the benefits of each of the optimizations that ACRoBat employs in more detail in §4.5.4.

On the other hand, we note that DyNet performs slightly better than ACRoBat for some configurations of the BiRNN and the NestedRNN models. Table 4.5 provides a breakdown of various runtime components for the BiRNN model. We see that while ACRoBat incurs much less runtime overheads for DFG construction, scheduling and memory transfer as compared to DyNet, it spends a significantly

[8] DyNet implements two scheduling schemes [85]—an agenda-based one, and a depth-based one. We consider the best of the two for each model configuration in our evaluation.

higher amount of time in kernel execution which outweighs the ACRoBat's savings in runtime overheads. We believe that better tensor kernel optimizations can help reduce the performance gap between ACRoBat and DyNet for these cases.

Beyond these reasons, ACRoBat performs better on specific benchmarks due to the following reasons: Unbatched execution of certain operators: We find that DyNet is unable to batch the execution of some operators certain model computations invoke. This is because of the following two reasons:

1. *Brittle heuristics:* We mentioned in §4.3.1 that ACRoBat's use of static analysis for inferring parameter reuse allows it to have accurate knowledge to exploit reuse during with static optimizations. On the other hand, DyNet employs a fully dynamic analysis for this purpose. 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 MVRNN 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 4.6.

2. *High framework development effort:* As described in §4.3, 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. For example, 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 valued 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 valued 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 4.6 shows.

Inability to exploit instance parallelism in DRNN: 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 sub-trees 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 §4.2.3. 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. DyNet's performance when this optimization is performed manually is shown in Table 4.6.

Table 4.6: Model execution times in $ms$ after the improvements described in §4.5.3 were made for the TreeLSTM, MVRNN and DRNN models. DN++ stands for DyNet with improvements.

| Model Size | Batch Size | TreeLSTM | | | MVRNN | | | DRNN | | |
|---|---|---|---|---|---|---|---|---|---|---|
| | | DyNet | DyNet++ | ACRoBat | DyNet | DyNet++ | ACRoBat | DyNet | DyNet++ | ACRoBat |
| 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 |

Table 4.7: Cortex vs. ACRoBat: Forward pass latencies in $ms$.

| Hidden Size | Batch Size | TreeLSTM | | MVRNN | | 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 |

Performance Comparison with Cortex

Table 4.7 compares the performance of ACRoBat with that of Cortex for the TreeLSTM, MVRNN and the BiRNN models. Cortex's restrictive API prohibits the implementation of the other models in Table 4.2. The input linear transformations that can be hoisted out of the recursive computation in the TreeLSTM and BiRNN models (as described in §4.2.1) are manually hoisted and offloaded to cuBLAS in the case of Cortex, while ACRoBat performs this hoisting automatically and relies on auto-scheduled kernels for the same.

We see that for the TreeLSTM and BiRNN models, Cortex performs up to 1.87× better than ACRoBat. This is because Cortex is specialized for recursive deep learning computations, allowing it to generate a single kernel implementing the entire computation and to take advantage of aggressive kernel fusion and model persistence, while reducing kernel call overheads. On the other hand, Cortex requires its users to manually optimize the generated kernel significantly increasing the developer effort[9] while ACRoBat relies on an auto-scheduler (§4.3.3). Note also that Cortex performs much worse than ACRoBat on the MVRNN 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, we see that ACRoBat delivers performance comparable to that of Cortex, while supporting a much wider range of deep learning computations with significantly lesser developer effort.

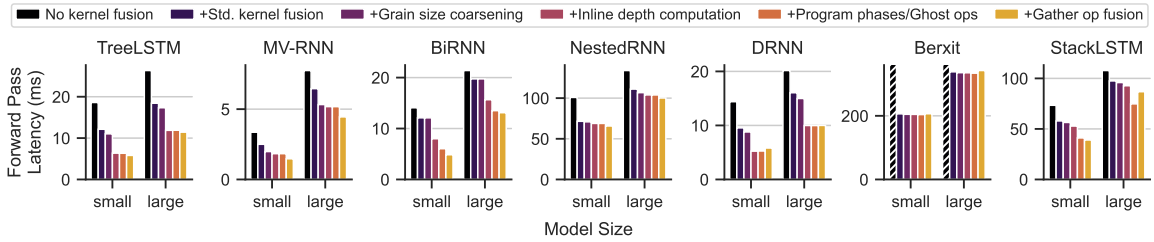



Figure 4.7: Benefits of different optimizations. The unfused executions of Berxit were killed due to out-of-memory errors.

Table 4.8: NestedRNN (small, batch size 8) execution times in *ms*, illustrating the benefits of using PGO invocation frequencies during auto-scheduling.

| Auto-scheduler iterations | Exe. time without PGO | Exe. time with PGO |
|---|---|---|
| 100 | 41.08 | 42.49 |
| 250 | 34.58 | 30.88 |
| 500 | 31.61 | 24.4 |
| 750 | 27.33 | 23.72 |
| 1000 | 25.63 | 24.34 |

### 4.5.4 Benefits of Optimizations

We now evaluate the relative benefits of the different optimizations ACRoBat performs. Figure 4.7 shows forward pass execution times for the models in Table 4.2 (for the large model size 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 §4.3.2) provides significant benefits for all models[10]. Grain size coarsening is beneficial in all models, but is most prominent in models with a relatively high amount of control flow such as TreeLSTM and MVRNN. Inline depth computation enables ACRoBat to skip the separate scheduling step, thus reducing overheads. This optimization also is most beneficial for models such as TreeLSTM and MVRNN. Further, turning this optimization on in the case of the DRNN model also enables ACRoBat to exploit the instance parallelism inherent in the computation as discussed in §4.2.3 leading to a drop in the execution time. The BiRNN model involves per-token output linear transformations as would be the case when performing token classification. Program phase annotations for this allow ACRoBat to avoid greedy scheduling as described in §4.2.1 and batch all these output linear transformations together. The StackRNN model executes different tensor operators, depending on the current parser action. This involves a conditional if-statement. Insertion of ghost operators here therefore again enables more optimal exploitation of parallelism leading to better performance.

Gather operator fusion leads to a reduction in the execution time for some benchmarks and an increase in 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

[9]For example, implementing the MVRNN model in Cortex requires 325 LoC in Python, while the same model in ACRoBat can be implemented in 79 LoC of Relay and 108 LoC of Python (187 LoC in total).

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

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 in ACRoBat, we perform the explicit memory gather when the input tensors are not already contiguous in memory. Tensors are more likely to be contiguous in such iterative models as compared to recursive ones, 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, we find that models with a relatively lower amount of control flow or a higher amount of tensor computations such as Berxit or NestedRNN benefit less from optimizations that reduce scheduling overheads. For the same reason, the larger models benefit less from such optimizations as compared to the smaller ones for all the benchmarks evaluated.

Benefit of PGO in Tensor Kernel Auto-Scheduling

We mentioned in §4.3.3 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. As Table 4.2 lists, NestedRNN 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 4.8 shows the execution times of the benchmark with and without PGO for different iterations of the auto-scheduler[11] which shows how ACRoBat can better prioritize auto-scheduling for the RNN operators with PGO turned on.

## 4.6 Related Work

ACRoBat improves upon past work on auto-batching (which we discussed in Chapter 2) by effectively exploiting all parallelism in a given computation, while incurring significantly lower runtime overheads. As part of its design, ACRoBat builds upon and borrows from a wide body of past work including the aforementioned work on auto-batching. For instance,

1. *Grain size coarsening:* Grain size coarsening has been explored in past work [36, 42, 116, 146, 151]. ACRoBat, however, performs coarsening statically in the context of general purpose auto-batching framework.
2. *Gather Operator Fusion:* The gather operator fusion optimization is similar to the gather and scatter fusion [22] performed for sparse GEMM operations in the CUTLASS library though ACRoBat is able to perform this optimization automatically as part of its compilation workflow.
3. *Local Padding and Local Partitioning:* As mentioned in §4.3.3, ACRoBat borrows some techniques from DietCode to auto-schedule tensor operators with variable loop extents. DietCode's techniques are

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

thus complementary to ours and it can be fully integrated into ACRoBat potentially for better kernel performance.

4. *Traditional Compiler Techniques:* ACRoBat borrows techniques from the large body of work on compilation for programs written in general-purpose languages. These techniques have been developed and used for a wide variety of applications in this field. Context-sensitivity [5] is a common technique used to increase the precision of of static compiler analyses. ACRoBat's inline depth computation and DFG scheduling more generally are similar to work on static [39, 43] and dynamic [94, 118] instruction scheduling in the past for pipelined and superscalar processors. ACRoBat however applies these techniques in the context of a deep learning framework. Taint analysis has been extensively used for security purposes [53, 71, 132]. Knowledge about program phases [114] allows system designers to adaptively optimize systems for different parts of a program for optimal performance [8, 52, 154]. Similarly, profile information can effectively guide a variety of optimization decisions [15, 47, 61].

Recent compiler framework and IR (intermediate representation) design for deep learning applications has often focused on expressivity. These include Relay, TorchScript [97] which is used in PyTorch [92] and MLIR [70]. While ACRoBat builds on the Relay compilation stack, we believe that our techniques can be transfered to the other commonly used IRs in a straightforward manner.

## 4.7 Chapter Summary

This chapter presents ACRoBat, a compiler and runtime framework that performs auto-batching of deep learning computations in the presence of dynamic control flow. ACRoBat employs hybird static+dynamic analysis to enable effective batching with low runtime overheads. Further, end-to-end code generation allows ACRoBat to generate hghly optimized tensor kernels for efficient execution. The design and proliferation of highly expressive compiler representations for deep learning computations such as Relay, MLIR and TorchScript signifies the importance of a compiler's ability to effectively represent dynamism. In this context, we believe that ACRoBat takes an important step forward in opening the door to more collaborative relationships between the various components of a deep learning framework such as the tensor compiler, the high-level language compiler as well as the runtime.

# 5 The CoRa Ragged Tensor Compiler: Efficient Batching for Shape Dynamism

In the last two chapters, we described techniques to perform auto-batching in the presence of control flow dynamism. We saw in Chapter 2 that shape dynamism is another kind of dynamism often exhibited by deep learning computations such as transformer-based models. In this chapter, we develop techniques to enable performant batched execution for such computations.

Recall that a computation is said to exhibit shape dynamism when its execution involves tensors of varying shapes across different inputs. Ragged tensors are a common abstraction used to represent data when executing such computations in a batched fashion. A simple ragged tensor operator is illustrated in Figure 5.1. Due to the impendance mismatch between current vendor libraries and tensor compilation infrastructures, which mostly support computations on sparse and dense tensors, ragged tensor operations, when executed using either of them are quite inefficient. Sparse tensor libraries and tensor compilers do not sufficiently exploit the properties of ragged tensors, and are optimized for tensors much sparser than the ragged tensors encountered in practice. On the other hand, using dense tensor infrastructure for performing ragged computations necessitates the use of padding or masking, which can be quite wasteful. Figure 5.2 plots the relative amount of computation (in FLOPs) involved in the forward pass of an encoder layer of the transformer model[1] with and without padding. We see that padding leads to a significant increase in the computational requirements of the layer, especially at larger batch sizes, increasing computation in an already computationally expensive model.

This chapter proposes CoRa (**Co**mpiler for **Ra**gged Tensors), a compiler-based solution enabling easy and more portable generation of performant code for ragged operators[2]. While sparse [65, 81] and dense [7, 13, 103, 135] tensor compilers have been well-studied, it is not straightforward to apply these techniques to ragged tensors, due to the following challenges:

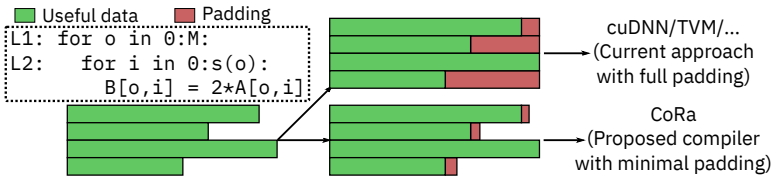



Figure 5.1: A simple elementwise operation on ragged tensors.

[1] The hyperparameters used are the same as those in §5.6.2.

[2] A paper [37] describing CoRa was published at MLSys 2022.

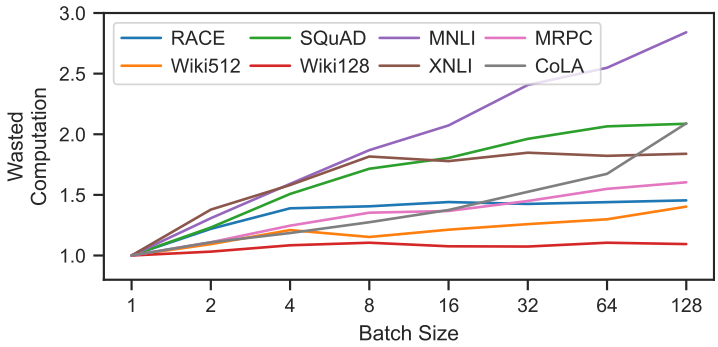



Figure 5.2: Wasted computation due to padding in a transformer encoder layer.

Table 5.1: Comparison between CoRa and current solutions for ragged operations.

| Framework | Portability | Operator implementation effort | Padding | Performance |
|---|---|---|---|---|
| Dense tensor compilers | High | Low | Full | Low |
| Sparse tensor compilers | High | Low | Minimal | Low |
| Dense vendor libraries | Low | High | Full | Low |
| Hand-optimized implementations | Low | High | Minimal | High |
| CoRa | High | Low | Minimal | High |

**C1 Irregularity in generated code:** While the data in ragged tensors are densely packed, the variable loop bounds can lead to irregular code, often causing a loss of performance on hardware substrates such as GPUs.

**C2 Insufficient compiler mechanisms:** Representing transformations on loops with variable bounds and on tensor dimensions with variable-sized slices is not straightforward due to the dependences that exist among loops and tensor dimensions respectively in ragged operators. Further, optimization decisions made by sparse tensor compilers may not always work for ragged tensors because sparse tensors are much sparser than ragged tensors.

**C3 Ill-fitting computation abstractions:** There is a mismatch between the interfaces and abstractions provided by current compilers and ragged operators. Such operators cannot be expressed in dense compilers, while sparse compilers do not adequately provide ways to express information relevant to efficient code generation.

CoRa is a tensor compiler that allows one to express and optimize ragged operations to easily target a variety of substrates such as CPUs and GPUs. To overcome challenge **C1**, CoRa enables minimal padding of ragged tensor dimensions (§5.3.1) in order to generate efficient code for targets such as GPUs as well as to specify thread remapping strategies to lower load imbalance (§5.3.1). CoRa uses uninterpreted functions (introduced in Chapter 3) to symbolically represent variable loop bounds and scheduling operations on the same (§5.4.1). CoRa's mechanisms (such as its storage lowering scheme discussed in §5.4.3) and optimizations are specialized for ragged tensors thereby tackling **C2**. Further, CoRa provides simple abstractions to convey to the compiler information essential to efficient code

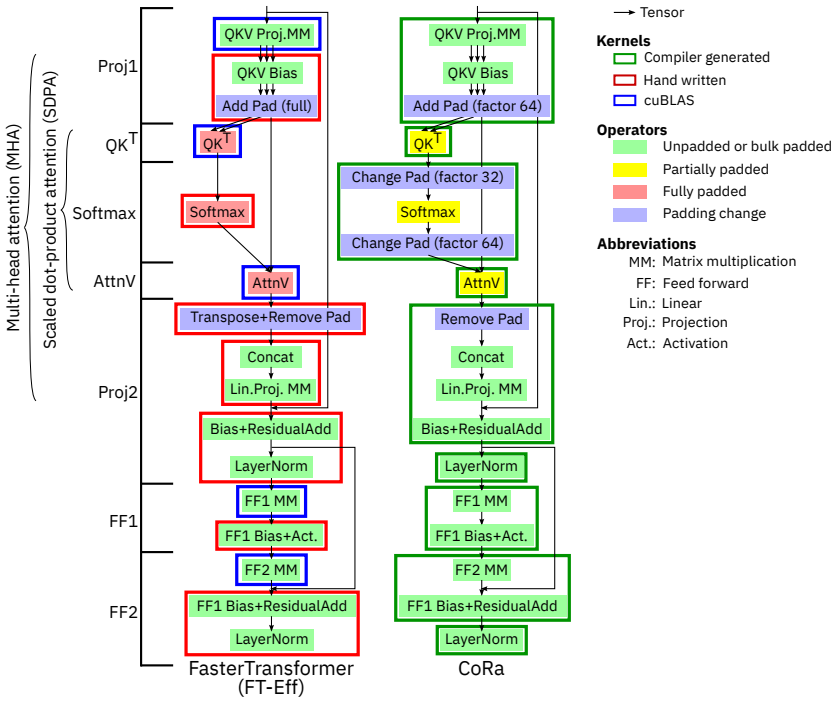



Figure 5.3: FasterTransformer (FT-Eff) and CoRa implementations of a transformer's encoder layer. Note how CoRa's fully compiler-based implementation uses only partial padding for SDPA as opposed to FasterTransformer's fully padded implementation. CoRa also enables more operator fusion (including fusing all the padding change operations) as opposed to FasterTransformer, which cannot do so in all cases as it relies on vendor libraries.

generation, such as padding or thread remapping specifications and raggedness patterns of tensors (§5.3). This overcomes challenge **C3**.

CoRa enables efficient code generation for ragged operators by significantly reducing padding (§5.6). As part of CoRa's implementation, we reuse past work by extending a tensor compiler [7, 13, 65, 103] and thus, provide familiar interfaces to CoRa's users. This also makes it easy in the future to use auto-scheduling [1, 14, 80, 117, 156] for optimizing ragged tensor operations. Table 5.1 compares CoRa with alternatives that are or could be used for ragged operators. Only CoRa achieves high performance and portability, with low operator implementation effort (and minimal padding).

## 5.1 CoRa Overview

CoRa's compiler-based approach enables the generation of performant code in a portable manner. This is reflected in Figure 5.3, which compares CoRa's implementation of a transformer encoder layer with FasterTransformer. The highly-optimized FasterTransformer relies heavily on kernels implemented in cuBLAS (Nvidia's BLAS library), which are shown as blue outlines in the figure, and on manually implemented kernels, shown as red outlines. On the other hand, CoRa's implementation exclusively employs compiler generated kernels (shown as green outlines), making it more portable.

Further, CoRa's compiler approach allows it to exploit more kernel fusion opportunities, evident from the fact that CoRa's implementation launches nine kernels as opposed to FasterTransformer's twelve. Both the implementations in the figure use minimal padding for all operators except for those in the scaled dot-product attention (SDPA) sub-module, where CoRa's specialized approach enables it to get away with lower padding as compared to FasterTransformer. We further discuss these implementations in §5.6.

CoRa's ability to generate performant code that employs minimal padding in a portable manner relies on the following two insights:

**I1** In ragged operations, the pattern of raggedness is usually known before the tensor is actually computed, and is the same across multiple tensors involved in the operation.

**I2** Ragged tensors, like dense tensors, allow $O(1)$ accesses (§5.4.3). This is unlike sparse formats such as compressed sparse row (CSR), where accesses require a search over an array. The HASH [20] sparse format, while allowing $O(1)$ accesses, is unsuitable for accelerators such as GPUs due to its highly irregular storage.

Insight **I1** allows CoRa to precompute the auxiliary data structures needed to access ragged tensors without knowledge of the computation (or values of its input tensors) that produces the ragged tensor. This and insight **I2** enable CoRa to generate efficient code for ragged operations.

Let us now look at CoRa's overall compilation and execution pipeline, as illustrated in Figure 5.4. The user first expresses ① and schedules ② their computation using an API similar to that of past tensor compilers (§5.3). This specification of the computation and the scheduling primitives are then lowered ③ to an SSA-based IR ④. As part of this lowering step, CoRa generates code ⑦ to initialize some auxiliary data structures it needs to be able to lower accesses to ragged tensors (§5.4.3) and to enable loop fusion in ragged loop nests (§5.4.1). We refer to this code as the *prelude* code. Compilation then continues with CoRa lowering tensor accesses to raw memory offsets by making use of the data structures generated by the prelude. Finally, CoRa generates ⑤ target-dependent code ⑨ such as C or CUDA C++. During execution, the formats of the input ragged tensors ⑥ are first processed by the generated prelude code ⑦ which creates the auxiliary data structures ⑧. This prelude code is not computationally expensive (§5.6.4) and hence is executed on the host CPU. These data structures and the ragged tensors are then passed to the generated target dependent code ⑨ which executes on devices such as CPUs or GPUs.

We will now look these stages in more detail below.

## 5.2 Terminology

Ragged operators have one or more loops with bounds that are functions of iteration variables of outer loops. We refer to such loops as *variable loops* or *vloops* while loops with constant bounds are referred to as constant loops, or *cloops*. A loop nest with at least one vloop is referred to as a vloop nest. Further, tensors can be stored in memory with or without padding. When stored without full padding, the size

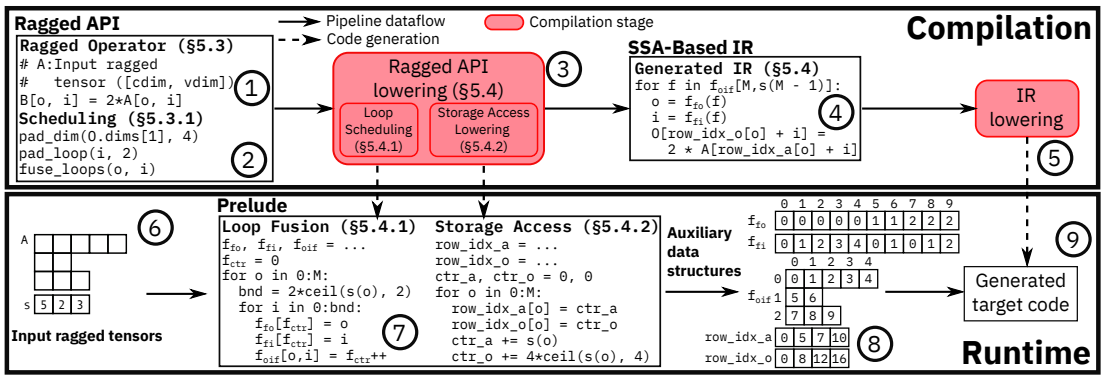



Figure 5.4: Overview of CoRa's compilation and runtime pipeline.

of some tensor dimensions depends on outer tensor dimensions. Such dimensions are referred to as *variable dimensions*, or *vdims* and those with constant sizes are *constant dimensions* or *cdims*. A tensor stored such that it has no vdim (i.e., a fully padded tensor) is referred to as a dense tensor, while a tensor with at least one vdim is a ragged tensor. Note that ragged tensors may still be padded to some extent.

