DANLING NESTEDTENSOR: COMPOSABLE MULTI-RAGGED TENSORS FOR DEEP LEARNING

Anonymous authors
Paper under double-blind review

ABSTRACT

Variable-size inputs are common in deep learning, but dense batching allocates a shared envelope and spends computation on padding. The cost multiplies across varying axes: an explicit pair state allocates $BN_{\max}^2$ positions instead of $\sum_i N_i^2$. Packing removes that waste, but composing packed operations still requires the logical axes and sample boundaries a flat buffer no longer exposes. We present DanLing NestedTensor, a PyTorch tensor abstraction that makes multi-ragged structure a property of the tensor itself. Packed values carry tensor-backed partitions and logical dimension order, so broadcasting creates ragged axes, feature transformations retain them, and reductions consume them. The same representation carries through autograd and both eager and compiled execution. On an A100, the geometric-mean speedup over same-mode padding is 2.74× eager and 3.39× compiled across four BERT scales, and 1.97× eager across four FCN backbones. A four-block Pairformer-style workload runs 2.40–4.32× faster than padding with native PyTorch kernels across square length regimes in eager execution, with peak allocation falling from 38.08 to 5.41 GiB on its high-variation batch. The tensor interface lets model code built from its supported operators compose efficient variable-size computation without managing offsets at any call site. Code will be released publicly upon publication.

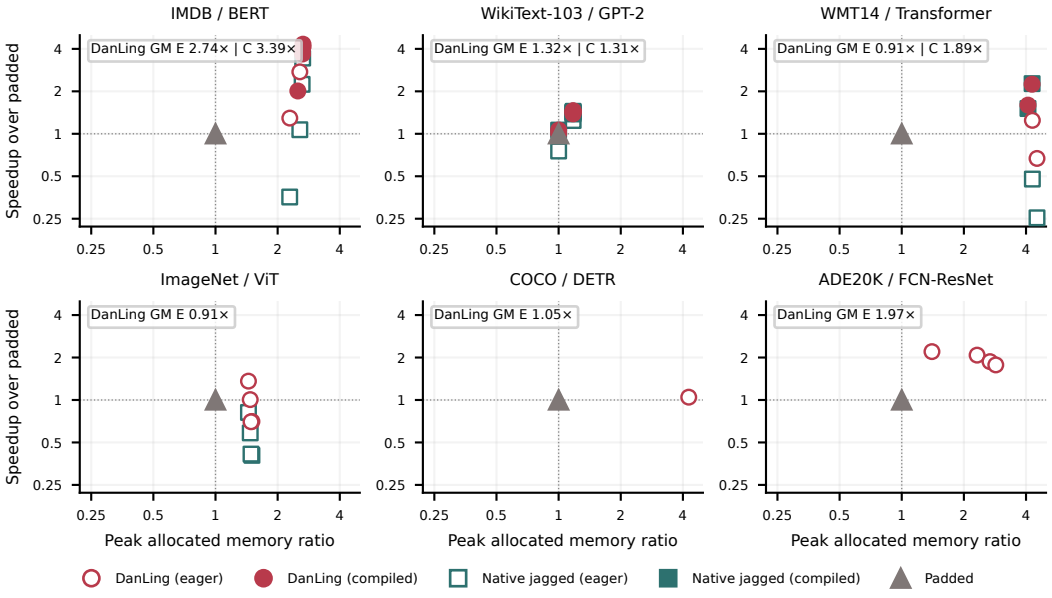

Figure 1: **Speed and memory efficiency relative to same-mode padded execution.** One panel per workload; one point per model scale and mode, against that cell's own padded baseline at $(1, 1)$. Shape and colour identify the method, fill the mode (hollow eager, solid compiled). Both axes use steady-state execution (Section 4.1), so the coordinates share one measurement window; Appendix Table 3 reports compilation counts and full-trace cost. Inline text gives DanLing's geometric mean per mode. All completed paired measurements are shown; other outcomes use the codes of Appendix C.

Reviewers: please read the Reviewer Guidelines (iclr.cc/Conferences/2027/ReviewerGuidelines) and the AI Policy for Reviewers (iclr.cc/Conferences/2027/AIPolicyForReviewers). If you used AI to expand, edit, or polish your review, please provide the input text to the LLM. Better still, consider skipping the LLM and submitting your original text: we, and the authors, are much more interested in your unedited thoughts than in what an LLM has to say. AI-assisted or not, you are putting your name and reputation behind your review: LLM-generated falsehoods, hallucinations or misrepresentations are subject to disciplinary action, which may include desk-rejecting all papers you have authored.

1 INTRODUCTION

Data vary in size; dense tensor batches do not. Sequence packing already handles the one-dimensional case, and for language models it is close to sufficient. Much of what people train is not one-dimensional: a protein structure model refines an $N_i \times N_i$ residue-pair state through dozens of blocks (Jumper et al., 2021; Abramson et al., 2024), and vision models are trained at variable resolution and aspect ratio (Dehghani et al., 2023). Convolution must keep height and width separate to address neighbourhoods within each image. Padding places these examples inside a common envelope, consuming memory and computation at empty positions (Fegade et al., 2022). The cost multiplies across varying axes: a pair representation over a length-N_i sample holds N_i^2 entries, so a padded batch allocates $BN_{\max}^2$ of them rather than $\sum_i N_i^2$; a sample at half the maximum length wastes three quarters of its block rather than half. On our Pairformer-style workload padded allocation barely moves: 34.76–38.09 GiB whether lengths are near-uniform or badly skewed (Table 2).

Collecting that saving is harder than allocating a smaller buffer, because the pair state is not where the program ends. A single–pair–single block broadcasts two single representations into a pair tensor, a residual feature transformation updates it, and a reduction over one pair axis returns singles. Packing the valid pair cells removes the padded storage, but the flat buffer no longer shows which cells share a sample and a row. The reduction needs that grouping, and its result must carry a structure the next operator can accept. When this structure is maintained outside the tensor, model code must supply indexing and reconstruction logic at each structural transition.

Existing approaches address different parts of this problem. Bucketing groups examples of similar size, which reduces padding but ties batch composition to input geometry and can change the order and statistics of optimisation (Morishita et al., 2017; Kocmi & Bojar, 2017). Packing keeps the chosen batch, and variable-length attention kernels execute attention on its packed storage (Dao et al., 2022; Dao, 2024). Neither specifies how logical axes and sample boundaries are maintained when a model constructs, transforms and reduces an explicit pair state, and PyTorch's `torch.jagged` admits only one ragged dimension (PyTorch contributors, 2026c).

DanLing NestedTensor treats this bookkeeping the way ordinary tensors treat shape and dtype: as state the tensor carries, not state the model rebuilds. The packed payload travels with exact element sizes, hierarchical partitions and logical dimension order, and each handler updates that structure as it computes, so autograd and compilation consume the same representation. Figure 1 summarises the speed and memory this yields; composition is the sharper test. A four-block Pairformer-style application, which creates, transforms and reduces two ragged pair axes in every block, compiles and trains under this representation with default compiler settings, while the padded and hand-packed implementations we tested do not—although the simpler block compiles for all three.

This paper makes three contributions.

- **A representation that composes.** Multiple varying logical axes are carried together, including non-leading axes and rectangular pair states, without conflating them with the flattened storage axis. Each handler consults and updates those axes as it already consults shape—broadcasting creates one, feature transformations preserve it, reductions consume it—and returns a structure the next operator can use, so a complete single–pair–single program composes without offset bookkeeping at any call site. Handlers of this kind cover 404 distinct operators across PyTorch's two dispatch protocols (Sections 3.1 and 3.2).
- **Differentiation and compilation across the representation boundary.** Bridges in both directions keep the gradient connected where a structured wrapper meets its packed payload, so a compiled block can return singles and pairs to an eager loss and still receive gradients through changes of rank and layout. Compilation fixes the structural schema at each program point while sizes and partitions stay runtime tensor inputs, so supported length-dynamic programs reuse one compiled graph rather than specialising against each length vector, which is what lets the complete pair application compile (Section 3.4, Table 3).
- **Efficiency, and where it comes from.** Four BERT scales give a 3.39× compiled geometric-mean speedup over padding and FCN-ResNet50 drops from 28.38 to 12.25 GiB; a four-block Pairformer-style workload runs 4.32× faster than padding with native PyTorch kernels on its high-variation batch. Against explicit packing, a matched-granularity control separates cost from strategy: at equal call granularity the tensor interface adds at most 1.60% latency, while batch-level segmented execution runs 1.53–3.94× faster than per-sample calls (Sections 4.2 and 4.4).

2 Related Work

Ragged representations. TensorFlow RaggedTensor, Awkward Array and FBGEMM jagged operators represent variable-size data through numerical buffers and structural metadata (TensorFlow contributors, 2026; Pivarski et al., 2020; FBGEMM contributors, 2026). PyTorch's native `torch.jagged` combines a tensor interface with packed storage, differentiation and compilation for its supported operations; its documented layout admits one ragged dimension (PyTorch contributors, 2026c).

Ragged and dynamic compilation. CoRa provides a programming interface and compiler for ragged computation, using techniques including selective padding and operation splitting (Fegade et al., 2022). PyTorch 2 integrates graph capture, differentiation and backend compilation, while Relax makes symbolic shapes explicit across computational graphs and tensor programs (Ansel et al., 2024; Lai et al., 2025). NestedTensor integrates its representation with PyTorch, exposing changing sizes and partitions as tensor data while preserving the structural schema of each program point.

