THE COR TENSOR COMPILER: COMPIL TION FOR R GGED TENSORS WITH MINIM L P DDING

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

BSTR CT

There is often variation in the shape and size of input data used for deep learning. In many cases, such data can be represented using tensors with non-uniform shapes, or *ragged tensors*. Due to limited and non-portable support for efficient execution on ragged tensors, current deep learning frameworks generally use techniques such as padding and masking to make the data shapes uniform and then offload the computations to optimized kernels for dense tensor algebra. Such techniques can, however, lead to a lot of wasted computation and therefore, a loss in performance. This paper presents COR , a tensor compiler that allows users to easily generate efficient code for ragged tensor operators targeting a wide range of CPUs and GPUs. Evaluating COR on a variety of operators on ragged tensors as well as on an encoder layer of the transformer model, we find that COR (i) performs competitively with hand-optimized implementations of the operators and the transformer encoder and (ii) achieves a 1 6× geomean speedup over PyTorch for the encoder on an Nvidia GPU and a 1 37× geomean speedup over TensorFlow for the multi-head attention module used in transformers on a 64-core RM CPU.

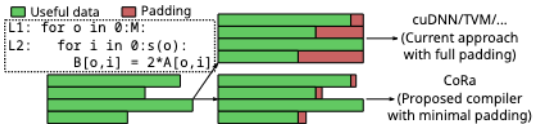


Figure 1: n example operation on ragged tensors.

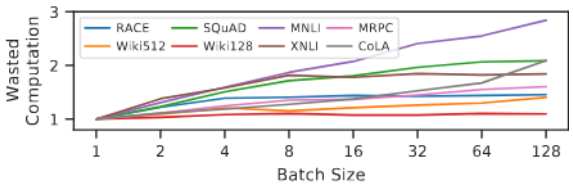


Figure 2: Wasted computation due to padding in a transformer encoder layer.

1 INTRODUCTION

Deep learning (DL) is used for a variety of computational tasks on different kinds of data including *sequential data* like text (Tai et al., 2015; Vaswani et al., 2017), audio (van den Oord et al., 2016) and music (Briot et al., 2017; Huang et al., 2018) and *spatial data* like images (He et al., 2016). Simultaneously, DL models have become more and more computationally expensive. More efficient execution of these models is, therefore, a priority.

There is often variation in the sizes of the data that we process using DL. Images can be of different resolutions, textual sentences and documents can be of different lengths, and audio can be of different durations. Processing such data exhibiting variation in shape, or *shape dynamism* (Shen et al., 2020), using the same model and in the same mini-batch is therefore important. n example elementwise operation on such data is shown in Fig. 1, where the slices of the inner dimension of tensor have variable sizes. Such tensors and operators are referred to as *ragged tensors* and *ragged operators* respectively. Note how the shape dynamism translates to a variable bound for loop `L2`, which iterates over the variable-sized tensor slices.

Past work has developed hand-optimized kernels to accelerate some important ragged applications such as batched matrix multiplication with variable dimensions (Li et al., 2019; Nath et al., 2010), triangular matrix multiplication (Charara et al., 2016) and the widely-used transformer (Vaswani et al., 2017) models (ByteDance; Nvidia). Such hand-optimized kernels, however, require substantial development effort and, hence, are available only for a few operators. Further, they are not portable across different hardware substrates, which is problematic due to the rapid innovation in DL hardware.

While some DL frameworks have started providing support for ragged operators recently (TensorFlow Team, 2022; PyTorch Team, 2022), it is quite limited (TensorFlow Community; PyTorch Community; PyTorch Team) as discussed in §8. Therefore, frameworks usually rely on efficient dense tensor algebra kernels implemented in vendor libraries such as cuDNN (Chetlur et al., 2014) and oneDNN (Intel) or generated by tensor compilers such as TVM (Chen et al., 2018a)

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

Proceedings of the 5th MLSys Conference, Santa Clara, C , US , 2022.

to target parallel hardware. Padding (illustrated in the top right of Fig. 1) and masking[1] are therefore commonly used to eliminate shape dynamism in ragged tensors and enable the use of vendor libraries or dense tensor compilers (HuggingFace, 2020).

Padding and masking, however, lead to wasted computation as the padding or the masked data points are discarded after execution. Fig. 2 plots the relative amount of computation (computed analytically in FLOPs) involved in the forward pass of an encoder layer of the transformer model[2] 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.

Thus, current solutions for efficient ragged operator execution are unsatisfactory. Hence, we propose a compiler-based solution enabling easy and more portable generation of performant code for ragged operators. While sparse (Tian et al., 2021; Kjolstad et al., 2017) and dense (Chen et al., 2018a; Vasilache et al., 2018; Ragan-Kelley et al., 2013; Baghdadi et al., 2019) tensor compilers have been well-studied, it is not straightforward to apply these techniques to ragged tensors, due to the following challenges:

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.

With these challenges in mind, we present CoR (**Co**mpiler for **Ra**gged Tensors), 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**, CoR enables minimal padding of ragged tensor dimensions (§4.1) in order to generate efficient code for targets such as GPUs as well as to specify thread remapping strategies to lower load imbalance (§4.1). CoR uses uninterpreted functions (Strout et al., 2018) to symbolically represent variable loop bounds and scheduling operations on the same (§5.1). CoR 's mechanisms (such as its storage lowering scheme discussed in §5.3) and optimizations are specialized for ragged tensors thereby tackling **C2**. Further, CoR provides simple abstractions to convey to the compiler information essential to efficient code generation, such as padding or thread remapping specifications and raggedness patterns of tensors (§4). This overcomes challenge **C3**.

[1]Masking involves setting some tensor elements to a special value so that these elements are ignored in computations.

[2]The hyperparameters used are the same as those in §7.2.

Table 1: Comparison between CoR and current solutions for ragged operations. TC stands for tensor compilers.

Framework	Portability	Operator impl. effort	Padding	Performance
Dense TC	High	Low	Full	Low
Sparse TC	High	Low	Minimal	Low
Dense vendor libs.	Low	High	Full	Low
Hand-optimized impl.	Low	High	Minimal	High
CoR	High	Low	Minimal	High

CoR enables efficient code generation for ragged operators by significantly reducing padding (§7). s part of CoR 's implementation, we reuse past work by extending a tensor compiler (Ragan-Kelley et al., 2013; Chen et al., 2018a; Baghdadi et al., 2019; Kjolstad et al., 2017) and thus, provide familiar interfaces to CoR 's users. This also makes it easy in the future to use auto-scheduling (Mullapudi et al., 2016; dams et al., 2019; Chen et al., 2018b; Zheng et al., 2020a; Singh et al., 2021) for optimizing ragged tensor operations. Table 1 compares CoR with alternatives that are or could be used for ragged operators. Only CoR achieves high performance and portability, with low operator implementation effort (and minimal padding).

In summary, this paper makes the following contributions:

1. We present CoR , a tensor compiler for ragged tensors. To our knowledge, CoR is the first tensor compiler that allows efficient computation on ragged tensors.
2. s part of the design, we generalize the PI, abstractions and the mechanisms of tensor compilers and propose new scheduling primitives for ragged tensors.
3. We evaluate CoR on a variety of ragged operators. For a transformer encoder layer, we perform 1 6× better than PyTorch (Paszke et al., 2019) and as well as FasterTransformer (Nvidia), a highly optimized transformer implementation, on an Nvidia V100 GPU. On a 64-core RM CPU, we are 1 37× faster than TensorFlow (badi et al., 2016), for the multi-head attention (MH) module (Vaswani et al., 2017) used in transformers.

2 CoR Overview

CoR 's compiler-based approach enables the generation of performant code in a portable manner. This is reflected in Fig. 3, which compares CoR 's implementation of a transformer encoder layer with FasterTransformer. The highly-

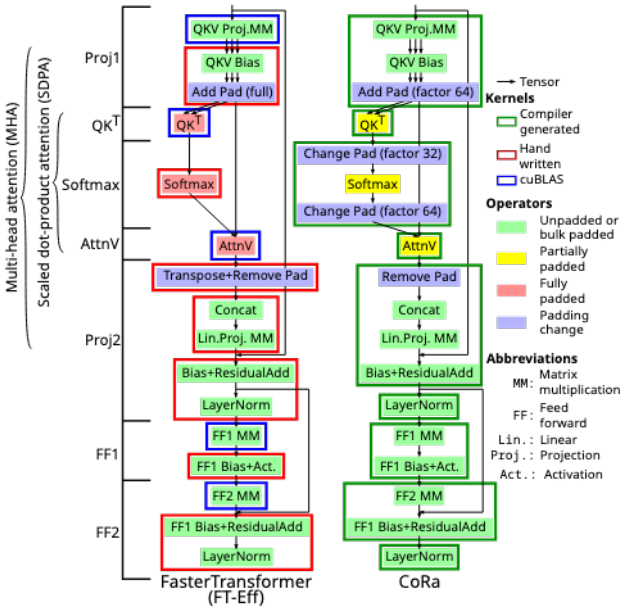


Figure 3: FasterTransformer (FT-Eff) and CoR implementations of a transformer's encoder layer. Note how CoR 's fully compiler-based implementation uses only partial padding for SDP as opposed to FasterTransformer's fully padded implementation. CoR also enables more operator fusion (including fusing all the padding change operations) as opposed to FasterTransformer, which cannot do so in all cases as it relies on vendor libraries.

optimized FasterTransformer relies heavily on kernels implemented in cuBL S (Nvidia's BL S library), which are shown as blue outlines in the figure, and on manually implemented kernels, shown as red outlines. On the other hand, CoR 's implementation exclusively employs compiler generated kernels (shown as green outlines), making it more portable. Further, CoR 's compiler approach allows it to exploit more kernel fusion opportunities, evident from the fact that CoR '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 (SDP) sub-module, where CoR 's specialized approach enables it to get away with lower padding as compared to FasterTransformer. We further discuss these implementations in §7.

CoR '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 (§5.3). This is unlike sparse formats such as compressed sparse row (CSR), where accesses require a search over an array. The H SH (Chou et al., 2018) sparse format, while allowing $O(1)$ accesses, is unsuitable for accelerators such as GPUs due to its highly irregular storage.

Insight **I1** allows CoR 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 CoR to generate efficient code for ragged operations.