## 5.3 CoRa's Ragged API

CoRa provides a simple API similar to that of past tensor compilers, as seen in Listing 5.1, which expresses the example computation from Figure 5.1 in CoRa. Apart from describing the computation as in a dense tensor compiler, CoRa also requires the user to specify the raggedness dependences of the computation (highlighted in Listing 5.1). This involves specifying vloop bounds as functions of outer loop variables and vdim extents as functions of indices of outer tensor dimensions. Given this information, CoRa automatically computes any derived data structures required (§5.4), making it easy for users to express their computations. CoRa uses named dimensions[3] (discussed further in §5.4.2) to name loops and corresponding tensor dimensions and to specify relationships between them. For example, the loop extent defined on linestates the dependence on the outer loop, referred to by the named dimension `batch_dim`.

### 5.3.1 Scheduling Primitives

In order to optimize the expressed computation, CoRa provides all the scheduling primitives commonly found in tensor compilers. Below, we describe some salient features and points of departure from past tensor compilers.

#### Loop Scheduling

Both cloops and vloops can be scheduled in CoRa. We saw how a vloop, say $L_v$, has a loop bound that is a function of the iteration variables of one or more outer loops, say $L_1$ to $L_k$. CoRa currently

[3] Recall that we also used named dimensions in Cortex.

```
################ Operator Description ################
batch_size = var('M')
# Declare named dimensions
batch_dim, len_dim = Dim(), Dim()
# Loop: Specify vloop extents
lens = input_tensor((batch_size,))
l_ext = Extent([batch_dim], lambda b: lens[b])
loop_exts = [batch_size, l_ext]
# Storage: Specify vdim extents
s_ext = Extent([batch_dim], lambda b: lens[b])
storage_format = [batch_size, s_ext]
# Define input ragged tensor
dims = [batch_dim, len_dim]
A = input_tensor(dims, storage_format)
# Express computation
B = compute(dims, loop_exts, lambda i, j: 2 * A[i, j])

################ Scheduling primitives ################
pad_loop(B.loops[1], 2)
pad_dimension(B.dimensions[1], 4)
fuse_loops(B.loops[0], B.loops[1])
```

Listing 5.1: Operator in Figure 5.1 expressed in a simplified version of CoRa's API.

does not allow reordering such a loop $L_v$ beyond any of the loops $L_1$ to $L_k$. Such a reordering would introduce invalid iterations in the iteration space that would need to be skipped over with the used of conditional expressions. Although possible with the introduction of conditional statements, we have not found a use case for such reordering.

### Operation Splitting

It can sometimes be beneficial to differently schedule different iterations of a loop in a vloop nest in order to more optimally handle the variation in loop bounds. CoRa allows one to split an operation into two or more operations by specifying split points for one or more of its loops, as Figure 5.5 shows. In our evaluation (§5.6.3), we use this transformation in conjunction with horizontal fusion (described below) to better handle the last few iterations of a tiled loop without the need for additional padding in the $QK^T$ and AttnV operators in the transformer layer (Figure 5.3).

### Horizontal Fusion

Past work [73] has proposed horizontal fusion, or *hfusion* for short, as an optimization to better utilize massively parallel hardware devices such as GPUs by executing multiple operators concurrently as part of a single kernel. With CoRa, we implement this optimization in a tensor compiler for the outermost loop of two or more operators. HFusion enables the concurrent execution of the multiple operators that result from using the operation splitting transform described above.

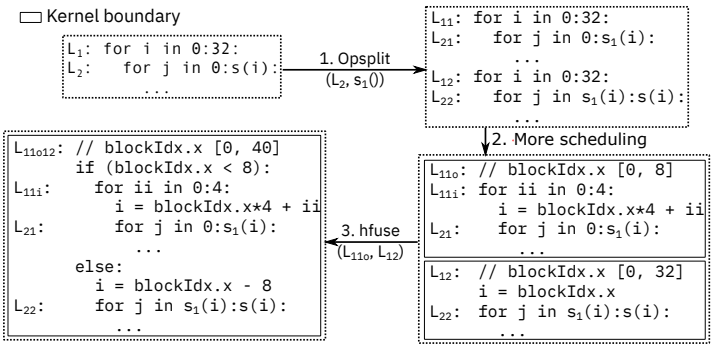



Figure 5.5: Operation splitting and horizontal fusion. Loop $L_2$ is first split in step 1 using operation splitting thus creating two loop nests, which are then horizontally fused together (step 3) so they execute concurrently as part of single kernel.

### Loop and Storage Padding

Despite the overheads of padding, a small amount of it is often useful in order to generate efficient vectorized and tiled code by eliding conditional checks. Accordingly, CoRa allows the user to specify padding for vloops and vdims as multiples of a constant. For example, on lineListing 5.1, the vloop associated with the dimension `len_dim` is asked to be padded to a multiple of 2 while the corresponding dimension of the output tensor is specified to be padded to a multiple of 4 on line 20. Such independent padding specification for loops and the underlying storage is allowed as long as the storage padding is at least as much as the loop padding (this ensures that the padded loop nest never accesses non-existent storage). This ability allows CoRa to fuse padding change operators as is illustrated in Figure 5.3. We show in §5.6.4 that this partial padding does not lead to much wasted computation.

### Tensor Dimension Scheduling

CoRa allows users to split, fuse and reorder dimensions of dense and ragged tensors. This can enable more optimal memory accesses. Fusing tensor dimensions in a way that mirrors the surrounding loop nest can allow for simpler memory accesses (§5.4.1).

### Load Balancing

The variable loop bounds in a vloop nest can lead to unbalanced load across execution units. As proposed by past work [41] on sparse tensor algebra, CoRa allows the user to redistribute work across different parallel processing elements by specifying a *thread remapping policy*. Given a parallel loop, this allows the user to specify a mapping between the loop iterations and the thread id (illustrated in Figure B.1 in the appendix). Depending on the hardware scheduling policy, this can influence the order in which the loop iterations are scheduled and lead to non-trivial performance gains as shown in §5.6.1.

In conclusion, CoRa provides familiar and simple interfaces to users, extended with a few abstractions and scheduling primitives specific to ragged tensors, enabling their application to support (efficient) ragged operations.

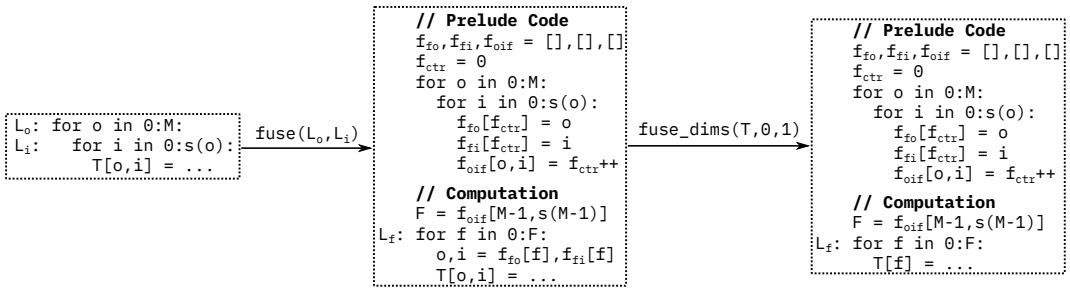



Figure 5.6: Fusing vloops and tensor dimensions.

## 5.4 CoRa's Ragged API Lowering

We now discuss some aspects of CoRa's Ragged API lowering that generates the SSA-based IR as shown in Figure 5.4.

### 5.4.1 Loop and Tensor Dimension Fusion

Consider the ragged loop nest shown on the top left corner of Figure 5.6. The loop bound of the inner loop `Li` is a function $s()$ of `o`, the iteration variable of the outer loop `Lo`. The loop `Lf` obtained by fusing `Lo` and `Li` is shown on the right of the figure. The loop bound `F` of the fused loop would be equal to $\sum_{o=0}^{M-1} s(o)$. Further note that while we have fused the loops `Lo` and `Li`, the tensor access `T[o,i]` in the body of the loop nest still uses variables `o` and `i`. Therefore, we need to compute the values of these two variables corresponding to the current value of `f`, the iteration variable of `Lf`. Because of the ragged nature of the loop nest, computing the loop bound `F` as well as the mapping between the iteration variables of the original and the fused loop nests is not straightforward. In CoRa, we generate code to compute these quantities and variable relationships (shown in the right pane of Figure 5.6) as part of the prelude which executes before the main kernel computation. We use vloop fusion as described above to implement the linear transformation operators (Proj1, Proj, FF1 and FF2) in the transformer encoder (Figure 5.3) with minimal padding.

Suppose now that the tensor `T` in Figure 5.6 has a storage format that mirrors the loop nest consisting of `Lo` and `Li`. This means that the 2-dimensional tensor has an outer cdim and an inner vdim the size of the $i^{th}$ slice of which is $s(i)$. Fusing these dimensions then enables CoRa to simplify the tensor access as shown in the bottom left pane of the figure.

### 5.4.2 Bounds Inference

We saw in Chapter 3, how a tensor compiler needs to perform a bounds inference pass in order to loop extents for all operators. Below, we describe some modifications CoRa makes to this pass.

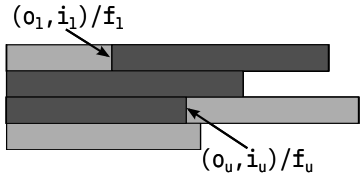



Figure 5.7: Iteration variable ranges during vloop fusion.

### Variable Loop Fusion

As we saw in §5.4.1, the application of scheduling transformations such as fusion can lead to a situation where the variables used in the tensor accesses in an operator's body are not the same as the loop iteration variables present after the transformations have been applied. This means that during bounds inference, one has to repeatedly translate iteration variable ranges between the transformed and the original variables. This is straightforward in the case of cloops, but gets slightly harder in the case of vloop fusion. For the loop nest in Figure 5.6, we provide the rules to translate between the ranges of iteration variables `o`, `i` and `f` below, and they are further visualized in Figure 5.7.

$$
\begin{aligned}
o \in [o_l, o_u] \land i \in [i_l, i_u] &\rightarrow f \in [f_{oif}(o_l, i_l), f_{oif}(o_u, i_u)] \\
f \in [f_l, f_u] &\rightarrow o \in [f_{fo}(f_l), f_{fo}(f_u)] \\
f \in [f_l, f_u] \land f_{fo}(f_l) \neq f_{fo}(f_u) &\rightarrow i \in [0, s(o)] \\
f \in [f_l, f_u] \land f_{fo}(f_l) = f_{fo}(f_u) &\rightarrow i \in [f_{fi}(f_l), f_{fi}(f_u)]
\end{aligned}
$$

Here, $s()$ represents the variable loop bound of the inner loop, while $f_{oif}$, $f_{fo}$ and $f_{fi}$ represent the relationships between the variables `o`, `i` and `f` such that $f_{fo}(f_0)$ and $f_{fi}(f_0)$ evaluate to values of `o` and `i`, respectively, corresponding to `f`$= f_0$. Similarly, $f_{oif}(o_0, i_0)$ evaluates to $f_0$[4].

### Named Dimensions

In §5.3, we described how the user uses named dimensions to specify relationships between loops as well as tensor dimensions. These dimensions play an important part in bounds inference as well. Along with the translation between fused and unfused loop iteration variables described above, one also needs to translate ranges of variables across producers and consumers during bounds inference. In CoRa, we use named dimensions to easily identify corresponding iteration variables across such producers and consumers to allow this translation.

[4] In the generated code, as seen in the right pane of Figure 5.6, these functions take the form of arrays initialized by the prelude. Further, the computation of the `f_oif` array can, in most cases, be optimized away to only compute the loop bound `F` of the fused loop.

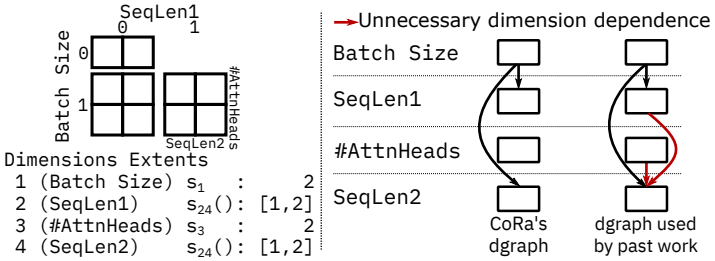



Figure 5.8: CoRa precisely models dimension dependences as compared to past schemes for sparse tensors.

### 5.4.3 Storage Access Lowering

In this section, we briefly describe how CoRa lowers accesses to ragged tensors. Consider the 4-dimensional attention matrix $X$ involved in a batched implementation of MHA shown in the left pane of Figure 5.8. Here, the first and the third dimensions are cdims and correspond to the batch size and the number of attention heads, respectively. The other two dimensions, corresponding to sequence lengths, are vdims.[5] For $X$, the size of a slice for both these vdims is the same function ($s_{24}()$) of the outermost batch dimension.

Due to the irregular nature of ragged tensor storage, we need some auxiliary data structures to be able to lower memory accesses to $X$. The lowering scheme used by past work on sparse tensors [20, 119] assumes that the number of non-zeros in a slice of a sparse dimension is, in general, a function of all outer dimensions. However, recall that for our example tensor $X$, the size of a slice of either vdim depends only on the outermost batch dimension. Being agnostic to such precise dependences between tensor dimensions (as illustrated via the *dimension graphs*, or *dgraphs* in Figure 5.8), past work would compute and store more auxiliary data as compared to CoRa.

CoRa's lowering scheme allows for cheap $O(1)$ accesses to ragged tensors. To enable this, we need to compute a memory offset within a constant number of operations. The reason sparse tensor formats such as the CSR format do not allow constant time tensor accesses is because they explicitly store indices of one or more dimensions along with every non-zero value. Thus, given a tensor index, one needs to perform a search over these stored indices to obtain the correct non-zero element. In the case of ragged tensors, however, we note that within a vdim slice, the data in densely packed with no intervening zero elements. Therefore, we can get away without storing explicit indices for any dimension. Accessing the precomputed memory offsets is also a constant time operation as CoRa's auxiliary data structures store these offsets using simple arrays.

We describe these lowering schemes further in the appendix in §B.2.1. In short, our storage access lowering scheme reduces the amount of auxiliary data that needs to be computed thus reducing the memory and computation overheads of the prelude code (§5.6.4), while allowing cheap tensor accesses.

[5] We use the same layout in CoRa's implementation in §5.6.2.

Table 5.2: Experimental environments used for evaluating CoRa.

| Hardware | Software (All instances ran Ubuntu 20.04.) |
|---|---|
| Nvidia Tesla V100 GPU (Google cloud n1-standard-8 instance) | CUDA 11.1, cuDNN 8.2.1, PyTorch 1.9.0, FasterTransformer v4.0 (commit dd4c071) |
| 8 core, 16 thread Intel CascadeLake CPU (Google cloud n2-standard-16 instance) | Intel MKL (v2021.3) |
| 8 core ARM Graviton2 CPU (AWS c6g.2xlarge instance) | PyTorch 1.10.0a0+git36449ea (with oneDNN 2.4 and Arm compute library 21.11), TensorFlow 2.6.0 (with oneDNN 2.3 and Arm compute library 21.05), OpenBLAS 0.3.10 |
| 64 core ARM Graviton2 CPU (AWS c6g.16xlarge instance) | |

## 5.5 Implementation

We prototype CoRa by extending TVM v0.6. Some details regarding this implementation are discussed below.

### Ragged API

Our prototype allows vdims to depend on at most one outer tensor dimension. This is not a fundamental limitation and can easily be overcome, though we have not needed to for our evaluation. We implement the operator splitting and hfusion transforms for non-reduction loops.

### Lowering

Our current prototype does not auto-schedule the expressed computation. The evaluation therefore uses implementations optimized using a combination of manual scheduling and grid search. For some operators, we auto-scheduled the corresponding dense tensor operator using past work [156] and manually applied the schedule to the ragged case. We find that this works well in most cases and therefore believe that the prototype could readily be extended with prior work on auto-scheduling (we discuss this further in Chapter 6). Our implementation currently expects users to correctly allocate memory (taking into account padding requirements as specified in the schedule) for tensors. Checks to report problems with this memory allocation could be easily implemented.

## 5.6 Evaluation

We evaluate CoRa against state-of-the-art baselines, first, on two ragged variants of the gemm (general matrix multiplication) operation in §5.6.1 and then on an encoder layer of the transformer model (Figure 5.3) in §5.6.2. Our experimental environment is described in Table 5.2. Below, we refer to the four platforms listed in the table as Nvidia GPU, Intel CPU, 8-core ARM CPU and 64-core ARM CPU. Our evaluation is performed with single-precision floating point numbers.

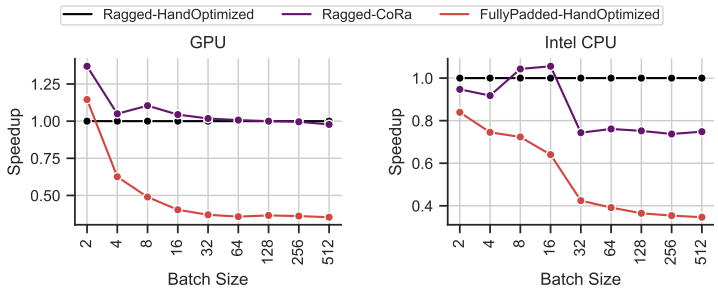



Figure 5.9: Performance comparison of CoRa's vgemm and hand-optimized implementations of vgemm and fully padded gemm.

### 5.6.1 Matrix Multiplication

We start by evaluating CoRa's performance on the variable-sized batched gemm (or vgemm) and the triangular matrix multiplication (or trmm) operators. As with all the implementations we compare against, the CoRa implementations of these operators use fully padded storage for all tensors.

#### Variable-Sized Batched Gemm

The vgemm operator consists of a batch of gemm operations, each with different dimensions. For this operator, we evaluate CoRa on the Nvidia GPU and Intel CPU backends and compare against hand-optimized implementations of vgemm and fully padded batched gemm in both cases. On the CPU, we compare against Intel MKL's implementations while on the GPU, we compare against past work [74] on vgemm and cuBLAS's implementation of fully padded batched gemm. We use synthetically generated data where matrix dimensions are uniformly randomly chosen multiples of 128 in $[512, 1408]$. CoRa's CPU implementation offloads the computation of inner gemm tiles to MKL, allowing us to obtain computational savings due to raggedness while also exploiting MKL's highly tuned microkernels. As Figure 5.9 shows, CoRa is effectively able to exploit raggedness on both CPUs and GPUs, performing as well as or better than the hand-optimized implementation on the GPU and obtaining better than 73% of the performance of MKL's vgemm for all batch sizes and performing better on a couple on the CPU. In all cases, CoRa is significantly better than the fully padded gemm operations, which perform worse at higher batch sizes as there is more wasted computation as batch size goes up for the batch sizes evaluated.

#### Triangular Matrix Multiplication

A triangular matrix, i.e. a matrix where all the elements above (or below) the diagonal are zero, can be thought of as a ragged tensor because all non-zero elements in a row are densely packed and their number per row is a function of the row index. Operations on triangular matrices can, thus, be effectively expressed and optimized using CoRa. In this section, we evaluate CoRa on the trmm operator wherein we multiply a square lower triangular matrix with a square dense matrix, on the Nvidia GPU. We compare against cuBLAS's trmm and fully padded gemm implementations. In trmm, the reduc-

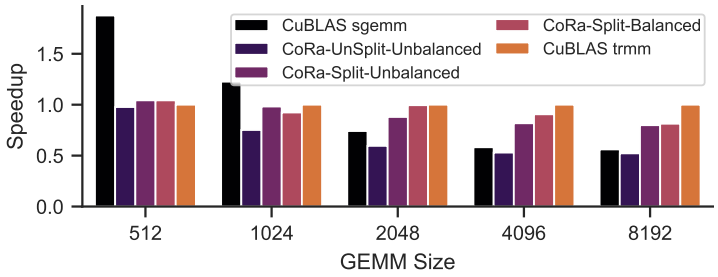



Figure 5.10: CoRa's trmm performance compared against cuBLAS's hand-optimized trmm and fully-padded gemm implementations.

Table 5.3: Datasets used for evaluating CoRa.

| Dataset (Short name, if any) | Sequence length statistics | | |
|---|---|---|---|
| | Minimum | Mean | Maximum |
| RACE [69] | 80 | 364 | 512 |
| English Wikipedia with SeqLen 512 (Wiki512) | 12 | 371 | 512 |
| SQuAD v2.0 [104] (SQuAD) | 39 | 192 | 384 |
| English Wikipedia with SeqLen 128 (Wiki128) | 14 | 117 | 128 |
| MNLI [142] | 9 | 43 | 128 |
| XNLI [25] | 9 | 70 | 128 |
| MRPC [31] | 21 | 59 | 102 |
| CoLA [140] | 6 | 13 | 37 |

tion loop is a vloop. In order to efficiently handle the last few iterations of this loop after tiling, we use operation splitting[6] (§5.3). Further, the raggedness in this loop leads to imbalanced load across the GPU thread blocks. We use thread remapping (§5.3.1) to schedule thread blocks with the most amount of work first, leading to more balanced load.