Packing and specialised kernels. Sequence packing and NaViT retain independent-example computation while reducing padding in language and vision workloads (Krell et al., 2021; Dehghani et al., 2023). FlashAttention and FlashAttention-2 optimise attention execution, and FlexAttention provides a programmable fused attention interface (Dao et al., 2022; Dao, 2024; Dong et al., 2025). NestedTensor reuses these execution paths while keeping projections, normalisation and residual operations packed around them. KeOps evaluates formula-defined pairwise computations lazily (Charlier et al., 2021), whereas our pair states persist across multiple feature transformations before a structural reduction.

3 NestedTensor: Representation and Execution

A single–pair–single block illustrates how NestedTensor carries logical structure through successive operations. For nonempty inputs $X_i \in \mathbb{R}^{N_i \times C}$, the block computes

$$\begin{aligned} P_i[a,b,:] &= u_\theta(X_i[a,:]) + v_\theta(X_i[b,:]), \\ \widetilde{P}_i &= P_i + h_\theta\big(\mathrm{LN}_{C_p}(P_i)\big)\,, \\ \widetilde{X}_i[a,:] &= X_i[a,:] + g_\theta\left(\frac{1}{N_i}\sum_{b=0}^{N_i-1}\widetilde{P}_i[a,b,:]\right). \end{aligned} \tag{1}$$

Here u_θ, v_θ project to pair width C_p, h_θ is a SwiGLU (Shazeer, 2020) feature update, and g_θ maps the reduced features back to width C. The block returns both updated singles and pairs, so the pair state is model state that later operations consume, not a score matrix that exists only inside a fused attention call. Figure 2 draws what the tensor holds at each stage of the block for a two-sample batch, with the exact packed shapes and offsets, and Figure 3 gives the measured implementation (Appendix A.1).

3.1 Packed Representation of Variable-Size Batches

Problem setting. A batch $\mathcal{X} = (X_0, \ldots, X_{B-1})$ contains individually dense tensors of common rank D; raggedness is variation between samples, not irregularity inside one. Let $n_{i,d}$ be the extent of element dimension d in sample i. An ordered tuple $\mathcal{R} = (r_1, \ldots, r_q)$ declares the ragged axes. The complementary static axes $\mathcal{S} = (s_1, \ldots, s_{D-q})$ have common extents $c_j = n_{i,s_j}$ across samples.

Numerical storage. The permutation $\pi = (\mathcal{R}, \mathcal{S})$ places ragged axes before static axes. We collapse each sample's ragged prefix and concatenate the resulting rows. Writing $p_i = \prod_{j=1}^{q} n_{i,r_j}$ for a sample's packed row count, $T = \sum_i p_i$ and $o_i = \sum_{k<i} p_k$, the value tensor $V \in \mathbb{R}^{T \times c_1 \times \cdots \times c_{D-q}}$ satisfies

$$V[o_i : o_{i+1}] = \mathrm{reshape}(\mathrm{permute}_\pi(X_i), (p_i, c_1, \ldots, c_{D-q}))\,. \tag{2}$$

Images (C, H_i, W_i) pack as $(\sum_i H_i W_i, C)$, pairs (N_i, M_i, C) as $(\sum_i N_i M_i, C)$, and non-leading layouts (S, N_i, C) as $(\sum_i N_i, S, C)$.

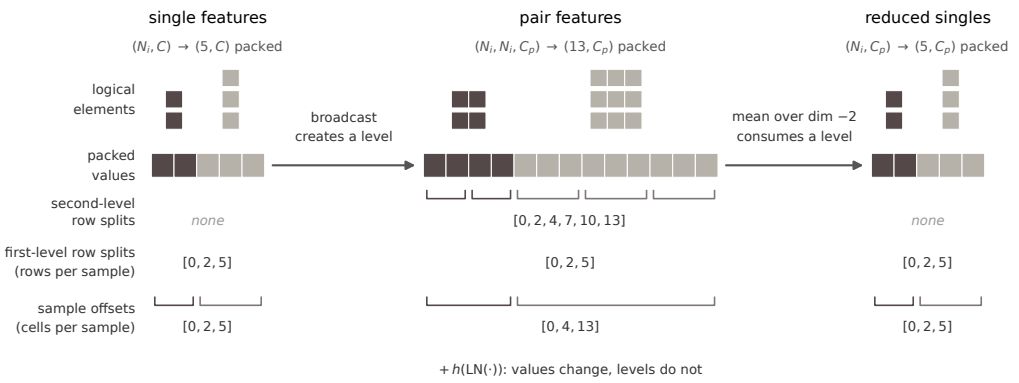

Figure 2: **Structure through a single–pair–single block.** For lengths $(2, 3)$, broadcasting expands five single positions into thirteen pair cells; column reduction returns five singles. First-level row splits count rows per sample, second-level splits delimit each pair row's cells, and sample offsets delimit complete elements in packed-cell space. First-level row splits and sample offsets coincide for singles but differ for pairs.

```
pair   = self.left_projection(left).unsqueeze(-2) + self.right_projection(right).unsqueeze(-3)
pair   = pair + self.transition(pair)
single = left + self.reduction_projection(pair.mean(dim=-2))

# the same pair construction, packed by hand:
pair   = self.left_projection(left)[rows] + self.right_projection(right)[columns]
```

Figure 3: **Nothing in the model code refers to the packing.** The measured implementation of Equation 1, with its three method bodies inlined: `left_projection` and `right_projection` are u_θ and v_θ, `transition` is the pre-normalised update $h_\theta \circ \mathrm{LN}_{C_p}$, and `reduction_projection` is g_θ. The tensor carries the partitions of Figure 2 through all three lines without appearing in any of them. The last line is the explicit-packed baseline's equivalent of the first, reaching the same values but requiring caller-supplied `rows` and `columns` gather indices.

Structural metadata. The wrapper retains the exact size matrix $A_{i,d} = n_{i,d}$, sample offsets o, dimension order π, and row splits at each ragged level. Layouts with explicitly declared ragged axes retain the partitions as tensors, so later operations and reconstructed wrappers can use them directly. Although sizes can generate these partitions, retaining them avoids recovering per-sample Python shapes at each reconstruction boundary, and exact sizes distinguish zero-volume shapes such as $(0, 3)$ and $(0, 7)$ whose offsets coincide. The public logical shape describes the padded envelope; numerical storage remains V.

3.2 Structure-Aware Operator Dispatch

A handler interprets its operands in logical coordinates, runs the numerical operation in packed coordinates, and constructs the structure of its result, so that result is ready for the next operator without model-level reconstruction. NestedTensor installs these handlers on high-level PyTorch functions through `__torch_function__` and on ATen operations through `__torch_dispatch__`, covering 404 distinct operators across the two protocols (Appendix B). For ragged coordinates $(a_1, \ldots, a_q)$ in sample i, the packed row is

$$\ell_i(a_1, \ldots, a_q) = o_i + \sum_{j=1}^{q} a_j \prod_{k=j+1}^{q} n_{i,r_k}. \tag{3}$$

Static axes map individually to axes of V, while the ragged coordinates jointly determine its leading index.

Creating axes through broadcasting. The projected singles in Equation 1 acquire complementary singleton dimensions, giving element shapes $(N_i, 1, C_p)$ and $(1, N_i, C_p)$. Their broadcast cre-

ates (N_i, N_i, C_p) pair states. The handler associates each pair cell with its sample-local row and column, gathers the corresponding projected features, and constructs the square output sizes and partitions. Packed row counts change from N_i to N_i^2, and in Figure 2 the sample offsets from $[0, 2, 5]$ to $[0, 4, 13]$. The first pair partition retains the single-row grouping, while the second records the columns within each row.

Retaining structure through feature transformations. Activations and feature-wise normalisation retain the pair axes and partitions. A projection updates the static feature extent, while the residual addition requires matching logical pair structure. These operations reuse the existing partitions instead of inferring new ones from their current extents.

Consuming one logical axis. For a rectangular pair state $P_i \in \mathbb{R}^{N_i \times M_i \times C_p}$, a column sum Y_i groups cells with the same sample and row coordinate:

$$V_Y[\eta_i + a, :] = \sum_{b=0}^{M_i - 1} V_P[o_i + aM_i + b, :], \qquad \eta_i = \sum_{k<i} N_k. \tag{4}$$

The output has shape (N_i, C_p) in each sample, with offsets η and the remaining row axis. The square-pair mean in Equation 1 additionally divides each group by N_i. Its projection can then be added to X_i because the handler has restored the single representation's logical structure as well as its values.

Compatibility and semantics. Elementwise alignment checks logical extents and dimension order, including the grouping represented by partitions. For example, shapes $(2, 6, C)$ and $(3, 4, C)$ both flatten to twelve feature rows but are incompatible for direct elementwise addition. Dense operands are aligned to logical axes before permutation or gathering. Each handler is specified so that interpreting its packed output logically agrees with running the reference operation on the logically interpreted inputs, which for sample-local operations is the corresponding dense element computation; Appendix A states that contract and the semantics of batch and global operations.

3.3 Efficient Execution on Packed Storage

Dense work across valid positions. Feature-wise operations reuse dense kernels on V. Projections compute $V' = VW^\mathsf{T} + b$, and normalisation acts on each row's static feature tail. This preserves a large numerical batch while eliminating padded positions: the feature transformations in Equation 1 process $\sum_i N_i^2$ pair cells rather than $BN_{\max}^2$.

Partition-dependent computation. Ragged-axis reductions derive group indices from retained coordinates and use indexed accumulation or scatter reduction, with means additionally using group counts. Pair broadcasting uses coordinate maps to gather the projected row and column features before their numerical combination. Large pair-coordinate maps are constructed on the values device from the per-sample lengths and sample offsets, which are batch-sized, and their construction, gathers and backward accumulation contribute to runtime and temporary storage. The partitions also provide the segment boundaries consumed by variable-length kernels.