Let us now look at CoR 's overall compilation and execution pipeline, as illustrated in Fig. 4. The user first expresses ① and schedules ② their computation using an PI similar to that of past tensor compilers (§4). This specification of the computation and the scheduling primitives are then lowered ③ to an SS -based IR ④. s part of this lowering step, CoR generates code ⑦ to initialize some auxiliary data structures it needs to be able to lower accesses to ragged tensors (§5.3) and to enable loop fusion in ragged loop nests (§5.1). We refer to this code as the *prelude* code. Compilation then continues with CoR lowering tensor accesses to raw memory offsets by making use of the data structures generated by the prelude. Finally, CoR generates ⑤ target-dependent code ⑨ such as C or CUD 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 (§7.4) 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.

3 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*.

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 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*. 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.

4 CoR 'S R GGED PI

CoR provides a simple PI similar to that of past tensor compilers, as seen in Listing 1, which expresses the example computation from Fig. 1 in CoR . part from describing the computation as in a dense tensor compiler,

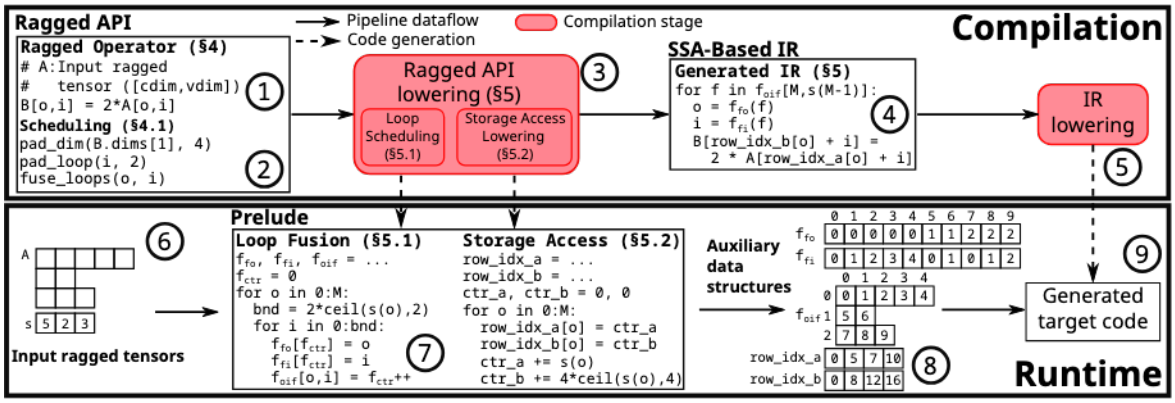


Figure 4: Overview of CoR 's compilation and runtime pipeline.

CoR also requires the user to specify the raggedness dependences of the computation (highlighted in Listing 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, CoR automatically computes any derived data structures required (§5), making it easy for users to express their computations. CoR uses identifiers called *named dimensions* (discussed further in §5.2) to name loops and corresponding tensor dimensions and to specify relationships between them. For example, the loop extent defined on line 7 in the listing states the dependence on the outer loop, referred to by the named dimension `batch_dim`.

```
1  ################ Operator  escription ################
2  batch_size = var('M')
3  #  eclare named dimensions
4  batch_dim, len_dim = Dim(), Dim()
5  # Loop: Specify vloop extents
6  lens = input_tensor((batch_size,))
7  l_ext = Extent([batch_dim], l mbd  b: lens[b])
8  loop_exts = [batch_size, l_ext]
9  # Storage: Specify vdim extents
10 s_ext = Extent([batch_dim], l mbd  b: lens[b])
11 storage_format = [batch_size, s_ext]
12 #  efine input ragged tensor
13 dims = [batch_dim, len_dim]
14   = input_tensor(dims, storage_format)
15 # Express computation
16 B = compute(dims, loop_exts, l mbd  i,j: 2* [i,j])
17 ############### Scheduling primitives ###############
18 pad_loop(B.loops[1], 2)
19 pad_dimension(B.dimensions[1], 4)
20 fuse_loops(B.loops[0], B.loops[1])
```

Listing 1: Operator in Fig. 1 expressed in a simplified version of CoR 's PI.

4.1 Scheduling Primitives

In order to optimize the expressed computation, CoR 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 CoR . 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. CoR currently does not allow reordering such a loop L_v beyond any of the loops L_1 to L_k. lthough possible with the introduction of conditional statements, we have not found a use case for such reordering.

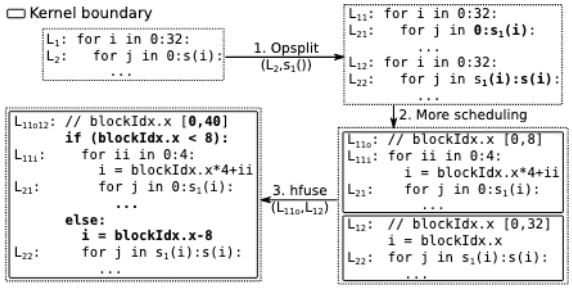


Figure 5: Operation splitting and horizontal fusion. Loop `L` is first split in step 1 using operation splitting thus creating two loop nests, which are then horizontally fused together (step 3) so they execute concurrently as part of single kernel.

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. CoR allows one to split an operation into two or more operations by specifying split points for one or more of its loops, as Fig. 5 shows. In our evaluation (§7.3), 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 ttnV operators in the transformer layer (Fig. 3).

Horizontal Fusion: Past work (Li et al., 2020) 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 CoR , 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.

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 condi-

tional checks. ccordingly, CoR allows the user to specify padding for vloops and vdims as multiples of a constant. For example, on line 18 of Listing 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 19. 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 CoR to fuse padding change operators as is illustrated in Fig. 3. We show in §7.4 that this partial padding does not lead to much wasted computation.

Tensor Dimension Scheduling: CoR 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 (§5.1).

Load Balancing: The variable loop bounds in a vloop nest can lead to unbalanced load across execution units. s proposed by past work (Gale et al., 2020) on sparse tensor algebra, CoR 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 Fig. 15 in the appendix). Depending on the hardware thread scheduling policy, this can influence the order in which iterations of the loop are scheduled and lead to non-trivial performance gains as shown in §7.1.

In conclusion, CoR 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.

5 CoR 's R GGED PI LOWERING

We now discuss some aspects of CoR 's Ragged PI lowering that generates the SS -based IR as shown in Fig 4.

5.1 Loop and Tensor Dimension Fusion

Consider the ragged loop nest shown on the top left corner of Fig 6. The loop bound of the inner loop $\mathtt{L_i}$ is a function s) of $\mathtt{o}$, the iteration variable of the outer loop $\mathtt{L_o}$. The loop $\mathtt{L_f}$ obtained by fusing $\mathtt{L_o}$ and $\mathtt{L_i}$ is shown on the right of the figure. The loop bound $\mathtt{F}$ of the fused loop would be equal to $\sum_{o=0}^{M} {}^{1} s\ o)$. Further note that while we have fused the loops $\mathtt{L_o}$ and $\mathtt{L_i}$, the tensor access `T[o,i]` in the body of the loop nest still uses variables $\mathtt{o}$ and `i`. Therefore, we need to compute the values of these two variables corresponding to the current value of `f`, the iteration variable of $\mathtt{L_f}$. 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

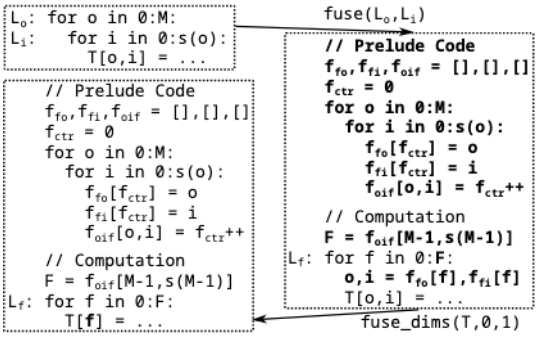


Figure 6: Fusing vloops and tensor dimensions

loop nests is not straightforward. In CoR , we generate code to compute these quantities and variable relationships (shown in the right pane of Fig. 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 (Fig. 3) with minimal padding.

Suppose now that the tensor `T` in Fig. 6 has a storage format that mirrors the loop nest consisting of $\mathtt{L_o}$ and $\mathtt{L_i}$. 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 CoR to simplify the tensor access as shown in the bottom left pane of the figure.

5.2 Bounds Inference

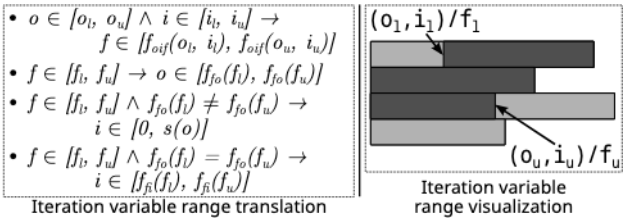


Figure 7: Iteration variable ranges during vloop fusion.

Variable Loop Fusion: During compilation, a tensor compiler infers loop bounds for all operators. In order to do so, the compiler usually proceeds from the outputs of the operator graph towards the inputs, inferring the region of a tensor t that needs to be computed and then using this information to infer the loop bounds for the operator that computes t. s we saw in §5.1, 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 Fig. 6, Fig 7 provides the rules to translate between the ranges of iteration variables $\mathtt{o}$, `i`

and f as well as a visualization of the ranges. 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[3].

Named Dimensions: In §4, 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. long 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 CoR , we use named dimensions to easily identify corresponding iteration variables across such producers and consumers to allow this translation.

5.3 Storage ccess Lowering

In this section, we briefly describe how CoR lowers accesses to ragged tensors. Consider the 4-dimensional attention matrix X involved in a batched implementation of MH shown in the left pane of Fig. 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.[4] 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 (Smith & Karypis, 2015; Chou et al., 2018) 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 Fig. 8), past work would compute and store more auxiliary data as compared to CoR .

CoR '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. ccessing the precomputed memory offsets is also a constant time operation as CoR 's auxiliary data structures store these offsets using simple arrays.

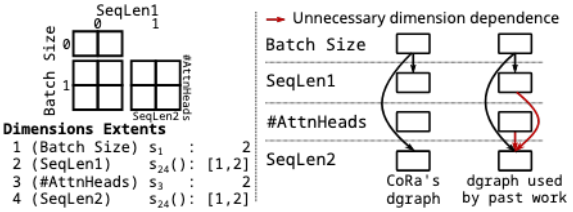


Figure 8: CoR precisely models dimension dependences as compared to past schemes for sparse tensors.