Figure 5.10 shows the performance of the aforementioned cuBLAS implementations and three implementations in CoRa—CoRa-unsplit-unbalanced, CoRa-split-unbalanced and CoRa-split-balanced—which progressively employ operation splitting and thread remapping, starting with neither. We see the trmm implementations—both cuBLAS's and CoRa's—are beneficial as compared to cuBLAS's dense sgemm operator only for larger matrices. In all cases, however, the CoRa-split-balanced implementation performs within 81.3% of cuBLAS's hand-optimized trmm implementation. Operation splitting leads to a significant increase in performance by allowing CoRa to elide conditional checks in the main body of the computation. Further, a better load distribution with thread remapping also helps CoRa achieve performance close to cuBLAS.

### 5.6.2 The Transformer Model

We now move on to look at how CoRa performs on various modules of the transformer model. We mainly focus on the GPU backend as it is more commonly used for these models. We use a 6 layer model with a hidden dimension of 512 and 8 attention heads each of size 64. The encoder layer

[6] HFusion is not applicable here as the split loop is a reduction loop and executing the split operators concurrently would require atomic instructions, which our prototype does not yet support.

Table 5.4: Transformer encoder layer execution latencies (in ms) for CoRa, PyTorch and the two manually-optimized variants of FasterTransformer on the Nvidia GPU. CoRa's execution latencies include prelude overheads assuming a 6 layer transformer encoder.

| Dataset | Batch Size | PyTorch | FT | CoRa | FT-Eff |
|---|---|---|---|---|---|
| RACE | 32 | 12.22 | 11.0 | 8.22 | 8.61 |
| | 64 | 24.46 | 21.88 | 15.91 | 16.75 |
| | 128 | 48.73 | 42.26 | 31.45 | 33.61 |
| Wiki512 | 32 | 12.26 | 11.0 | 9.1 | 9.32 |
| | 64 | 24.52 | 22.12 | 17.4 | 17.85 |
| | 128 | 48.72 | 42.43 | 32.17 | 33.66 |
| SQuAD | 32 | 8.17 | 7.56 | 4.15 | 4.69 |
| | 64 | 16.9 | 15.63 | 7.78 | 9.2 |
| | 128 | 34.18 | 30.62 | 15.36 | 17.91 |
| Wiki128 | 32 | 2.79 | 2.45 | 2.59 | 2.28 |
| | 64 | 5.12 | 4.61 | 4.72 | 4.35 |
| | 128 | 10.1 | 9.29 | 8.86 | 8.54 |
| MNLI | 32 | 2.22 | 2.04 | 1.11 | 1.03 |
| | 64 | 4.44 | 4.06 | 1.89 | 1.93 |
| | 128 | 9.53 | 8.86 | 3.53 | 3.78 |
| XNLI | 32 | 2.76 | 2.45 | 1.56 | 1.5 |
| | 64 | 5.13 | 4.62 | 2.94 | 2.86 |
| | 128 | 10.03 | 9.3 | 5.62 | 5.49 |
| MRPC | 32 | 1.85 | 1.73 | 1.32 | 1.27 |
| | 64 | 3.76 | 3.48 | 2.6 | 2.36 |
| | 128 | 7.42 | 6.89 | 4.55 | 4.55 |
| CoLA | 32 | 0.67 | 0.57 | 0.59 | 0.44 |
| | 64 | 1.02 | 0.93 | 0.77 | 0.63 |
| | 128 | 2.37 | 2.18 | 1.26 | 1.17 |

contains two feed-forward layers, the inner one of which has a dimension of 2048. These are the same hyperparameters used in the base model evaluated in [137].

We use sequence lengths from some commonly used NLP datasets listed in Table 5.3 for the evaluation on the transformer model. For each dataset, we use the sequence lengths corresponding to the text obtained after preprocessing as performed in the implementations corresponding to past work on various transformer models [27, 137, 148]. The Wiki512 and Wiki128 datasets, usually used for pre-training [27], are generated from a dump of the English Wikipedia website [141]. Each sequence in these two datasets was created by accumulating consecutive sentences from the dump until a sentence could no longer be added without exceeding the maximum sequence length used for training (which is a hyperparameter). This was done, in the transformer implementation, to reduce wasted computation due to padding as much as possible. As a result, these datasets do not provide as much opportunity for CoRa to exploit as do some of the other datasets. We see this reflected in Figure 5.2 for example. We focus on larger batch sizes (32, 64 and 128) because, as we saw in Figure 5.2, there is lesser opportunity to exploit raggedness for smaller batch sizes and hence other factors such as the quality of the schedules used in CoRa's implementations play a big role. In this section, CoRa's implementations use ragged tensor storage.

Transformer Encoder Layer

We first evaluate the forward pass latency of an encoder layer of the transformer model (Figure 5.3). We compare CoRa's performance with that of FasterTransformer and an implementation in PyTorch, a popular deep learning framework, with TorchScript [97] enabled. All the operators in the encoder layer except the ones in the SDPA sub-module process the hidden vectors associated with each word independently. Therefore, with manual effort, they can be implemented without any padding. The linear transformation operators Proj1, Proj2, FF1 and FF2 reduce to gemm operators in this case. FasterTransformer provides an option to perform this optimization, first introduced in EffectiveTransformers [12]. We compare against FasterTransformer both with and without this optimization. We refer to these two implementations as FT-Eff and FT, respectively. In the CoRa implementation, this optimization is applied simply by loop fusion, analogous to the illustration in Figure 5.6. In CoRa's implementation however, we pad this fused loop so that its bound is a multiple of 64. In other words, we add a padding sequence to the batch to ensure that the sum of the sequence lengths is a multiple of 64. We refer to this kind of padding as *bulk padding* (Figure 5.3). The relative amount of bulk padding added is usually quite low as the sum of sequence lengths in a batch is much higher.

Table 5.4 shows the forward execution latencies for the encoder layer for the aforementioned frameworks and datasets. The auxiliary data structures computed by CoRa's prelude are shared across multiple layers of the model as the raggedness pattern stays the same across layers, depending only on the sequence lengths in the mini-batch. The execution times shown for CoRa include per-layer prelude overheads assuming a 6 layer model. We further look at these overheads in §5.6.4. As we can see, the CoRa implementation is competitive with the manually-optimized FT-Eff implementation for all datasets, even performing better in a few cases, and performs significantly better as compared to the fully-padded PyTorch and FT implementations. Figure 5.11, which plots the overall performance of all these implementations for the batch sizes evaluated, makes this clear.

We now take a closer look at the FasterTransformer and CoRa implementations which are sketched in Figure 5.3.[7] The FT implementation is similar to the FT-Eff implementation except it uses full padding for all operations. The CoRa and FasterTransformer implementations differ in their operator fusion strategies. Therefore, the figure breaks the implementations down to the smallest sub-graphs that correspond to each other. Figure 5.13 shows a breakdown of the execution times for these implementations for the RACE dataset and batch size 128 at the level of these sub-graphs.[8] As Figure 5.3 shows, the FT-Eff and CoRa implementations differ significantly with respect to padding only in the SDPA sub-module where the FT-Eff implementation employs full padding while the CoRa employs partial padding. We see, in Figure 5.13, that the CoRa implementation performs better than FasterTransformer for all the SDPA operators ($QK^T$, Softmax and AttnV) despite the fact that the latter is

[7] FasterTransformer uses specialized implementations for different GPUs. For the purposes of evaluating CoRa, we limit our discussion to its implementation for the Nvidia V100 GPU we use for evaluation.

[8] The raw data for this plot is listed in Table B.5 in the appendix.

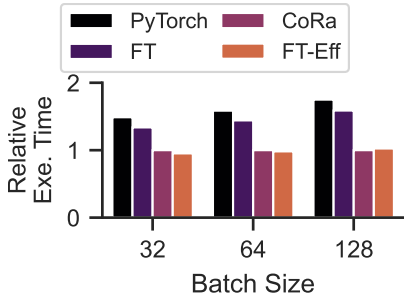



Figure 5.11: Relative GPU execution times for PyTorch, FasterTransformer and CoRa for the transformer encoder layer.

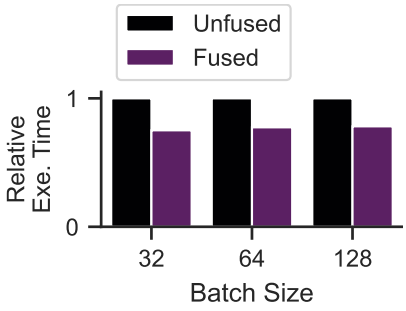



Figure 5.12: Relative MHA execution times with and without layout change operator fusion.

heavily hand optimized.[9] This is because CoRa's ability to handle raggedness enables it to perform less wasted computation. For the remaining operators where both the CoRa and FT-Eff implementations employ little to no padding, we see that the CoRa implementation is usually slower, but often close in performance to the FT-Eff implementation and significantly faster than the fully padded FT implementation. This is expected as FT-Eff calls into cuBLAS's extensively optimized gemm kernels for the linear transformation operators and into hand-optimized kernels for the rest. CoRa's performance drops slightly for datasets with smaller sequence lengths as well as for smaller batch sizes. As we discuss in §B.4.7, this performance difference can be reduced by further optimizing the schedules used for the projection and feed forward operators in CoRa's implementation for smaller batch sizes and sequence lengths. Further, we also note that the overheads associated with the prelude code and partial padding (§5.6.4) play a larger role in these cases, further contributing to increased execution latencies.

FasterTransformer's reliance on vendor libraries prevents it from fusing any of the gemm operations with surrounding elementwise operators, which CoRa can due to its compiler-based approach. Specifically CoRa can completely fuse all operators which add or remove padding in its implementation (as shown in Figure 5.3). This is as opposed to the FT-Eff implementation, which cannot. Fusing these padding change operators leads to a significant drop in CoRa's execution latency as seen in Figure 5.12, which shows the execution latencies of the MHA module for the RACE dataset in CoRa with and without this fusion enabled.

### Masked Scaled Dot-Product Attention

The decoder layer of a transformer uses a variant of MHA called *masked MHA* wherein the upper half of the attention matrix is masked for all attention heads during training. This masking only affects the SDPA module, the operators in which can now be seen as computing on a batch of lower triangular matrices. We saw in §5.6.1 that CoRa can effectively generate code for operations on triangular matrices. For batch size 128, an implementation of masked SDPA in CoRa which exploits this masking performs 1.56× faster than an implementation which does not for the RACE dataset and 1.29× for

[9]The execution times of the three SDPA operators is quadratically proportional to the sequence length, unlike the remaining operators which are linearly proportional. We discuss the performance of SDPA further in §B.4.7 of the appendix.

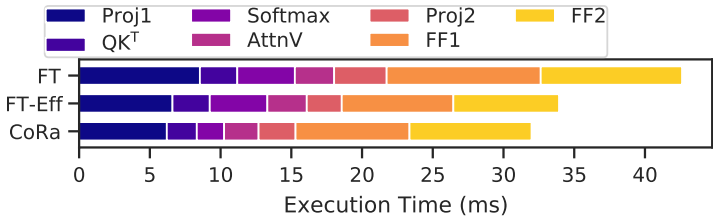



Figure 5.13: Breakdown of the encoder layer execution times for the RACE dataset at batch size 128. This data is obtained with profiling turned on and might deviate from Table 5.4.

Table 5.5: MHA execution latencies (in ms) on the 64-core ARM CPU for TensorFlow and CoRa.

| Dataset | Batch Size 32 | | | Batch Size 64 | | | Batch Size 128 | | |
|---|---|---|---|---|---|---|---|---|---|
| | TF | TF-UB | CoRa | TF | TF-UB | CoRa | TF | TF-UB | CoRa |
| RACE | 55 | 46 | **44** | 111 | 88 | **85** | 209 | **156** | 168 |
| Wiki512 | 53 | 53 | **47** | 106 | 96 | **91** | 205 | **172** | 176 |
| SQuAD | 35 | 27 | **20** | 68 | 49 | **39** | 137 | 79 | **76** |
| Wiki128 | 11 | 11 | **9** | 19 | 18 | **17** | 34 | 33 | **33** |
| MNLI | 9 | 9 | **4** | 16 | 14 | **7** | 30 | 23 | **14** |
| XNLI | 11 | 11 | **6** | 18 | 18 | **11** | 34 | 28 | **22** |
| MRPC | 9 | 8 | **5** | 14 | 14 | **10** | 26 | 23 | **18** |
| CoLA | 5 | 4 | **2** | 6 | 6 | **3** | 9 | 8 | **5** |

the MNLI dataset. The benefits are less pronounced for the MNLI dataset, which has smaller sequence lengths, as we pad vloops to be multiples of a constant regardless of the dataset. We provide more data and discussion on the implementation of masked SDPA in §B.4.2 in the appendix.

### Memory Consumption

We find that the use of ragged tensors leads to an overall 1.78× drop in the size of the forward activations (computed analytically) of the encoder layer across all datasets at batch size 64 (more details in §B.4.4). The reduction, however, is not uniform across the datasets and those with higher mean sequence lengths, such as Wiki512 and Wiki128, see only small benefits. Forward activations often consume significant memory during training. The use of ragged tensors can help alleviate the resulting memory bottlenecks along with other memory management techniques for training [57, 64].

### MHA Evaluation on ARM CPU

Table 5.5 shows the execution latencies of MHA implementations in TensorFlow and CoRa on the 64-core ARM CPU. We evaluate against two execution configurations of TensorFlow—TF, where the entire mini-batch is executed at once and TF-UB, where the mini-batch is executed as a series of smaller *micro-batches*, which enables execution with lower padding. Across the datasets and batch sizes evaluated, we see that CoRa's implementation is overall 1.57× faster than TF and 1.37× faster than TF-UB. In this case, too, CoRa's ability to save on wasted computation due to padding leads to significant performance gains over a popular deep learning framework. §B.4.7 of the appendix more extensively

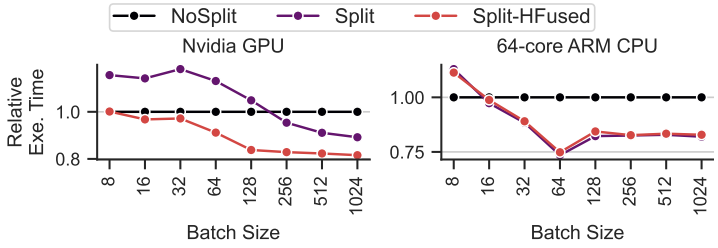



Figure 5.14: Benefits of operator splitting and hfusion for the AttnV operator. Note that the y-axis does not start at 0.

compares the performance of TensorFlow and PyTorch against CoRa on both the 8- and 64-core ARM CPUs.

### 5.6.3 Operation Splitting and Horizontal Fusion

We now evaluate operator splitting and hfusion on the AttnV operator, which is an instance of the vgemm problem. AttnV has two vloops, one of which is a reduction loop. We apply the optimizations to the non-reduction vloop allowing us to use a larger tile size (we use 64) without padding the vloop bound to be a multiple of this tile size. This especially benefits datasets with sequence lengths comparable to the tile size, such as MNLI. For this dataset, Figure 5.14 shows the relative execution times of three CoRa implementations of AttnV—NoSplit, Split and Split-HFused—in which we progressively perform the two optimizations, on the Nvidia GPU and 64-core ARM CPU backends. On the GPU, operation splitting causes a slowdown despite lower wasted computation as it reduces parallelism, which is restored by hfusion. This is more apparent at lower batch sizes when the amount of parallelism is lower. The effects of reduced parallelism due to operation splitting are less apparent on the CPU as it exposes lower levels of hardware parallelism. The lower CPU parallelism also mean that hfusion has no benefit in this case. We also evaluate these optimizations on the $QK^T$ operator in §B.4.5 in the appendix.

### 5.6.4 Overheads in CoRa

Let us now look at the sources of overheads in CoRa—the prelude code, the wasted computation due to partial padding and auxiliary data structure accesses in the generated code.

#### Prelude Overheads

The prelude code constructs the required auxiliary data structures (§5.4) and copies them to the accelerator's memory if needed. The table below lists the execution time (in ms) and memory (in kB) required for these tasks for a 6-layer transformer encoder on the GPU backend. It also shows the overheads associated with the storage lowering scheme used in past work we discussed in §5.4.3 (referred to as Sparse Storage in the table). As compared to this scheme, we see that CoRa's specialized lowering scheme significantly reduces the resources required to compute the data structures associated with tensor storage.

| Dataset / Batch Size | Sparse Storage Time / Mem. | CoRa Storage Time / Mem. | CoRa Loop Fusion Time / Mem. | CoRa-Copy Time |
|---|---|---|---|---|
| CoLA / 32 | 0.09 / 267.97 | 3.80e-03 / 2.93 | 5.35e-03 / 32.15 | 0.24 |
| CoLA / 128 | 0.35 / 1047.22 | 5.76e-03 / 11.18 | 0.02 / 104.22 | 0.27 |
| RACE / 32 | 0.52 / 1607.97 | 4.15e-03 / 2.93 | 0.09 / 666.54 | 0.42 |
| RACE / 128 | 2.02 / 6300.02 | 6.30e-03 / 11.18 | 0.34 / 2609.58 | 0.99 |

The overheads associated with loop fusion are higher than those associated with storage as we need to compute and store the relationship between all values of the fused and unfused loop iteration variables (§5.4.1). Copying the generated data structures to the GPU's memory is, however, the major source of the overhead. Overall, the overheads range from 0.7% (RACE dataset at batch size 128) to about 7% (CoLA dataset at batch size 32) of the total execution time of the encoder layer on the GPU. On the CPU, the overheads are a very small fraction of the execution times, because the execution times are much higher and because the memory copy costs are absent. We discuss some simple optimizations to reduce prelude overheads in §B.4.6 of the appendix.

#### Partial Padding Overheads

We saw that in CoRa, small amounts of padding can be specified for vloops (both unfused vloops and fused ones with bulk padding) and tensor storage to enable efficient code generation. While this leads to some wasted computation, we find that it is generally quite low. For the transformer encoder layer, we see a 3.5% increase in the amount of computation (computed analytically) over the ideal case without padding for a batch size of 32 and a 2.3% increase for a batch size of 128 across all the datasets evaluated. The overheads decrease with increasing batch size as bulk padding ensures that the sum of the sequence lengths in a batch is a multiple of a constant (64, in this case) irrespective of the batch size leading to a higher relative amount of padding at lower batch sizes. We provide further data and discussion in §B.4.6 of the appendix.

#### Ragged Tensor Overheads and Load Hoisting

CoRa's generated code accesses the auxiliary data structures generated by the prelude leading to frequent indirect memory accesses. We measure the overheads caused by these accesses for the operators used in MHA. While the data and more discussion are provided in §B.4.6, we note here that the indirect memory accesses do not cause any significant slowdown for the Proj1, Softmax, AttnV and the Proj2 operators. The accesses do lead to a higher slowdown in the $QK^T$ operator, which is the only operator where we fuse two vloops leading to complex memory access expressions. For this case, we find that hoisting data structures accesses outside loops when possible helps recover the lost performance.

### 5.6.5 Evaluation Against Sparse Tensor Compilers

We saw that ragged tensors are similar to sparse tensors as both involve irregular storage. In order to evaluate the suitability of using sparse tensor compilers for ragged tensors, we compared CoRa's perfor-

mance with Taco, a state-of-the-art sparse tensor compiler. Specifically, we measured the performance of a few operators on triangular matrices implemented in Taco using the CSR and blocked CSR formats. We provide more detailed discussion in §B.4.3, but note here that these implementations showed slowdowns ranging from 1.33× to 95.37× compared to the corresponding CoRa implementations. As we discuss later, this is essentially due to a mismatch between the use case of ragged tensors and the general sparse tensor computations that Taco is designed for. For example, ragged tensors are usually much denser as compared to traditionally used sparse tensors and the applications each is used in are quite different.

## 5.7 Related Work

### Tensor Compilers

Past work on both sparse and dense tensor compilers (Chapter 2) has informed CoRa's design. We generalize the abstractions provided by dense tensor compilers to ragged tensors, while enabling efficient code generation for the latter. We discuss in §5.6.5 and further in §B.4.3, how despite the similarity between ragged and sparse tensors, sparse tensor compilers are unable to effectively exploit the properties of ragged tensors to enable efficient execution.

CoRa techniques are complementary to a lot past work on optimizing control flow dynamism, such as Nimble, Janus, Terra and PyTorch's LazyTensor. CoRa can therefore potentially be used as part of its pipeline. CoRa's use of uninterpreted functions and named dimensions has been inspired by their use in Cortex which (as discussed in §3.7) is itself based on the Sparse Polyhedral Framework [79, 82, 123]. Named dimensions are also similar to the index labels in Comet. CoRa implements a limited form of the hfusion optimization, first proposed in [73], as part of a tensor compiler.

### Deep Learning Frameworks and Graph Optimizations

Deep learning frameworks have recently begun adding support for ragged tensors with the RaggedTensor [129] class in TensorFlow and the NestedTensor [98] module for PyTorch. Very few operators are, however, supported for ragged tensors at this point [95, 128].[10] CoRa can be used to expand the set of ragged operators supported in these frameworks. CoRa's techniques are complementary to graph optimizations for efficient deep learning execution such as data layout optimizations [56], kernel fusion [157] and operator scheduling [30], and can be used in conjunction with them.

### Sparse Tensor Algebra

There have been decades of past work on efficient execution of sparse tensor operators. This work has been revisited recently in the context of deep learning by work on exploiting block sparsity in model

[10] Tensor contraction and similar operators such as batched gemm and convolution are generally not supported. PyTorch's NestedTensor further supports only a few elementwise and reduction operators [96] at this point.

weights [46] as well as for tuning sparse kernels for the sparsity patterns and distributions usually encountered in deep learning [41]. The thread remapping strategy discussed in §5.3.1 was implemented first in [41].

CoRa's compiler-based approach further improves over the hand-optimized ragged tensor operator implementations we described in Chapter 2.

