PyCoRa: A PyTorch Implementation of Ragged Tensors

Dev Singh
dsingh14@illinois.edu
University of Illinois Urbana-Champaign
Urbana, Illinois, USA

Yanni Zhuang
yanniz3@illinois.edu
University of Illinois Urbana-Champaign
Urbana, Illinois, USA

1 Introduction

In modern deep learning applications, there is an increasing reliance on data with significant variations in shape and size. This phenomenon is prevalent in domains such as natural language processing (NLP), where sentences vary in length, and audio processing, where clips vary in duration. These types of data are best represented as "ragged tensors," where the dimensions can vary dynamically across a batch.

However, current deep learning frameworks and hardware accelerators, such as GPUs, are optimized primarily for dense, uniform tensor algebra. To work around this limitation, ragged tensors are typically handled via padding and masking to force irregular data length into uniform, rectangular shapes. This approach has two critical disadvantages. Firstly, padding necessitates processing a large number of zero elements that are eventually discarded, incurring significant performance overhead through unnecessary computation. As batch sizes increase, the probability of encountering a long "outlier" sequence rises, forcing every other sequence in the batch to be padded to this maximum length. Consequently, standard padding strategies become increasingly inefficient at the large batch sizes required for high-throughput training. Secondly, storing these padding zeros incurs substantial GPU memory overhead, which can limit the maximum batch size or model depth.

An alternative to padding is the use of hand-optimized kernels to deal with the irregular shape of ragged data. While these implementations succeed in minimizing padding and offer higher performance, the manual development work and lack of portability across different hardware architectures makes this approach not suitable for real-world deployment.

This project aims to implement a flexible, compiler-based solution for ragged tensors that interacts natively with their irregular shapes to achieve high performance, while maintaining the ease of use of high-level frameworks. By leveraging PyTorch 2.0's TorchDynamo and TorchInductor [1] compiler frameworks, along with GPU kernels written in Triton [10], we aim to replicate the minimal-padding benefits of the CoRa compiler [4] as proposed by Fegade et al. without relying on the more complex TVM [2] stack. This approach enables efficient, portable, and automated code generation for ragged operators, effectively addressing the limitations of both full padding and rigid hand-optimized kernels.

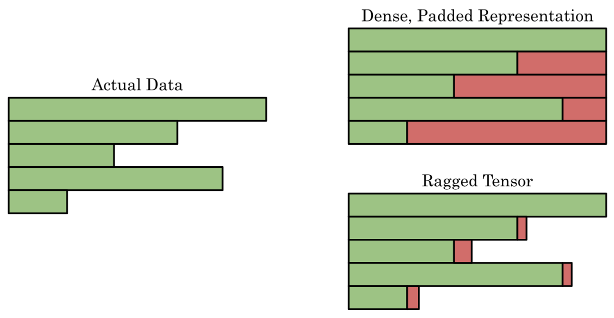

Figure 1. An example of PyCoRa ragged tensors vs traditional dense tensors.

2 Existing Work

There are a number of pre-existing approaches to this challenge of processing and operating upon variable-length data.

Hand-Optimized Kernels: High-performance libraries have been developed to handle specific ragged tensor applications. For example, specialized kernels exist for Transformer models, such as NVIDIA's FasterTransformer [7], which builds on the original transformer architecture proposed by Vaswani et al. [11]. These implementations often remove padding to process sequences as a continuous stream and use specialized algorithms to fuse operators. However, these solutions are generally monolithic and difficult to extend to new, generalized operators.

Dense Tensor Compilers Compilers such as TVM and Halide [8] generate efficient code for dense tensors by optimizing loops and memory accesses. However, these same patterns apply poorly to ragged tensors because they assume uniform loop bounds and rectangular data shapes. The dependencies between the data and loop bounds are inherently inexpressible, and thus these approaches must revert to full padding.