We describe these lowering schemes further in the appendix in §B.1. In short, our storage access lowering scheme reduces the amount of auxiliary data that needs to be computed thus reducing overheads of the prelude code (§7.4), while allowing cheap tensor accesses.

6 Implement tion

We prototype CoR by extending TVM (Chen et al., 2018a) v0.6, a DL framework and a tensor compiler. Some details regarding this implementation are discussed below.

Ragged PI: 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 (Zheng et al., 2020a) 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. Our implementation currently expects users to correctly allocate memory (taking into account padding requirements as specified in the schedule) for tensors. Checks to report these problems could be easily implemented.

7 Ev lu tion

We evaluate CoR against state-of-the-art baselines, first, on two ragged variants of the gemm (general matrix multiplication) operation in §7.1 and then on an encoder layer of the transformer model (Fig. 3) in §7.2. Our experimental environment is described in Table 2. Below, we refer to the

[3] In the generated code, as seen in the right pane of Fig. 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.

[4] We use the same layout in CoR 's implementation in §7.2.

Table 2: Our experimental environment

Hardware	Software (ll instances ran Ubuntu 20.04.)
Nvidia Tesla V100 GPU (Google cloud n1-standard-8 instance)	CUD 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 RM Graviton2 CPU (WS c6g.2xlarge instance)	PyTorch 1.10.0a0+git36449ea (with oneDNN 2.4 and rm compute library 21.11), TensorFlow 2.6.0 (with oneDNN 2.3 and rm compute library 21.05), OpenBL S 0.3.10
64 core RM Graviton2 CPU (WS c6g.16xlarge instance)	

four platforms listed in the table as Nvidia GPU, Intel CPU, 8-core RM CPU and 64-core RM CPU. Our evaluation is performed with single-precision floating point numbers.

7.1 Matrix Multiplication

We start by evaluating CoR 's performance on the variable-sized batched gemm (or vgemm) and the triangular matrix multiplication (or trmm) operators. s with all the implementations we compare against, the CoR 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 CoR 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 (Li et al., 2019) on vgemm and cuBL S's implementation of fully padded batched gemm. We use synthetically generated data where matrix dimensions are uniformly randomly chosen multiples of 128 in $[512, 1408]$. CoR '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 optimized microkernels.

s Fig. 9 shows, CoR 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, CoR 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: 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 CoR . In this section, we evaluate CoR on the trmm operator wherein we multiply

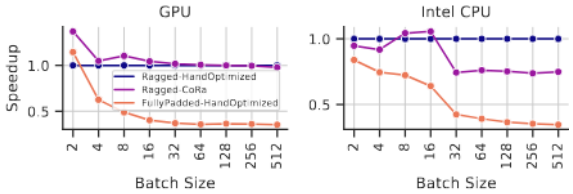


Figure 9: Performance comparison of CoR 's vgemm and hand-optimized implementations of vgemm and fully padded gemm.

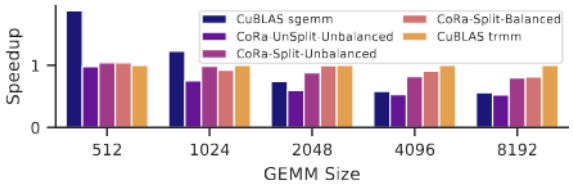


Figure 10: CoR 's trmm performance compared against cuBL S's hand-optimized trmm and fully-padded gemm implementations.

a square lower triangular matrix with a square dense matrix, on the Nvidia GPU. We compare against cuBL S's trmm and fully padded gemm implementations. In trmm, the reduction loop is a vloop. In order to efficiently handle the last few iterations of this loop after tiling, we use operation splitting[5] (§4). Further, the raggedness in this loop leads to imbalanced load across the GPU thread blocks. We use thread remapping (§4.1) to schedule thread blocks with the most amount of work first, leading to more balanced load.

Fig. 10 shows the performance of the aforementioned cuBL S implementations and three implementations in CoR —CoR -unsplit-unbalanced, CoR -split-unbalanced and CoR -split-balanced—which progressively employ operation splitting and thread remapping, starting with neither. We see the trmm implementations—both cuBL S's and CoR 's—are beneficial as compared to cuBL S's dense sgemm operator only for larger matrices. In all cases, however, the CoR -split-balanced implementation performs within 81.3% of cuBL S's hand-optimized trmm implementation. Operation splitting leads to a significant increase in performance by allowing CoR to elide conditional checks in the main body of the computation. Further, a better load distribution with thread remapping also helps CoR achieve performance close to cuBL S.

7.2 The Transformer Model

We now move on to look at how CoR performs on various modules of the transformer model. We mainly focus on the GPU backend as it is more commonly used for these

[5] 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 3: Datasets used in our evaluation

Dataset (Short name, if any)	Min. / Mean / Max. SeqLength
R CE (Lai et al., 2017)	80 / 364 / 512
English Wikipedia with SeqLen 512 (Wiki512)	12 / 371 / 512
SQu D v2.0 (Rajpurkar et al., 2018) (SQu D)	39 / 192 / 384
English Wikipedia with SeqLen 128 (Wiki128)	14 / 117 / 128
MNLI (Williams et al., 2018)	9 / 43 / 128
XNLI (Conneau et al., 2018)	9 / 70 / 128
MRPC (Dolan & Brockett, 2005)	21 / 59 / 102
CoL (Warstadt et al., 2019)	6 / 13 / 37

Table 4: Transformer encoder layer execution latencies (in ms) for CoR , PyTorch and the two manually-optimized variants of FasterTransformer on the Nvidia GPU. CoR 's execution latencies include prelude overheads assuming a 6 layer transformer encoder.

Dataset	Batch Size	PyTorch	FT	CoR	FT-Eff
R CE	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
SQu D	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
CoL	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

models. We use a 6 layer model with a hidden dimension of 512 and 8 attention heads each of size 64. The encoder layer 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 (Vaswani et al., 2017). We use sequence lengths from some commonly used NLP datasets listed in Table 3.[6] We focus on larger batch sizes (32, 64 and 128) because, as we saw in Fig. 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 CoR 's implementations play a big role. In this section, CoR 's implementations use ragged tensor storage.

Transformer Encoder Layer: We first evaluate the forward pass latency of an encoder layer of the transformer model (Fig. 3). We compare CoR 's performance with that of FasterTransformer and an implementation in PyTorch, a popular DL framework, with TorchScript (PyTorch Team, 2020) enabled. ll the operators in the encoder layer except the ones in the SDP 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 operations in this case. FasterTransformer provides an option to perform this optimization, first introduced in EffectiveTransformers (ByteDance). We compare against FasterTransformer both with and without this optimization. We refer to these two implementations as FT-Eff and FT, respectively. In the CoR implementation, this optimization is applied simply by loop fusion, analogous to the illustration in Fig. 6. In CoR '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* (Fig. 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 4 shows the forward execution latencies for the encoder layer for the aforementioned frameworks and datasets. The auxiliary data structures computed by CoR '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 CoR include per-layer prelude overheads assuming a 6 layer model. We further look at these overheads in §7.4. s we can see, the CoR 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. Fig. 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 CoR implementations which are sketched in Fig. 3.[7] The FT implementation is similar to the FT-Eff implementation except it uses full padding for all operations. The CoR 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. Fig. 13 shows a breakdown of the execution times for these implementations for the R CE dataset and batch size 128 at the level of these sub-graphs.[8] s Fig. 3 shows, the FT-Eff and CoR implementations differ significantly with respect to padding only in the SDP sub-module where the FT-Eff implementation employs full padding while the CoR employs partial padding. We see, in Fig 13, that the CoR implementation performs better than FasterTransformer for all the SDP operators (QK^T,

[6] More details can be found in §D.1.

[7] FasterTransformer uses specialized implementations for different GPUs. In this paper, 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 10 in the appendix.

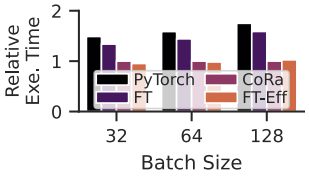


Figure 11: Relative execution times of the transformer encoder layer on the GPU.

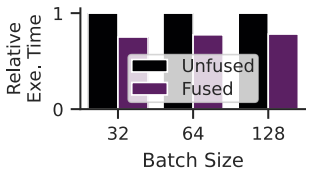


Figure 12: Benefits of padding change operator fusion for the MH module.

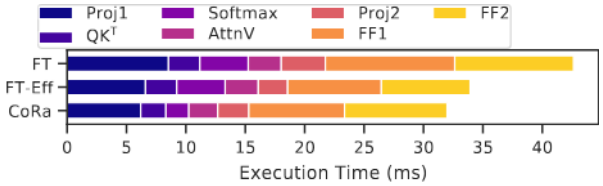


Figure 13: Breakdown of the encoder layer execution times for the R CE dataset at batch size 128. This data is obtained with profiling turned on and might deviate from Table 4.

Softmax and ttnV) despite the fact that the latter is heavily hand optimized.[9] This is because CoR 's ability to handle raggedness enables it to perform less wasted computation. For the remaining operators where both the CoR and FT-Eff implementations employ little to no padding, we see that the CoR 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 cuBL S's extensively optimized gemm kernels for the linear transformation operators and into hand-optimized kernels for the rest. CoR 's performance drops slightly for datasets with smaller sequence lengths as well as for smaller batch sizes. s we discuss in §D.8, this performance difference can be reduced by further optimizing the schedules used for the projection and feed forward operators in CoR 's implementation for smaller batch sizes and sequence lengths. Further, we also note that the overheads associated with the prelude code and partial padding (§7.4) 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 CoR can due to its compiler-based approach. Specifically CoR can completely fuse all operators which add or remove padding in its implementation (as shown in Fig. 3). This is as opposed to the FT-Eff implementation, which cannot. Fusing these padding change operators leads to a significant drop in CoR 's execution latency as seen in Fig. 12, which shows the execution latencies of the MH module for the R CE dataset in CoR with and without this fusion enabled.