## 5.8 Chapter Summary

This chapter presented CoRa, a tensor compiler for expressing and optimizing ragged operators to portably target CPUs and GPUs using simple and familiar abstractions. CoRa's approach, specialized for ragged tensors, reduces overheads associated with techniques such as masking and padding. With deep learning being applied to an ever-increasing set of fields and the models getting more resource-intensive, we believe that efficiently handling the shape dynamism that naturally arises in many settings is important. CoRa extends past work on tensor compilers by supporting efficient operators on ragged tensors. Our work can also be seen as a step towards unifying past work on sparse and dense tensor compilation. In the future, we plan to make CoRa easier to use, potentially with the help of auto-scheduling techniques.

# 6 Conclusion and Future Directions

In this thesis, we have tried to tackle the problem of performing efficient auto-batching for deep learning computations in the presence of dynamism. While we believe that the work presented here makes significant progress towards this goal, there a lot of avenues for futher improvements. Below, we describe a few.

## 6.1 Improvements to ACRoBat

### 6.1.1 Auto-batching During Backpropagation

We belive that ACRoBat's techniques, as described in Chapter 4, can be applied to both training and inference of deep learning models. However, we have implemented and evaluated the techniques primarily for inference in ACRoBat's prototype. Extending the techniques to training would involve some non-trivial challenges.

Fully dynamic frameworks such as DyNet and TensorFlow's eager mode rely on a gradient tape in order to support backpropagation. On the other hand, extensions to ACRoBat to support training would have to employ static auto-differentation techniques in order to allow compiler optimizations for the backward pass of the computation. Due to the automated generation of the backward pass of the computation, ACRoBat would no longer have user annotations (which currently enable ACRoBat to exploit control flow parallelism, and exploit program phases, for instance) for this pass. One would therefore have to develop further static analyses in order to reduce ACRoBat's reliance on user annotations. Once again, past work on traditional compilation would be a great point of reference here. Further, as implemented in TVM, Relay's auto-differentation transformations rely heavily on state mutations. ACRoBat's analysis and transformations have currently only been implemented for the functional subset of Relay and hence do not support state mutations. Extending these to handle state mutations would be straightforward[1], but might involve a non-trivial amount of implementation effort.

### 6.1.2 Other Improvements

As described in Chapter 4, ACRoBat needs to auto-schedule kernels with dynamic shapes. While it uses some of DietCode's techniques, a tighter integration between the two will certainly lead to better

[1] As we have discussed, ACRoBat's static analyses derive heavily from traditional compilation techniques, most of which have been developed for imperative programming languages and IRs and therefore support state mutations by design. Extending ACRoBat's analysis to also handle mutations, would therefore be straightforward.

end-to-end performance. ACRoBat's prototype currently only supports GPU execution. This can also be extended to enable ACRoBat to target a wider variety of hardwares.

## 6.2 Improvements and Extensions to CoRa

### 6.2.1 Autoscheduling for Ragged Tensor Compilation

As described in Chapter 5, CoRa currently does not support auto-scheduling for ragged tensor operators. With the area of auto-scheduling in the presence of dynamic tensor shapes just being started to be explored, such approaches can be further extended in order to support ragged tensor computations. We already saw in Chapter 5, how in order to optimize some ragged kernels for CoRa's evaluation, we manually applied schedules generated by TVM's auto-scheduler for a corresponding dense tensor kernel. This suggests that auto-scheduling techniques could work well for ragged kernels. Extending such techniques to ragged tensor kernels would require working with tensor size distributions due to the large number of unknown tensor shapes (each tensor in the batch, for a batched transformer, for example).

### 6.2.2 Graph Optimizations for Ragged Tensors

CoRa develops techniques to enable the optimization and code generation for individual ragged tensor operators. Deep learning practitioners would however not be able to benefit from CoRa unless further abstractions and techniques have been designed to support ragged tensor operators in the deep learning frameworks today. This includes enabling graph-level optimizations such as tensor layout selection, kernel fusion and memory planning to ragged tensor operators. The use of sparse tensors in deep learning has also been steadily rising as deep learning models become larger and larger. Due to the similaties between sparse and ragged tensors, general abstractions for can be developed that support working with both kinds of tensors simultaneously.

## 6.3 Auto-batching for Control Flow and Shape Dynamism Simultaneously

In this thesis, we have discussed techniques to enable efficient auto-batching for deep learning models exhibiting either control flow or shape dynamism. However, deep learning computations such as beam search decoding with transformers [139] can exhibit both kinds of dynamism simultaneously. We now briefly discuss how the techniques we have described can enable auto-batching for such computations as well.

In order to exploit batch and instance parallelism in the presence of dynamic control flow, we would need to rely on ACRoBat's techniques. Due to the shape dynamism exhibited by the input computation, however, the batch kernels ACRoBat's generates will now need to perform computations on ragged tensors. Here's we can rely on CoRa's infrastructure to generate efficient code for the batched kernels. We discussed already how it should be possible to extend current work on auto-scheduling techniques to

handle ragged operators as well. This will then enable us to automatically optimize the ragged batched kernels ACRoBat generates, thereby enabling efficient auto-batching for computations simultaneously exhibiting both control flow and shape dynamism.

## 6.4 Beyond Auto-Batching

Auto-batching is just one of the many problems that arise when supporting dynamism in deep learning computations. We believe that the techniques developed in this thesis can be applied fruitfully in order to solve other related problems as well. We discuss briefly two of these problems below.

### 6.4.1 Memory Planning

Memory planning is an important optimization step, both during training as well as inference. The presence of memory allocation operations during inference often reduces performance [113]. Further, on edge devices, memory is a scarce resource and thus memory planning is essential [26]. During training, one needs to persist the intermediate tensor activations of the forward pass as they are used during the backward pass. As a result, memory is often also a scarce resource when training large models. Techniques such as rematerialization and tensor swapping are often used to alleviate these problems during training, but have been explored mostly for the case of static deep learning computations.

We believe that traditional compiler techniques, as used in ACRoBat, can also be effective here. Escape analysis [19, 122] can be useful in uncovering opportunities for tensor reuse in the presence of control flow dynamism. Planning when to swap out tensors during training and when to swap them back in can benefit from work in prefetching. Graph-level abstractions to support ragged tensors, as discussed above, can also help extend current memory planning techniques to ragged tensors.

### 6.4.2 Graph-level Optimizations

Current deep learning frameworks usually perform optimizations such as kernel fusion over one or more DFGs extracted from the input computation (Chapter 2). While this is an easy way to extend the scope of existing graph-level optimizations to computations exhibiting control flow dynamism, it also limits the scope of the optimizations. In order to support efficient execution of control flow dynamism, we need to develop techniques to perform such optimizations while not being limited by the boundaries of control flow. This includes, for example, being able to perform kernel fusion across control flow structures such as conditional statements and function calls. Similarly, other optimizations which are often expressed as graph rewrites (such as the ones explored in [60]) today would also need to be extended to work with general control flow dynamism.

## 6.5 Conclusion

In this thesis, we first surveyed control flow and shape dynamism in deep learning computations in Chapter 2. For such dynamic deep learning computations, we have developed compiler and runtime-based techniques to enable efficient auto-batching. First, we looked at the specialized class of recursive computations. We identified how such computations can easily be separated into the lightweight but recursive data structure linearization and the tensor computations. This allowed us to design Cortex, a compiler for expressing and optimizing such recursive computations. Cortex's end-to-end compilation of the computations enables it to achieve up to 14× better performance over generalized dynamic batching implementations. Next, we described ACRoBat in Chapter 4, as we attempted to generalize Cortex's insights to handle unrestricted control flow. This broadening of scope meant that we had to co-design both static and dynamic program analysis as opposed to Cortex's aggressive focus on static optimizations. Despite this, we saw how ACRoBat achieves up to 8× faster execution as compared to past work. Changing our focus to shape dynamism, we looked at how we can go about generating efficient implementations of ragged tensor operators in Chapter 5. We saw that due to similarities between ragged and dense tensor computations, CoRa could reuse large parts of dense tensor compilation infrastructure, while generalizing aspects relating to tensor storage and loop transformations. We also saw that a CoRa-generated implementation of the transformer model performed on-par with a highly hand-optimized implementation of the same.

Thus, we believe that the work described in this thesis makes significant progress towards enabling efficient auto-batching of dynamic deep learning computations. We hope that work in this direction continues, enabling deep learning researchers to propose more expressive models to tackle problems.

# Appendices

# A Appendix for Cortex

## A.1 Caching Tensors Indexed by Non-Affine Expressions

We saw in §3.4.1 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 occurs when a 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 3.2, for the purposes for the accesses `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.

```
for b_idx = 0:num_internal_batches:
  for n_idx = 0:batch_sizes[b_idx]:
    node = internal_batches[b_idx, n_idx]
    for i = 0:256:
      rnn_cache[b_idx, n_idx, i, 0] = rnn[left[node], i]
      rnn_cache[b_idx, n_idx, i, 1] = rnn[right[node], i]

for b_idx = 0:num_internal_batches:
  for n_idx = 0:batch_sizes[b_idx]:
    node = internal_batches[b_idx, n_idx]
    for i = 0:256:
      rnn[node,i] = tanh(rnn_cache[b_idx, n_idx, i, 0] +
                         rnn_cache[b_idx, n_idx, i, 1])
```

## A.2 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 loop-carried dependencies in the generated ILIR code. This can be seen in the

generated ILIR for the running example, in Listing 3.3. In the listing, the data written to tensor `rnn` in loops `L2` and `L7` is read by loops `L5` and `L6`. This dependence only exists across a node and its children. We are also guaranteed, by the properties described in §3.1 and the way the data structure linearizer works, that no node in batch may be a child of any other node in the same batch. The, therefore, 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 `L3`. We therefore designed a modification to the pass to insert the barrier in the outer loop which actually carries the dependence.

## A.3 Other Optimizations during ILIR Lowering

Below, we discuss a couple less important optimizations and scheduling knobs we implemented.

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

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

## A.4 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 allows us to lower the batches into two arrays — `batch_begin` and `batch_length`, which store the starting node and the length, respectively, of every batch. Thus, node `n` is in batch `i` if `batch_begin[i] <= n < batch_begin[i] + batch_length[i]`. This numbering scheme also ensures that all leaf nodes are numbered higher than all internal nodes. This reduces the cost of checking if a node is a leaf. When nodes are numbered this way, a leaf check involves a single comparison as opposed 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 is arbitrary. This scheme thus generally reduces the overheads of iterating over batches and leaf checks.

## A.5 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 leads to high register pressure. We find that recursive unrolling precludes us from using persistence for the TreeLSTM and TreeRNN models discussed in

§3.6.4. 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 [105] and [109] can potentially be applied in our context to alleviate this issue.

# B Appendix for CoRa

We now look at additional details regarding CoRa's mechanism in §B.1, §B.2 and §B.3, and discuss further aspects of the evaluation in §B.4. Notably, we look at how CoRa can exploit masking in masked MHA to obtain further savings in §B.4.2, discuss how CoRa's overheads are quite low, allowing it to effectively exploit raggedness (§B.4.6) and look more closely at CoRa's performance on the transformer model and where the benefits come from in §B.4.7.

## B.1 Ragged API

### B.1.1 Thread Remapping Policy

We discussed, in §5.3, that CoRa allows users to specify a thread remapping policy to influence how iterations of a parallel loop are scheduled on the execution units in the hardware substrate. This is illustrated in Figure B.1.

## B.2 Ragged API Lowering

### B.2.1 Tensor Storage Lowering

In §5.4.3, we briefly discussed the storage lowering schemes used by past work on sparse tensor compilers and by CoRa. Both are illustrated in Figure B.2 and discussed more below.

#### Sparse Storage Access Lowering Scheme Used in Past Work

Recall the 4-dimensional attention tensor $X$ we discussed in §5.4.3 and which is illustrated again in Figure B.2. We saw that the first and the third dimensions of $X$ are cdims and correspond to the batch size ($s_1$) and the number of attention heads ($s_3$) respectively. The other two dimensions, which correspond to sequence lengths are vdims, the size of a slice for which is the same function ($s_{24}()$) of the outermost batch dimension.

The sparse tensor compiler Taco [65], the performance of which look at in §B.4.3, uses a tree-based modular scheme (first proposed in the work [119] on the Compressed Sparse Fiber tensor format) to model sparse tensor storage. In this scheme, illustrated in Figure B.2 for tensor $X$, tensor storage is modeled as hierarchical tree structure, where each tensor dimension corresponds to a tree level. Note that this tree abstraction exists only at compile time. As mentioned before, this scheme assumes that the number of non-zeros in a slice of a sparse tensor dimension can depend on the indices of all outer

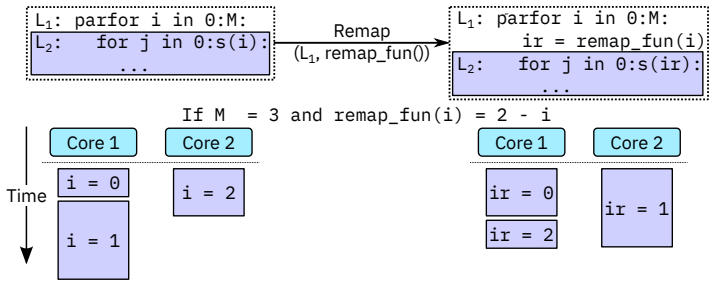



Figure B.1: Thread remapping allows users to influence the scheduling of iterations to allow for better load balancing.

dimensions in general. We saw that this is not the case with ragged tensors and that this is the source of sub-optimality in this lowering scheme for the applications we look at. Because every slice may have a different number of non-zero elements, when used to store a ragged tensor, this storage scheme would store auxiliary data proportional to the number of slices for a given vdim. For our example tensor $X$ in Figure B.2, the outer of the two vdims (the second dimension) has $s_1$ slices while the number of slices in the inner vdim (the fourth dimension) is $s_3 \sum_{i=0}^{s_1} s_{24}(i)$. Therefore, the amount of auxiliary data computed and stored would be equal to $s_1 + s_3 \sum_{i=0}^{s_1} s_{24}(i)$, which as we saw in §5.6.4 can be much larger than CoRa's specialized scheme.

**Algorithm 1** Procedure to lower ragged tensor accesses

**procedure** LowerAccess($[b_1, ..., b_n]$)
  $offset \leftarrow 0$
  $relaxed \leftarrow [b_1, ..., b_n]$
  **for** $i \leftarrow n$ to 1 **do** ▷ Compute $D_i(\overrightarrow{B_{\leq i}})$
    $D \leftarrow 1$
    **if** $O_G(i) \neq \emptyset$ **then**
      $D \leftarrow A_i(relaxed[j])$
    **else**
      $D \leftarrow relaxed[i]$
    **end if**
    **for** $j$ in $S(i) - \{i\}$ **do**
      **if** $O_G(j) \neq \emptyset$ **then**
        $D \leftarrow D * A_j(relaxed[j])$
      **else**
        $D \leftarrow D * s_j(relaxed[I_G(j)])$
      **end if**
    **end for**
    $relaxed[i] \leftarrow s_i(relaxed[I_G(i)])$
    $offset \leftarrow offset + D$
  **end for**
  **return** $offset$
**end procedure**

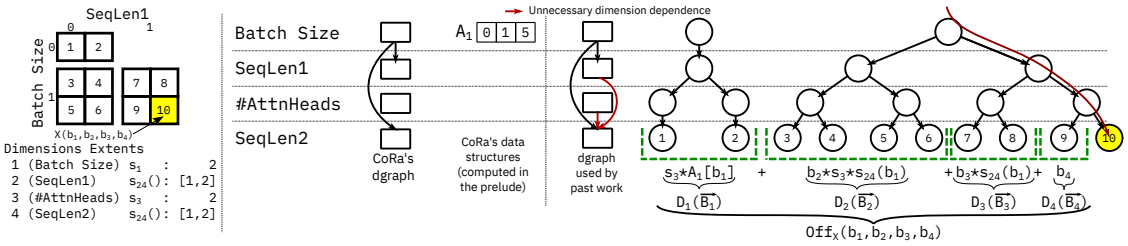



Figure B.2: Comparing CoRa's storage lowering with the tree-based scheme used by past work on sparse tensors.

CoRa's Storage Access Lowering Scheme

We saw that CoRa's storage access lowering scheme is specialized for ragged tensors and enables us to reduce the amount of auxiliary data that needs to be computed as compared to the scheme used by past work while allowing $O(1)$ accesses to ragged tensor storage. Such $O(1)$ accesses are enabled by the memory offsets that CoRa precomputes as part of its auxiliary data structures. Below, we describe exactly how these data structures are computed and how they are used to lower memory accesses.

Let $T$ be an n-dimensional tensor with dimensions numbered 1 to $n$ such that dimension 1 is the outermost dimension. Given a tensor access $T(b_1, .., b_n)$, we need to generate a flat memory access as part of lowering. In other words, we need to generate a memory offset $\mathit{Off}_T(b_1, .., b_n)$ to access the tensor.

Given a tensor and its corresponding storage layout, we define what we refer to as the *dimension graph* or *dgraph* for short (Figure B.2). The dgraph $G$ of the n-dimensional tensor $T$ is a pair $(D, E)$ where $D$ is the set of all dimensions $\{1, ..., n\}$ and $E$ is a set of directed edges. An edge $d_1 \rightarrow d_2$ belongs to $E$ if the size of a slice of dimension $d_2$ depends on the index $b_{d_1}$ in the tensor access $T(b_1, .., b_n)$. Thus, a cdim will not have any incoming edge in the dgraph, while a vdim would. It also follows, for example, that the outermost dimension of the tensor, which is always a cdim, will not have any incoming edges. More generally, we note that the dgraph of a given tensor is always acyclic as the size of a slice of a given vdim depends only on the indices of outer dimensions. Further, given a dimension $d$, let $O_G(d) = \{d_2 | (d, d_2) \in E\}$ and $I_G(d) = \{d_1 | (d_1, d) \in E\}$ be the set of outgoing and incoming dimensions, respectively, for $d$ in the dimension graph. The size of a slice of a vdim $d$ can now be written as $s_d(I_G(d))$. For cdims, this quantity is constant as $I_G$ for a cdim in the empty set. Let $O^*_G(d)$ denote the transitive closure of $O_G(d)$. Also, let $O^{ex}_G(d) = O_G(d) - \bigcup_{i \in O_G(d)} O^*_G(i)$.

We present the procedure to compute $\mathit{Off}_T(b_1, .., b_n)$ in Algorithm 1. For brevity, we refer to the index vector $[b_1, .., b_n]$ as $\overrightarrow{B}$. Also, let $\overrightarrow{B_{\geq i}} = [b_i, ..., b_n]$. We can correspondingly defined $\overrightarrow{B_{\leq i}}$. We abuse notation to represent $\mathit{Off}_T(b_1, ..., b_{i-1}, b_i, 0, ..., 0)$ as $\mathit{Off}_T(\overrightarrow{B_{\leq i}})$. Then, we can expand the offset $\mathit{Off}_T(\overrightarrow{B_{\leq n}})$ as follows:

$$\begin{aligned} \mathit{Off}_T(\overrightarrow{B_{\leq n}}) &= \sum_{i=1}^{n} (\mathit{Off}_T(\overrightarrow{B_{\leq i}}) - \mathit{Off}_T(\overrightarrow{B_{<i}})) \\ &= \sum_{i=1}^{n} D_i(\overrightarrow{B_{\leq i}}) \end{aligned}$$

During compilation, the procedure in Algorithm 1 computes the memory offset expression using two nested loops. Each iteration of the outer loop (line 4) corresponds to one dimension $i$ and computes $D_i(\overrightarrow{B_{\leq i}})$. For a dimension $i$, $D_i(\overrightarrow{B_{\leq i}})$, is further computed (in the inner loop on line 11) as a product of contributions corresponding each of the inner dimensions $j$ such that $j \geq i$ (Figure B.2 shows the values of $D_i$s for the 4 dimensions in our example tensor $X$ at the bottom of the tree in green in the rightmost pane.). In the case of a dense tensor, $D_i(\overrightarrow{B_{\leq i}}) = b_i \prod_{j=i+1}^{n} s_j$. For a ragged tensor, however, due to the dependences between dimensions, the contribution of each dimension $j$ to $D_i$ cannot be computed independently. Specifically, we compute the contribution of an inner dimension $j$ along with all the dimensions dependent on it, directly or indirectly (i.e. $O_G^*(j)$) as a single quantity as a call to the function $A_j()$. This function is similar to the `row_index` array in the CSR matrix format which stores the start and ends of variable-sized rows. Given a ragged tensor format (in the form of the length functions $s_d$ for all dimensions $d$), we need to precompute the values of the function $A_d$ for all dimensions such that $O_G(d)$ is non-empty. We perform this computation as part of the prelude discussed in §5.1. The function $A_d()$ for the batch dimension (the first dimension) of our example tensor $X$ in Figure 5.8 is shown as the array $\mathsf{A}_1$ where $\mathsf{A}_1\texttt{[}i\texttt{]} = \sum_{j=1}^{i} s_{24}(j) * s_{24}(j)$.

As discussed above, for a dimension $d$, because, $A_d()$ includes the contributions from all dimensions in $O_G^*(d)$, we need to exclude those dimensions to avoid double counting them during the inner loop. Therefore, the inner loop of the procedure iterates over the set $S(d)$ (defined recursively as $S(n) = \{n\}$ and $S(d) = \{d\} \cup (S(d+1) - O_G^*(d))$) which excludes these dimensions. Given a dimension $d$, the function $A_d$ can be computed recursively as follows.

$$A_d(B_{\leq d}) = \begin{cases} s_d(B_{\leq d}), & \text{if } O_G(d) = \emptyset \\ \sum_{i=0}^{i=b_d} \Big( \prod_{d_i \in O_G^{ex}(d_i)} A_{d_i}(relaxed_d[I_G(d_i)]) \Big) & \text{otherwise} \end{cases}$$

where $relaxed_d$ is the value of the vector $relaxed$ in Algorithm 1 in the iteration of the outer loop corresponding to the dimension $d$.