Sparse Tensor Compilers Sparse tensor compilers such as TACO [5] and COMET [6] optimize operations on tensors with many zeros. While ragged tensors resemble sparse tensors in that padded regions contain zeros, standard sparse formats like compressed sparse row matrices (CSR) are inefficient for ragged deep learning workloads. Sparse formats like CSR compress data by storing only non-zero values along with their coordinates. To access a specific element at position (i, j), these formats must first locate row i (in $O(1)$ using row pointers), then search through the column indices of non-zeros in that row—either via binary search (in $O(\log k)$ time, where k is the number of non-zeros in row i) or linear scan ($O(k)$). Consequently, general sparse compilers often introduce unnecessary indexing and searching costs when applied to ragged data, despite both involving non-rectangular shapes.

The CoRa Tensor Compiler The most relevant work is the CoRa tensor compiler as presented by Fegade et al, which introduced a compiler specifically for ragged tensors. CoRa allows users to generate efficient code targeting various hardware with minimal padding. It addresses the limitations of sparse compilers by using a specialized storage lowering scheme that precomputes memory offsets for $O(1)$ access. CoRa also introduces scheduling primitives like "operation splitting" and "horizontal fusion" to handle load imbalances caused by variable loop lengths. This project builds upon the algorithmic insights of CoRa but implements them within the modern PyTorch/Triton ecosystem, in contrast to the original TVM implementation.

3 Algorithm Design and Architecture

3.1 RaggedTensor Data Structure

The core of our implementation is the `RaggedTensor` class, which represents a batch of variable-length sequences with minimal padding. Unlike traditional batched tensors that use a dense $(B \times L_{\max} \times D)$ representation where B is the batch size, $L_{\max}$ is the maximum sequence length, and D is the feature dimension, our ragged representation stores only the actual data in a flattened tensor.

A `RaggedTensor` consists of four key components:

- **data**: A 2D tensor of shape (T_{total}, D) containing the concatenated sequence data, where $T_{\text{total}} = \sum_{i=1}^{B} \lceil L_i/A \rceil \cdot A$ is the total number of tokens after alignment padding, and A is the alignment factor (64 in our implementation and in Fegade et al.).
- **offsets**: A 1D tensor of length $B+1$ where `offsets[`i`]` indicates the starting position of sequence i in the flattened data tensor. This enables $O(1)$ random access to any sequence.
- **lengths**: A 1D tensor of length B containing the actual (unpadded) length of each sequence.
- **padded_lengths**: A 1D tensor of length B storing the aligned length $\lceil L_i/A \rceil \cdot A$ for each sequence, which is critical for cache alignment.

The alignment to 64-element boundaries ensures that each sequence begins at a cacheline-aligned address, improving memory access patterns and enabling vectorized operations. This design reduces wasted computation on padding tokens while maintaining efficient GPU memory access.

3.2 CoRa Auxiliary Metadata

Following the CoRa framework, we precompute auxiliary data structures that enable $O(1)$ access to the flattened value array, allowing efficient execution of operations on ragged tensors. These metadata structures map the standard 2D grid of thread blocks onto the ragged structure:

- **grid_mapping**: A 1D tensor that maps each global thread block ID to its corresponding batch index. For a batch with sequences of padded lengths $\{P_1, P_2, \ldots, P_B\}$ and block size M, sequence i requires $\lceil P_i/M \rceil$ thread blocks. The grid mapping concatenates these batch indices, creating a mapping from linearized block IDs to batch indices.
- **block_offsets**: A 1D tensor of length $B+1$ where `block_offsets[`i`]` stores the cumulative number of thread blocks for all sequences before sequence i. This enables each thread block to determine its local block ID within a sequence.

These auxiliary structures are computed once per ragged tensor in an effort to minimize overhead. They enable thread blocks to independently identify which sequence they process and their position within that sequence.

3.3 Ragged Flash Attention

Our flash attention implementation modifies the attention algorithm from Dao et al. [3] to operate on ragged sequences. The key challenge is that standard flash attention assumes a fixed sequence length across all examples in a batch, which is incompatible with variable-length sequences.

We modify the flash attention algorithm to use the CoRa metadata for batch-aware processing:

1. Each thread block uses its global ID to look up its batch index from `grid_mapping`.
2. The batch index is used to retrieve the sequence length and starting offset for that specific example.
3. Within-sequence block indexing is computed using `block_offsets` to determine the local position.
4. The attention kernel processes only the actual tokens for each sequence, with masking applied at sequence boundaries.