Masked Scaled Dot-Product ttention: The decoder layer of a transformer uses a variant of MH called *masked MH* wherein the upper half of the attention matrix is masked for all attention heads during training. This masking only affects the SDP module, the operators in which can now be seen as computing on a batch of lower triangular matrices. We saw in §7.1 that CoR can effectively generate code for operations on triangular matrices. For batch size 128, an implementation of masked SDP in CoR which exploits this masking performs 1 56× faster than an implementation which does not for the R CE dataset and 1 29× for 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 SDP in §D.3 in the appendix.

[9]The execution times of the three SDP operators is quadratically proportional to the sequence length, unlike the remaining operators which are linearly proportional. We discuss the performance of SDP further in §D.8 of the appendix.

Table 5: MH execution latencies (in ms) on the 64-core RM CPU for TensorFlow and CoR .

Dataset	Batch Size 32			Batch Size 64			Batch Size 128		
	TF	TF-UB	CoR	TF	TF-UB	CoR	TF	TF-UB	CoR
R CE	55	46	**44**	111	88	**85**	209	**156**	168
Wiki512	53	53	**47**	106	96	**91**	205	**172**	176
SQu D	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**
CoL	5	4	**2**	6	6	**3**	9	8	**5**

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 §D.5). 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 and ragged tensors can help alleviate memory bottlenecks along with other memory management techniques for training (Kirisame et al., 2021; Jain et al., 2020).

MH Evaluation on RM CPU: Table 5 shows the execution latencies of MH implementations in TensorFlow and CoR on the 64-core RM 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. cross the datasets and batch sizes evaluated, we see that CoR 's implementation is overall 1 57× faster than TF and 1 37× faster than TF-UB. In this case, too, CoR 's ability to save on wasted computation due to padding leads to significant performance gains over a popular DL framework. §D.8 of the appendix more extensively compares the performance

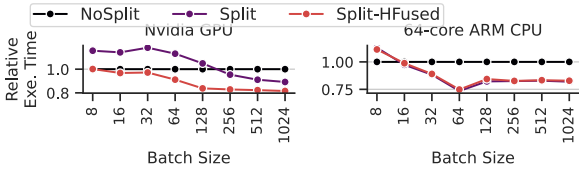


Figure 14: Benefits of operator splitting and hfusion. Note that the y-axis does not start at 0.

of TensorFlow and PyTorch against CoR on both the 8- and 64-core RM CPUs.

7.3 Operation Splitting and Horizontal Fusion

We now evaluate operator splitting and hfusion on the ttnV operator, which is an instance of the vgemm problem. t-tnV 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, Fig. 14 shows the relative execution times of three CoR implementations of ttnV—NoSplit, Split and Split-HFused—in which we progressively perform the two optimizations, on the Nvidia GPU and 64-core RM 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 levels of CPU parallelism also mean that hfusion has no benefit in this case. We also evaluate these optimizations on the QK^T operator in §D.6 in the appendix.

7.4 Overheads in CoR

Let us now look at the sources of overheads in CoR —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) 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 §5.3 (referred to as Sparse Storage in the table). s compared to this scheme, we see that CoR 's specialized lowering scheme significantly reduces the resources required to compute the data structures associated with tensor storage. 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 (§5.1). 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% (R CE dataset at batch size 128) to about 7% (CoL 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 §D.7 of the appendix.

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

Partial Padding Overheads: We saw that in CoR , 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 §D.7 of the appendix.

Ragged Tensor Overheads and Load Hoisting: CoR '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 MH . While the data and more discussion are provided in §D.7, we note here that the indirect memory accesses do not cause any significant slowdown for the Proj1, Softmax, ttnV 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.

7.5 Evaluation gainst 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 CoR 's performance with Taco, a state-of-the-art sparse tensor compiler on a few triangular matrix operators (implemented in Taco using the CSR and blocked CSR formats). These implementations perform 1 33× to 95 37× slower than the corresponding CoR implementa-

tions. While §D.4 of the appendix provides further details and discussion, we note here that this is essentially due to a mismatch between the use case of ragged tensors and the general sparse tensor computations for which Taco is designed. 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.

8 Rel ted Work

Tensor Compilers: There has been extensive work on tensor compilers such as (i) TVM (Chen et al., 2018a), Halide (Ragan-Kelley et al., 2013), Tiramisu (Baghdadi et al., 2019), Tensor Comprehensions (Vasilache et al., 2018), Fireiron (Hagedorn et al., 2020), Stripe (Zerrell & Bruestle, 2019), KG (Zhao et al., 2021) and work by (Gysi et al., 2021) and (Bhaskaracharya et al., 2020) for dense tensors and (ii) Taco (Kjolstad et al., 2017), COMET (Tian et al., 2021) and work by (Bik et al., 2022) and (Hsu et al., 2021) for sparse tensors. This work has informed CoR '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 §7.5 and further in §D.4, 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.

Past work on DL compilers has also looked at handling dynamism. Nimble (Shen et al., 2020) develops dynamism-aware compiler abstractions from the ground up. Its handling of shape dynamism is limited to variation across mini-batches. CoR is therefore complementary to Nimble's techniques. Cortex (Fegade et al., 2021) handles recursive models by essentially lowering the recursive control flow into sequential control flow on ragged tensors. CoR can therefore potentially be used as part of its pipeline. CoR 's use of uninterpreted functions and named dimensions has been inspired by their use in Cortex and past work on the Sparse Polyhedral Framework (Strout et al., 2018; Mohammadi et al., 2019; Nandy et al., 2018). Named dimensions are also similar to the index labels in COMET. CoR implements a limited form of the hfusion optimization, first proposed in (Li et al., 2020), as part of a tensor compiler.

DL Frameworks and Graph Optimizations: DL frameworks have recently begun adding support for ragged tensors with the RaggedTensor (TensorFlow Team, 2022) class in TensorFlow and the NestedTensor (PyTorch Team, 2022) module for PyTorch. Very few operators are, however, supported for ragged tensors at this point (TensorFlow Community; PyTorch Community).[10] CoR can be used to expand the set of ragged operators supported in these frameworks. CoR 's techniques are complementary to graph optimizations for efficient DL execution such as data layout optimizations (Ivanov et al., 2020), kernel fusion (Zheng et al., 2020b) and operator scheduling (Ding et al., 2020), and can be used in conjunction with them.

Hand Optimized Implementations: There has been work on efficient implementations of certain important ragged tensor operations. This includes the work on variable-sized batched gemm operations (Li et al., 2019; Nath et al., 2010), as well as the work on Effective Transformers and Faster-Transformers, which we compared CoR 's performance against in §7. This past work informs our work on CoR as we saw with the operator splitting transform in §4.1.

Sparse Tensor lgebra: There have been decades of past work on efficient execution of sparse tensor operators. This work has been revisited recently in the context of DL by work on exploiting block sparsity in model weights (Gray et al.) as well as for tuning sparse kernels for the sparsity patterns and distributions usually encountered in DL (Gale et al., 2020). The thread remapping strategy discussed in §4.1 was implemented first in (Gale et al., 2020).

9 Conclusion

This paper presented CoR , a tensor compiler for expressing and optimizing ragged operators to portably target CPUs and GPUs using simple and familiar abstractions. CoR 's approach, specialized for ragged tensors, reduces overheads associated with techniques such as masking and padding. With DL 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. CoR 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 CoR easier to use, potentially with the help of auto-scheduling techniques.

CKNOWLEDGMENTS

This work was supported in part by grants from the National Science Foundation, Oracle and IBM, and by the Parallel Data Lab (PDL) Consortium (mazon, Facebook, Google, Hewlett-Packard Enterprise, Hitachi, IBM, Intel, Microsoft, Net pp, Oracle, Pure Storage, Salesforce, Samsung, Seagate, TwoSigma and Western Digital). We would like to thank Saman marasinghe, Dominic Chen, Stephen Chou, Chris Fallin, Graham Neubig, Olatunji Ruwase, the Catalyst Research Group at Carnegie Mellon University and the anonymous reviewers at MLSys for their valuable suggestions and feedback on our work.

[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 (PyTorch Team) at this point.

REFERENCES

badi, M., Barham, P., Chen, J., Chen, Z., Davis, ., Dean, J., Devin, M., Ghemawat, S., Irving, G., Isard, M., Kudlur, M., Levenberg, J., Monga, R., Moore, S., Murray, D. G., Steiner, B., Tucker, P., Vasudevan, V., Warden, P., Wicke, M., Yu, Y., and Zheng, X. TensorFlow: system for large-scale machine learning. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*, pp. 265–283, Savannah, G , November 2016. USENIX ssociation. ISBN 978-1-931971-33-1. URL https://www.usenix.org/conference/osdi16/technical-sessions/presentation/abadi.

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

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

Bhaskaracharya, S. G., Demouth, J., and Grover, V. utomatic kernel generation for Volta tensor cores. *CoRR*, abs/2006.12645, 2020. URL https://arxiv.org/abs/2006.12645.

Bik, . J. C., Koanantakool, P., Shpeisman, T., Vasilache, N., Zheng, B., and Kjolstad, F. Compiler support for sparse tensor computations in MLIR, 2022.

Briot, J., Hadjeres, G., and Pachet, F. Deep learning techniques for music generation - survey. *CoRR*, abs/1709.01620, 2017. URL http://arxiv.org/abs/1709.01620.

ByteDance. Effective Transformer. URL https://github.com/bytedance/effective_transformer. Last accessed Sept 09, 2021.

Charara, ., Ltaief, H., and Keyes, D. Redesigning triangular dense matrix computations on GPUs. In Dutot, P.-F. and Trystram, D. (eds.), *Euro-Par 2016: Parallel Processing*, pp. 477–489, Cham, 2016. Springer International Publishing. ISBN 978-3-319-43659-3.

Chen, T., Moreau, T., Jiang, Z., Zheng, L., Yan, E., Shen, H., Cowan, M., Wang, L., Hu, Y., Ceze, L., Guestrin, C., and Krishnamurthy, . TVM: n automated end-to-end optimizing compiler for deep learning. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, pp. 578–594, Carlsbad, C , October 2018a. USENIX ssociation. ISBN 978-1-939133-08-3. URL https://www.usenix.org/conference/osdi18/presentation/chen.

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

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

Chou, S., Kjolstad, F., and marasinghe, S. Format abstraction for sparse tensor algebra compilers. *Proc. CM Program. Lang.*, 2(OOPSL), October 2018. doi: 10.1145/3276493. URL https://doi.org/10.1145/3276493.