### B.2.2 Variable Loop Fusion

In §5.4.1, we discussed how we need to precompute certain quantities as part of the prelude to support vloop fusion. During lowering, we represent these quantities as opaque or uninterpreted functions. For example, §5.4.2 describes how the functions $f_{fo}$, $f_{fi}$ and $f_{oif}$ represent the relationships between the

iteration variables `o`, `i` and `f` in Figure 5.6. In the generated code, as we can see in Figure 5.4, these functions take the form of arrays that are initialized by the prelude. Similar to Cortex as discussed in Appendix A, in order to perform simplification over expressions containing calls to these functions as well as for proving if certain bound checks are redundant, we use the Z3 SMT solver during compilation. In order to enable Z3 process these uninterpreted functions, we provide it with the following relationships between these functions:

$$\begin{aligned} \forall f, f_{oif}(f_{fo}(f), f_{fi}(f)) &= f \\ \forall o, i, f_{fo}(f_{oif}(o, i)) &= o \\ \forall o, i, f_{fi}(f_{oif}(o, i)) &= i \end{aligned}$$

## B.3 Additional Implementation Details

As we mentioned in §5.5, we have prototyped CoRa for the common cases encountered when expressing and optimizing ragged operations. In our evaluation, we implement and compare the performance of an encoder layer of the transformer model in CoRa. Our prototype currently allows us to generate code for individual (potentially fused) ragged operators at a time as opposed to entire model graphs. Therefore, for our implementation of the transformer layer, we individually optimized and generated code for each operator and then invoked it as part of a separate program that ties the operators together to form the layer. CoRa's implementation of the hfusion optimization currently is limited to the outermost loops of the operators one would like to fuse. On a GPU, this means that our prototype implementation allows one to execute multiple operators concurrently as part of the same GPU grid, but not the same GPU thread block. Implementing the general transform is not fundamentally difficult, however.

## B.4 Supplementary Evaluation and Additional Details

### B.4.1 Load Balancing

We briefly discussed the challenge of ensuring a balanced workload across multiple execution units in §5.3.1. On a CPU, these execution units take the form of CPU cores, while a GPU has a hierarchy composed of thread blocks, warps and threads. In all the kernels we evaluate on (except the Softmax kernel in the transformer layer), dense inner cloops or partial padding allow us to prevent imbalance across GPU warps in the same thread block. Imbalance across multiple thread blocks exists, most commonly in gemm-like operations where the reduction loop is a vloop such as the AttnV operator in the SDPA module. We handle this imbalance using either thread remapping (§5.3 and §B.1.1) or, in the case of kernels that are part of the transformer layer, by sorting the sequences in the mini-batch in

descending order of sequence lengths so that thread blocks with the most amount of work are scheduled first.

### B.4.2 Masked Scaled Dot-Product Attention

As we briefly mentioned in §5.6.2, the decoder layer of a transformer uses a variant of MHA called masked MHA wherein the upper triangular half of the attention matrix is masked for all attention heads during training. This is done to prevent the model from attending to words that would not be known during inference at a given time step. In this section, we provide further details and data regarding how CoRa can exploit this masking and further save on wasted computation in the SDPA sub-module, which is the only portion affected by the masking.

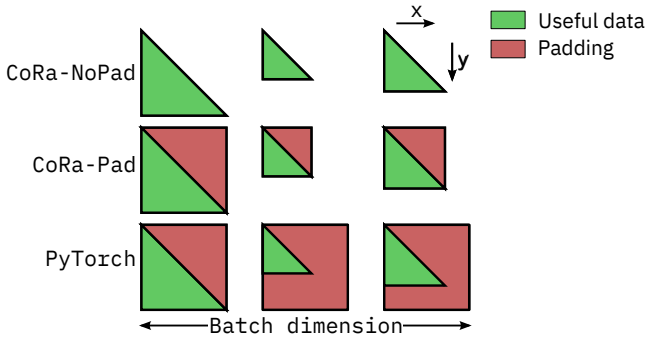



Figure B.3: The attention matrices of the masked MHA module as implemented in the implementations discussed in §5.6.2 and compared in Figure B.4. In the figure, for simplicity, the number of attention heads is assumed to be 1, partial padding is not shown and the batch size is assumed to be 3. The x and y directions denote increasing matrix indices.

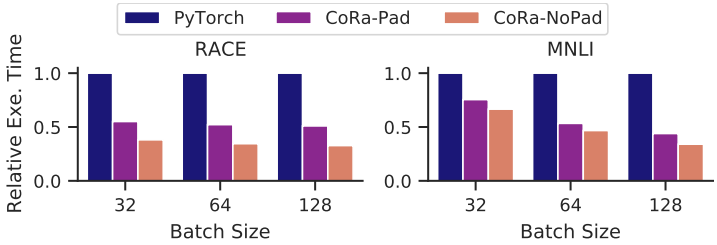



Figure B.4: Execution time of masked SDPA in PyTorch and CoRa, with and without padding for the attention matrix.

We also mentioned in §5.6.2 that with masking, the SDPA computation is essentially composed of batched lower triangular matrix operations. Implemented this way, these operations have one vloop corresponding to the variable sequence lengths and another inner vloop corresponding to the triangular matrix rows. Figure B.4 shows the performance of three implementations of masked SDPA—CoRa-NoPad, where both the vloops are only partially padded, CoRa-Pad, where the outer vloop is partially padded while the inner one is fully padded and a PyTorch implementation, where both the vloops are fully padded. The padding in the three implementations is illustrated in Figure B.3. As Figure B.4

shows, CoRa-NoPad can effectively exploit the reduction in computation in the masked case by avoiding full padding. This leads to 1.34× and 2.46× faster execution as compared to CoRa-Pad and PyTorch respectively across the datasets and batch sizes evaluated in Figure B.4. As we saw, the performance of MNLI dataset improves to a smaller degree due to the padding employed in CoRa-NoPad.

### B.4.3 Evaluation Against Sparse Tensor Compilers

We saw in §5.7 that there are some similarities between ragged and sparse tensors. In this section, we explore using sparse tensor compilers in order to express ragged tensor operations. Specifically, we look at using Taco in order to implement three operations on triangular matrices—the triangular matrix multiplication (trmm) operation we saw in §5.6.1, elementwise addition of two square triangular matrices (we refer to this operation as tradd, for short) and a similar elementwise multiplication of two square triangular matrices (referred to as trmul, for short). Taco does not natively support the storage of ragged tensors. Therefore for this study, we use the compressed sparse row (CSR) and the blocked compressed sparse row (BCSR) matrix formats to store the triangular matrices. We use a block size of 32 for the BCSR format. Table B.1 lists the execution times (in ms) for the aforementioned operations and formats. As the table shows, CoRa performs better than Taco for all the cases evaluated. We discuss the reasons for this below.

#### Storage Layouts

A part of the slowdown in Taco stems from the sub-optimal storage format (CSR or BCSR) used for the triangular matrices. The overheads of traversing the auxiliary data structures to access the sparse tensor storage therefore decrease when we go from the CSR format to the BCSR format, thereby leading to increased performance, despite the additional padding in the latter. For the operations evaluated, the output matrices are stored in a dense manner because using the compressed formats prevents parallelization in some cases in the Taco implementations.

#### Degree of Sparsity

The optimizations, scheduling primitives and code generation techniques used in Taco have been designed for tensors with a high degree of sparsity. We have seen, however, that ragged tensors are much closer to their dense counterparts with respect to the amount of useful data they store. Therefore, optimization decisions that work well for sparse tensors do not always work for ragged tensors.

#### Properties of Ragged Tensors

Finally, due to its design as a tensor compiler for general sparse tensors, Taco is unable to exploit certain properties specific to ragged tensors and the applications they are used for, such as the insight **I1** we discussed in §5.1. Therefore, Taco assumes that the two triangular input matrices in the tradd and trmul operations have differing sparsity patterns. Taco, therefore, has to generate code to iterate over

Table B.1: Execution times (in ms) for the trmm, tradd and trmul operations implemented in Taco using the CSR and the BCSR matrix formats and in CoRa. The table also shows Taco's slowdowns with respect to CoRa.

| Op | Matrix Dim. | CoRa | Taco-CSR | | Taco-BCSR | |
|---|---|---|---|---|---|---|
| | | | Time | Slowdown | Time | Slowdown |
| trmm | 128 | 0.043 | 0.062 | 1.44 | 0.467 | 10.92 |
| | 512 | 0.082 | 1.347 | 16.43 | 1.112 | 13.56 |
| | 2048 | 0.893 | 75.12 | 84.19 | 47.497 | 53.24 |
| | 8192 | 50.905 | 4854.31 | 95.37 | 4252.33 | 83.54 |
| tradd | 128 | 0.004 | 0.057 | 15.61 | - | - |
| | 512 | 0.004 | 0.223 | 61.68 | - | - |
| | 2048 | 0.033 | 1.538 | 46.94 | - | - |
| | 8192 | 0.476 | 7.883 | 16.58 | - | - |
| trmul | 128 | 0.004 | 0.057 | 15.89 | 0.008 | 2.08 |
| | 512 | 0.004 | 0.225 | 57.21 | 0.016 | 3.87 |
| | 2048 | 0.033 | 1.544 | 47.26 | 0.077 | 2.34 |
| | 8192 | 0.476 | 7.92 | 16.67 | 0.632 | 1.33 |

all the coordinates representing the union of the non-zeroes in the input matrices for the tradd operator. This is unlike an intersection that is performed in trmul. This prevented us from scheduling the tradd operator using the BCSR format in a way similar to the trmul operator. Further, Taco currently does not allow users to specify padding for loops and tensor dimensions which would help elide conditional checks in the generated code.

Therefore, while Taco achieves performance comparable to CoRa's in some cases (such as the trmul operator), we conclude that Taco's programming model and optimizations are designed for highly sparse tensors which can lead to poor performance in a lot of cases involving ragged tensors.

### B.4.4 Memory Consumption

We mentioned in §5.6.2 that the use of ragged tensors leads to a significant drop in the memory required to store the forward activations of the encoder layer. Figure B.5 shows this for the datasets in Table 5.3 for batch size 64. It plots the relative total memory consumption (computed analytically) of the forward activations of a transformer encoder layer for CoRa's implementation with and without the use of ragged tensors. We take into account any partial padding that the ragged implementation requires. The relative memory consumption for the other batch sizes is also similar. We also saw how only small improvements are observed for the Wiki512 and Wiki128 datasets which have higher sequence lengths and hence low opportunity for CoRa to exploit.

### B.4.5 Operation Splitting and Horizontal Fusion

In §5.6.3 of the main text, we looked at the benefits of operation splitting and hfusion on the AttnV operator. We now look at the $QK^T$ operator, which is also an instance of the vgemm problem. Each gemm instance in this case has two non-reduction vloops. We first look at the case where the optimizations are applied to the outer one of these two vloops in Figure B.6. The figure shows the normalized execution times, for the $QK^T$ operator, of the three implementations described in §5.6.3. We see that

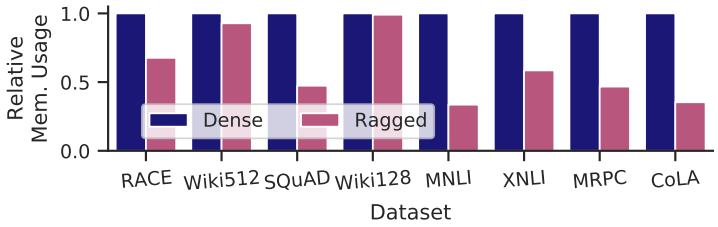



Figure B.5: Relative sizes of the forward activations of a transformer encoder layer with and without ragged tensors.

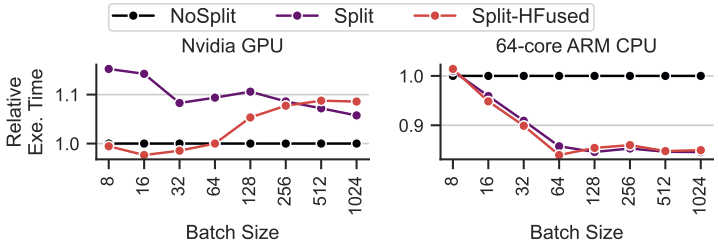



Figure B.6: Operation splitting and hfusion for $QK^T$. Note that the y-axis does not start at 0.

on the CPU backend, similar to the AttnV operator, operation splitting has a significant benefit but hfusion does not, due to low parallelism exposed by the CPU. On the GPU backend, however, we see that the combination of the optimizations gives slightly better performance for lower batch sizes but performs worse as the batch size increases. Profiling data shows that applying the optimization in this case leads to an increase in the number of integer instructions executed as well as an increase in the number of memory load requests. One possible explanation for this is that the CUDA compiler does not effectively hoist memory access expressions in order to avoid high register pressure (the compiled code does not contain any spilled registers). While the optimizations generally lead to more complicated code, the fact that $QK^T$ has two vloops that we fuse when scheduling further exacerbates this problem.

When applied to both the vloops, the optimizations slow the execution down as seen in Figure B.7. In that figure, we compare the performance of three CoRa implementations—NoSplit, which does not use either of the optimizations on either vloop, Split1-HFused, which employs both the optimizations for the outer vloop and Split2-HFused, which employs the optimizations for both vloops—on the Nvidia GPU and the 64-core ARM CPU backends. We see that on both backends, optimizing both vloops is no better than optimizing just one vloop and is, in fact, quite slower on the GPU. On the GPU, we find that despite the decrease in the computation performed and hence the number of floating point instructions executed, the total number of executed instructions is higher in the case Split2-HFused case as compared to the NoSplit case. We therefore believe, that in this case too, the overheads of performing the optimizations are much higher than their benefits (the reduced wasted computation).

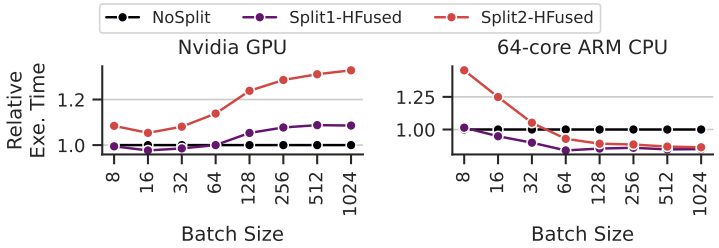



Figure B.7: Efficacy of operation splitting and hfusion when applied to one or both vloops of the $QK^T$ operator. Note that the y-axis does not start at 0.

Table B.2: Prelude execution times (in ms) for a 6-layer transformer encoder with and without redundant computation.

| Dataset | Batch Size | CoRa-Redundant | | | CoRa-Optimized | | |
|---|---|---|---|---|---|---|---|
| | | CoRa Storage | CoRa Loop Fusion | CoRa-Copy Time | CoRa Storage | CoRa Loop Fusion | CoRa-Copy Time |
| CoLA | 32 | 0.004 | 0.006 | 0.232 | 0.002 | 0.002 | 0.088 |
| CoLA | 128 | 0.006 | 0.015 | 0.261 | 0.003 | 0.004 | 0.094 |
| RACE | 32 | 0.005 | 0.085 | 0.419 | 0.002 | 0.015 | 0.121 |
| RACE | 128 | 0.007 | 0.339 | 0.985 | 0.003 | 0.053 | 0.209 |

### B.4.6 CoRa Overheads

#### Prelude Overheads

As we discussed in §B.3, CoRa's prototype allows us to generate code for operator kernels one at a time. For each kernel, CoRa generates all the prelude code required for its execution. Therefore, when these generated kernels are invoked to form a larger model graph, as in our implementation of the transformer encoder layer, there is a lot of redundant computation in the prelude code. This is because (i) each operator computes the auxiliary data structures needed for all of its input and output tensors, which leads to these data structures being generated twice for every tensor in the graph, and (ii) the vloops in the schedules for all operators except the $QK^T$ and the AttnV operators in CoRa's implementation of the layer are fused similarly and can reuse the same auxiliary data structures, which are also currently computed separately for every operator. Tables B.2 and B.3 compare, for a 6-layer transformer encoder, the execution time and memory consumption of the prelude code respectively, as present in CoRa's current implementation (referred to as CoRa-Redundant in the table) with an optimized implementation (referred to as CoRa-Optimized) which has all of this redundant computation removed. We see that when appropriately reused, the time and memory resources required to compute the auxiliary data structures in the prelude are quite low as compared to the those required for the execution of the kernel computation.

#### Overheads Due to Partial Padding

In Figure B.8, we show the relative amount of computation (computed analytically as in Figure 5.2) for the transformer encoder layer for all datasets at batch sizes 32 and 128 for three cases—the fully padded dense case, the actual computation as evaluated in §5.6 with partial padding, and the ideal case with no

Table B.3: Prelude memory usage (in kB) for a 6-layer transformer encoder with and without redundant computation.

| Dataset | Batch Size | CoRa-Redundant | | CoRa-Optimized | |
|---|---|---|---|---|---|
| | | CoRa Storage | CoRa Loop Fusion | CoRa Storage | CoRa Loop Fusion |
| CoLA | 32 | 2.93 | 32.15 | 1.2 | 5.27 |
| CoLA | 128 | 11.18 | 104.22 | 4.58 | 17.5 |
| RACE | 32 | 2.93 | 666.54 | 1.2 | 106.87 |
| RACE | 128 | 11.18 | 2609.58 | 4.58 | 418.06 |

padding. We see that partial padding leads to a very small increase in the amount of computation (3.5% across datasets for batch size 32 and 2.3% for batch size 128). Because we generally pad individual sequence lengths or their sum (as part of bulk padding) so that the quantity is a constant multiple of a small quantity (such as 32, or 64), the relative amount of padding added is higher for smaller batch sizes and datasets with smaller sequence lengths. Even in these cases, however, the added padding is much lower as compared to the benefits obtained with the use of ragged tensors. Further we note that the amount of padding added is a scheduling and optimization decision and can be changed if needed.

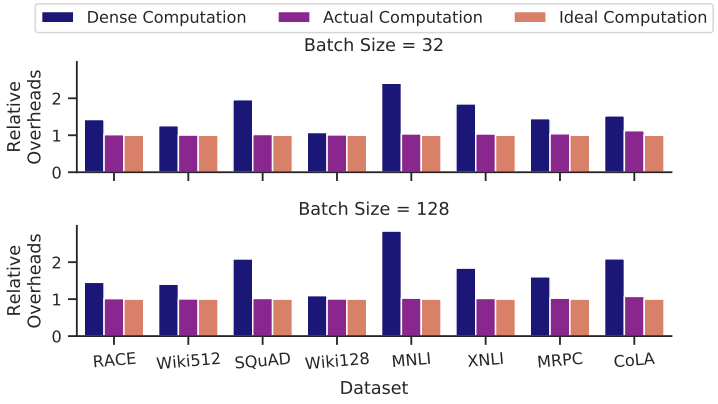



Figure B.8: Overheads due to partial padding.

Ragged Tensor Overheads and Load Hoisting

We now take a closer look at the effects of auxiliary data structure accesses on the performance of CoRa-generated code. These data structure accesses arise in the generated code, as we have seen, due to the use of vloop fusion and ragged tensor storage. We focus on the five operators that make up the MHA module here. We measure the execution times of four implementations of each operator. The Dense implementation does not use ragged tensor storage or ragged computations. The +vloops implementation uses ragged computations, but the tensors are stored with full padding in a dense fashion. The +vdims implementation uses both ragged computations as well as ragged tensor storage. The +LoadHoist implementation is same as +vdims but hoists accesses to the auxiliary data structures out of loops as much as possible. In order to ensure that we perform the same amount of computation in all cases, we use a synthetic dataset where all sequences have the same length (512). The relative

performance of these implementations for the operators on the Nvidia GPU is shown in Figure B.9. Apart from the overheads due to indirect memory accesses, the use of vloops and/or vdims also lead to overheads associated with the prelude code. In order to focus on the former overheads, however, we exclude prelude costs in the figure.

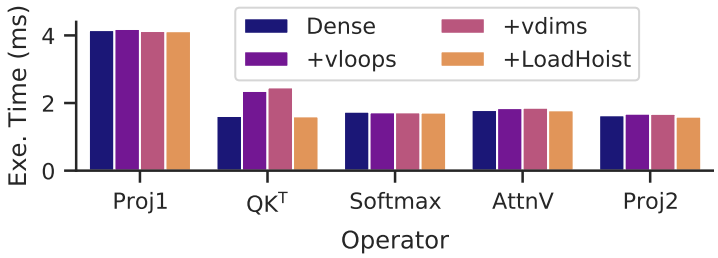



Figure B.9: Overheads of using ragged computations and ragged tensor storage, and the benefits of load hoisting, measured for a synthetic dataset where all sequence lengths are 512. The batch size used is 64.

As the figure shows, the use of vloops and vdims leads to a slight slowdown for the Proj1, Softmax, Attnv and Proj2 operators. The slowdown is significant, however, for the $QK^T$ operator, which has two vloops in its loop nest. As part of scheduling, we fuse both these vloops as well as the loop that the vloop bounds depend on (i.e. the loop that iterates over the mini-batch), leading to complex auxiliary data structure accesses. We believe that the CUDA compiler is unable to effectively hoist these accesses in this case. CoRa however has more knowledge about these accesses and can hoist them to recover the lost performance.

### B.4.7 Discussion on Transformer Layer Evaluation

In this section, we provide further analysis of our evaluation of the transformer encoder layer on the Nvidia GPU and ARM CPU backends. We break down the execution time of the encoder layer for a few cases. As in Figure 5.13, these per-operator execution times are obtained under profiling and might deviate slightly from the data in Tables 5.4 and 5.5.

#### Nvidia GPU Backend

Table B.5 provides the raw data for the breakdown of the execution times for the RACE dataset at batch size 128 of the transformer encoder layer shown in Figure 5.13 in the main text. Apart from improvements in the $QK^T$ and AttnV operators discussed in §5.6.2, we note that CoRa's implementation is significantly faster for the Softmax operator as compared to the FasterTransformer implementations. While we perform less computation on this operator as compared to the fully padded implementation in FasterTransformer, part of CoRa's performance benefits also stem from a better schedule. Specifically, the FasterTransformer implementation performs parallel reductions across GPU thread blocks. This leads to a significant number of barriers at the thread block-level which have execution overheads. Further, the FasterTransformer implementation uses conditional checks to ensure that it never accesses attention scores for the added padding. In CoRa we use warp-wide parallel reductions which are much

cheaper due to their lower synchronization costs but also provide a lower amount of parallelism. We, therefore, only partially parallelize the reductions and compensate with the high parallelism available in the other loops of the operator. Further, this means that we do not have to additionally employ conditional checks to avoid accessing invalid data (that is part of the partial padding we add).