This design maintains the tiled, online softmax computation of flash attention while adapting to variable-length inputs. Each thread block computes attention for a 64×64 tile independently, updating running statistics (max and sum) in an online fashion to compute the correct softmax normalization.

3.4 Ragged Batch Matrix Multiplication

For feed-forward layers in transformers, we implement a custom ragged batch matrix multiplication kernel. Given a ragged tensor A of shape (T_{total}, K) representing B sequences and a dense batch tensor B of shape (B, K, N), we compute

$C = AB$ where each sequence i is multiplied by its corresponding weight matrix $B[i]$.

The kernel uses a tiled matrix multiplication approach with blocks of size $64 \times 64 \times 64$. Similar to flash attention, each thread block:

1. Determines its batch assignment using `grid_mapping`
2. Loads tiles from the ragged tensor A
3. Loads tiles from the corresponding slice of the batch tensor B
4. Accumulates partial products and writes results back to the ragged output tensor

This approach ensures that each sequence is processed by the correct weight matrix while maintaining high arithmetic intensity through tiled computation.

4 Implementation and Evaluation

4.1 Implementation Details

The codebase consists of several key components:

Core Data Structures

- `RaggedTensor`: The primary data structure implementing operator overloading for addition, multiplication, and matrix multiplication. It provides methods for conversion to/from dense padded representations.
- `CoRa Prelude`: Computes and caches the grid mapping and block offsets for a given ragged tensor. Results are cached to avoid redundant computation.

Triton Kernels We implement two primary kernels using Triton:

- Implements flash attention as proposed by Dao et. al [3] for ragged sequences with automatic tuning over block sizes (32, 64, 128) and warp configurations (4, 8 warps).
- Performs batch matrix multiplication with automatic tuning over block sizes and configurations.

Both kernels are registered with PyTorch's custom operator framework using `@triton_op` decorators, enabling integration with `torch.compile` and the TorchDynamo graph construction.

Transformer Components. We provide ragged-aware versions of standard transformer components as presented in Vaswani et al.

- `RaggedMultiheadAttention`: Multi-head self-attention using our flash attention kernel
- `RaggedLayerNorm`: Layer normalization operating on flattened ragged data
- `RaggedLinear`, `RaggedReLU`, `RaggedDropout`: Element-wise operations preserving ragged structure
- `RaggedTransformerEncoderLayer`: Complete encoder layer with pre-normalization
- `RaggedTransformerEncoder`: Full transformer encoder with configurable depth

All ragged operations preserve the offset and length metadata for reduced conversion overhead and use native PyTorch operations where possible.

4.2 Experimental Setup

We evaluate our implementation against PyTorch's native `TransformerEncoder` using padding masks on an NVIDIA RTX 3060 (12GB VRAM) with an AMD Ryzen 5 9600X CPU. The software environment consists of PyTorch 2.9.1, CUDA 13.0, GCC 13.3.0, and 16GB DDR5 RAM. We use `torch.compile` with `mode="reduce-overhead"` for both implementations to ensure fair comparison.

Fegade et al. present eight NLP datasets with varying characteristics: RACE, Wikipedia English Sequence variants, SQuAD, MNLI, XNLI, MRPC, and CoLA. These datasets exhibit different sequence length distributions and sizes, allowing us to evaluate performance across diverse workloads. We test batch sizes of 32, 64, and 128 to examine scaling behavior. All models use 6 layers, 8 attention heads, hidden dimension 512, and feedforward dimension 2048, matching standard transformer configurations as presented in Fegade et al.

Due to reproducibility concerns and dataset access issues, we elected to generate synthetic data using Beta distributions parameterized to match the statistical properties (min, max, and mean) reported in Fegade et al. rather than using the original datasets directly. This approach does not affect the performance evaluation, as the computational characteristics are determined by the sequence length distributions rather than the semantic content of the data. For each benchmark, we generate batches with sequence lengths sampled from a Beta distribution with concentration parameter 5.0 and mean normalized to the provided ranges. Each experiment runs 10 iterations after 5 warmup iterations.