Conneau, ., Rinott, R., Lample, G., Williams, ., Bowman, S. R., Schwenk, H., and Stoyanov, V. XNLI: Evaluating cross-lingual sentence representations. In *Proceedings of the 2018 Conference on Empirical Methods in Natural Language Processing*. ssociation for Computational Linguistics, 2018.

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

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

Ding, Y., Zhu, L., Jia, Z., Pekhimenko, G., and Han, S. IOS: inter-operator scheduler for CNN acceleration. *CoRR*, abs/2011.01302, 2020. URL https://arxiv.org/abs/2011.01302.

Dolan, W. B. and Brockett, C. utomatically constructing a corpus of sentential paraphrases. In *Proceedings of the Third International Workshop on Paraphrasing (IWP2005)*, 2005. URL https://aclanthology.org/I05-5002.

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

182be0c5cdcd5072bb1864cdee4d3d6e-Paper.pdf.

Gale, T., Zaharia, M., Young, C., and Elsen, E. *Sparse GPU Kernels for Deep Learning*. IEEE Press, 2020. ISBN 9781728199986.

Gray, S., Radford, ., and Kingma, D. P. GPU kernels for block-sparse weights. URL https://cdn.openai.com/blocksparse/blocksparsepaper.pdf.

Gysi, T., Müller, C., Zinenko, O., Herhut, S., Davis, E., Wicky, T., Fuhrer, O., Hoefler, T., and Grosser, T. Domain-specific multi-level IR rewriting for GPU: The open earth compiler for GPU-accelerated climate simulation. *CM Trans. rchit. Code Optim.*, 18(4), September 2021. ISSN 1544-3566. doi: 10.1145/3469030. URL https://doi.org/10.1145/3469030.

Hagedorn, B., Elliott, . S., Barthels, H., Bodik, R., and Grover, V. Fireiron: data-movement-aware scheduling language for GPUs. In *Proceedings of the CM International Conference on Parallel rchitectures and Compilation Techniques*, P CT '20, pp. 71–82, New York, NY, US , 2020. ssociation for Computing Machinery. ISBN 9781450380751. doi: 10.1145/3410463.3414632. URL https://doi.org/10.1145/3410463.3414632.

He, K., Zhang, X., Ren, S., and Sun, J. Deep residual learning for image recognition. In *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pp. 770–778, 2016. doi: 10.1109/CVPR.2016.90.

Hsu, O., Yadav, R., Chou, S., Olukotun, K., marasinghe, S., and Kjolstad, F. Compilation of sparse array programming models. 2021.

Huang, C. ., Vaswani, ., Uszkoreit, J., Shazeer, N., Hawthorne, C., Dai, . M., Hoffman, M. D., and Eck, D. n improved relative self-attention mechanism for transformer with application to music generation. *CoRR*, abs/1809.04281, 2018. URL http://arxiv.org/abs/1809.04281.

HuggingFace. Everything you always wanted to know about padding and truncation, 2020. URL https://huggingface.co/transformers/v3.0.2/preprocessing.html#everything-you-always-wanted-to-know-about-padding-and-truncation. Last accessed Sept 22, 2021.

Intel. Intel one PI Deep Neural Network Library. URL https://software.intel.com/content/www/us/en/develop/tools/oneapi/components/onednn.html#gs.ah65z4. Last accessed September 2, 2021.

Ivanov, ., Dryden, N., Ben-Nun, T., Li, S., and Hoefler, T. Data movement is all you need: case study on optimizing transformers. *CoRR*, abs/2007.00072, 2020. URL https://arxiv.org/abs/2007.00072.

Jain, P., Jain, ., Nrusimha, ., Gholami, ., bbeel, P., Gonzalez, J., Keutzer, K., and Stoica, I. Checkmate: Breaking the memory wall with optimal tensor rematerialization. In Dhillon, I., Papailiopoulos, D., and Sze, V. (eds.), *Proceedings of Machine Learning and Systems*, volume 2, pp. 497–511, 2020. URL https://proceedings.mlsys.org/paper/2020/file/084b6fbb10729ed4da8c3d3f5a3ae7c9-Paper.pdf.

Kirisame, M., Lyubomirsky, S., Haan, ., Brennan, J., He, M., Roesch, J., Chen, T., and Tatlock, Z. Dynamic tensor rematerialization. In *International Conference on Learning Representations*, 2021. URL https://openreview.net/forum?id=Vfs_2RnOD0H.

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

Lai, G., Xie, Q., Liu, H., Yang, Y., and Hovy, E. R CE: Large-scale Re ding comprehension dataset from examinations. In *Proceedings of the 2017 Conference on Empirical Methods in Natural Language Processing*, pp. 785–794, Copenhagen, Denmark, September 2017. ssociation for Computational Linguistics. doi: 10.18653/v1/D17-1082. URL https://aclanthology.org/D17-1082.

Li, ., Zheng, B., Pekhimenko, G., and Long, F. utomatic horizontal fusion for GPU kernels. *CoRR*, abs/2007.01277, 2020. URL https://arxiv.org/abs/2007.01277.

Li, X., Liang, Y., Yan, S., Jia, L., and Li, Y. 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, pp. 229–241, New York, NY, US , 2019. ssociation for Computing Machinery. ISBN 9781450362252. doi: 10.1145/3293883.3295734. URL https://doi.org/10.1145/3293883.3295734.

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

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

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

Nath, R., Tomov, S., and Dongarra, J. n improved magma gemm for Fermi graphics processing units. *The International Journal of High Performance Computing pplications*, 24(4):511–515, 2010. doi: 10.1177/1094342010385729. URL https://doi.org/10.1177/1094342010385729.

Nvidia. FasterTransformer. URL https://github.com/NVIDI /FasterTransformer. Last accessed Sept 09, 2021.

OpenBL S Community. OpenBL S: n optimized BL S library. URL https://www.openblas.net/. Last accessed Oct 08, 2021.

Paszke, ., Gross, S., Massa, F., Lerer, ., Bradbury, J., Chanan, G., Killeen, T., Lin, Z., Gimelshein, N., ntiga, L., Desmaison, ., Kopf, ., Yang, E., DeVito, Z., Raison, M., Tejani, ., Chilamkurthy, S., Steiner, B., Fang, L., Bai, J., and Chintala, S. PyTorch: n imperative style, high-performance deep learning library. In Wallach, H., Larochelle, H., Beygelzimer, ., d' lché-Buc, F., Fox, E., and Garnett, R. (eds.), *dvances in Neural Information Processing Systems 32*, pp. 8024–8035. Curran ssociates, Inc., 2019. URL http://papers.neurips.cc/paper/9015-pytorch-an-imperative-style-high-performance-deep-learning-library.pdf.

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

PyTorch Team. The nestedtensor package prototype: Readme.md. URL https://github.com/pytorch/nestedtensor/blob/master/nestedtensor/csrc/RE DME.md. Last accessed Oct 15, 2021.

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

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

Ragan-Kelley, J., Barnes, C., dams, ., Paris, S., Durand, F., and marasinghe, S. Halide: language and compiler for optimizing parallelism, locality, and recomputation in image processing pipelines. In *Proceedings of the 34th CM SIGPL N Conference on Programming Language Design and Implementation*, PLDI '13, pp. 519–530, New York, NY, US , 2013. ssociation for Computing Machinery. ISBN 9781450320146. doi: 10.1145/2491956.2462176. URL https://doi.org/10.1145/2491956.2462176.

Rajpurkar, P., Jia, R., and Liang, P. Know what you don't know: Unanswerable questions for SQu D. *CoRR*, abs/1806.03822, 2018. URL http://arxiv.org/abs/1806.03822.

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

Singh, S., Steiner, B., Hegarty, J., and Leather, H. Using graph neural networks to model the performance of deep neural networks, 2021.

Smith, S. and Karypis, G. Tensor-matrix products with a compressed sparse tensor. In *Proceedings of the 5th Workshop on Irregular pplications: rchitectures and lgorithms*, I 3'15, New York, NY, US , 2015. ssociation for Computing Machinery. ISBN 9781450340014. doi: 10.1145/2833179.2833183. URL https://doi.org/10.1145/2833179.2833183.

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

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

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

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

Tian, R., Guo, L., Li, J., Ren, B., and Kestor, G. high-performance sparse tensor algebra compiler in multi-level IR. *CoRR*, abs/2102.05187, 2021. URL https://arxiv.org/abs/2102.05187.

van den Oord, ., Dieleman, S., Zen, H., Simonyan, K., Vinyals, O., Graves, ., Kalchbrenner, N., Senior, . W., and Kavukcuoglu, K. Wavenet: generative model for raw audio. *CoRR*, abs/1609.03499, 2016. URL http://arxiv.org/abs/1609.03499.

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

Vaswani, ., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, . N., Kaiser, L., and Polosukhin, I. ttention is all you need. *CoRR*, abs/1706.03762, 2017. URL http://arxiv.org/abs/1706.03762.

Warstadt, ., Singh, ., and Bowman, S. R. Neural network acceptability judgments. *Transactions of the ssociation for Computational Linguistics*, 7:625–641, March 2019. doi: 10.1162/tacl_a_00290. URL https://aclanthology.org/Q19-1040.

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

Williams, ., Nangia, N., and Bowman, S. broad-coverage challenge corpus for sentence understanding through inference. In *Proceedings of the 2018 Conference of the North merican Chapter of the ssociation for Computational Linguistics: Human Language Technologies, Volume 1 (Long Papers)*, pp. 1112–1122, New Orleans, Louisiana, June 2018. ssociation for Computational Linguistics. doi: 10.18653/v1/N18-1101. URL https://aclanthology.org/N18-1101.

Yang, Z., Dai, Z., Yang, Y., Carbonell, J. G., Salakhutdinov, R., and Le, Q. V. XLNet: Generalized autoregressive pretraining for language understanding. *CoRR*, abs/1906.08237, 2019. URL http://arxiv.org/abs/1906.08237.

Zerrell, T. and Bruestle, J. Stripe: Tensor compilation via the nested polyhedral model. *CoRR*, abs/1903.06498, 2019. URL http://arxiv.org/abs/1903.06498.

