TRIMSO(2): FAST AND MEMORY-EFFICIENT TRAINING OF EQUIVARIANT INTERATOMIC POTENTIALS

Anonymous authors
Paper under double-blind review

ABSTRACT

Equivariant interatomic potentials enable accurate and efficient simulations of molecules and materials. Among them, models based on SO(2) convolutions reduce the cost of tensor products and achieve state-of-the-art accuracy, but their training remains time-consuming and memory-intensive. Existing training implementations materialize large edge tensors for representation transformations surrounding dense matrix multiplications, increasing memory usage and data movement. We observe that these transformations can be integrated into the model's existing dataflow, avoiding separate intermediate materialization. We introduce TrimSO(2), a fast and memory-efficient algorithm for SO(2)-based equivariant models. It realizes this principle through rotation on gather, recombination with activation, and inverse rotation on reduction, with support for the higher-order differentiation required for training. By avoiding intermediate materialization, TrimSO(2) delivers 2.71–16.73× operator speedups across forward, backward, and double backward and reduces model-level high-bandwidth memory traffic by 3.81–9.88×. Finally, TrimSO(2) achieves 2.07–2.94× training speedups and 1.53–3.16× reductions in peak training memory over PyTorch implementations for official eSEN, UMA, and EquiformerV3 workloads on A100 and H100 GPUs.

1 INTRODUCTION

Machine learning interatomic potentials learn atomic interactions from quantum mechanical data, enabling accurate and efficient molecular and materials simulations (Unke et al., 2021). To improve accuracy, leading models exploit the rotational symmetry of atomic systems: energies are invariant to rotations, while forces transform equivariantly (Batzner et al., 2022). Equivariant neural networks extend this symmetry to their internal representations (Thomas et al., 2018). Their features include scalars, vectors, and higher-order geometric tensors that carry directional information between layers while obeying the corresponding rotation laws (Geiger & Smidt, 2022). Models using these representations have demonstrated high data efficiency and state-of-the-art accuracy on molecular and materials benchmarks (Batzner et al., 2022; Batatia et al., 2022; Liao et al., 2024).

Mainstream equivariant architectures combine SO(3) representations through tensor products, whose computational cost can scale as $O(L^6)$ with the maximum representation degree L (Passaro & Zitnick, 2023). This computational cost makes training expensive, motivating progress in algorithm design as well as systems optimization. On the algorithmic side, eSCN reduces this cost to $O(L^3)$ by reformulating SO(3) convolutions as SO(2) convolutions in local edge frames (Passaro & Zitnick, 2023). This formulation enables models to scale to higher-degree representations and further improve accuracy (Liao et al., 2024). On the systems side, OpenEquivariance and FlashTP accelerate SO(3) tensor products through optimized GPU implementations, achieving model-level speedups (Bharadwaj et al., 2025; Lee et al., 2025). However, systems optimizations tailored to the training of SO(2)-based models remain limited, leaving the practical benefits of their lower algorithmic complexity only partially realized.

In SO(2) models, tensor products are conducted in local edge frames, replacing sparse SO(3) contractions with efficient dense matrix multiplications. This reformulation introduces three representation transformations around the dense matrix products: node features must be rotated into local edge frames, matrix-product outputs must be recombined through additions and subtractions, and edge messages must be rotated back to the global frame (Passaro & Zitnick, 2023). Existing training

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.

implementations materialize large intermediate edge tensors when performing these transformations, requiring repeated writes and reads that increase memory usage and data movement. The higher-order derivatives required for training machine learning interatomic potentials (Fu et al., 2025; Lee et al., 2025; Bharadwaj et al., 2025) further extend the lifetimes of intermediate values retained for differentiation, increasing storage demands.

We observe that these representation transformations can be performed as part of the model's existing dataflow, avoiding separate writes and reads of their intermediate results. We introduce TrimSO(2), a fast and memory-efficient algorithm for SO(2)-based equivariant models. This design extends across energy evaluation, force differentiation, and loss backpropagation, improving training efficiency while preserving the model's original computational formulation. Across eSEN (Fu et al., 2025), UMA (Wood et al., 2026), and EquiformerV3 (Liao et al., 2026) on A100 and H100 GPUs, TrimSO(2) achieves 3.81–9.88× reductions in HBM traffic, 2.07–2.94× training speedups, and 1.53–3.16× reductions in peak training memory over official PyTorch implementations.

TrimSO(2) integrates these transformations into the model's training dataflow through three corresponding strategies:

1. **Rotation on gather.** For each edge-feature tile, we gather the required node features and rotate them directly into the matrix-product input layout, avoiding a separate tensor of gathered edge features.
2. **Recombination with activation.** For each edge-feature tile, we load the matrix-product outputs, perform recombination, and immediately apply the activation, avoiding a separate recombined edge tensor.
3. **Inverse rotation on reduction.** For each node-output tile, we inverse-rotate incoming edge messages and accumulate their contributions on chip before writing the result, avoiding a separate tensor of global-frame edge messages.

2 Background

Equivariant representations. Equivariant interatomic potentials encode directional information in atomic features that transform under three-dimensional rotations (Thomas et al., 2018; Batzner et al., 2022). Features are organized by degree ℓ, with $2\ell + 1$ angular components $m = -\ell, \ldots, \ell$ per channel. A Wigner rotation matrix $D^{(\ell)}$ mixes these components within each degree and acts identically on all channels. With C channels and maximum representation degree L, a node feature is $x_i \in \mathbb{R}^{d \times C}$, where $d = \sum_{\ell=0}^{L}(2\ell + 1) = (L + 1)^2$. These rotation rules allow features to be expressed in different coordinate frames while describing the same atomic environment.