Attention through packed blocks. Projections produce token-major queries, keys, and values with static head and feature axes. The variable-length attention path passes these buffers and cumulative lengths to FlashAttention (Dao et al., 2022; Dao, 2024), and the output adopts the query structure, allowing different query and key lengths in cross-attention. FlexAttention paths map packed indices to local positions and restrict interactions to matching sample identifiers (Dong et al., 2025). Projections, feature normalisations, and residual operations share the packed representation around attention, avoiding conversions at each block boundary.

Convolution over variable-size images. Convolution requires neighbourhoods in each image's own coordinate system. The spatial path gathers sample-local input neighbourhoods, including the receptive-field halo, into batches of equal-size tiles; native convolution processes those tiles and valid outputs are scattered back into packed storage, with coordinates and boundary padding defined within each image so that no interaction crosses a sample boundary. Backward accumulates

overlapping input-gradient contributions and sums parameter gradients across tiles. This replaces a batch-wide padded envelope with tile buffers, trading padded computation for halo duplication and gather/scatter work; Appendix B gives the halo geometry and the tiling tradeoff.

Structural costs. Execution cost divides into numerical kernels, structural indexing and data movement; packed execution pays off when the saved padded work outweighs the indexing and movement it adds. Retained partitions additionally let an operator issue one segmented call over the batch instead of one call per sample.

3.4 Autograd and Dynamic Compilation

Gradients across representations. Gradient connectivity must be preserved where a structured wrapper meets its packed payload. Autograd records its graph over the payload while the model's forward returns the wrapper, and a compiled block can hand structured singles and pairs to an eager consumer, so that boundary is crossed in both directions within one step and has to be differentiable itself. A wrapper-to-packed bridge passes the numerical payload to such a consumer and, in backward, rebuilds a gradient carrying the same partitions and dimension order as the forward value. The inverse bridge attaches packed numerical results to their output wrappers and aligns incoming gradients with the packed dimension order. Together they preserve the dependency through changes in rank, layout and representation; for Equation 1 they connect the eager loss back to the compiled projections, pair update and reduction.

Runtime sizes and structural schemas. Compilation fixes the structural schema at each program point: rank, declared ragged axes, logical dimension order, and static feature layout. The primary length-dynamic evaluation also fixes batch size; we report no compiled results for traces whose batch size changes. Different program points may have different schemas: in Equation 1 broadcasting creates a pair and reduction returns singles, so its three program points do not share one schema, though each is fixed across calls. Per-sample sizes and partitions remain runtime tensor inputs, so supported length-dynamic programs reuse a compiled graph rather than specialising on each length vector. Output reconstruction combines the fixed schema with these runtime tensors, and custom-operation boundaries represent data-dependent packed extents symbolically. Explicit declarations preserve an axis's meaning even when observed sizes happen to coincide, so a batch whose lengths are momentarily equal still compiles to the ragged schema.

4 Experiments

The evaluation asks when the abstraction improves speed and memory over padding, whether it carries a complete multi-ragged computation, and where the benefit comes from.

4.1 Setup

Workloads and methods. The models are randomly initialised systems workloads, measured for execution cost rather than task quality. The dataset study spans BERT (Devlin et al., 2019) on IMDB (Maas et al., 2011), GPT-2 (Radford et al., 2019) on WikiText-103 (Merity et al., 2016), an encoder–decoder Transformer (Vaswani et al., 2017) on WMT14 (Bojar et al., 2014), a variable-resolution ViT (Dosovitskiy et al., 2021) on ImageNet-1K (Russakovsky et al., 2015), DETR-R50 (Carion et al., 2020) on COCO (Lin et al., 2014) and FCN-ResNet (He et al., 2016; Long et al., 2015) on ADE20K (Zhou et al., 2019); Appendix C specifies the model scales and configurations. The main comparison is padded execution, DanLing, and native `torch.nested` using its `torch.jagged` layout. Three analysis controls isolate separate effects: comparing attention-only with whole-program packing assesses the benefit of keeping computation packed beyond attention; explicit packing is a direct-programming reference that manages offsets in model code; and matched-granularity execution separates the cost of the tensor interface from the benefit of batch-level call granularity. Each paired comparison uses the same examples in the same order, with the same initial weights, objective and logical batch partitions. Eager and Inductor-compiled execution are parallel experiments, each normalised to padding in the same mode.

Measurement. The experiments use an A100-SXM4-80GB with PyTorch `2.13.0+cu132` and BF16 parameters and floating-point inputs. TF32 and stochastic layers are disabled for the comparisons.

Table 1: **Single–pair–single program, eager and compiled.** Full-trace mean ms/step / peak allocated GiB; the final column is speedup over padding. Both modes run the same logical workloads.

Layout	Mode	Padded	Explicit packed	DanLing	Speedup
square	eager	25.1 / 5.59	8.7 / 1.60	11.1 / 1.61	2.26×
square	compiled	21.2 / 4.64	7.6 / 1.35	9.5 / 1.25	2.23×
rectangular	eager	24.2 / 4.99	9.1 / 1.62	11.6 / 1.62	2.09×
rectangular	compiled	18.1 / 4.14	8.0 / 1.36	10.1 / 1.26	1.79×
nonleading	eager	25.7 / 5.53	8.5 / 1.63	11.4 / 1.63	2.26×
nonleading	compiled	22.1 / 4.59	7.5 / 1.37	9.8 / 1.27	2.25×

Host-wall timing covers forward computation, loss, backward and runtime structural work; preparation, input transfer and optimiser updates are outside this compute boundary. Comparisons against padding use memory efficiency $E_m = M_{\mathrm{pad}}/M_m$ and speedup $S_m = T_{\mathrm{pad}}/T_m$, both improving above one. Results use one of two windows throughout: *full-trace* pools every measured step, keeping any in-trace compilation in the total, while *steady-state* pools the steps on which *neither* method compiled, so both sides average identical batches. Time and peak allocated memory are taken over the selected steps; the windows coincide in eager execution. Figure 1 reports steady-state; Figure 4 and the result tables report full-trace; Appendix Table 3 gives both where they diverge, and Appendix B the retained fraction per cell. Configuration ranges describe variation across model scales rather than independent-run uncertainty.

Validation protocol. Values, logical structure and gradients are checked against matching references at the same precision. The performance matrix retains all completed paired measurements; other outcomes are reported separately using the codes defined in Appendix C.

4.2 Speed and Memory Across Workloads

Across the four BERT scales, DanLing gives geometric-mean speedups of 2.74× in eager and 3.39× in compiled execution. BERT-Base uses 61.95% less peak allocated memory than padding. The four eager FCN backbones give a 1.97× geometric-mean speedup, with a 28.38-to-12.25 GiB allocation reduction for ResNet50. GPT-2 on WikiText-103 gives an eager geometric-mean speedup of 1.32×. Peak reserved memory favours packing less and, for GPT-2 Small to Large, exceeds padding (Appendix C).

The vision results separate the two kinds of benefit: variable-resolution ViT has a 0.91× eager geometric-mean throughput ratio while improving allocated-memory efficiency by 1.48×. DETR is close to padded throughput (1.05×) with 4.27× memory efficiency, against the matched padded reference that preserves DETR's valid-boundary semantics; Appendix C.7's conventional padded workflow changes those semantics and runs faster still.

Eager WMT-Base saves substantial memory but runs at 0.67× padded throughput. Its compiled speedup is 1.60× in steady state and 58.18× over the full trace, reflecting four additional padded recompilations that DanLing's fixed compiled schema avoids. Native jagged reaches steady-state throughput close to DanLing's in the completed compiled GPT-2 and WMT14 cells, while their eager results differ more widely across workloads.

4.3 Composing Multiple Ragged Axes

The single–pair–single program of Equation 1 directly exercises the paper's central design, with both of its outputs used by an eager, sample-weighted squared loss. The same program is expressed in square, rectangular and non-leading layouts, with $B = 8$, single width 384 and pair width 128. Padded, explicit-packed and DanLing executions share the feature transformations and objective.

NestedTensor runs 1.79–2.26× faster than padding across these layouts and modes (Table 1). The compiled program reuses its graph across the tested changing partitions and extents. NestedTensor and explicit packing both run at whole-batch granularity here, the latter through gather indices and a segmented reduction, so NestedTensor's 25–34% higher latency is the structural overhead of the

Table 2: **Four-block Pairformer-style application, eager execution.** Full-trace mean ms/step / peak allocated GiB; the final column is speedup over padding. The square regimes span length variability.

Shape regime	Padded	Explicit packed	DanLing	Speedup
Square, near-uniform ($B = 8$)	634.2 / 38.09	397.9 / 17.81	264.5 / 17.43	2.40×
Square, moderate ($B = 8$)	588.2 / 35.02	403.0 / 12.97	212.0 / 12.78	2.77×
Square, high variation ($B = 8$)	641.8 / 38.08	396.4 / 5.44	148.7 / 5.41	4.32×
Square, extreme variation ($B = 16$)	638.8 / 34.76	802.7 / 3.04	203.7 / 3.06	3.14×
Non-leading ($B = 8$)	651.5 / 33.12	488.5 / 22.83	494.6 / 22.83	1.32×
Rectangular ($B = 8$)	650.6 / 38.43	542.9 / 23.93	547.8 / 23.93	1.19×

tensor interface, which cheap operators amortise least; Section 4.4 measures the same interface at 1.60% on fused kernels.