Zhao, J., Li, B., Nie, W., Geng, Z., Zhang, R., Gao, X., Cheng, B., Wu, C., Cheng, Y., Li, Z., Di, P., Zhang, K., and Jin, X. KG: utomatic kernel generation for neural processing units using polyhedral transformations. In *Proceedings of the 42nd CM SIGPL N International Conference on Programming Language Design and Implementation*, PLDI 2021, pp. 1233–1248, New York, NY, US , 2021. ssociation for Computing Machinery. ISBN 9781450383912. doi: 10.1145/3453483.3454106. URL https://doi.org/10.1145/3453483.3454106.

Zheng, L., Jia, C., Sun, M., Wu, Z., Yu, C. H., Hajli, ., Wang, Y., Yang, J., Zhuo, D., Sen, K., Gonzalez, J. E., and Stoica, I. nsor: Generating high-performance tensor programs for deep learning. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pp. 863–879. USENIX sociation, November 2020a. ISBN 978-1-939133-19-9. URL https://www.usenix.org/conference/osdi20/presentation/zheng.

Zheng, Z., Zhao, P., Long, G., Zhu, F., Zhu, K., Zhao, W., Diao, L., Yang, J., and Lin, W. FusionStitching: Boosting memory intensive computations for deep learning workloads. *CoRR*, abs/2009.10924, 2020b. URL https://arxiv.org/abs/2009.10924.

We now look at additional details regarding CoR 's mechanism in § , §B and §C, and discuss further aspects of the evaluation in §D. Notably, we look at how CoR can exploit masking in masked MH to obtain further savings in §D.3, discuss how CoR 's overheads are quite low, allowing it to effectively exploit raggedness (§D.7) and look more closely at CoR 's performance on the transformer model and where the benefits come from in §D.8.

R GGED PI

.1 Thread Remapping Policy

We discussed, in §4, that CoR 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 Fig 15.

B R GGED PI LOWERING

B.1 Tensor Storage Lowering

In §5.3, we briefly discussed the storage lowering schemes used by past work on sparse tensor compilers and by CoR . Both are illustrated in Fig. 16 and discussed more below.

Sparse Storage ccess Lowering Scheme Used in Past Work: Recall the 4-dimensional attention tensor X we discussed in §5.3 and which is illustrated again in Fig. 16. 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.

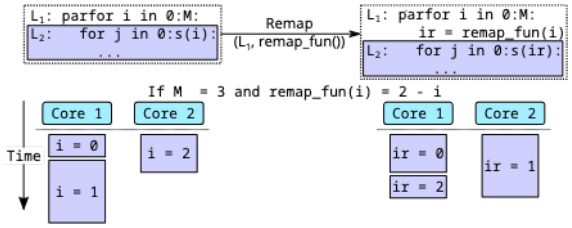


Figure 15: Thread remapping allows users to influence the scheduling of iterations to allow for better load balancing.

The sparse tensor compiler Taco (Kjolstad et al., 2017), the performance of which look at in §D.4, uses a tree-based modular scheme (first proposed in the work (Smith & Karypis, 2015) on the Compressed Sparse Fiber tensor format) to model sparse tensor storage. In this scheme, illustrated in Fig. 16 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. s 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 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 Fig. 16, 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 $s_3 \sum_{i=0}^{s} s_{24}\ i)$. Therefore, the amount of auxiliary data computed and stored would be equal to $s_1 + s_3 \sum_{i=0}^{s} s_{24}\ i)$, which as we saw in §7.4 can be much larger than CoR 's specialized scheme.

lgorithm 1 Procedure to lower ragged tensor accesses

procedure LOWER CCESS($[b\ , ..., b_n]$)
  $o\ \ set \leftarrow 0$
  $relaxed \leftarrow [b\ , ..., b_n]$
  for $i \leftarrow n$ to 1 **do** Compute $D_i\ \vec{B}_i$)
    $D \leftarrow 1$
    if $O\ \ i) \neq \emptyset$ **then**
      $D \leftarrow A_i\ \ relaxed[j])$
    else
      $D \leftarrow relaxed[i]$
    end if
    for j in $S\ i)\ \ \{i\}$ **do**
      if $O\ \ j) \neq \emptyset$ **then**
        $D \leftarrow D * A_j\ \ relaxed[j])$
      else
        $D \leftarrow D * s_j\ \ relaxed[I\ \ j)])$
      end if
    end for
    $relaxed[i] \leftarrow s_i\ \ relaxed[I\ \ i)])$
    $o\ \ set \leftarrow o\ \ set + D$
  end for
  return $o\ \ set$
end procedure

CoR 's Storage ccess Lowering Scheme: We saw that CoR '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 CoR 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 $O\ \ _T\ b_1,\ \ , b_n)$ 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 (Fig. 16). The dgraph G of the n-dimensional tensor

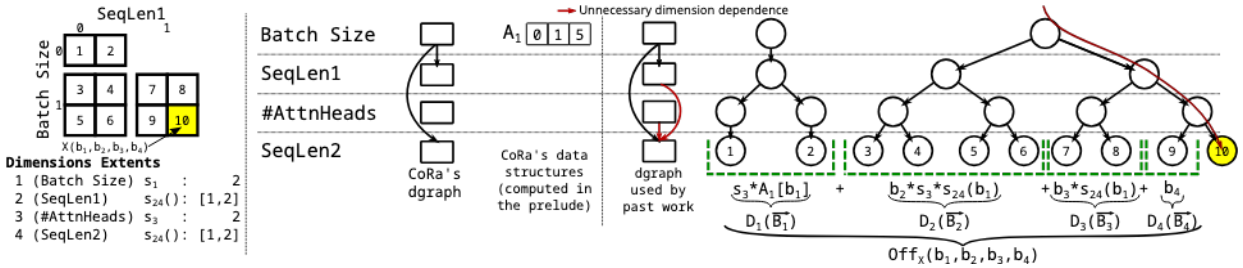


Figure 16: Comparing CoR 's storage lowering with the tree-based scheme used by past work on sparse tensors.

T is a pair $D, E)$ where D is the set of all dimensions $\{1, \ , n\}$ and E is a set of directed edges. n 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 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 $O_G\ d)$. lso, let $O_G^{ex}\ d) = O_G\ d) \ \bigcup_{i \in O\ \ d)} O_G^*\ i)$.

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

$$O\ \ _T\ B_{\leq n}^{\rightarrow}) = \sum_{i=1}^{n} O\ \ _T\ B_{\leq i}^{\rightarrow}) \ \ O\ \ _T\ B^{\rightarrow}\ i))$$
$$= \sum_{i=1}^{n} D_i\ B_{\leq i}^{\rightarrow})$$

During compilation, the procedure in lgorithm 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 $D_i\ B_{\leq i}^{\rightarrow})$. For a dimension i, $D_i\ B_{\leq i}^{\rightarrow})$, is further computed (in the inner loop on line 11) as a product of contributions corresponding each of the inner dimensions j such that $j \geq i$ (Fig. 16 shows the values of D_is 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, $D_i\ B_{\leq i}^{\rightarrow}) = b_i \prod_{j=i+1}^{n} s_j$. 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 §2. The function $A_d\ )$ for the batch dimension (the first dimension) of our example tensor X in Fig. 8 is shown as the array $_1$ where $_1[i] = \sum_{j=1}^{i} s_{24}\ j) * s_{24}\ j)$.

s 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.

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

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

B.2 Variable Loop Fusion

In §5.1, 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, §5.2 describes how the functions f_{fo}, f_{fi} and f_{oif} represent the relationships between the iteration variables `o`, `i` and `f` in Fig. 6. In the generated code, as we can see in Fig. 4, these functions take the form of arrays that are initialized by the prelude. During compilation, 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 (De Moura & Bjørner, 2008). In order to

enable Z3 process these uninterpreted functions, we provide it with the following relationships between these functions:

$$\forall f, f_{oif}\ f_{fo}\ f), f_{fi}\ f)) = f$$
$$\forall o, i, f_{fo}\ f_{oif}\ o, i)) = o$$
$$\forall o, i, f_{fi}\ f_{oif}\ o, i)) = i$$

C ddition l Implement tion Det ils

s we mentioned in §6, we have prototyped CoR 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 CoR . 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. CoR '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.

D Supplement ry Ev lu tion nd ddition l Det ils

D.1 Datasets

We use the datasets listed in Table 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 (Vaswani et al., 2017; Devlin et al., 2018; Yang et al., 2019). The Wiki512 and Wiki128 datasets, usually used for pre-training (Devlin et al., 2018), are generated from a dump of the English Wikipedia website (Wikipedia). 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. s a result, these datasets do not provide as much opportunity for CoR to exploit as do some of the other datasets. We saw this reflected in Fig. 2 as well as in the evaluation in §7.

D.2 Load Balancing

We briefly discussed the challenge of ensuring a balanced workload across multiple execution units in the main text. 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 ttnV operator in the SDP module. We handle this imbalance using either thread remapping (§4 and § .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.

D.3 Masked Scaled Dot-Product ttention

s we briefly mentioned in §7.2, the decoder layer of a transformer uses a variant of MH called masked MH 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 CoR can exploit this masking and further save on wasted computation in the SDP sub-module, which is the only portion affected by the masking.

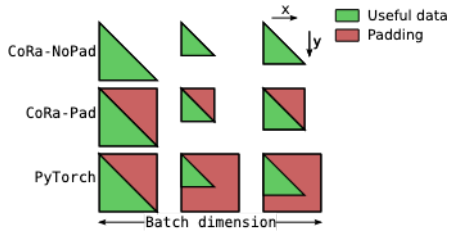


Figure 17: The attention matrices of the masked MH module as implemented in the implementations discussed in §7.2 and compared in Fig. 18. In the figure, for simplicity, the number of attention heads is assumed to be 1, partial padding is not shown and the batch size is assumed to be 3. The x and y directions denote increasing matrix indices.

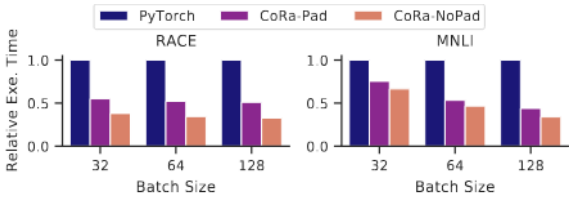


Figure 18: Execution time of masked SDP in PyTorch and CoR , with and without padding for the attention matrix.