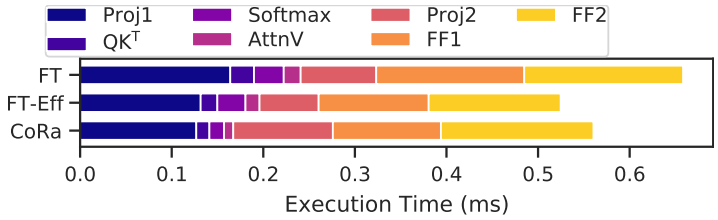
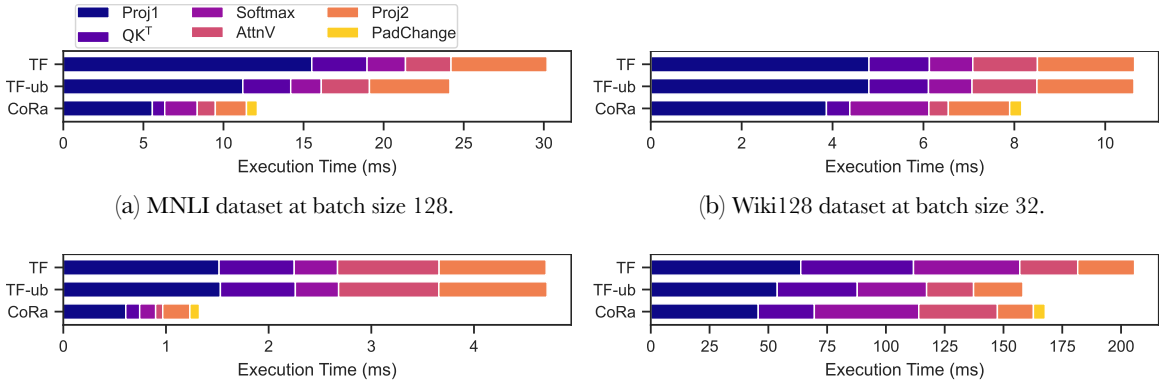



Figure B.10: Breakdown of execution times of the encoder layer for the CoLA dataset at batch size 32 on the GPU.




(a) MNLI dataset at batch size 128.

(b) Wiki128 dataset at batch size 32.

(c) CoLA dataset at batch size 32.

(d) RACE dataset at batch size 128.

Figure B.11: Breakdown of execution times of the MHA module for four cases on the 64-core ARM CPU backend.

We now look at the execution time breakdown for the CoLA dataset at batch size 32 on the Nvidia GPU shown in Figure B.10. We see that CoRa performs slightly worse than FT-Eff for this case. Most of CoRa's slowdown stems from worse performance on the linear transformation operators Proj2, FF1 and FF2. CoRa performs slightly better than FT-Eff for the Proj1 operator, which is also a linear transformation operator. From this data, we conclude that CoRa's schedules for the Proj2, FF1 and FF2 operators can be improved to close this performance gap. We note that, even in this case, CoRa performs much better on the SDPA module (the $QK^T$, Softmax and AttnV operators) as compared to FasterTransformer.

### ARM CPU Backends

In §5.6.2, we saw how CoRa performs better than TensorFlow for the MHA module on the 8- and 64-core ARM CPUs. In this section, we discuss these implementations in more detail and provide more extensive evaluation.

Table B.4: MHA execution latencies (in ms) on 8- and 64-core ARM CPUs. uBS stands for the optimal micro-batch size.

| Dataset | Batch Size | 8-core ARM CPU | | | | | 64-core ARM CPU | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| | | PT | PT-UB / uBS | TF | TF-UB / uBS | CoRa | PT | PT-UB / uBS | TF | TF-UB / uBS | CoRa |
| RACE | 32 | 627 | **209** / 2 | 300 | 228 / 8 | 263 | 4373 | 127 / 2 | 55 | 46 / 16 | **44** |
| | 64 | 1267 | **411** / 2 | 596 | 432 / 8 | 515 | 8724 | 253 / 2 | 111 | 88 / 32 | **85** |
| | 128 | 2558 | **810** / 2 | 1189 | 835 / 8 | 1009 | 17431 | 511 / 2 | 209 | **156** / 32 | 168 |
| Wiki512 | 32 | 620 | **227** / 2 | 294 | 246 / 8 | 285 | 4294 | 123 / 2 | 53 | 53 / 32 | **47** |
| | 64 | 1267 | **443** / 2 | 597 | 466 / 8 | 561 | 8727 | 239 / 2 | 106 | 96 / 32 | **91** |
| | 128 | 2563 | **875** / 2 | 1184 | 904 / 16 | 1094 | 17427 | 660 / 2 | 205 | **172** / 32 | 176 |
| SQuAD | 32 | 324 | **101** / 4 | 189 | 117 / 8 | 113 | 1904 | 94 / 4 | 35 | 27 / 16 | **20** |
| | 64 | 770 | **192** / 4 | 383 | 210 / 8 | 219 | 4953 | 181 / 4 | 68 | 49 / 32 | **39** |
| | 128 | 1580 | **364** / 4 | 780 | 390 / 8 | 424 | 10236 | 357 / 4 | 137 | 79 / 32 | **76** |
| Wiki128 | 32 | 53 | **52** / 16 | 53 | 52 / 32 | 54 | 76 | 76 / 32 | 11 | 11 / 32 | **9** |
| | 64 | 133 | 101 / 16 | 101 | **100** / 64 | 102 | 330 | 141 / 16 | 19 | 18 / 64 | **17** |
| | 128 | 353 | 196 / 16 | 199 | **190** / 64 | 200 | 1544 | 273 / 16 | 34 | 33 / 128 | **33** |
| MNLI | 32 | 41 | 26 / 8 | 39 | 29 / 8 | **20** | 69 | 30 / 4 | 9 | 9 / 32 | **4** |
| | 64 | 100 | 47 / 8 | 82 | 52 / 16 | **38** | 204 | 51 / 8 | 16 | 14 / 32 | **7** |
| | 128 | 260 | 90 / 16 | 177 | 93 / 16 | **76** | 399 | 87 / 16 | 30 | 23 / 64 | **14** |
| XNLI | 32 | 53 | 36 / 8 | 52 | 42 / 16 | **33** | 76 | 58 / 2 | 11 | 11 / 32 | **6** |
| | 64 | 133 | 68 / 8 | 101 | 73 / 16 | **65** | 324 | 95 / 8 | 18 | 18 / 64 | **11** |
| | 128 | 351 | 131 / 16 | 199 | 134 / 32 | **128** | 1549 | 179 / 16 | 34 | 28 / 64 | **22** |
| MRPC | 32 | 38 | 31 / 8 | 37 | 33 / 16 | **27** | 71 | 46 / 4 | 9 | 8 / 32 | **5** |
| | 64 | 86 | 59 / 8 | 75 | 61 / 16 | **52** | 172 | 80 / 8 | 14 | 14 / 64 | **10** |
| | 128 | 187 | 110 / 16 | 151 | 111 / 32 | **103** | 351 | 153 / 8 | 26 | 23 / 64 | **18** |
| CoLA | 32 | 10 | 9 / 16 | 12 | 11 / 32 | **8** | 7 | 7 / 16 | 5 | 4 / 32 | **2** |
| | 64 | 21 | 16 / 16 | 21 | 18 / 32 | **14** | 11 | 13 / 16 | 6 | 6 / 64 | **3** |
| | 128 | 46 | 29 / 32 | 37 | 29 / 32 | **25** | 23 | 18 / 32 | 9 | 8 / 128 | **5** |

Micro-Batching for PyTorch and TensorFlow: We saw, in Figure 5.2, that the amount of padding and wasted computation increases with the batch size. On devices that expose low levels of parallelism such as CPUs, it is therefore possible to trade-off batch parallelism for reduced padding, and therefore reduced wasted computation, for frameworks such as PyTorch and TensorFlow. In effect, this amounts to executing a mini-batch sorted by sequence lengths as a series of smaller *micro-batches*. Overall, this reduces the amount of padding needed as each micro-batch is only padded to the length of the longest sequence in that micro-batch, rather than the entire mini-batch as illustrated in Figure B.12. We search over micro-batch sizes that are powers of 2 starting from the lowest micro-batch size of 2. In Table B.4, we provide the execution latencies as well as the optimal micro-batch sizes for PyTorch and TensorFlow (these configurations are referred to as PT-UB and TF-UB respectively) for an 8-core as well as a 64-core ARM CPU. For reference, we also provide the latencies corresponding to naive executions of PyTorch and TensorFlow (referred to as PT and TF respectively) where the micro-batch size is equal to the mini-batch size.

CoRa's MHA Implementation: As in CoRa's vgemm implementation on the Intel CPU backend, we offload the computation of the dense inner tiles of the Proj1 and Proj2 operators in CoRa's MHA implementation on the ARM backends to gemm calls in the OpenBLAS [90] library. Due to limitations of our prototype implementation, however, offloading the computation this way means that we cannot fuse the padding change operators with other computational operators in this case. We see

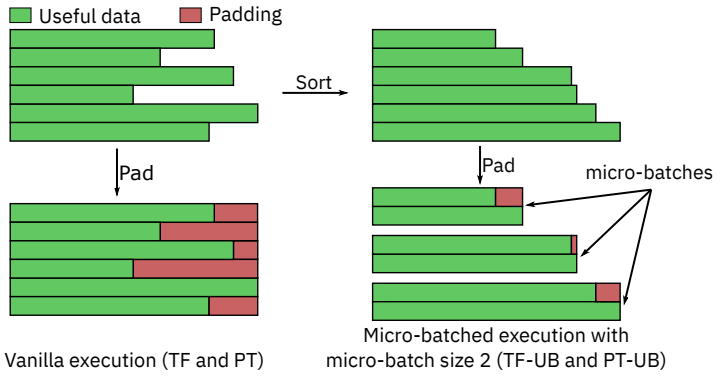



Figure B.12: Comparison of vanilla and micro-batched execution for PyTorch and TensorFlow.

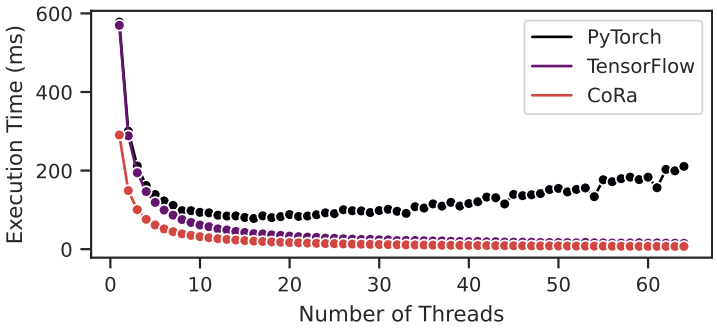



Figure B.13: Execution latencies of PT, TF and CoRa as the number of threads is increased for the MNLI dataset at a batch size of 64. These measurements were performed on the 64-core CPU by changing the number of threads launched by OpenMP. Due to this, the measurements may not exactly be equal to the ones in Table B.4.

in Figure B.11, however, that these pad fusion operators are relatively cheap to perform on the CPU backend.

Overall Performance Comparison: Table B.4 shows the inference latencies for the PyTorch, TensorFlow and CoRa implementations of the MHA module on the 8- and 64-core ARM CPUs. We saw that TF-UB trades-off parallelism for reduced wasted computation. It, therefore, performs the best when there is high parallelism in the workload (i.e. for datasets with longer sequence lengths at higher batch sizes) and it performs the worst when the workload has low parallelism (i.e. for datasets with shorter sequences at lower batch sizes). This is because in the presence of high parallelism in the workload, TF-UB can reduce the micro-batch size much more (leading to much lower wasted padding) as compared to the case of a workload with low parallelism. This is seen reflected in the optimal micro-batch sizes shown in Table B.4. TF-UB also performs better on the 8-core CPU which exposes lower parallelism as compared to the 64-core CPUs. This is again reflected in the optimal micro-batch sizes which are generally higher (leading to higher padding) on the 64-core CPU as compared the 8-core CPU. Overall, we see that TF-UB and CoRa perform similarly on the 8-core ARM CPU, while CoRa outperforms TF-UB by about 1.37× as the hardware parallelism increases on the 64-core CPU. In both the cases, CoRa performs significantly better than the TF configuration of executing TensorFlow.

On the 8-core CPU, PyTorch in the PT-UB configuration performs better than both TF-UB and CoRa for datasets with higher sequence lengths. Similar to TF-UB, PT-UB can more effectively trade-off batch parallelism in these cases due to the high parallelism. Overall, across all the datasets and batch sizes evaluated, CoRa and PT-UB perform similarly, while TF-UB is about 6% slower than both on the 8-core CPU. We find that on the 64-core CPU, however, PyTorch's performance does not scale well with the number of cores (this is apparent in Figure B.13) as compared to TensorFlow and CoRa. Therefore, below, we only consider TensorFlow for further analysis.

Per-Operator Execution Time Breakdown: Let us now look more closely at the execution times of the TensorFlow and CoRa implementations. Figure B.11 provides a breakdown of the execution times for four cases: (1) the MNLI dataset at a batch size of 128 and the Wiki128 dataset at a batch size of 32, which have the most and the least potential for savings on wasted computation due to padding as Figure 5.2 shows, and (2) the RACE dataset at a batch size of 128 and the CoLA dataset at a batch size of 32, which represent the best and worst cases for the TF-UB configuration.

TF-UB and TF perform similarly for the CoLA dataset at batch size 32, as that represents the worst case for TF-UB, and on the Wiki128 dataset at batch size 128 as there is little potential for computational savings due to reduction in padding for that case. In the remaining two cases, TF-UB performs better than TF as expected. For the RACE dataset at batch size 128, which represents the best case for TF-UB, TF-UB performs slightly better than CoRa. In cases where CoRa performs better than TensorFlow, we find that a lot of the reduction in CoRa's absolute execution time stems from computational savings in the Proj1 and Proj2 two operators, which consume a significant portion of the execution time. The $QK^T$ and AttnV operators, however, show a higher relative reduction in execution time as they are quadratically proportional to the sequence lengths as opposed to Proj1 and Proj2 which are linearly proportional to sequence lengths. This difference in proportionality is also reflected in the data for the Wiki128 dataset. TensorFlow generally does well on the Softmax operator, performing better than CoRa for the RACE and Wiki128 datasets. We believe this is due to better optimized implementations and that this gap can be reduced with more time spent optimizing CoRa's implementation of the operator.

Table B.5: Breakdown of the encoder layer execution time for FasterTransformer and CoRa on the Nvidia GPU backend for the RACE dataset at batch size 128. Per-layer prelude code overheads are included in these latencies for CoRa. Both FasterTransformer and CoRa implementations normally execute CUDA kernels asynchronously. For the purposes of profiling (i.e., this table only), these calls were made synchronous, which can lead to slower execution. We also show the end-to-end execution times under profiling for reference.

| Op sub-graphs | FT Ops | FT | FT-Eff | CoRa | CoRa Ops |
|---|---|---|---|---|---|
| Proj1 | QKV Proj. MM | 7.16 | 5.4 | 6.2 | QKV Proj. |
| | QKV Bias + AddPad | 1.39 | 1.21 | | |
| $QK^T$ | $QK^T$ | 2.65 | 2.64 | 2.12 | AddPad + $QK^T$ |
| Softmax | Softmax | 4.08 | 4.08 | 1.93 | ChangePad + Softmax + ChangePad |
| AttnV | AttnV | 2.78 | 2.79 | 2.44 | AttnV |
| Proj2 | Transpose + RemovePad | 0.78 | 0.29 | | |
| | Linear Proj. MM | 2.42 | 1.82 | 2.31 | RemovePad + Linear Proj. MM + Bias + ResidualAdd |
| | Linear Proj. Bias + ResidualAdd + LayerNorm | 0.52 | 0.38 | 0.31 | LayerNorm |
| FF1 | FF1 MM | 9.52 | 6.92 | 8.06 | FF1 MM + Bias + Activation |
| | FF1 Bias + Activation | 1.38 | 0.98 | | |
| FF2 | FF2 MM | 9.47 | 7.1 | 8.33 | FF2 MM + Bias + ResidualAdd |
| | FF2 Bias + ResidualAdd + LayerNorm | 0.53 | 0.38 | 0.31 | LayerNorm |
| | Total Execution Time | 42.82 | 34.12 | 31.99 | Total Execution Time |

# 7 Bibliography


[1] Andrew Adams, Karima Ma, Luke Anderson, Riyadh Baghdadi, Tzu-Mao Li, Michaël Gharbi, Benoit Steiner, Steven Johnson, Kayvon Fatahalian, Frédo Durand, and Jonathan Ragan-Kelley. Learning to optimize Halide with tree search and random programs. *ACM Trans. Graph.*, 38(4), July 2019. ISSN 0730-0301. doi: 10.1145/3306346.3322967. URL https://doi.org/10.1145/3306346.3322967. 2.2.2, 3, 5

[2] Ashish Agarwal. Static automatic batching in TensorFlow. In Kamalika Chaudhuri and Ruslan Salakhutdinov, editors, *Proceedings of the 36th International Conference on Machine Learning*, volume 97 of *Proceedings of Machine Learning Research*, pages 92–101. PMLR, 09–15 Jun 2019. URL https://proceedings.mlr.press/v97/agarwal19a.html. 2.3.1

[3] Ashish Agarwal and Igor Ganichev. Auto-vectorizing TensorFlow graphs: Jacobians, auto-batching and beyond. *CoRR*, abs/1903.04243, 2019. URL http://arxiv.org/abs/1903.04243. 2.3.1

[4] Govind Agrawal, Alan Sussman, and Joel Saltz. Integrated runtime and compile-time approach for parallelizing structured and block structured applications. *Parallel and Distributed Systems, IEEE Transactions on*, 6:747 – 754, 08 1995. doi: 10.1109/71.395403. 3.7

[5] Alfred V Aho, Monica S Lam, Ravi Sethi, and Jeffrey D Ullman. *Compilers: principles, techniques, & tools*. Pearson Education India, 2007. 3, 4

[6] David Alvarez-Melis and T. Jaakkola. Tree-structured decoding with doubly-recurrent neural networks. In *ICLR*, 2017. 2.1.1, 2.1, 4.2

[7] Riyadh Baghdadi, Jessica Ray, Malek Ben Romdhane, Emanuele Del Sozzo, Abdurrahman Akkas, Yunming Zhang, Patricia Suriana, Shoaib Kamil, and Saman Amarasinghe. Tiramisu: A polyhedral compiler for expressing fast and portable code. In *Proceedings of the 2019 IEEE/ACM International Symposium on Code Generation and Optimization*, CGO 2019, page 193–205. IEEE Press, 2019. ISBN 9781728114361. 2.2.2, 3, 5, 5

[8] R.D. Barnes, E.M. Nystrom, M.C. Merten, and W.W. Hwu. Vacuum packing: extracting hardware-detected program phases for post-link optimization. In *35th Annual IEEE/ACM International Symposium on Microarchitecture, 2002. (MICRO-35). Proceedings.*, pages 233–244, 2002. doi: 10.1109/MICRO.2002.1176253. 4

[9] Samuel R. Bowman, Jon Gauthier, Abhinav Rastogi, Raghav Gupta, Christopher D. Manning, and Christopher Potts. A fast unified model for parsing and sentence understanding. *CoRR*, abs/1603.06021, 2016. URL http://arxiv.org/abs/1603.06021. 2.3.1

[10] James Bradbury and Chunli Fu. Automatic batching as a compiler pass in PyTorch. In *Workshop on Systems for ML*, 2018. 2.3.1

[11] Tom B. Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, Sandhini Agarwal, Ariel Herbert-Voss, Gretchen Krueger, Tom Henighan, Rewon Child, Aditya Ramesh, Daniel M. Ziegler, Jeffrey Wu, Clemens Winter, Christopher Hesse, Mark Chen, Eric Sigler, Mateusz Litwin, Scott Gray, Benjamin Chess, Jack Clark, Christopher Berner, Sam McCandlish, Alec Radford, Ilya Sutskever, and Dario Amodei. Language models are few-shot learners. *CoRR*, abs/2005.14165, 2020. URL https://arxiv.org/abs/2005.14165. 2.1.2

[12] ByteDance. Effective Transformer. URL https://github.com/bytedance/effective_transformer. Last accessed Sept 09, 2021. 2.3.2, 5.6.2

[13] Tianqi Chen, Thierry Moreau, Ziheng Jiang, Lianmin Zheng, Eddie Yan, Haichen Shen, Meghan Cowan, Leyuan Wang, Yuwei Hu, Luis Ceze, Carlos Guestrin, and Arvind Krishnamurthy. TVM: An automated end-to-end optimizing compiler for deep learning. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, pages 578–594, Carlsbad, CA, October 2018. USENIX Association. ISBN 978-1-939133-08-3. URL https://www.usenix.org/conference/osdi18/presentation/chen. 2.2.2, 3, 3.5, 5, 5

[14] Tianqi Chen, Lianmin Zheng, Eddie Q. Yan, Ziheng Jiang, Thierry Moreau, Luis Ceze, Carlos Guestrin, and Arvind Krishnamurthy. Learning to optimize tensor programs. *CoRR*, abs/1805.08166, 2018. URL http://arxiv.org/abs/1805.08166. 2.2.2, 3, 5

[15] Wen-ke Chen, Sanjay Bhansali, Trishul Chilimbi, Xiaofeng Gao, and Weihaw Chuang. 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. 4

[16] Xinchi Chen, Xipeng Qiu, Chenxi Zhu, and Xuanjing Huang. 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)*, pages 1744–1753, Beijing, China, July 2015. Association for Computational Linguistics. doi: 10.3115/v1/P15-1168. URL https://aclanthology.org/P15-1168. 2.1