4.3 Results and Analysis

Table 1 presents the latency comparison between PyTorch's padded implementation and our ragged tensor implementation. Results are reported as mean latency in milliseconds over 10 iterations.

Our results reveal several important insights:

Dataset-Dependent Performance. Performance gains are highly dependent on dataset characteristics. The characteristics of the SQuAd dataset shows the best speedups (1.54×–1.63×), while datasets with shorter average sequences like CoLA show significant regression (0.31×–0.47×). This suggests that our implementation's overhead is amortized effectively only when sequences are sufficiently long and variable in length. This generally matches the observations of Fegade et al.

Batch Size Scaling. Larger batch sizes generally improve our relative performance, with the most pronounced gains at batch size 128. This is expected as larger batches provide more parallelism to saturate the GPU, and thus the CoRa metadata computation overhead becomes amortized.

PyTorch Evolution Context. A critical observation is that our speeds are lower than those reported in Fegade et al. Since 2021, PyTorch has undergone significant improvements, causing the performance difference between padded and ragged approaches to be less pronounced.

- **FlashAttention integration**: PyTorch now includes optimized FlashAttention implementations that reduce memory bandwidth requirements even with padding masks. [9]
- **torch.compile**: The PyTorch 2.x compiler applies graph-level optimizations, kernel fusion, and CUDA graph capture that substantially improve baseline performance. [1]
- **Hardware Evolution**: Both the original CoRa paper and our work use different GPU architectures, making direct comparison difficult.

Table 1. Transformer encoder latency comparison (ms). Lower is better. Speedup computed as Ragged time / PyTorch time.

Dataset	Batch	PyTorch	Ragged	Speedup
RACE	32	132.68	110.51	1.20×
	64	265.08	216.15	1.23×
	128	543.43	411.81	1.32×
Wikipedia English Seq512	32	135.24	119.48	1.13×
	64	278.12	206.59	1.35×
	128	563.98	435.18	1.30×
SQuAd	32	84.88	55.20	1.54×
	64	170.04	110.74	1.54×
	128	345.40	211.69	1.63×
Wikipedia English Seq128	32	26.71	29.50	0.91×
	64	52.77	57.18	0.92×
	128	105.64	111.92	0.94×
MNLI	32	21.25	18.90	1.12×
	64	45.99	31.50	1.46×
	128	84.34	67.70	1.25×
XNLI	32	25.23	25.29	1.00×
	64	51.77	46.00	1.13×
	128	95.81	90.66	1.06×
MRPC	32	19.74	21.54	0.92×
	64	39.38	39.11	1.01×
	128	80.52	72.31	1.11×
CoLA	32	5.40	17.51	0.31×
	64	10.20	28.94	0.35×
	128	25.98	55.73	0.47×

These improvements explain why our ragged implementation shows more modest gains (and occasional regressions) compared to the original CoRa results. The padded baseline has become significantly more competitive, particularly for smaller sequences where padding overhead is minimal.

Our results suggest ragged tensors are most beneficial when:

1. Sequences have substantial length variance (high standard deviation)
2. Average sequence length is moderate to long
3. Batch sizes are large enough to amortize metadata computation

Implementation Overhead. The regressions on short-sequence datasets (Wikipedia English Seq128, CoLA) highlight the overhead of our custom Triton kernels and metadata management. For very short sequences, the cost of computing grid mappings, launching custom kernels, and managing ragged structure outweighs the savings from avoiding padding. Modern PyTorch's highly optimized standard operations are extremely efficient for small-scale problems, even with padding.

In summary, while ragged tensors can provide meaningful speedups on appropriate workloads (up to 1.63× on SQuAd), the evolution of PyTorch's ecosystem has reduced the performance gap compared to earlier comparisons. Future work could explore more aggressive kernel fusion, mixed ragged/-dense strategies, and applying heuristics to determine when to use ragged representations.