We also mentioned in the main text that with masking, the

SDP 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. Fig. 18 shows the performance of three implementations of masked SDP —CoR -NoPad, where both the vloops are only partially padded, CoR -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 Fig. 17. s Fig. 18 shows, CoR -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 CoR -Pad and PyTorch respectively across the datasets and batch sizes evaluated in Fig. 18. s we saw, the performance of MNLI dataset improves to a smaller degree due to the padding employed in CoR -NoPad.

D.4 Evaluation gainst Sparse Tensor Compilers

We saw in the main text of the paper 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 §7.1, 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 6 lists the execution times (in ms) for the aforementioned operations and formats. s the table shows, CoR performs better than Taco for all the cases evaluated. We discuss the reasons for this below.

Storage Layouts: 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.

Table 6: Execution times (in ms) for the trmm, tradd and trmul operations implemented in Taco using the CSR and the BCSR matrix formats and in CoR . The table also shows Taco's slowdowns with respect to CoR .

Op	Matrix Dim.	CoR	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

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 §2. 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 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 CoR '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.

D.5 Memory Consumption

We mentioned in §7.2 that the use of ragged tensors leads to a significant drop in the memory required to store the forward activations of the encoder layer. Fig. 19 shows this for the datasets in Table 3 for batch size 64. It plots the relative total memory consumption (computed analytically) of the forward activations of a transformer encoder layer for CoR '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 CoR to exploit.

D.6 Operation Splitting and Horizontal Fusion

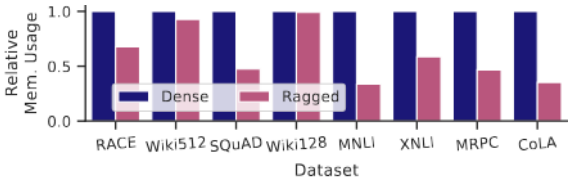


Figure 19: Relative sizes of the forward activations of a transformer encoder layer with and without ragged tensors.

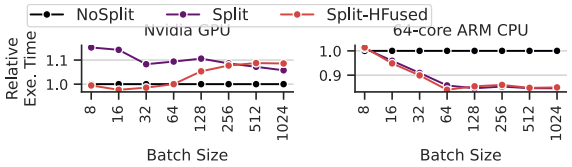


Figure 20: Operation splitting and hfusion for QK^T.

In §7.3 of the main text, we looked at the benefits of operation splitting and hfusion on the ttnV 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 Fig. 20. The figure shows the normalized execution times, for the QK^T operator, of the three implementations described in §7.3. We see that on the CPU backend, similar to the ttnV 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 CUD 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 Fig. 21. In that figure, we compare the performance of three CoR 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 RM 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).

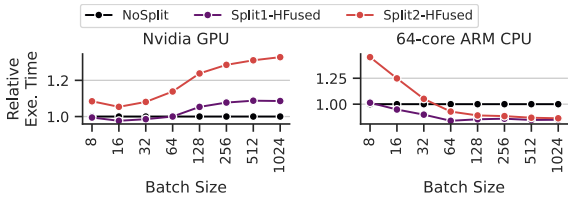


Figure 21: Efficacy of operation splitting and hfusion when applied to one or both vloops of the QK^T operator.

D.7 CoR Overheads

Prelude Overheads: s we discussed in §C, CoR 's prototype allows us to generate code for operator kernels one at a time. For each kernel, CoR 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 ttnV operators in CoR '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 7 and 8 compare, for a 6-layer transformer encoder, the execution time and memory consumption of the prelude code respectively, as present in CoR 's current implementation (referred to as CoR -Redundant in the table) with an optimized implementation (referred to as CoR -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 Fig. 22, we show the relative amount of computation (computed analytically as in Fig. 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 §7 with partial padding, and the ideal case with no padding. We see that partial padding leads to a very small increase in the amount of computation (3.5% across datasets for batch

Table 7: Prelude execution times (in ms) for a 6-layer transformer encoder with and without redundant computation.

Dataset	Batch Size	CoR -Redundant			CoR -Optimized		
		CoR Storage	CoR Loop Fusion	CoR -Copy Time	CoR Storage	CoR Loop Fusion	CoR -Copy Time
CoL	32	0.004	0.006	0.232	0.002	0.002	0.088
CoL	128	0.006	0.015	0.261	0.003	0.004	0.094
R CE	32	0.005	0.085	0.419	0.002	0.015	0.121
R CE	128	0.007	0.339	0.985	0.003	0.053	0.209

Table 8: Prelude memory usage (in kB) for a 6-layer transformer encoder with and without redundant computation.

Dataset	Batch Size	CoR -Redundant		CoR -Optimized	
		CoR Storage	CoR Loop Fusion	CoR Storage	CoR Loop Fusion
CoL	32	2.93	32.15	1.2	5.27
CoL	128	11.18	104.22	4.58	17.5
R CE	32	2.93	666.54	1.2	106.87
R CE	128	11.18	2609.58	4.58	418.06

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.

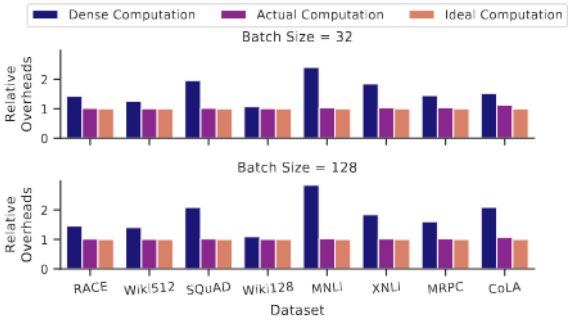


Figure 22: Overheads due to partial padding.

Ragged Tensor Overheads and Load Hoisting: We now take a closer look at the effects of auxiliary data structure accesses on the performance of CoR -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 MH 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 Fig. 23. part 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.

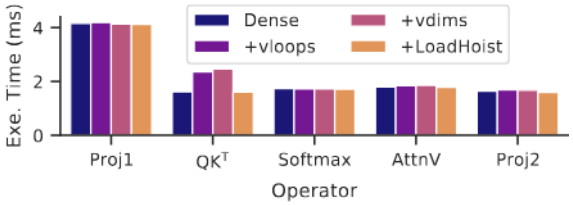


Figure 23: Overheads of using ragged computations and ragged tensor storage, and the benefits of load hoisting, measured for a synthetic dataset where all sequence lengths are 512. The batch size used is 64.

s the figure shows, the use of vloops and vdims leads to a slight slowdown for the Proj1, Softmax, ttnv and Proj2 operators. The slowdown is significant, however, for the QK^T operator, which has two vloops in its loop nest. s 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 CUD compiler is unable to effectively hoist these accesses in this case. CoR however has more knowledge about these accesses and can hoist them to recover the lost performance.

D.8 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 RM CPU backends. We break down the execution time of the encoder layer for a few cases. s in Fig. 13, these per-operator execution times are obtained under profiling and might deviate slightly from the data in Tables 4 and 5.

Nvidia GPU Backend: Table 10 provides the raw data for the breakdown of the execution times for the R CE dataset at batch size 128 of the transformer encoder layer shown in Fig. 13 in the main text. part from improvements in the QK^T and ttnV operators discussed in §7.2, we note

that CoR '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 CoR '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 CoR 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).

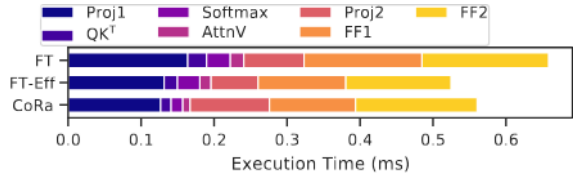
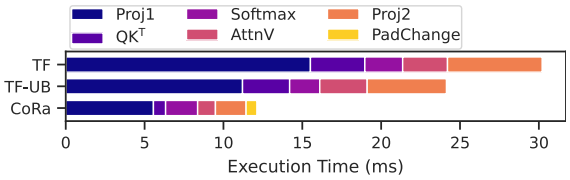


Figure 24: Breakdown of execution times of the encoder layer for the CoL dataset at batch size 32 on the GPU.

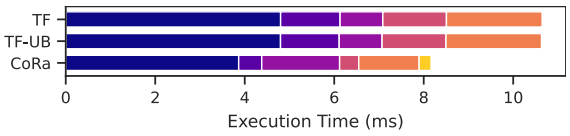
We now look at the execution time breakdown for the CoL dataset at batch size 32 on the Nvidia GPU shown in Fig. 24. We see that CoR performs slightly worse than FT-Eff for this case. Most of CoR 's slowdown stems from worse performance on the linear transformation operators Proj2, FF1 and FF2. CoR performs slightly better than FT-Eff for the Proj1 operator, which is also a linear transformation operator. From this data, we conclude that CoR 's schedules for the Proj2, FF1 and FF2 operators can be improved to close this performance gap. We note that, even in this case, CoR performs much better on the SDP module (the QK^T, Softmax and ttnV operators) as compared to FasterTransformer.

RM CPU Backends: In §7.2, we saw how CoR performs better than TensorFlow for the MH module on the 8- and 64-core RM CPUs. In this section, we discuss these implementations in more detail and provide more extensive evaluation.

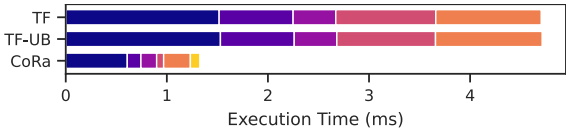
Micro-Batching for PyTorch and TensorFlow: We saw, in Fig 2, that the amount of padding and wasted computation increases with the batch size. On devices that expose low levels of parallelism such as CPUs, it is therefore possible to trade-off batch parallelism for reduced padding, and therefore reduced wasted computation, for frameworks such as



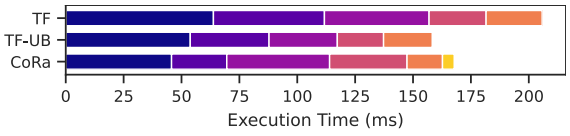
(a) MNLI dataset at batch size 128.



(b) Wiki128 dataset at batch size 32.



(c) CoL dataset at batch size 32.



(d) R CE dataset at batch size 128.

Figure 25: Breakdown of execution times of the MH module for four cases on the 64-core RM CPU backend.

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 Fig. 26. We search over micro-batch sizes that are powers of 2 starting from the lowest micro-batch size of 2. In Table 9, we provide the execution latencies as well as the optimal micro-batch sizes for PyTorch and TensorFlow (these configurations is referred to as PT-UB and TF-UB respectively) for an 8-core as well as a 64-core RM 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.