A larger pair application. A four-block Pairformer-style workload extends the comparison to repeated triangle multiplication, starting- and ending-node triangle attention, single attention and feature transitions (Abramson et al., 2024), at single width 384 and pair width 128. Padding runs the triangle updates as native `einsum` contractions and masked attention over the envelope. For square pairs, explicit packing, which again manages offsets itself, invokes OOps (a fused triangle-kernel library) dense operators per sample, while NestedTensor invokes the corresponding segmented operators over the batch; both run pair-biased single attention per sample on unpacked slices.

The measured square regimes give 2.40–4.32× speedups over padding, which combine removed padding with fused kernels; explicit-packed and NestedTensor allocations are similar to each other and both well below padded. The non-leading and rectangular variants run per sample in both packed implementations, which then differ by no more than the interface overhead of Section 4.4.

Compiled execution. Under the default Inductor configuration, DanLing completes compiled execution in all four square regimes, while the tested padded and explicit-packed implementations fail during compilation. They fail at different points: padded execution fails an Inductor divisibility check on a symbolic pair extent, and explicit packing fails fake-tensor propagation on its data-dependent gather indices (Appendix B). Explicit packing's failure is the kind the runtime-tensor metadata of Section 3.4 avoids; padding's is a limitation of Inductor's fusion. The complete pair application therefore compiles, not only the single–pair–single block of Equation 1. Disabling Inductor fusion entirely, though not epilogue fusion alone, lets the padded implementation compile as a full graph, but only 1.13–1.17× faster than its eager execution; DanLing compiled is 2.23–6.08× faster again, a gap combining padding removal with fusion and kernel choice. Relative to its own eager execution, DanLing is 1.06–2.20× faster in steady state, with two of the four regimes recompiling once during the measured trace.

4.4 Performance Analysis

Whole-program packing. Figure 4 separates four implementations of the same BERT workload, from padding the whole model to keeping every surrounding operation packed.

For random batching, replacing padded attention helps, but the larger gain comes from also keeping the surrounding feature computation packed. Attention-only packing uses DanLing's variable-length backend with every other operation left padded, so it separates the backend from the rest: 1.47× padded throughput eager and 1.51× compiled, against DanLing's 3.94× and 4.31×. In compiled execution, DanLing and explicit packing differ by at most 0.19% in full-trace throughput under either batching policy, so a tensor interface can approach direct packed performance without model-level offset bookkeeping; eager execution retains a larger gap, as the multi-ragged program also shows.

Separating interface cost from execution strategy. Once padded cells are gone, how the remaining work is issued still matters. On the Pairformer workload NestedTensor is 1.50–3.94× faster than explicit packing, the opposite direction to the single–pair–single program of Section 4.3, because

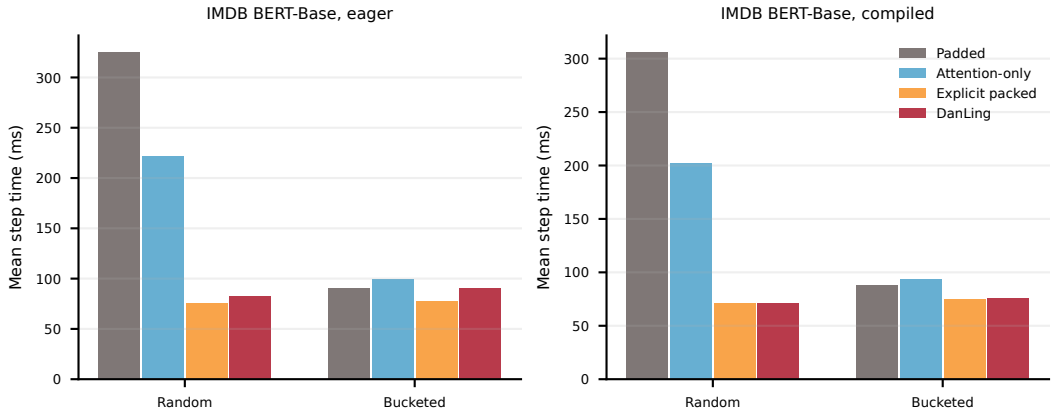

Figure 4: **Packing attention versus packing the complete program.** The four controls use the same IMDB BERT-Base workload. Attention-only packing includes unpadding and repadding; explicit packing and DanLing keep surrounding operations packed. Eager and compiled panels use full-trace mean step time (Section 4.1).

that baseline also differs in how it calls the kernels. A third implementation separates the two effects, keeping the NestedTensor interface but issuing one dense call per sample as the baseline does. At matched granularity the interface adds at most 1.60% to latency across the four square regimes, while moving to one segmented call per operator over the whole batch gives 1.53–3.94× (Appendix Table 7); the speedups above are that gain net of the interface overhead, up to run-to-run variation.

Batching policy. How much redundant work exists to remove depends on the batch: for sequences the valid fraction is $\rho_{\text{seq}} = \sum_i N_i / BN_{\max}$, with the image and pair analogues in Appendix C. Random and bucketed IMDB traces hold the same examples, partitioned differently, and bucketing raises token occupancy from 31.0% to 97.7%. The eager DanLing/padded throughput ratio then changes from 3.94× to 1.01×, and the compiled ratio from 4.31× to 1.17×. Packing and bucketing are therefore complementary: the representation preserves the chosen batch, while the gain available depends on that batch's geometry.

Model scale and shape variability. BERT speedups increase with model scale before levelling off at Base and Large, while length heterogeneity moves a different workload: across the square Pairformer regimes at $B = 8$ the speedup over padding rises from 2.40× under near-uniform lengths to 4.32× under high variability, because padding's cost tracks the batch maximum whatever the distribution. At that near-uniform end the lengths span only 208–224, for a pair occupancy of 94.4% (Appendix Table 4), so its speedup comes from the fused kernels rather than from removed padding. The extreme regime additionally doubles the batch to $B = 16$ and is reported separately (Table 2; Appendix Figure 5).

5 Discussion and Conclusion

NestedTensor keeps logical structure available after packing and transforms that structure together with the numerical values, through a common PyTorch tensor interface. The discriminating test is the complete four-block pair application, which compiles and trains under this representation with default compiler settings where the padded and hand-packed implementations we measured do not, although the simpler single–pair–single block compiles for all three. The measurements then separate where the efficiency comes from: on BERT, packing the whole program rather than attention alone; on the pair workload, issuing batch-level segmented calls rather than per-sample ones. How much there is to gain stays conditional—bucketing recovers most of it on IMDB, and memory savings can coexist with slower execution—but the composition it rests on does not.

Structure that survives into execution is useful twice: it removes padded work, and it leaves the execution layer enough information to choose how the remaining work is issued.

AI USE STATEMENT

Generative AI assistants were used to support software development, experiment tooling, analysis, and manuscript preparation. They were not used for research ideation or for literature search and related-work discovery: the research question, the design of the representation and its operator rules, and the choice and framing of the experiments are the author's own, and every cited work was located and read by the author. No reported quantity was produced by a language model: every number in the text, tables and figures is emitted by one script from the frozen benchmark records. AI-assisted code was reviewed by running it and comparing its output against a reference implementation; AI-assisted text was checked against the stored artifacts, and its cross-references, macros and citations verified, by automated scans over the sources. The author reviewed all AI-assisted work and takes responsibility for the implementation, analysis, claims, and final manuscript.

REPRODUCIBILITY STATEMENT

The representation and its structural transformations are specified in Section 3, with reference semantics in Appendix A and execution paths in Appendix B. Appendix C records the environment, model configurations, measurement boundaries and the outcome codes used when a cell has no completed measurement; Section 4.1 defines the two timing windows, and Appendix B reports how many steps the steady-state window retains.

ETHICS STATEMENT

The experiments use existing public NLP and vision datasets under their respective access terms and licenses. They measure execution efficiency without collecting new human-subject data or deploying trained models. The models are randomly initialised, so no model capable of downstream use is produced or released.

REFERENCES

Josh Abramson, Jonas Adler, Jack Dunger, Richard Evans, Tim Green, Alexander Pritzel, Olaf Ronneberger, Lindsay Willmore, Andrew J Ballard, Joshua Bambrick, Sebastian W Bodenstein, David A Evans, Chia-Chun Hung, Michael O'Neill, David Reiman, Kathryn Tunyasuvunakool, Zachary Wu, Akvilė Žemgulytė, Eirini Arvaniti, Charles Beattie, Ottavia Bertolli, Alex Bridgland, Alexey Cherepanov, Miles Congreve, Alexander I Cowen-Rivers, Andrew Cowie, Michael Figurnov, Fabian B Fuchs, Hannah Gladman, Rishub Jain, Yousuf A Khan, Caroline M R Low, Kuba Perlin, Anna Potapenko, Pascal Savy, Sukhdeep Singh, Adrian Stecula, Ashok Thillaisundaram, Catherine Tong, Sergei Yakneen, Ellen D Zhong, Michal Zielinski, Augustin Žídek, Victor Bapst, Pushmeet Kohli, Max Jaderberg, Demis Hassabis, and John M Jumper. Accurate structure prediction of biomolecular interactions with AlphaFold 3. *Nature*, 630(8016):493–500, June 2024.

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. 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*, pp. 929–947, New York, NY, USA, April 2024. ACM.