[17] Xinyun Chen, Chang Liu, and Dawn Song. Tree-to-tree neural networks for program translation. *CoRR*, abs/1802.03691, 2018. URL http://arxiv.org/abs/1802.03691. 2.1.1, 2.1

[18] Kyunghyun Cho, Bart van Merrienboer, Çaglar Gülçehre, Fethi Bougares, Holger Schwenk, and Yoshua Bengio. Learning phrase representations using RNN encoder-decoder for statistical machine translation. *CoRR*, abs/1406.1078, 2014. URL http://arxiv.org/abs/1406.1078. 2.1.1, 2.1

[19] Jong-Deok Choi, Manish Gupta, Mauricio Serrano, Vugranam C Sreedhar, and Sam Midkiff. Escape analysis for java. *Acm Sigplan Notices*, 34(10):1–19, 1999. 6.4.1

[20] Stephen Chou, Fredrik Kjolstad, and Saman Amarasinghe. Format abstraction for sparse tensor algebra compilers. *Proc. ACM Program. Lang.*, 2(OOPSLA):123:1–123:30, October 2018. ISSN 2475-1421. doi: 10.1145/3276493. URL http://doi.acm.org/10.1145/3276493. **I2**, 5.4.3

[21] Boost Community. Boost.Fiber, 2022. URL https://www.boost.org/doc/libs/1_79_0/libs/fiber/doc/html/index.html. Last accessed July 1, 2022. 4.2.3

[22] CUTLASS Community. Gather and Scatter Fusion, 2022. URL https://github.com/NVIDIA/cutlass/tree/master/examples/36_gather_scatter_fusion. Last accessed July 25, 2022. 2

[23] PyTorch Community. TorchDynamo. URL https://github.com/pytorch/torchdynamo. Last accessed Oct 03, 2022. 2.2.1

[24] 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. 7

[25] Alexis Conneau, Ruty Rinott, Guillaume Lample, Adina Williams, Samuel R. Bowman, Holger Schwenk, and Veselin Stoyanov. XNLI: Evaluating cross-lingual sentence representations. In *Proceedings of the 2018 Conference on Empirical Methods in Natural Language Processing*. Association for Computational Linguistics, 2018. 4.2, 5.3

[26] Robert David, Jared Duke, Advait Jain, Vijay Janapa Reddi, Nat Jeffries, Jian Li, Nick Kreeger, Ian Nappier, Meghna Natraj, Tiezhen Wang, Pete Warden, and Rocky Rhodes. Tensorflow lite micro: Embedded machine learning for tinyml systems. In A. Smola, A. Dimakis, and I. Stoica, editors, *Proceedings of Machine Learning and Systems*, volume 3, pages 800–811, 2021. URL https://proceedings.mlsys.org/paper/2021/file/d2ddea18f00665ce8623e36bd4e3c7c5-Paper.pdf. 6.4.1

[27] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. BERT: Pre-training of deep bidirectional transformers for language understanding. In *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1 (Long and Short Papers)*, pages 4171–4186, Minneapolis, Minnesota, June 2019. Association for Computational Linguistics. doi: 10.18653/v1/N19-1423. URL https://www.aclweb.org/anthology/N19-1423. 2.1.2, 4.5.1, 5.6.2

[28] Gregory Diamos, Shubho Sengupta, Bryan Catanzaro, Mike Chrzanowski, Adam Coates, Erich Elsen, Jesse Engel, Awni Hannun, and Sanjeev Satheesh. Persistent RNNs: Stashing recurrent weights on-chip. In *Proceedings of the 33rd International Conference on International Conference on Machine Learning - Volume 48*, ICML'16, page 2024–2033. JMLR.org, 2016. 3.7

[29] Shuoyang Ding and Philipp Koehn. Parallelizable stack long short-term memory. In *Proceedings of the Third Workshop on Structured Prediction for NLP*, pages 1–6, Minneapolis, Minnesota, June 2019. Association for Computational Linguistics. doi: 10.18653/v1/W19-1501. URL https://aclanthology.org/W19-1501. 2.3.1

[30] Yaoyao Ding, Ligeng Zhu, Zhihao Jia, Gennady Pekhimenko, and Song Han. IOS: inter-operator scheduler for CNN acceleration. *CoRR*, abs/2011.01302, 2020. URL https://arxiv.org/abs/2011.01302. 5.7

[31] William B. Dolan and Chris Brockett. Automatically constructing a corpus of sentential paraphrases. In *Proceedings of the Third International Workshop on Paraphrasing (IWP2005)*, 2005. URL https://aclanthology.org/I05-5002. 5.3

[32] Andrew Drozdov, Pat Verga, Mohit Yadav, Mohit Iyyer, and Andrew McCallum. Unsupervised latent tree induction with deep inside-outside recursive autoencoders. In *North American Association for Computational Linguistics*, 2019. 2.1

[33] Chris Dyer, Miguel Ballesteros, Wang Ling, Austin Matthews, and Noah A. Smith. Transition-based dependency parsing with stack long short-term memory. *CoRR*, abs/1505.08075, 2015. URL http://arxiv.org/abs/1505.08075. 2.1.1, 2.1, 4.2

[34] Maha Elbayad, Jiatao Gu, Edouard Grave, and Michael Auli. Depth-adaptive transformer. *CoRR*, abs/1910.10073, 2019. URL http://arxiv.org/abs/1910.10073. 2.1.1, 2.1

[35] William Fedus, Barret Zoph, and Noam Shazeer. Switch transformers: Scaling to trillion parameter models with simple and efficient sparsity. *CoRR*, abs/2101.03961, 2021. URL https://arxiv.org/abs/2101.03961. 2.1.1, 2.1

[36] Pratik Fegade, Tianqi Chen, Phillip Gibbons, and Todd Mowry. Cortex: A compiler for recursive deep learning models. In A. Smola, A. Dimakis, and I. Stoica, editors, *Proceedings of Machine Learning and Systems*, volume 3, pages 38–54, 2021. URL https://proceedings.mlsys.

org/paper/2021/file/182be0c5cdcd5072bb1864cdee4d3d6e-Paper.pdf. 1, 4.2.2, 1

[37] Pratik Fegade, Tianqi Chen, Phillip Gibbons, and Todd Mowry. The CoRa tensor compiler: Compilation for ragged tensors with minimal padding. In D. Marculescu, Y. Chi, and C. Wu, editors, *Proceedings of Machine Learning and Systems*, volume 4, pages 721–747, 2022. URL https://proceedings.mlsys.org/paper/2022/file/d3d9446802a44259755d38e6d163e820-Paper.pdf. 2

[38] Oleksandr Ferludin, Arno Eigenwillig, Martin Blais, Dustin Zelle, Jan Pfeifer, Alvaro Sanchez-Gonzalez, Sibon Li, Sami Abu-El-Haija, Peter Battaglia, Neslihan Bulut, Jonathan Halcrow, Filipe Miguel Gonçalves de Almeida, Silvio Lattanzi, André Linhares, Brandon Mayer, Vahab Mirrokni, John Palowitch, Mihir Paradkar, Jennifer She, Anton Tsitsulin, Kevin Villela, Lisa Wang, David Wong, and Bryan Perozzi. TF-GNN: Graph neural networks in TensorFlow, 2022. URL https://arxiv.org/abs/2207.03522. 2.1.2

[39] Joseph A 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. 4

[40] Roy Frostig, Matthew Johnson, and Chris Leary. Compiling machine learning programs via high-level tracing. 2018. URL https://mlsys.org/Conferences/doc/2018/146.pdf. 2.3.1

[41] Trevor Gale, Matei Zaharia, Cliff Young, and Erich Elsen. *Sparse GPU Kernels for Deep Learning*. IEEE Press, 2020. ISBN 9781728199986. 5.3.1, 5.7

[42] Pin Gao, Lingfan Yu, Yongwei Wu, and Jinyang Li. 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. 2.3.1, 4.2.2, 1

[43] Philip B Gibbons and Steven S Muchnick. Efficient instruction scheduling for a pipelined architecture. In *Proceedings of the 1986 SIGPLAN symposium on Compiler construction*, pages 11–16, 1986. 4

[44] Ross B. Girshick. Fast R-CNN. *CoRR*, abs/1504.08083, 2015. URL http://arxiv.org/abs/1504.08083. 2.1

[45] Ross B. Girshick, Jeff Donahue, Trevor Darrell, and Jitendra Malik. Rich feature hierarchies for accurate object detection and semantic segmentation. *CoRR*, abs/1311.2524, 2013. URL http://arxiv.org/abs/1311.2524. 2.1

[46] Scott Gray, Alec Radford, and Diederik P. Kingma. GPU kernels for block-sparse weights. URL https://cdn.openai.com/blocksparse/blocksparsepaper.pdf. 5.7

[47] Rajiv Gupta, Eduard Mehofer, and Youtao Zhang. Profile guided compiler optimizations, 2002. 4

[48] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Spatial pyramid pooling in deep convolutional networks for visual recognition. In *Computer Vision – ECCV 2014*, pages 346–361. Springer International Publishing, 2014. doi: 10.1007/978-3-319-10578-9_23. URL https://doi.org/10.1007%2F978-3-319-10578-9_23. 2.1.2

[49] Sepp Hochreiter and Jürgen Schmidhuber. Long short-term memory. *Neural Comput.*, 9(8):1735–1780, November 1997. ISSN 0899-7667. doi: 10.1162/neco.1997.9.8.1735. URL https://doi.org/10.1162/neco.1997.9.8.1735. (document), 1.1, 3, 2.1

[50] Matthew D. Hoffman and Andrew Gelman. The no-u-turn sampler: Adaptively setting path lengths in hamiltonian monte carlo, 2011. 2.1.1, 2.1

[51] Connor Holmes, Daniel Mawhirter, Yuxiong He, Feng Yan, and Bo Wu. GRNN: Low-latency and scalable RNN inference on GPUs. In *Proceedings of the Fourteenth EuroSys Conference 2019*, EuroSys '19, New York, NY, USA, 2019. Association for Computing Machinery. ISBN 9781450362818. doi: 10.1145/3302424.3303949. URL https://doi.org/10.1145/3302424.3303949. 3.6.1, 3.6.4, 3.7

[52] Michael Huang, Jose Renau, and Josep Torrellas. Profile-based energy reduction in high-performance processors. In *4th Workshop on Feedback-Directed and Dynamic Optimization (FDDO-4)*, 2001. 4

[53] Wei Huang, Yao Dong, Ana Milanova, and Julian Dolby. Scalable and precise taint analysis for Android. In *Proceedings of the 2015 International Symposium on Software Testing and Analysis*, pages 106–117, 2015. 4

[54] Intel. Intel math kernel library, 2020. URL https://software.intel.com/content/www/us/en/develop/tools/math-kernel-library.html. Last accessed July 18, 2020. 2.3.2

[55] Intel. OpenVINO toolkit, 2020. URL https://docs.openvinotoolkit.org/. Last accessed Oct 06, 2020. 2.2.1, 3.7

[56] Andrei Ivanov, Nikoli Dryden, Tal Ben-Nun, Shigang Li, and Torsten Hoefler. Data movement is all you need: A case study on optimizing transformers. *CoRR*, abs/2007.00072, 2020. URL https://arxiv.org/abs/2007.00072. 5.7

[57] Paras Jain, Ajay Jain, Aniruddha Nrusimha, Amir Gholami, Pieter Abbeel, Joseph Gonzalez, Kurt Keutzer, and Ion Stoica. Checkmate: Breaking the memory wall with optimal tensor rematerialization. In I. Dhillon, D. Papailiopoulos, and V. Sze, editors, *Proceedings of Machine Learning*

*and Systems*, volume 2, pages 497–511, 2020. URL https://proceedings.mlsys.org/paper/2020/file/084b6fbb10729ed4da8c3d3f5a3ae7c9-Paper.pdf. 5.6.2

[58] Eunji Jeong, Joo Seong Jeong, Soojeong Kim, Gyeong-In Yu, and Byung-Gon Chun. Improving the expressiveness of deep learning frameworks with recursion. In *Proceedings of the Thirteenth EuroSys Conference*, EuroSys '18, New York, NY, USA, 2018. Association for Computing Machinery. ISBN 9781450355841. doi: 10.1145/3190508.3190530. URL https://doi.org/10.1145/3190508.3190530. 2.2.1

[59] Eunji Jeong, Sungwoo Cho, Gyeong-In Yu, Joo Seong Jeong, Dong-Jin Shin, and Byung-Gon Chun. 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)*, pages 453–468, Boston, MA, February 2019. USENIX Association. ISBN 978-1-931971-49-2. URL https://www.usenix.org/conference/nsdi19/presentation/jeong. 2.2.1

[60] Zhihao Jia, Oded Padon, James Thomas, Todd Warszawski, Matei Zaharia, and Alex Aiken. TASO: Optimizing deep learning computation with automatic generation of graph substitutions. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, SOSP '19, page 47–62, New York, NY, USA, 2019. Association for Computing Machinery. ISBN 9781450368735. doi: 10.1145/3341301.3359630. URL https://doi.org/10.1145/3341301.3359630. 2.2.1, 6.4.2

[61] Erik Johansson and Sven-Olof Nyström. Profile-guided optimization across process boundaries. *SIGPLAN Not.*, 35(7):23–31, jan 2000. ISSN 0362-1340. doi: 10.1145/351403.351411. URL https://doi.org/10.1145/351403.351411. 4

[62] Yigitcan Kaya and Tudor Dumitras. How to stop off-the-shelf deep neural networks from overthinking. *CoRR*, abs/1810.07052, 2018. URL http://arxiv.org/abs/1810.07052. 2.1.1, 2.1

[63] Taebum Kim, Eunji Jeong, Geon-Woo Kim, Yunmo Koo, Sehoon Kim, Gyeongin Yu, and Byung-Gon Chun. Terra: Imperative-symbolic co-execution of imperative deep learning programs. In M. Ranzato, A. Beygelzimer, Y. Dauphin, P.S. Liang, and J. Wortman Vaughan, editors, *Advances in Neural Information Processing Systems*, volume 34, pages 1468–1480. Curran Associates, Inc., 2021. URL https://proceedings.neurips.cc/paper/2021/file/0b32f1a9efe5edf3dd2f38b0c0052bfe-Paper.pdf. 2.2.1

[64] Marisa Kirisame, Steven Lyubomirsky, Altan Haan, Jennifer Brennan, Mike He, Jared Roesch, Tianqi Chen, and Zachary Tatlock. Dynamic tensor rematerialization. In *International Conference on Learning Representations*, 2021. URL https://openreview.net/forum?id=Vfs_2RnOD0H. 5.6.2

[65] Fredrik Kjolstad, Shoaib Kamil, Stephen Chou, David Lugato, and Saman Amarasinghe. The tensor algebra compiler. *Proc. ACM Program. Lang.*, 1(OOPSLA):77:1–77:29, October 2017. ISSN 2475-1421. doi: 10.1145/3133901. URL http://doi.acm.org/10.1145/3133901. 2.2.2, 5, 5, B.2.1

[66] Fredrik Kjolstad, Peter Ahrens, Shoaib Kamil, and Saman Amarasinghe. Tensor algebra compilation with workspaces. pages 180–192, 2019. URL http://dl.acm.org/citation.cfm?id=3314872.3314894. 3.7

[67] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. ImageNet classification with deep convolutional neural networks. In F. Pereira, C.J. Burges, L. Bottou, and K.Q. Weinberger, editors, *Advances in Neural Information Processing Systems*, volume 25. Curran Associates, Inc., 2012. URL https://proceedings.neurips.cc/paper/2012/file/c399862d3b9d6b76c8436e924a68c45b-Paper.pdf. 1

[68] Faisal Ladhak, Ankur Gandhe, Markus Dreyer, Lambert Mathias, Ariya Rastrow, and Bjorn Hoffmeister. LATTICE RNN: Recurrent neural networks over lattices. In *Interspeech 2016*, 2016. URL https://www.amazon.science/publications/lattice-rnn-recurrent-neural-networks-over-lattices. 2.1

[69] Guokun Lai, Qizhe Xie, Hanxiao Liu, Yiming Yang, and Eduard Hovy. RACE: Large-scale ReAding Comprehension dataset from Examinations. In *Proceedings of the 2017 Conference on Empirical Methods in Natural Language Processing*, pages 785–794, Copenhagen, Denmark, September 2017. Association for Computational Linguistics. doi: 10.18653/v1/D17-1082. URL https://aclanthology.org/D17-1082. 5.3

[70] Chris Lattner, Mehdi Amini, Uday Bondhugula, Albert Cohen, Andy Davis, Jacques Pienaar, River Riddle, Tatiana Shpeisman, Nicolas Vasilache, and Oleksandr Zinenko. MLIR: A compiler infrastructure for the end of Moore's law, 2020. URL https://arxiv.org/abs/2002.11054. 2.2.1, 3.7, 4.6

[71] Johannes Lerch, Ben Hermann, Eric Bodden, and Mira Mezini. FlowTwist: Efficient context-sensitive inside-out taint analysis for large codebases. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE 2014, page 98–108, New York, NY, USA, 2014. Association for Computing Machinery. ISBN 9781450330565. doi: 10.1145/2635868.2635878. URL https://doi.org/10.1145/2635868.2635878. 4

[72] levskaya and mattjj. JAX - The Sharp Bits. URL https://jax.readthedocs.io/en/latest/notebooks/Common_Gotchas_in_JAX.html. Last accessed Oct 03, 2022. 2.3.1

[73] Ao Li, Bojian Zheng, Gennady Pekhimenko, and Fan Long. Automatic horizontal fusion for GPU kernels. *CoRR*, abs/2007.01277, 2020. URL https://arxiv.org/abs/2007.01277. 5.3.1, 5.7

[74] Xiuhong Li, Yun Liang, Shengen Yan, Liancheng Jia, and Yinghan Li. A coordinated tiling and batching framework for efficient GEMM on GPUs. In *Proceedings of the 24th Symposium on Principles and Practice of Parallel Programming*, PPoPP '19, page 229–241, New York, NY, USA, 2019. Association for Computing Machinery. ISBN 9781450362252. doi: 10.1145/3293883.3295734. URL https://doi.org/10.1145/3293883.3295734. 2.3.2, 5.6.1

[75] Moshe Looks, Marcello Herreshoff, DeLesley Hutchins, and Peter Norvig. Deep learning with dynamic computation graphs. *CoRR*, abs/1702.02181, 2017. URL http://arxiv.org/abs/1702.02181. 2.3.1, **C.3**, 3, 3.2, 4.2.1

[76] Daniel Lustig and Margaret Martonosi. Reducing GPU offload latency via fine-grained CPU-GPU synchronization. In *2013 IEEE 19th International Symposium on High Performance Computer Architecture (HPCA)*, pages 354–365. IEEE, 2013. 3.6.2

[77] Jiaqi Ma, Zhe Zhao, Xinyang Yi, Jilin Chen, Lichan Hong, and Ed H Chi. 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*, pages 1930–1939, 2018. 2.1.1, 2.1

[78] Michael McCool, Arch D. Robison, and James Reinders. Chapter 8 - fork–join. In Michael McCool, Arch D. Robison, and James Reinders, editors, *Structured Parallel Programming*, pages 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. 2.1.1, 4.2.3

[79] Mahdi Soltan Mohammadi, Kazem Cheshmi, Maryam Mehri Dehnavi, Anand Venkat, Tomofumi Yuki, and Michelle Mills Strout. Extending index-array properties for data dependence analysis. In Mary Hall and Hari Sundar, editors, *Languages and Compilers for Parallel Computing*, pages 78–93, Cham, 2019. Springer International Publishing. ISBN 978-3-030-34627-0. 3.7, 5.7

[80] Ravi Teja Mullapudi, Andrew Adams, Dillon Sharlet, Jonathan Ragan-Kelley, and Kayvon Fatahalian. Automatically scheduling Halide image processing pipelines. *ACM Trans. Graph.*, 35(4), July 2016. ISSN 0730-0301. doi: 10.1145/2897824.2925952. URL https://doi.org/10.1145/2897824.2925952. 2.2.2, 3, 5

[81] Erdal Mutlu, Ruiqin Tian, Bin Ren, Sriram Krishnamoorthy, Roberto Gioiosa, Jacques Pienaar, and Gokcen Kestor. COMET: A domain-specific compilation of high-performance computa-

tional chemistry. *CoRR*, abs/2102.06827, 2021. URL https://arxiv.org/abs/2102.06827. 2.2.2, 5

[82] Payal Nandy, Mary Hall, Eddie C Davis, Catherine Olschanowsky, Mahdi Soltan Mohammadi, Wei He, and Michelle Strout. Abstractions for specifying sparse matrix data transformations. In *Proceedings of the Eighth International Workshop on Polyhedral Compilation Techniques*, 2018. 3.7, 5.7

[83] Rajib Nath, Stanimire Tomov, and Jack Dongarra. An improved magma GEMM for Fermi graphics processing units. *The International Journal of High Performance Computing Applications*, 24(4): 511–515, 2010. doi: 10.1177/1094342010385729. URL https://doi.org/10.1177/1094342010385729. 2.3.2

[84] Graham Neubig, Chris Dyer, Yoav Goldberg, Austin Matthews, Waleed Ammar, Antonios Anastasopoulos, Miguel Ballesteros, David Chiang, Daniel Clothiaux, Trevor Cohn, et al. DyNet: The dynamic neural network toolkit. *arXiv preprint arXiv:1701.03980*, 2017. URL https://arxiv.org/abs/1701.03980. 1.5, **C.3**

[85] Graham Neubig, Yoav Goldberg, and Chris Dyer. On-the-fly operation batching in dynamic computation graphs. In *Proceedings of the 31st International Conference on Neural Information Processing Systems*, NIPS'17, page 3974–3984, Red Hook, NY, USA, 2017. Curran Associates Inc. ISBN 9781510860964. 1.5, 2.3.1, 3, 3.6, 4.2.1, 4.2.1, 4.5.3, 8

[86] John Nickolls, Ian Buck, Michael Garland, and Kevin Skadron. Scalable parallel programming with CUDA: Is CUDA the parallel programming model that application developers have been waiting for? *Queue*, 6(2):40–53, mar 2008. ISSN 1542-7730. doi: 10.1145/1365490.1365500. URL https://doi.org/10.1145/1365490.1365500. 2.3.1