CoR 's MH Implementation: s in CoR 's vgemm implementation on the Intel CPU backend, we offload the computation of the dense inner tiles of the Proj1 and Proj2 operators in CoR 's MH implementation on the RM backends to gemm calls in the OpenBL S (OpenBL S Community) 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

Table 9: MH execution latencies (in ms) on 8- and 64-core RM CPUs. uBS stands for the optimal micro-batch size.

Dataset	Batch Size	8-core RM CPU					64-core RM CPU				
		PT	PT-UB / uBS	TF	TF-UB / uBS	CoR	PT	PT-UB / uBS	TF	TF-UB / uBS	CoR
R CE	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
SQu D	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**
CoL	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**

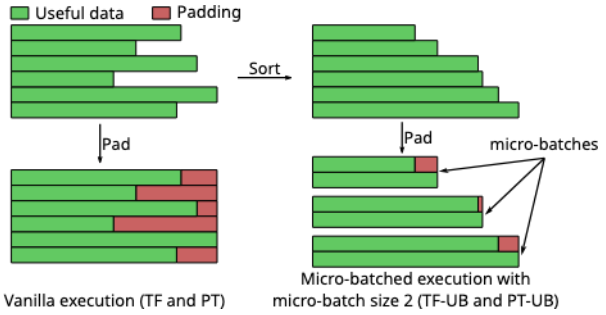


Figure 26: Comparison of vanilla and micro-batched execution for PyTorch and TensorFlow.

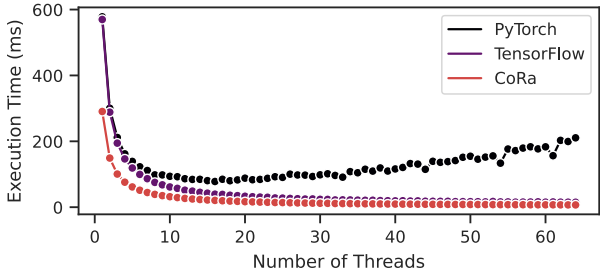


Figure 27: Execution latencies of PT, TF and CoR as the number of threads is increased for the MNLI dataset at a batch size of 64. These measurements were performed on the 64-core CPU by changing the number of threads launched by OpenMP. Due to this, the measurements may not exactly be equal to the ones in Table 9.

see in Fig. 25, however, that these pad fusion operators are relatively cheap to perform on the CPU backend.

Overall Performance Comparison: Table 9 shows the inference latencies for the PyTorch, TensorFlow and CoR implementations of the MH module on the 8- and 64-core RM 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 9. 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 CoR perform similarly on the 8-core RM CPU, while CoR outperforms TF-UB by about 1.37× as the hardware parallelism increases on the 64-core CPU. In both the cases, CoR 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 CoR 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, CoR 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 Fig. 27) as compared to Ten-

sorFlow and CoR . 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 CoR implementations. Fig. 25 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 Fig. 2 shows, and (2) the R CE dataset at a batch size of 128 and the CoL 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 CoL 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 R CE dataset at batch size 128, which represents the best case for TF-UB, TF-UB performs slightly better than CoR . In cases where CoR performs better than TensorFlow, we find that a lot of the reduction in CoR '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 ttnV 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 CoR for the R CE and Wiki128 datasets. We believe this is due to better optimized implementations and that this gap can be reduced with more time spent optimizing CoR 's implementation of the operator.

Table 10: Breakdown of the encoder layer execution time for FasterTransformer and CoR on the Nvidia GPU backend for the R CE dataset at batch size 128. Per-layer prelude code overheads are included in these latencies for CoR . Both FasterTransformer and CoR implementations normally execute CUD 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	CoR	CoR Ops
Proj1	QKV Proj. MM QKV Bias + ddPad	7.16 1.39	5.4 1.21	6.2	QKV Proj.
QK^T	QK^T	2.65	2.64	2.12	ddPad + QK^T
Softmax	Softmax	4.08	4.08	1.93	ChangePad + Softmax + ChangePad
ttnV	ttnV	2.78	2.79	2.44	ttnV
Proj2	Transpose + RemovePad Linear Proj. MM Linear Proj. Bias + Residual dd + LayerNorm	0.78 2.42 0.52	0.29 1.82 0.38	 2.31 0.31	 RemovePad + Linear Proj. MM + Bias + Residual dd LayerNorm
FF1	FF1 MM FF1 Bias + ctivation	9.52 1.38	6.92 0.98	8.06	FF1 MM + Bias + ctivation
FF2	FF2 MM FF2 Bias + Residual dd + LayerNorm	9.47 0.53	7.1 0.38	8.33 0.31	FF2 MM + Bias + Residual dd LayerNorm
	Total Execution Time	42.82	34.12	31.99	Total Execution Time

RTIF CT PPENDIX

.1 bstract

This appendix describes how to reproduce the results described above in §7 and §D. The experiments in the paper evaluate CoR on an Nvidia V100 GPU, an 8-core, 16-thread Intel CascadeLake CPU, an 8-core RM Neoverse N1 CPU and a 64-core RM Neoverse N1 CPU. Below, we provide instructions to set up and execute CoR as well as other frameworks used in the evaluation on each of these platforms. s we start with publicly available Docker containers in all cases, only a few GPU-related dependencies (such as cuDNN) as well as CoR 's dependencies (such as the Z3 SMT solver, LLVM and OpenBL S) need to be installed. We provide instructions for each of these. In each case, 100 GB of disk space should be more than enough for the evaluation.

.2 rtifact check-list (meta-information)

- **Compilation:** We reply on the publicly available compilers `nvcc` (the CUD compiler to compile CUD code generated by CoR), `gcc` (to build CoR and other dependencies) and LLVM (to facilitate CoR 's code generation for CPUs).
- **Data set:** We use sequence lengths for 8 different commonly used NLP datasets, all of which are included in the repositories described below.
- **Run-time environment:** The artifact has been tested on Ubuntu 20.04 and with the following versions of different dependencies.
- **Hardware:** We use an Nvidia V100 GPU, an 8-core, 16-thread Intel CascadeLake CPU, an 8-core RM Neoverse N1 CPU and a 64-core RM Neoverse N1 CPU for our evaluation.
- **Metrics:** We use execution time as the primary execution metric of evaluation.
- **Output:** We generate CSV files, and also provide Python scripts to generate plots from this raw data.
- **Experiments:** Python/bash scripts are provided to replicate the results.
- **Disk space required:** 100 GB on each backend.
- **Time needed to prepare workflow:** few hours on each backend.
- **Time needed to complete experiments:** Running all experiments on a backend takes several hours. The experiments on the RM CPUs are particularly slow, taking 2-3 days to complete.
- **Public availability:** Yes, in the form of GitHub repositories (linked later) as well as on Zenodo (`https://doi.org/10.5281/zenodo.6326455`).

.3 Description

.3.1 *How delivered*

Source code in the form of Github repositories and archived on Zenodo.

.3.2 *Hardware dependencies*

We use an Nvidia V100 GPU, an 8-core, 16-thread Intel CascadeLake CPU, an 8-core RM Neoverse N1 CPU and a 64-core RM Neoverse N1 CPU for our evaluation.

.3.3 *Software dependencies*

GPU: Below, we describe the environment we use across the different hardware backends we evaluate CoR on. On all of the backends, we use Ubuntu 20.04. Some of the frameworks below are already installed as part of the Docker images we start with (described below), while some need to be manually or installed. This is described below in § .4.

Dependencies Common across Backends: CoR requires the following frameworks on all platforms: Z3 4.8.8, LLVM 9.0.0, cmake $\geq$ 3.5 and g++ $\geq$ 5.0.

Nvidia GPU: CUD 11.1 (V11.1.105), cuDNN 8.2.1, PyTorch 1.9.0+cu111, FasterTransformer (modified on top of FasterTransformer v4.0 (commit `dd4c071`) and provided as part of the `cora_benchmarks` repository). Make sure that `nvcc` is on the `P TH`.

RM CPUs: OpenBL S 0.3.10, PyTorch 1.10.0 with RM Compute Library 21.12, TensorFlow 2.6.0 with RM Compute Library 21.09.

Intel CPU: Intel one PI MKL v2021.3.

.3.4 *Data sets*

ll datasets are included in the `cora_benchmarks` repository.

.4 Installation

Refer to the file `ae_appendix_supplement.pdf` at the root of the `cora` repository.

.5 Experiment workflow

s we saw above, CoR is a tensor compiler. It takes as input a description of a tensor operator and generates LLVM or CUD code for the kernel depending on whether the target is a CPU or a GPU. For evaluating individual kernels, such as trmm, vgemm or any of the nine kernels that make up the transformer layer (illustrated in Figure 3 of the paper), merely executing the python script implementing the kernel will compile the kernel, generate the code and execute it. For evaluating the entire transformer layer, or parts of it (such the self-attention or the MH modules), we

separate the steps of code generation and execution (§C). We first generate compiled code for each of the operators in the form of shared libraries, which are then loaded and executed to form the layer to benchmark layer performance. The scripts provided automate all of these steps.

.6 Evaluation Notes and Instructions

Refer to the file `ae_appendix_supplement.pdf` at the root of the `cora` repository for instructions on how to perform the evaluation on the platforms referred to in the paper.

.7 Experiment customization

1. **GPU Evaluation:** The GPU evaluation has been tested on Nvidia GPUs with compute capability 70. While the evaluation may run on other Nvidia GPUs as well, we have not yet tested it thoroughly. CoR can currently only support Nvidia GPUs as it only supports the generation of CUD code.

2. **CPU Evaluation:** The Intel and the RM CPU evaluation should work for other CPUs beyond the CascadeLake and the Neoverse N1 CPUs we have tested it on. However, given that our schedules have been tuned for these CPUs, the performance might not be optimal. Further, when running on other CPUs, the LLVM target triple would need to be changed (in the file `cora_benchmarks/scripts/common.py` on line 10). The number of threads would also be need to be changed for PyTorch evaluation in the files `cora_benchmarks/bert_layer/pytorch/layer_cpu_micro_batch.py` and `cora_benchmarks/bert_layer/pytorch/layer_cpu.py` on the appropriate branch of the `cora_benchmarks` repository.