Ondřej Bojar, Christian Buck, Christian Federmann, Barry Haddow, Philipp Koehn, Johannes Leveling, Christof Monz, Pavel Pecina, Matt Post, Herve Saint-Amand, Radu Soricut, Lucia Specia, and Aleš Tamchyna. Findings of the 2014 workshop on statistical machine translation. In Ondřej Bojar, Christian Buck, Christian Federmann, Barry Haddow, Philipp Koehn, Christof Monz, Matt

Post, and Lucia Specia (eds.), *Proceedings of the Ninth Workshop on Statistical Machine Translation*, pp. 12–58, Baltimore, Maryland, USA, June 2014. Association for Computational Linguistics. doi: 10.3115/v1/W14-3302. URL https://aclanthology.org/W14-3302/.

Nicolas Carion, Francisco Massa, Gabriel Synnaeve, Nicolas Usunier, Alexander Kirillov, and Sergey Zagoruyko. End-to-end object detection with transformers. In Andrea Vedaldi, Horst Bischof, Thomas Brox, and Jan-Michael Frahm (eds.), *Computer Vision – ECCV 2020*, pp. 213–229. Springer International Publishing, Cham, 2020. ISBN 978-3-030-58452-8.

Benjamin Charlier, Jean Feydy, Joan Alexis Glaunès, François-David Collin, and Ghislain Durif. Kernel operations on the GPU, with autodiff, without memory overflows. *Journal of Machine Learning Research*, 22(74):1–6, 2021. URL http://jmlr.org/papers/v22/20-275.html.

Tri Dao. FlashAttention-2: Faster attention with better parallelism and work partitioning. In *The Twelfth International Conference on Learning Representations*, 2024. URL https://openreview.net/forum?id=mZn2Xyh9Ec.

Tri Dao, Dan Fu, Stefano Ermon, Atri Rudra, and Christopher Ré. FlashAttention: Fast and memory-efficient exact attention with IO-awareness. In S. Koyejo, S. Mohamed, A. Agarwal, D. Belgrave, K. Cho, and A. Oh (eds.), *Advances in Neural Information Processing Systems*, volume 35, pp. 16344–16359. Curran Associates, Inc., 2022. doi: 10.52202/068431-1189. URL https://proceedings.neurips.cc/paper_files/paper/2022/file/67d57c32e20fd0a7a302cb81d36e40d5-Paper-Conference.pdf.

Mostafa Dehghani, Basil Mustafa, Josip Djolonga, Jonathan Heek, Matthias Minderer, Mathilde Caron, Andreas Peter Steiner, Joan Puigcerver, Robert Geirhos, Ibrahim Alabdulmohsin, Avital Oliver, Piotr Padlewski, Alexey A. Gritsenko, Mario Lucic, and Neil Houlsby. Patch n' pack: NaViT, a vision transformer for any aspect ratio and resolution. In *Thirty-seventh Conference on Neural Information Processing Systems*, 2023. URL https://openreview.net/forum?id=VpGFHmI7e5.

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

Juechu Dong, Boyuan Feng, Driss Guessous, Yanbo Liang, and Horace He. FlexAttention: A programming model for generating fused attention variants. In M. Zaharia, G. Joshi, and Y. Lin (eds.), *Proceedings of Machine Learning and Systems*, volume 7. MLSys, 2025. URL https://proceedings.mlsys.org/paper_files/paper/2025/file/61a9278dfef5f871b5e472389f8d6fa1-Paper-Conference.pdf.

Alexey Dosovitskiy, Lucas Beyer, Alexander Kolesnikov, Dirk Weissenborn, Xiaohua Zhai, Thomas Unterthiner, Mostafa Dehghani, Matthias Minderer, Georg Heigold, Sylvain Gelly, Jakob Uszkoreit, and Neil Houlsby. An image is worth 16x16 words: Transformers for image recognition at scale. In *International Conference on Learning Representations*, 2021. URL https://openreview.net/forum?id=YicbFdNTTy.

FBGEMM contributors. Jagged tensor operators. FBGEMM documentation, 2026. URL https://docs.pytorch.org/FBGEMM/fbgemm_gpu/overview/jagged-tensor-ops/JaggedTensorOps.html. Accessed 2026-09-16.

Pratik Fegade, Tianqi Chen, Phillip Gibbons, and Todd Mowry. The CoRa tensor compiler: Compilation for ragged tensors with minimal padding. In D. Marculescu, Y. Chi, and C. Wu (eds.), *Proceedings of Machine Learning and Systems*, volume 4, pp. 721–747, 2022. URL https://proceedings.mlsys.org/paper_files/paper/2022/file/afe8a4577080504b8bec07bbe4b2b9cc-Paper.pdf.

Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, June 2016.

John Jumper, Richard Evans, Alexander Pritzel, Tim Green, Michael Figurnov, Olaf Ronneberger, Kathryn Tunyasuvunakool, Russ Bates, Augustin Žídek, Anna Potapenko, Alex Bridgland, Clemens Meyer, Simon A A Kohl, Andrew J Ballard, Andrew Cowie, Bernardino Romera-Paredes, Stanislav Nikolov, Rishub Jain, Jonas Adler, Trevor Back, Stig Petersen, David Reiman, Ellen Clancy, Michal Zielinski, Martin Steinegger, Michalina Pacholska, Tamas Berghammer, Sebastian Bodenstein, David Silver, Oriol Vinyals, Andrew W Senior, Koray Kavukcuoglu, Pushmeet Kohli, and Demis Hassabis. Highly accurate protein structure prediction with AlphaFold. *Nature*, 596(7873):583–589, August 2021.

Tom Kocmi and Ondřej Bojar. Curriculum learning and minibatch bucketing in neural machine translation. In Ruslan Mitkov and Galia Angelova (eds.), *Proceedings of the International Conference Recent Advances in Natural Language Processing, RANLP 2017*, pp. 379–386, Varna, Bulgaria, September 2017. INCOMA Ltd. doi: 10.26615/978-954-452-049-6_050. URL https://aclanthology.org/R17-1050/.

Mario Michael Krell, Matej Kosec, Sergio P. Perez, and Andrew Fitzgibbon. Efficient sequence packing without cross-contamination: Accelerating large language models without impacting performance, 2021. URL https://arxiv.org/abs/2107.02027. Revised version: arXiv:2107.02027v2, 2022.

Ruihang Lai, Junru Shao, Siyuan Feng, Steven Lyubomirsky, Bohan Hou, Wuwei Lin, Zihao Ye, Hongyi Jin, Yuchen Jin, Jiawei Liu, Lesheng Jin, Yaxing Cai, Ziheng Jiang, Yong Wu, Sunghyun Park, Prakalp Srivastava, Jared Roesch, Todd C. Mowry, and Tianqi Chen. Relax: Composable abstractions for end-to-end dynamic machine learning. In *Proceedings of the 30th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*, ASPLOS '25, pp. 998–1013, New York, NY, USA, 2025. Association for Computing Machinery. ISBN 9798400710797. doi: 10.1145/3676641.3716249. URL https://doi.org/10.1145/3676641.3716249.

Tsung-Yi Lin, Michael Maire, Serge Belongie, James Hays, Pietro Perona, Deva Ramanan, Piotr Dollár, and C. Lawrence Zitnick. Microsoft COCO: Common objects in context. In David Fleet, Tomas Pajdla, Bernt Schiele, and Tinne Tuytelaars (eds.), *Computer Vision – ECCV 2014*, pp. 740–755, Cham, 2014. Springer International Publishing. ISBN 978-3-319-10602-1.

Jonathan Long, Evan Shelhamer, and Trevor Darrell. Fully convolutional networks for semantic segmentation. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, June 2015.

Andrew L. Maas, Raymond E. Daly, Peter T. Pham, Dan Huang, Andrew Y. Ng, and Christopher Potts. Learning word vectors for sentiment analysis. In Dekang Lin, Yuji Matsumoto, and Rada Mihalcea (eds.), *Proceedings of the 49th Annual Meeting of the Association for Computational Linguistics: Human Language Technologies*, pp. 142–150, Portland, Oregon, USA, June 2011. Association for Computational Linguistics. URL https://aclanthology.org/P11-1015/.

Stephen Merity, Caiming Xiong, James Bradbury, and Richard Socher. Pointer sentinel mixture models, 2016. URL https://arxiv.org/abs/1609.07843.

Makoto Morishita, Yusuke Oda, Graham Neubig, Koichiro Yoshino, Katsuhito Sudoh, and Satoshi Nakamura. An empirical study of mini-batch creation strategies for neural machine translation. In Thang Luong, Alexandra Birch, Graham Neubig, and Andrew Finch (eds.), *Proceedings of the First Workshop on Neural Machine Translation*, pp. 61–68, Vancouver, August 2017. Association for Computational Linguistics. doi: 10.18653/v1/W17-3208. URL https://aclanthology.org/W17-3208/.

Jim Pivarski, Peter Elmer, and David Lange. Awkward arrays in Python, C++, and Numba. *EPJ Web of Conferences*, 245:05023, 2020.

PyTorch contributors. PyTorch benchmark. PyTorch Tutorials, 2026a. URL https://docs.pytorch.org/tutorials/recipes/recipes/benchmark.html. Accessed 2026-09-16.

PyTorch contributors. CUDA semantics. PyTorch main documentation, 2026b. URL https://docs.pytorch.org/docs/main/notes/cuda.html. Accessed 2026-09-16.

PyTorch contributors. torch.nested. PyTorch main documentation, 2026c. URL https://docs.pytorch.org/docs/main/nested.html. Accessed 2026-09-16.

Alec Radford, Jeffrey Wu, Rewon Child, David Luan, Dario Amodei, and Ilya Sutskever. Language models are unsupervised multitask learners. 2019.