[87] Hiroshi Noji and Yohei Oseki. Effective batching for recurrent neural network grammars. *CoRR*, abs/2105.14822, 2021. URL https://arxiv.org/abs/2105.14822. 2.3.1

[88] Nvidia. FasterTransformer. URL https://github.com/NVIDIA/FasterTransformer. Last accessed Sept 09, 2021. 1.5, 2.3.2

[89] NVIDIA. NVIDIA TensorRT programmable inference accelerator, 2020. URL https://developer.nvidia.com/tensorrt. Last accessed July 18, 2020. 2.2.1, 3.7

[90] OpenBLAS Community. OpenBLAS: An optimized BLAS library. URL https://www.openblas.net/. Last accessed Oct 08, 2021. B.4.7

[91] Tom Le Paine, Pooya Khorrami, Shiyu Chang, Yang Zhang, Prajit Ramachandran, Mark A. Hasegawa-Johnson, and Thomas S. Huang. Fast WaveNet generation algorithm. *CoRR*, abs/1611.09482, 2016. URL http://arxiv.org/abs/1611.09482. 2.1.1

[92] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Kopf, Edward Yang, Zachary DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. PyTorch: An imperative style, high-performance deep learning library. In H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E. Fox, and R. Garnett, editors, *Advances in Neural Information Processing Systems 32*, pages 8024–8035. Curran Associates, Inc., 2019. URL http://papers.neurips.cc/paper/9015-pytorch-an-imperative-style-high-performance-deep-learning-library.pdf. 2.2.1, 4.6

[93] Matt Pharr and William R. Mark. ispc: A SPMD compiler for high-performance cpu programming. In *2012 Innovative Parallel Computing (InPar)*, pages 1–13, 2012. doi: 10.1109/InPar.2012.6339601. 2.3.1

[94] D. Ponomarev, G. Kucuk, and K. Ghose. Reducing power requirements of instruction scheduling through dynamic allocation of multiple datapath resources. In *Proceedings. 34th ACM/IEEE International Symposium on Microarchitecture. MICRO-34*, pages 90–101, 2001. doi: 10.1109/MICRO.2001.991108. 4

[95] PyTorch Community. NestedTensor Project Progress. URL https://github.com/pytorch/pytorch/issues/25032. Last accessed Oct 15, 2021. 5.7

[96] PyTorch Team. The nestedtensor package prototype: Readme.md. URL https://github.com/pytorch/nestedtensor/blob/master/nestedtensor/csrc/README.md. Last accessed Oct 15, 2021. 10

[97] PyTorch Team. TorchScript, 2020. URL https://pytorch.org/docs/stable/jit.html. Last accessed Sept 09, 2021. 2.2.1, 4.6, 5.6.2

[98] PyTorch Team. The nestedtensor package prototype, 2022. URL https://github.com/pytorch/nestedtensor. Last accessed Sept 09, 2021. 5.7

[99] Yuchen Qiao and Kenjiro Taura. An automatic operation batching strategy for the backward propagation of neural networks having dynamic computation graphs, 2019. URL https://openreview.net/forum?id=SkxXwo0qYm. 2.3.1

[100] Alec Radford, Karthik Narasimhan, Tim Salimans, Ilya Sutskever, et al. Improving language understanding by generative pre-training. 2018. 2.1.2

[101] Alec Radford, Jeffrey Wu, Rewon Child, David Luan, Dario Amodei, Ilya Sutskever, et al. Language models are unsupervised multitask learners. *OpenAI blog*, 1(8):9, 2019. 2.1.2

[102] Alexey Radul, Brian Patton, Dougal Maclaurin, Matthew Hoffman, and Rif A. Saurous. Automatically batching control-intensive programs for modern accelerators. In I. Dhillon, D. Papailiopoulos, and V. Sze, editors, *Proceedings of Machine Learning and Systems*, volume 2, pages 390–399. 2020. URL https://proceedings.mlsys.org/paper/2020/file/140f6969d5213fd0ece03148e62e461e-Paper.pdf. 2.3.1, 2.3.1, 3.7

[103] Jonathan Ragan-Kelley, Connelly Barnes, Andrew Adams, Sylvain Paris, Frédo Durand, and Saman Amarasinghe. Halide: A language and compiler for optimizing parallelism, locality, and recomputation in image processing pipelines. In *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '13, page 519–530, New York, NY, USA, 2013. Association for Computing Machinery. ISBN 9781450320146. doi: 10.1145/2491956.2462176. URL https://doi.org/10.1145/2491956.2462176. 2.2.2, 3, 5, 5

[104] Pranav Rajpurkar, Robin Jia, and Percy Liang. Know what you don't know: Unanswerable questions for SQuAD. *CoRR*, abs/1806.03822, 2018. URL http://arxiv.org/abs/1806.03822. 5.3

[105] Prashant Singh Rawat, Fabrice Rastello, Aravind Sukumaran-Rajam, Louis-Noël Pouchet, Atanas Rountev, and P. Sadayappan. Register optimizations for stencils on GPUs. In *Proceedings of the 23rd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '18, page 168–182, New York, NY, USA, 2018. Association for Computing Machinery. ISBN 9781450349826. doi: 10.1145/3178487.3178500. URL https://doi.org/10.1145/3178487.3178500. A.5

[106] Jared Roesch, Steven Lyubomirsky, Marisa Kirisame, Josh Pollock, Logan Weber, Ziheng Jiang, Tianqi Chen, Thierry Moreau, and Zachary Tatlock. Relay: A high-level IR for deep learning. *CoRR*, abs/1904.08368, 2019. URL http://arxiv.org/abs/1904.08368. 2.1, 2.2.1, 4.1

[107] Nadav Rotem, Jordan Fix, Saleem Abdulrasool, Summer Deng, Roman Dzhabarov, James Hegeman, Roman Levenstein, Bert Maher, Nadathur Satish, Jakob Olesen, Jongsoo Park, Artem Rakhov, and Misha Smelyanskiy. Glow: Graph lowering compiler techniques for neural networks. *CoRR*, abs/1805.00907, 2018. URL http://arxiv.org/abs/1805.00907. 2.2.1, 3.7

[108] David E Rumelhart, Geoffrey E Hinton, and Ronald J Williams. Learning representations by back-propagating errors. *nature*, 323(6088):533–536, 1986. 1.1, 2.1

[109] Putt Sakdhnagool, Amit Sabne, and Rudolf Eigenmann. RegDem: Increasing GPU performance via shared memory register spilling. *CoRR*, abs/1907.02894, 2019. URL http://arxiv.org/abs/1907.02894. A.5

[110] Franco Scarselli, Marco Gori, Ah Chung Tsoi, Markus Hagenbuchner, and Gabriele Monfardini. The graph neural network model. *IEEE Transactions on Neural Networks*, 20(1):61–80, 2009. doi: 10.1109/TNN.2008.2005605. 2.1.2

[111] Mike Schuster and Kuldip K Paliwal. Bidirectional recurrent neural networks. *IEEE transactions on Signal Processing*, 45(11):2673–2681, 1997. 4.3.1

[112] Noam Shazeer, Azalia Mirhoseini, Krzysztof Maziarz, Andy Davis, Quoc V. Le, Geoffrey E. Hinton, and Jeff Dean. Outrageously large neural networks: The sparsely-gated mixture-of-experts layer. *CoRR*, abs/1701.06538, 2017. URL http://arxiv.org/abs/1701.06538. 2.1.1, 2.1

[113] Haichen Shen, Jared Roesch, Zhi Chen, Wei Chen, Yong Wu, Mu Li, Vin Sharma, Zachary Tatlock, and Yida Wang. Nimble: Efficiently compiling dynamic neural networks for model inference. In A. Smola, A. Dimakis, and I. Stoica, editors, *Proceedings of Machine Learning and Systems*, volume 3, pages 208–222, 2021. URL https://proceedings.mlsys.org/paper/2021/file/4e732ced3463d06de0ca9a15b6153677-Paper.pdf. 2.2.1, 4.4, 6.4.1

[114] Timothy Sherwood, Suleyman Sair, and Brad Calder. Phase tracking and prediction. *ACM SIGARCH Computer Architecture News*, 31(2):336–349, 2003. 4

[115] Bing Shuai, Zhen Zuo, Gang Wang, and Bing Wang. DAG-recurrent neural networks for scene labeling. *CoRR*, abs/1509.00552, 2015. URL http://arxiv.org/abs/1509.00552. 2.1.1, 2.1, 3.2

[116] Franyell Silfa, José-María Arnau, and Antonio González. E-BATCH: energy-efficient and high-throughput RNN batching. *CoRR*, abs/2009.10656, 2020. URL https://arxiv.org/abs/2009.10656. 2.3.1, 4.2.2, 1

[117] Shikhar Singh, Benoit Steiner, James Hegarty, and Hugh Leather. Using graph neural networks to model the performance of deep neural networks, 2021. 2.2.2, 5

[118] J.E. Smith. Dynamic instruction scheduling and the astronautics zs-1. *Computer*, 22(7):21–35, 1989. doi: 10.1109/2.30730. 4

[119] Shaden Smith and George Karypis. Tensor-matrix products with a compressed sparse tensor. In *Proceedings of the 5th Workshop on Irregular Applications: Architectures and Algorithms*, IA3'15, New York, NY, USA, 2015. Association for Computing Machinery. ISBN 9781450340014. doi: 10.1145/2833179.2833183. URL https://doi.org/10.1145/2833179.2833183. 5.4.3, B.2.1

[120] Richard Socher, Brody Huval, Christopher D. Manning, and Andrew Y. Ng. Semantic compositionality through recursive matrix-vector spaces. In *Proceedings of the 2012 Joint Conference on*

*Empirical Methods in Natural Language Processing and Computational Natural Language Learning*, EMNLP-CoNLL '12, page 1201–1211, USA, 2012. Association for Computational Linguistics. 2.1, 3.2, 4.2

[121] Richard Socher, Alex Perelygin, Jean Wu, Jason Chuang, Christopher D Manning, Andrew Y Ng, and Christopher Potts. Recursive deep models for semantic compositionality over a sentiment treebank. In *Proceedings of the 2013 conference on empirical methods in natural language processing*, pages 1631–1642, 2013. 3.2, 4.2

[122] Lukas Stadler, Thomas Würthinger, and Hanspeter Mössenböck. Partial escape analysis and scalar replacement for java. In *Proceedings of Annual IEEE/ACM International Symposium on Code Generation and Optimization*, pages 165–174, 2014. 6.4.1

[123] M. M. Strout, M. Hall, and C. Olschanowsky. The sparse polyhedral framework: Composing compiler-generated inspector-executor code. *Proceedings of the IEEE*, 106(11):1921–1934, 2018. 3.4.1, 3.7, 5.7

[124] Alex Suhan, Davide Libenzi, Ailing Zhang, Parker Schuh, Brennan Saeta, Jie Young Sohn, and Denys Shabalin. LazyTensor: combining eager execution with domain-specific compilers. *CoRR*, abs/2102.13267, 2021. URL https://arxiv.org/abs/2102.13267. 2.2.1

[125] Kai Sheng Tai, Richard Socher, and Christopher D Manning. Improved semantic representations from tree-structured long short-term memory networks. *arXiv preprint arXiv:1503.00075*, 2015. 2.1.1, 2.1, 3.2, 4.2

[126] XLA Team. XLA - tensorflow, compiled, 2017. URL https://developers.googleblog.com/2017/03/xla-tensorflow-compiled.html. Last accessed Oct 04, 2020. 2.2.1, 3.7

[127] Surat Teerapittayanon, Bradley McDanel, and H. T. Kung. BranchyNet: Fast inference via early exiting from deep neural networks. *CoRR*, abs/1709.01686, 2017. URL http://arxiv.org/abs/1709.01686. 2.1.1, 2.1

[128] TensorFlow Community. Github issue: End2end transformer training by using ragged tensor. URL https://github.com/tensorflow/tensorflow/issues/40965. Last accessed Oct 15, 2021. 5.7

[129] TensorFlow Team. Ragged tensors, 2022. URL https://www.tensorflow.org/api_docs/python/tf/RaggedTensor?version=nightly. Last accessed Sept 09, 2021. 1.3, 5.7

[130] Ruiqin Tian, Luanzheng Guo, Jiajia Li, Bin Ren, and Gokcen Kestor. A high-performance sparse tensor algebra compiler in multi-level IR. *CoRR*, abs/2102.05187, 2021. URL https://arxiv.org/abs/2102.05187. 2.2.2

[131] Christoph Tillmann and Hermann Ney. Word reordering and a dynamic programming beam search algorithm for statistical machine translation. *Computational Linguistics*, 29(1):97–133, 2003. doi: 10.1162/089120103321337458. URL https://www.aclweb.org/anthology/J03-1005. 1, 2.1

[132] Omer Tripp, Marco Pistoia, Stephen J. Fink, Manu Sridharan, and Omri Weisman. 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. 4

[133] Aäron van den Oord, Sander Dieleman, Heiga Zen, Karen Simonyan, Oriol Vinyals, Alex Graves, Nal Kalchbrenner, Andrew W. Senior, and Koray Kavukcuoglu. WaveNet: A generative model for raw audio. *CoRR*, abs/1609.03499, 2016. URL http://arxiv.org/abs/1609.03499. 2.1.1

[134] Harmen L. A. van der Spek, C. W. Mattias Holm, and Harry A. G. Wijshoff. How to unleash array optimizations on code using recursive data structures. In *Proceedings of the 24th ACM International Conference on Supercomputing*, ICS '10, page 275–284, New York, NY, USA, 2010. Association for Computing Machinery. ISBN 9781450300186. doi: 10.1145/1810085.1810123. URL https://doi.org/10.1145/1810085.1810123. 3.7

[135] Nicolas Vasilache, Oleksandr Zinenko, Theodoros Theodoridis, Priya Goyal, Zachary DeVito, William S Moses, Sven Verdoolaege, Andrew Adams, and Albert Cohen. Tensor comprehensions: Framework-agnostic high-performance machine learning abstractions. *arXiv preprint arXiv:1802.04730*, 2018. 2.2.2, 5

[136] Nicolas Vasilache, Oleksandr Zinenko, Aart J. C. Bik, Mahesh Ravishankar, Thomas Raoux, Alexander Belyaev, Matthias Springer, Tobias Gysi, Diego Caballero, Stephan Herhut, Stella Laurenzo, and Albert Cohen. Composable and modular code generation in MLIR: A structured and retargetable approach to tensor compiler construction. *CoRR*, abs/2202.03293, 2022. URL https://arxiv.org/abs/2202.03293. 2.2.2

[137] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin. Attention is all you need. *CoRR*, abs/1706.03762, 2017. URL http://arxiv.org/abs/1706.03762. (document), 1.1, 5.6.2

[138] Minjie Wang, Lingfan Yu, Da Zheng, Quan Gan, Yu Gai, Zihao Ye, Mufei Li, Jinjing Zhou, Qi Huang, Chao Ma, Ziyue Huang, Qipeng Guo, Hao Zhang, Haibin Lin, Junbo Zhao, Jinyang Li, Alexander J. Smola, and Zheng Zhang. Deep graph library: Towards efficient and scalable deep learning on graphs. *CoRR*, abs/1909.01315, 2019. URL http://arxiv.org/abs/1909.01315. 3.7

[139] Xiaohui Wang, Ying Xiong, Yang Wei, Mingxuan Wang, and Lei Li. LightSeq: A high performance inference library for transformers. In *Proceedings of the 2021 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies: Industry Papers*, pages 113–120, Online, June 2021. Association for Computational Linguistics. doi: 10.18653/v1/2021.naacl-industry.15. URL https://aclanthology.org/2021.naacl-industry.15. 6.3

[140] Alex Warstadt, Amanpreet Singh, and Samuel R. Bowman. Neural network acceptability judgments. *Transactions of the Association for Computational Linguistics*, 7:625–641, March 2019. doi: 10.1162/tacl_a_00290. URL https://aclanthology.org/Q19-1040. 5.3

[141] Wikipedia. Wikipedia. URL https://en.wikipedia.org/wiki/Main_Page. Last accessed Mar. 14, 2022. 5.6.2

[142] Adina Williams, Nikita Nangia, and Samuel Bowman. A broad-coverage challenge corpus for sentence understanding through inference. In *Proceedings of the 2018 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1 (Long Papers)*, pages 1112–1122, New Orleans, Louisiana, June 2018. Association for Computational Linguistics. doi: 10.18653/v1/N18-1101. URL https://aclanthology.org/N18-1101. 5.3

[143] S. Xiao and W. Feng. Inter-block GPU communication via fast barrier synchronization. In *2010 IEEE International Symposium on Parallel Distributed Processing (IPDPS)*, pages 1–12, 2010. 3.6.2

[144] Ji Xin, Raphael Tang, Jaejun Lee, Yaoliang Yu, and Jimmy Lin. Deebert: Dynamic early exiting for accelerating BERT inference. *CoRR*, abs/2004.12993, 2020. URL https://arxiv.org/abs/2004.12993. 2.1.1, 2.1

[145] Ji Xin, Raphael Tang, Yaoliang Yu, and Jimmy Lin. 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*, pages 91–104, 2021. 4.2

[146] Shizhen Xu, Hao Zhang, Graham Neubig, Wei Dai, Jin Kyu Kim, Zhijie Deng, Qirong Ho, Guangwen Yang, and Eric P. Xing. Cavs: An efficient runtime system for dynamic neural networks. In *2018 USENIX Annual Technical Conference (USENIX ATC 18)*, pages 937–950, Boston, MA, July 2018. USENIX Association. ISBN 978-1-939133-01-4. URL https://www.usenix.org/conference/atc18/presentation/xu-shizen. 2.3.1, 3.6, 3.6.2, 4.2.2, 1

[147] Kevin Yang, Violet Yao, John DeNero, and Dan Klein. A streaming approach for efficient batched beam search. *CoRR*, abs/2010.02164, 2020. URL https://arxiv.org/abs/2010.02164. 2.3.1

[148] Zhilin Yang, Zihang Dai, Yiming Yang, Jaime G. Carbonell, Ruslan Salakhutdinov, and Quoc V. Le. XLNet: Generalized autoregressive pretraining for language understanding. *CoRR*, abs/1906.08237, 2019. URL http://arxiv.org/abs/1906.08237. 5.6.2

[149] Jiaxuan You, Rex Ying, Xiang Ren, William L. Hamilton, and Jure Leskovec. Graphrnn: A deep generative model for graphs. *CoRR*, abs/1802.08773, 2018. URL http://arxiv.org/abs/1802.08773. 2.1.1, 2.1

[150] Yuan Yu, Martín Abadi, Paul Barham, Eugene Brevdo, Mike Burrows, Andy Davis, Jeff Dean, Sanjay Ghemawat, Tim Harley, Peter Hawkins, et al. Dynamic control flow in large-scale machine learning. In *Proceedings of the Thirteenth EuroSys Conference*, pages 1–15, 2018. 2.2.1

[151] Sheng Zha, Ziheng Jiang, Haibin Lin, and Zhi Zhang. Just-in-time dynamic-batching. *CoRR*, abs/1904.07421, 2019. URL http://arxiv.org/abs/1904.07421. 2.3.1, 4.2.2, 1

[152] Lingqi Zhang, Mohamed Wahib, and Satoshi Matsuoka. Understanding the overheads of launching CUDA kernels. 3.6.2

[153] Minjia Zhang, Samyam Rajbhandari, Wenhan Wang, and Yuxiong He. DeepCPU: Serving RNN-based deep learning models 10x faster. In *2018 USENIX Annual Technical Conference (USENIX ATC 18)*, pages 951–965, Boston, MA, July 2018. USENIX Association. ISBN 978-1-939133-01-4. URL https://www.usenix.org/conference/atc18/presentation/zhang-minjia. 3.7

[154] Zhiming Zhang and J. Morris Chang. A cool scheduler for multi-core systems exploiting program phases. *IEEE Transactions on Computers*, 63(5):1061–1073, 2014. doi: 10.1109/TC.2012.283. 4

[155] Bojian Zheng, Ziheng Jiang, Cody Hao Yu, Haichen Shen, Joshua Fromm, Yizhi Liu, Yida Wang, Luis Ceze, Tianqi Chen, and Gennady Pekhimenko. DietCode: Automatic optimization for dynamic tensor programs. In D. Marculescu, Y. Chi, and C. Wu, editors, *Proceedings of Machine Learning and Systems*, volume 4, pages 848–863, 2022. URL https://proceedings.mlsys.org/paper/2022/file/fa7cdfad1a5aaf8370ebeda47a1ff1c3-Paper.pdf. 2.2.2, 4.3.3

[156] Lianmin Zheng, Chengfan Jia, Minmin Sun, Zhao Wu, Cody Hao Yu, Ameer Haj-Ali, Yida Wang, Jun Yang, Danyang Zhuo, Koushik Sen, Joseph E. Gonzalez, and Ion Stoica. Ansor: Generating high-performance tensor programs for deep learning. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pages 863–879. USENIX Association, November 2020. ISBN 978-1-939133-19-9. URL https://www.usenix.org/conference/osdi20/presentation/zheng. 2.2.2, 3, 4.3.3, 5, 5.5

[157] Zhen Zheng, Pengzhan Zhao, Guoping Long, Feiwen Zhu, Kai Zhu, Wenyi Zhao, Lansong Diao, Jun Yang, and Wei Lin. FusionStitching: Boosting memory intensive computations for deep learning workloads. *CoRR*, abs/2009.10924, 2020. URL https://arxiv.org/abs/2009.10924. 2.2.1, 5.7

[158] Kai Zhu, Wenyi Zhao, Zhen Zheng, Tianyou Guo, Pengzhan Zhao, Junjie Bai, Jun Yang, Xiaoyong Liu, Lansong Diao, and Wei Lin. DISC: A dynamic shape compiler for machine learning workloads. *CoRR*, abs/2103.05288, 2021. URL https://arxiv.org/abs/2103.05288. 2.2.2