References

[1] Jason Ansel, Edward Yang, Horace He, Natalia Gimelshein, Animesh Jain, Michael Voznesensky, Bin Bao, Peter Bell, David Berard, Evgeni Burovski, Geeta Chauhan, Anjali Chourdia, Will Constable, Alban Desmaison, Zachary DeVito, Elias Ellison, Will Feng, Jiong Gong, Michael Gschwind, Brian Hirsh, Sherlock Huang, Kshiteej Kalambarkar, Laurent Kirsch, Michael Lazos, Mario Lezcano, Yanbo Liang, Jason Liang, Yinghai Lu, C. K. Luk, Bert Maher, Yunjie Pan, Christian Puhrsch, Matthias Reso, Mark Saroufim, Marcos Yukio Siraichi, Helen Suk, Shunting Zhang, Michael Suo, Phil Tillet, Xu Zhao, Eikan Wang, Keren Zhou, Richard Zou, Xiaodong Wang, Ajit Mathews, William Wen, Gregory Chanan, Peng Wu, and Soumith Chintala. 2024. PyTorch 2: Faster Machine Learning Through Dynamic Python Bytecode Transformation and Graph Compilation. In *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2* (La Jolla, CA, USA) *(ASPLOS '24)*. Association for Computing Machinery, New York, NY, USA, 929–947. doi:10.1145/3620665.3640366

[2] Tianqi Chen, Thierry Moreau, Ziheng Jiang, Lianmin Zheng, Eddie Yan, Meghan Cowan, Haichen Shen, Leyuan Wang, Yuwei Hu, Luis Ceze, Carlos Guestrin, and Arvind Krishnamurthy. 2018. TVM: An Automated End-to-End Optimizing Compiler for Deep Learning. arXiv:1802.04799 [cs.LG] https://arxiv.org/abs/1802.04799

[3] Tri Dao, Daniel Y. Fu, Stefano Ermon, Atri Rudra, and Christopher Ré. 2022. FlashAttention: Fast and Memory-Efficient Exact Attention with IO-Awareness. arXiv:2205.14135 [cs.LG] https://arxiv.org/abs/2205.14135

[4] Pratik Fegade, Tianqi Chen, Phillip B. Gibbons, and Todd C. Mowry. 2022. The CoRa Tensor Compiler: Compilation for Ragged Tensors with Minimal Padding. arXiv:2110.10221 [cs.LG] https://arxiv.org/abs/2110.10221

[5] Fredrik Kjolstad, Stephen Chou, David Lugato, Shoaib Kamil, and Saman Amarasinghe. 2017. Taco: A tool to generate tensor algebra kernels. In *2017 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE)*. 943–948. doi:10.1109/ASE.2017.8115709

[6] Erdal Mutlu, Ruiqin Tian, Bin Ren, Sriram Krishnamoorthy, Roberto Gioiosa, Jacques Pienaar, and Gokcen Kestor. 2021. COMET: A Domain-Specific Compilation of High-Performance Computational Chemistry. arXiv:2102.06827 [cs.MS] https://arxiv.org/abs/2102.06827

[7] NVIDIA. 2019. FasterTransformer. https://github.com/NVIDIA/FasterTransformer

[8] Jonathan Ragan-Kelley, Andrew Adams, Dillon Sharlet, Connelly Barnes, Sylvain Paris, Marc Levoy, Saman Amarasinghe, and Frédo Durand. 2017. Halide: decoupling algorithms from schedules for high-performance image processing. *Commun. ACM* 61, 1 (Dec. 2017), 106–115. doi:10.1145/3150211

[9] PyTorch Team. 2023. Accelerated PyTorch 2 Transformers. https://pytorch.org/blog/accelerated-pytorch-2/

[10] Philippe Tillet, H. T. Kung, and David Cox. 2019. Triton: an intermediate language and compiler for tiled neural network computations. In *Proceedings of the 3rd ACM SIGPLAN International Workshop on Machine Learning and Programming Languages* (Phoenix, AZ, USA) *(MAPL 2019)*. Association for Computing Machinery, New York, NY, USA, 10–19. doi:10.1145/3315508.3329973

[11] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin. 2023. Attention Is All You Need. arXiv:1706.03762 [cs.CL] https://arxiv.org/abs/1706.03762