Olga Russakovsky, Jia Deng, Hao Su, Jonathan Krause, Sanjeev Satheesh, Sean Ma, Zhiheng Huang, Andrej Karpathy, Aditya Khosla, Michael Bernstein, Alexander C Berg, and Li Fei-Fei. ImageNet large scale visual recognition challenge. *International Journal of Computer Vision*, 115 (3):211–252, December 2015.

Noam Shazeer. GLU variants improve transformer, 2020. URL https://arxiv.org/abs/2002.05202.

TensorFlow contributors. Ragged tensors. TensorFlow Core documentation, 2026. URL https://www.tensorflow.org/guide/ragged_tensor. Accessed 2026-09-16.

Hugo Touvron, Matthieu Cord, Matthijs Douze, Francisco Massa, Alexandre Sablayrolles, and Hervé Jégou. Training data-efficient image transformers & distillation through attention. In Marina Meila and Tong Zhang (eds.), *Proceedings of the 38th International Conference on Machine Learning*, volume 139 of *Proceedings of Machine Learning Research*, pp. 10347–10357. PMLR, 18–24 Jul 2021. URL https://proceedings.mlr.press/v139/touvron21a.html.

Iulia Turc, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. Well-read students learn better: On the importance of pre-training compact models, 2019. URL https://arxiv.org/abs/1908.08962.

Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. Attention is all you need. In I. Guyon, U. Von Luxburg, S. Bengio, H. Wallach, R. Fergus, S. Vishwanathan, and R. Garnett (eds.), *Advances in Neural Information Processing Systems*, volume 30. Curran Associates, Inc., 2017. URL https://proceedings.neurips.cc/paper/2017/file/3f5ee243547dee91fbd053c1c4a845aa-Paper.pdf.

Bolei Zhou, Hang Zhao, Xavier Puig, Tete Xiao, Sanja Fidler, Adela Barriuso, and Antonio Torralba. Semantic understanding of scenes through the ADE20K dataset. *International Journal of Computer Vision*, 127(3):302–321, March 2019.

A Representation, Semantics, and the Running Program

Structural invariants. Ragged axes in this work vary between individually dense elements of a common rank. The representation retains static feature axes, an ordered tuple of ragged axes, and the batch-position convention. For the batch-leading exposition, element axis d is public axis $d+1$. The implementation also supports its sequence-first convention; the batch axis is not an arbitrary permutable element dimension. Sample offsets satisfy $o_0 = 0$, $o_{i+1} - o_i = p_i$, and $o_B = T$. Zero-volume elements may have coincident consecutive offsets, while the exact size matrix distinguishes their remaining logical dimensions.

Sizes and offsets play different roles. For example, element shapes $(2, 6, C)$ and $(3, 4, C)$ both occupy twelve packed feature rows but cannot be aligned as the same logical pair tensor. Hierarchical row splits identify the groups consumed by reductions. For rectangular elements (N_i, M_i, C), the innermost splits contain $1 + \sum_i N_i$ entries; they need not be $O(B)$ metadata. Canonical sizes and offsets stay on the CPU, with device mirrors and execution indices used by GPU consumers.

Reference semantics. Let $\mathcal{D}$ denote the logical interpretation of a structured tensor; it acts componentwise on argument tuples and leaves dense tensors unchanged. A deterministic functional handler $\widehat{f}$ is specified against its reference f_{ref} by $\mathcal{D}(\widehat{f}(\mathbf{Z})) = f_{\mathrm{ref}}(\mathcal{D}(\mathbf{Z}))$, the contract cited in Section 3.2. Sample-local operations are compared with independent dense element computations using the same weights, masks, coordinate conventions, and loss weighting. Global reductions and operations combining samples require their corresponding batch reference. A mean over all stored entries weights entries equally. The supplied API's single batch-axis reduction instead performs a complete

reduction within each element and stacks the results: for elements $[1, 2]$ and $[3, 4]$, its `sum(dim=0)` gives $[3, 7]$. This convention is distinct from a dense column-wise batch reduction. The cross-entropy handler uses a trailing class dimension, so a reference must align the objective accordingly.

The validity mask uses true for valid positions, while attention interfaces can use different boolean conventions. The adapter translates those conventions and combines padding validity with model masks. Padded accessors are interoperability operations; the canonical numerical payload remains packed. That contract is a specification; derivative and structural agreement are tested and reported separately.

A.1 Measured single–pair–single program

The measured mechanism uses the projections, residual pre-normalised SwiGLU update, and column mean in Equation 1. Pair construction is the sum of the two projected single features; there is no additional activation at that construction step. Both returned outputs are consumed outside the compiled forward by

$$\mathcal{L} = \frac{\sum_i \alpha_i \|\widetilde{X}_i\|_F^2}{SC \sum_i N_i} + \frac{\sum_i \beta_i \|\widetilde{P}_i\|_F^2}{SC_p \sum_i N_i M_i}, \qquad \alpha_i, \beta_i > 0. \tag{5}$$

Here $M_i = N_i$ for square pairs, $S = 1$ for leading layouts, and $S = 3$ for the non-leading variant. The rectangular variant has distinct row and column inputs and divides its column mean by M_i. The sample weights are $\alpha_i = 1 + i/B$ and $\beta_i = 1.5 - i/(2B)$ for $i = 0, \ldots, B-1$. The sample weights and losses are identical across representations; the squared sums accumulate in at least FP32. This mechanism has no trainable readout after the returned pair state. The separate Pairformer application does include a trainable pair readout in its eager loss.

The mechanism uses $B = 8$, $C = 384$, $C_p = 128$, and transition expansion four. Eight validation patterns cover changed partitions with fixed totals, changed packed extents and maxima, and fresh equal structures. Each measured method/layout/mode cell has three traces of two steps. All eight patterns pass the recorded structure, output, gradient and one-update checks for all 18 combinations.

B Implementation and Compilation Scope

Execution paths. Operator availability, a packed implementation, and fullgraph compilation are distinct properties. The two registries hold 708 keys for 404 distinct operators, since one operator owns several keys, as `torch.add`, `Tensor.add`, `Tensor.__add__` and `aten::add.Tensor` do; operators are counted after dropping module paths and overload suffixes, and an in-place operator counts separately from its out-of-place form because it mutates packed storage through its own handler. Feature-wise operations reuse dense packed kernels. Ragged reductions and broadcasts use coordinate/group maps; selected attention paths use native variable-length or FlexAttention execution. Spatial paths gather tile neighbourhoods and invoke dense convolution, or use direct pooling kernels; selected pooling paths instead address packed pixels directly through per-sample shapes and offsets. Pointwise convolution reuses the dense packed path after aligning the channel axis with the static feature dimension. For output tile dimensions (t_h, t_w), stride s, dilation δ and kernel size k, the gathered input tile has dimensions $u_h = (t_h - 1)s_h + \delta_h(k_h - 1) + 1$ and $u_w = (t_w - 1)s_w + \delta_w(k_w - 1) + 1$. Tile size therefore balances numerical batching against overlapping halo reads, boundary tiles and gather/scatter work; chunking bounds the number of simultaneous tile buffers, the dense backend may use channels-last storage, and backward can regenerate input tiles from the saved packed input for weight gradients. Some bindings, including segmented `cdist` and `cumprod`, retain per-sample native kernel invocations behind a custom-operation boundary. Eager fallback and unsupported compiled argument combinations remain part of the implementation's declared scope; a workload counts as passing only on its own measured outcome, never on operator availability.

Dynamic metadata and differentiation. The principal length-dynamic contract fixes batch size and the structural schema at each program point. The flattening protocol exposes one differentiable payload child alongside nondifferentiable structural metadata. Tensor-backed sizes and row splits are runtime inputs; standalone FakeTensor construction retains the Python metadata needed where symbolic data-dependent extents are unavailable. Eager reconstructions propagate weak dynamic

annotations before entering a compiled region. This prevents a ragged extent from being accidentally identified with an unrelated metadata dimension of the same observed size. The optimised eager propagation recognises the state installed by the public marker and preserves a public-API path for missing marker metadata, tensor subclasses, and FakeTensor execution.

Wrapper-to-packed and packed-to-wrapper autograd bridges preserve the corresponding numerical dependency and packed dimension order, so changing partitions reuse the compiled graph for the evaluated program and gradients remain connected across the compiled/eager boundary.

Current compiled application outcomes. The ordinary-operation mechanism completes in all three layouts. The dataset matrix records upstream native-jagged BERT guard errors, failed padded vision compilation, and NestedTensor FCN fake-propagation failures. Under the default Inductor configuration the full Pairformer compiles for DanLing but not for either baseline (Section 4.3). The recorded failures are identical across the four square regimes: padded raises `InductorError` with `CantSplit: 128*s83 not divisible by ((s83**2)//s83)`, and explicit packing raises `TorchRuntimeError` during fake-tensor propagation. The padded failure has no user frame: Inductor's scheduler raises it while fusing nodes whose iteration extents are s^2 and $128s$, a constraint that holds for every positive integer s but that it cannot establish. Padded therefore compiles under static shapes, and under dynamic shapes once fusion is disabled (`max_fusion_size=1`; `epilogue_fusion=False` alone still fails), running at 522–556 ms/step in one harness whose padded eager reproduces Table 2. Explicit packing fails before any fusion decision and was not retested. Of DanLing's four compiled regimes, two recompile once inside the measured trace, unlike the sequence workloads, where the compiled schema is reused throughout. For FCN, padded ResNet18/101 reach the recompilation limit, ResNet152 exhausts the recorded two-hour budget without a completed outcome, and the standalone ResNet50 padded compiled cell has no result artifact. The four NestedTensor FCN compiled cells fail during convolution fake propagation.

Steady-state execution versus in-trace recompilation. A measured step whose shape was not already covered by a compiled graph triggers a new frontend/backend capture; eager execution has no such event, so the distinction only matters for compiled runs. Figure 1's compiled-mode speedup pools only steps that did not themselves trigger a capture, isolating kernel and packing efficiency from compiler behaviour. Every steady-state ratio pools the steps on which neither method compiled, so numerator and denominator average identical batches. That intersection retains all paired steps except in six compiled cells: 63 of 64 for WikiText Small, Medium and Large, 124 of 128 for WMT14-Base, 125 of 128 for WMT14-Big, and four of six for WikiText-XL, whose short trace makes it the only cell where the restriction is material (it moves the ratio by 4.1%; elsewhere the shift is at most 0.48%). Table 3 reports the complement for the compiled cells whose full-trace and steady-state ratios diverge most (Section 4.2): how many graphs each method compiled over its lifetime, including calibration, and how many of those compilations were shape-driven recompilations occurring after calibration, alongside the resulting steady-state and full-trace per-step times. In these recorded traces the padded implementations recompile more often than DanLing and native jagged, which reuse one compiled schema in five of the six cells; WikiText-XL is the exception, where its six-step trace leaves every method recompiling once or twice.

C Benchmark Protocol and Complete Dataset Matrix

Environment and source cohorts. The PyTorch build is `2.13.0+cu132`, commit `cf30153c4c131c8164ee7798e5022d810682e2cb`, with CUDA 13.2 and Triton 3.7.1 on A100-SXM4-80GB GPUs. Parameters use BF16; TF32 is disabled, float32 matmul precision is `highest`, and stochastic layers are disabled for these comparisons. The validation optimiser is SGD at 10^{-5}, with BF16 parameters and no separate FP32 master-weight copy.

The reported tables and figures are not all measurements of a single checkout: the dataset and execution-control timings, the mechanism snapshot, and the square Pairformer timings were each frozen at a different source revision. Every exported table and figure is tied to frozen manifests recording the exact source revision and input-file hashes, distributed with the artifact. The Pairformer cells additionally depend on the OOps kernel library, whose public release postdates these measurements; the artifact therefore records that dependency as a per-file source snapshot rather than a citable revision.

Table 3: **Dynamic-compilation evidence.** "Graphs" counts lifetime frontend/backend invocations, including calibration; "recompile" counts only those occurring after calibration, during the measured trace. "Compile step" is the mean measured time of a recompiling step, execution and compilation combined; it is not an isolated compiler stopwatch. Steady-state time pools non-recompiling steps; full-trace time pools every measured step.

Workload	Method	Graphs	Recompiles	Compile step (ms)	Steady (ms)	Full trace (ms)
WMT14 / Base	Padded	8	4	49567	43.7	1591.3
	Native jagged	2	0	–	28.9	28.9
	DanLing	2	0	–	27.4	27.4
WMT14 / Big	Padded	6	3	56811	87.8	1417.3
	Native jagged	2	0	–	38.7	38.7
	DanLing	2	0	–	39.1	39.1
WikiText / Small	Padded	3	1	28563	200.7	643.9
	Native jagged	2	0	–	138.4	138.4
	DanLing	2	0	–	138.3	138.3
WikiText / Medium	Padded	3	1	54697	504.1	1350.9
	Native jagged	2	0	–	353.5	353.5
	DanLing	2	0	–	362.1	362.1
WikiText / Large	Padded	4	1	106792	997.4	2650.4
	Native jagged	2	0	–	701.8	701.8
	DanLing	2	0	–	709.9	709.9
WikiText / XL	Padded	5	2	128865	454.2	43257.6
	Native jagged	4	1	96345	444.8	16428.1
	DanLing	4	1	153093	444.6	25886.1

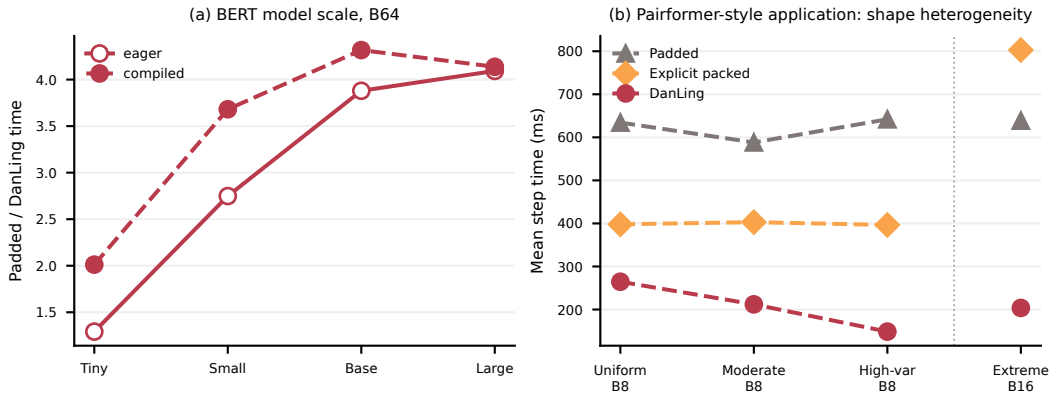

Figure 5: **Model-scale and shape-heterogeneity sensitivity. (a)** BERT model-scale speedup (padded / DanLing mean step time) at $B = 64$, eager (hollow) and compiled (solid). **(b)** Eager mean step time for the same four-block Pairformer-style workload as Table 2, for padded, explicit-packed and DanLing across four length-heterogeneity regimes at fixed model size, from nearly uniform lengths to one long chain among many short ones; the rightmost regime additionally doubles the batch to $B = 16$. Every point compares methods within the same execution mode.

The image and pair occupancy fractions accompanying ρ_{seq} in Section 4.4 are $\rho_{\text{image}} = \sum_i H_i W_i / B H_{\max} W_{\max}$ and $\rho_{\text{pair}} = \sum_i N_i M_i / B N_{\max} M_{\max}$. Table 4 lists the lengths and ρ_{pair} behind Tables 1 and 2.

Table 4: **Length regimes of the pair workloads.** Lengths span every measured sample (rows $\times$ columns for rectangular pairs); ρ_{pair} pools valid and padded pair cells over the measured batches. A cell's eager and compiled runs, and the matched-granularity run of a square regime (Table 7), use the same length schedule.

Workload	Shape regime	Lengths	ρ_{pair}
Single–pair–single	Square ($B = 8$)	66–317	27.2%
Single–pair–single	Rectangular ($B = 8$)	66–307 $\times$ 66–309	29.6%
Single–pair–single	Non-leading ($B = 8$)	40–182	28.0%
Pairformer-style	Square, near-uniform ($B = 8$)	208–224	94.4%
Pairformer-style	Square, moderate ($B = 8$)	137–216	71.6%
Pairformer-style	Square, high variation ($B = 8$)	88–224	27.9%
Pairformer-style	Square, extreme variation ($B = 16$)	44–160	14.8%
Pairformer-style	Non-leading ($B = 8$)	90–130	68.5%
Pairformer-style	Rectangular ($B = 8$)	130–221 $\times$ 148–224	66.2%

Architecture and batch sizes. IMDB uses BERT Tiny/Small configurations from Turc et al. (2019) and Base/Large configurations from Devlin et al. (2019), with (L, H, C) equal to $(2, 2, 128)$, $(4, 8, 512)$, $(12, 12, 768)$, and $(24, 16, 1024)$, all at $B = 64$. WikiText uses GPT-2 Small/Medium/Large/XL shapes with (L, H, C) equal to $(12, 12, 768)$, $(24, 16, 1024)$, $(36, 20, 1280)$, and $(48, 25, 1600)$. Its main document-length cap is 4096, with $B = 8$ except XL at $B = 2$. The WMT Base/Big settings use six encoder and six decoder layers, widths 512/1024, and 8/16 heads at $B = 64$. ViT Ti/S use the DeiT-Ti/S scale configurations of Touvron et al. (2021), while B/L follow Dosovitskiy et al. (2021); all use patch size 16, widths 192/384/768/1024, depths 12/12/12/24, and 3/6/12/16 heads at $B = 32$. DETR uses ResNet50, six encoder and six decoder layers, hidden width 256, eight heads, and 100 queries at $B = 32$. FCN uses ResNet18/50/101/152 and 150 classes at $B = 32$.

Measurement accounting. Timing follows PyTorch's benchmarking and CUDA synchronisation guidance (PyTorch contributors, 2026a;b). Every cell fixes one execution mode. Batch identities and sizes come from the frozen sibling manifest, not placeholder configuration lengths or a mutable external manifest. Completed traces are checked for matching IDs, repetition IDs and step counts. Latency sums all measured steps; throughput divides all measured examples by that elapsed time. Memory takes the maximum allocated or reserved bytes across all measured traces. No trace is removed for containing compilation. Validation and calibration timings are outside the compute measurement, and their cache priming means that a timed capture is not a measurement of all compilation costs from a pristine process. The primary tables record one repetition and descriptive step variation. Inference about independent-run uncertainty or maximum feasible batch size is outside these records.

Execution outcomes. Where a cell has no completed measurement, ND marks no run on record, BL marks a padded reference that itself failed validation (blocking the comparison), G marks a dynamic-shape guard error during compiled dispatch, CF marks any other compilation failure, NR marks a run that ended without a completed outcome, and NE marks a method not evaluated because the workload falls outside its structural support (e.g. native jagged on COCO/ADE20K); unfinished measurements keep NR rather than a diagnosed cause. The complete grids below report each configuration as two rows: mean throughput in samples/s, then peak allocated / reserved memory in GiB. Reserved memory additionally counts blocks the caching allocator holds for reuse, and favours packing less than allocated memory: for GPT-2 Small to Large, DanLing and native jagged reserve more than padding.

C.1 IMDB / BERT

Size	Mode		Padded	Native jagged	DanLing
tiny	eager	samples/s	4635.7	1651.7	5990.2
		GiB alloc/res.	0.67 / 0.82	0.29 / 0.63	0.29 / 0.63
tiny	compiled	samples/s	6625.6	G	13330.0
		GiB alloc/res.	0.61 / 0.64	–	0.24 / 0.43
small	eager	samples/s	932.3	996.2	2562.6
		GiB alloc/res.	4.50 / 4.66	1.76 / 4.05	1.76 / 4.05
small	compiled	samples/s	1036.1	G	3813.8
		GiB alloc/res.	4.40 / 4.48	–	1.67 / 3.91
base	eager	samples/s	197.1	442.0	764.8
		GiB alloc/res.	18.85 / 19.06	7.17 / 16.23	7.17 / 16.23
base	compiled	samples/s	208.9	G	901.7
		GiB alloc/res.	18.75 / 18.86	–	7.07 / 16.12
large	eager	samples/s	67.1	231.0	274.7
		GiB alloc/res.	49.50 / 49.65	18.72 / 42.45	18.72 / 42.55
large	compiled	samples/s	69.8	G	289.0
		GiB alloc/res.	49.39 / 49.54	–	18.61 / 42.34

C.2 WIKITEXT-103 / GPT-2

Size	Mode		Padded	Native jagged	DanLing
small	eager	samples/s	37.4	46.4	54.4
		GiB alloc/res.	22.44 / 22.45	19.08 / 39.21	19.08 / 39.21
small	compiled	samples/s	12.4	57.8	57.8
		GiB alloc/res.	22.43 / 26.21	19.07 / 39.20	19.07 / 39.20
medium	eager	samples/s	15.2	21.0	21.9
		GiB alloc/res.	37.93 / 37.95	32.29 / 53.00	32.29 / 53.00
medium	compiled	samples/s	5.9	22.6	22.1
		GiB alloc/res.	37.93 / 41.73	32.29 / 52.37	32.29 / 52.37
large	eager	samples/s	7.7	11.0	11.2
		GiB alloc/res.	59.80 / 63.54	50.97 / 75.59	50.97 / 75.59
large	compiled	samples/s	3.0	11.4	11.3
		GiB alloc/res.	59.80 / 59.88	50.97 / 76.00	50.97 / 76.00
xl	eager	samples/s	4.6	3.5	4.6
		GiB alloc/res.	25.31 / 35.16	25.31 / 30.87	25.31 / 30.87
xl	compiled	samples/s	0.046	0.12	0.077
		GiB alloc/res.	25.31 / 35.25	25.31 / 30.95	25.31 / 30.95

C.3 WMT14 / TRANSFORMER

Size	Mode		Padded	Native jagged	DanLing
base	eager	samples/s	1322.0	335.4	884.2
		GiB alloc/res.	12.71 / 39.89	2.81 / 4.88	2.81 / 4.88
base	compiled	samples/s	40.2	2214.8	2339.9
		GiB alloc/res.	12.70 / 39.96	2.81 / 7.29	2.81 / 7.29
big	eager	samples/s	702.7	335.6	875.1
		GiB alloc/res.	16.92 / 42.41	3.94 / 8.39	3.94 / 8.39
big	compiled	samples/s	45.2	1652.2	1636.4
		GiB alloc/res.	16.89 / 42.44	3.95 / 8.37	3.95 / 8.32

C.4 IMAGENET / VIT

Size	Mode		Padded	Native jagged	DanLing
ti	eager	samples/s	498.8	202.1	351.3
		GiB alloc/res.	1.46 / 2.01	0.97 / 1.21	0.97 / 1.23
ti	compiled	samples/s	BL	BL	BL
		GiB alloc/res.	–	–	–
s	eager	samples/s	490.1	202.7	343.4
		GiB alloc/res.	2.82 / 3.02	1.90 / 3.11	1.90 / 3.11
s	compiled	samples/s	BL	BL	BL
		GiB alloc/res.	–	–	–
base	eager	samples/s	341.6	199.6	343.7
		GiB alloc/res.	5.65 / 6.06	3.84 / 6.17	3.84 / 6.17
base	compiled	samples/s	BL	BL	BL
		GiB alloc/res.	–	–	–
l	eager	samples/s	135.4	110.5	184.2
		GiB alloc/res.	14.49 / 15.14	10.04 / 15.88	10.04 / 15.88
l	compiled	samples/s	BL	BL	BL
		GiB alloc/res.	–	–	–

C.5 COCO / DETR

Size	Mode		Padded	Native jagged	DanLing
official	eager	samples/s	78.1	NE	81.8
		GiB alloc/res.	7.79 / 14.43	–	1.83 / 4.02
official	compiled	samples/s	BL	NE	BL
		GiB alloc/res.	–	–	–

C.6 ADE20K / FCN-RESNET

Size	Mode		Padded	Native jagged	DanLing
18	eager	samples/s	135.2	NE	297.8
		GiB alloc/res.	9.89 / 50.46	–	7.06 / 23.40
18	compiled	samples/s	CF	NE	CF
		GiB alloc/res.	–	–	–
50	eager	samples/s	63.7	NE	132.4
		GiB alloc/res.	28.38 / 63.50	–	12.25 / 26.61
50	compiled	samples/s	ND	NE	CF
		GiB alloc/res.	–	–	–
101	eager	samples/s	43.4	NE	81.2
		GiB alloc/res.	46.07 / 77.27	–	17.21 / 31.90
101	compiled	samples/s	CF	NE	CF
		GiB alloc/res.	–	–	–
152	eager	samples/s	33.8	NE	59.8
		GiB alloc/res.	61.67 / 77.51	–	21.60 / 37.92
152	compiled	samples/s	NR	NE	CF
		GiB alloc/res.	–	–	–

C.7 CONVENTIONAL SPATIAL PADDING

The conventional FCN workflow applies interpolation over the padded envelope, whereas the matched reference applies it at the valid image extent. Intermediate convolution and pooling boundaries also differ. Frozen BatchNorm eliminates one source of coupling but does not identify every source of the observed differences. DETR comparisons include predictions, matching, loss and gradients. WMT has no corresponding spatial-boundary comparator.

Table 5: **Matched and conventional spatial padding.** Cells are full-trace mean ms/step / peak allocated GiB. The conventional workflow changes boundary and interpolation semantics, so it is a different computation rather than a faster version of the same one.

Model	Matched padded	Conventional padded	DanLing
FCN-R50	504.1 / 28.38	364.8 / 21.34	243.8 / 12.25
DETR-R50	403.8 / 7.79	313.2 / 4.21	387.3 / 1.83

C.8 Execution and granularity controls

Table 6: **Five-way execution controls.** Cells are full-trace samples/s. Random and bucketed policies are distinct frozen batch partitions of the same measured example cohort.

Dataset	Batching	Mode	Padded	Attention-only	Explicit packed	DanLing	Native jagged
IMDB	random	eager	196.6	288.9	843.0	774.6	448.8
IMDB	random	compiled	208.9	316.1	899.4	901.1	G
IMDB	bucket	eager	706.3	644.3	824.9	710.0	431.2
IMDB	bucket	compiled	729.9	683.0	851.7	850.5	G
WMT14	random	eager	1317.4	1139.4	1694.9	898.8	341.8
WMT14	random	compiled	39.3	136.0	2369.3	2404.4	2352.3
WMT14	bucket	eager	1497.6	1216.6	1666.5	882.8	311.9
WMT14	bucket	compiled	30.0	60.7	137.4	143.7	184.9

Attention-only packing includes Q/K/V unpadding and output repadding. Explicit packing and DanLing keep surrounding feature computations packed. The attention-only, explicit-packed and DanLing controls request native variable-length attention; padded attention runs masked scaled dot-product attention over the padded envelope, and native jagged follows its own SDPA dispatch. Compiled IMDB reaches similar DanLing and explicit-packed throughput in both policies. The IMDB random-policy cells repeat the BERT-Base configuration of Figure 1 in a separate run, so their ratios differ slightly from that figure's. WMT combines a larger eager implementation gap with timed compilation outliers in several compiled methods. Method-level kernel and metadata differences remain part of these comparisons.

The matched-granularity control supports two further comparisons on the Pairformer workload: DanLing with per-sample calls against explicit packing with per-sample calls isolates the interface, and DanLing with segmented calls against DanLing with per-sample calls isolates the execution strategy. Table 7 reports the three step times for each square regime.

Table 7: **Matched-granularity control on the four-block Pairformer-style workload, eager execution.** Full-trace mean ms/step. Explicit packing and DanLing per-sample call each OOps triangle operator once per sample; DanLing segmented calls it once per batch. Interface cost is the per-sample DanLing slowdown relative to explicit packing; granularity gain is per-sample over segmented DanLing time. The three methods share one run per regime, separate from Table 2, so both ratios come from that run and its times differ slightly from that table's.

Shape regime	Explicit packed	DanLing per-sample	DanLing segmented	Interface cost	Granularity gain
Square, near-uniform ($B = 8$)	404.8	406.8	265.4	0.49%	1.53×
Square, moderate ($B = 8$)	396.4	399.9	210.3	0.86%	1.90×
Square, high variation ($B = 8$)	391.7	398.0	148.3	1.60%	2.68×
Square, extreme variation ($B = 16$)	801.3	804.8	204.1	0.44%	3.94×