SO(2) convolutions. SO(2) convolutions reduce the tensor-product computation by aligning a local coordinate frame with each edge (Passaro & Zitnick, 2023). Rotations about the aligned axis form SO(2), under which components with matching angular orders can be mixed across degrees and channels. This permits dense matrix products for each order. In a real-valued basis, the positive and negative components of a nonzero order are paired. For each $m > 0$, let T_+ and T_- collect the components of orders $+m$ and $-m$, with one edge per row and degree and channel indices in the columns. Applying the two learned weight maps $W_{m,0}$ and $W_{m,1}$ to both inputs produces four output blocks:

$$\begin{pmatrix} Q_{00} & Q_{01} \\ Q_{10} & Q_{11} \end{pmatrix} = \begin{pmatrix} T_+ \\ T_- \end{pmatrix} \begin{bmatrix} W_{m,0} & W_{m,1} \end{bmatrix}, \qquad u_+ = Q_{00} - Q_{11}, \quad u_- = Q_{10} + Q_{01}. \tag{1}$$

We refer to the additions and subtractions that form the paired-order features u_+ and u_- from these product blocks as *recombination*. These features feed the activation after the first SO(2) linear layer and inverse rotation after the second.

To use these local computations in an atomic graph, the model gathers neighboring node features and rotates them into each edge frame. For an edge from atom j to atom i, let D_{ij} denote this *rotation*. Message generation and aggregation take the form

$$\tilde{x}_{ij} = D_{ij} x_j, \qquad m_{ij} = D_{ij}^{-1} \Phi_\theta(\tilde{x}_{ij}), \qquad x'_i = \sum_{j \in \mathcal{N}(i)} m_{ij}, \tag{2}$$

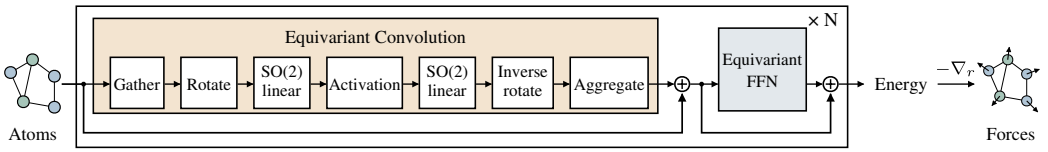


Figure 1: General architecture of an SO(2)-based interatomic potential. The model predicts energy and obtains forces by differentiating it with respect to atomic positions.

where Φ_θ applies two SO(2) linear layers with an intervening activation and $\mathcal{N}(i)$ contains the neighbors of atom i. The *inverse rotation* D_{ij}^{-1} transforms the messages back to a common global frame before they are summed at the receiving atom. Models may retain only orders $|m| \leq M$ locally, setting omitted components to zero before inverse rotation. As shown in Figure 1, SO(2)-based potentials adopt a Transformer-like architecture with stacked blocks, each combining an SO(2) convolution, an equivariant feed-forward network, and residual connections (Fu et al., 2025; Wood et al., 2026; Liao et al., 2026). See Appendix A for a detailed description.

Training of interatomic potentials. The model predicts energy $E_\theta(\mathbf{r}, Z)$ from atomic positions $\mathbf{r}$ and atom types Z. Forces follow from $F = -\nabla_{\mathbf{r}} E_\theta$, enforcing the conservative relation between energy and force required by the modeled atomic systems (Fu et al., 2025). Training on force labels, however, requires differentiating through this energy gradient with respect to the model parameters. For an energy–force objective $\mathcal{L} = \lambda_E \mathcal{L}_E + \lambda_F \mathcal{L}_F$, the parameter gradient includes a mixed second-derivative term:

$$\nabla_\theta \mathcal{L} = \lambda_E \nabla_\theta \mathcal{L}_E - \lambda_F \left[\frac{\partial}{\partial \theta} \left(\nabla_{\mathbf{r}} E_\theta \right) \right]^{\mathsf{T}} \nabla_F \mathcal{L}_F. \tag{3}$$

Optimizing training therefore requires accounting for all three stages: energy evaluation, force differentiation, and loss backpropagation.

3 Method

TrimSO(2) reduces storage and data movement by integrating the representation transformations of SO(2) convolutions into the model's existing dataflow. We preserve the dense matrix products and extend this principle through the backward and double-backward computations required for training. The full implementation also applies the same principle to node-side equivariant feed-forward networks.

3.1 Workload analysis

SO(2) convolutions expose dense matrix products that can use optimized GPU routines. For each positive order m, Equation (1) multiplies a $2E \times k_m$ input by a $k_m \times 2h_m$ weight matrix, where E is the number of directed edges and k_m, h_m are the input and output widths across degrees and channels. Batching edges into rows enables reuse of weights across edges and input features across output channels.

However, the representation transformations introduced by SO(2) convolutions incur storage and data movement overhead. In the upper half of Figure 2, gathering, rotation, recombination, and inverse rotation are separate steps connected by writes to and reads from HBM. For N atoms with C-channel node features, gathering expands NdC node values into EdC edge values. The rotated and recombined arrays likewise scale with the number of edges. Efficient matrix products also require four layout conversions between grouping by degree and grouping by order around the two SO(2) linear layers. Each conversion reads and rewrites a complete feature array. Training can further extend the lifetimes of intermediate values retained for differentiation.

Gathering provides the node features needed for rotation, activation consumes recombined product outputs, and reduction sums inverse-rotated messages. These relationships allow each transforma-

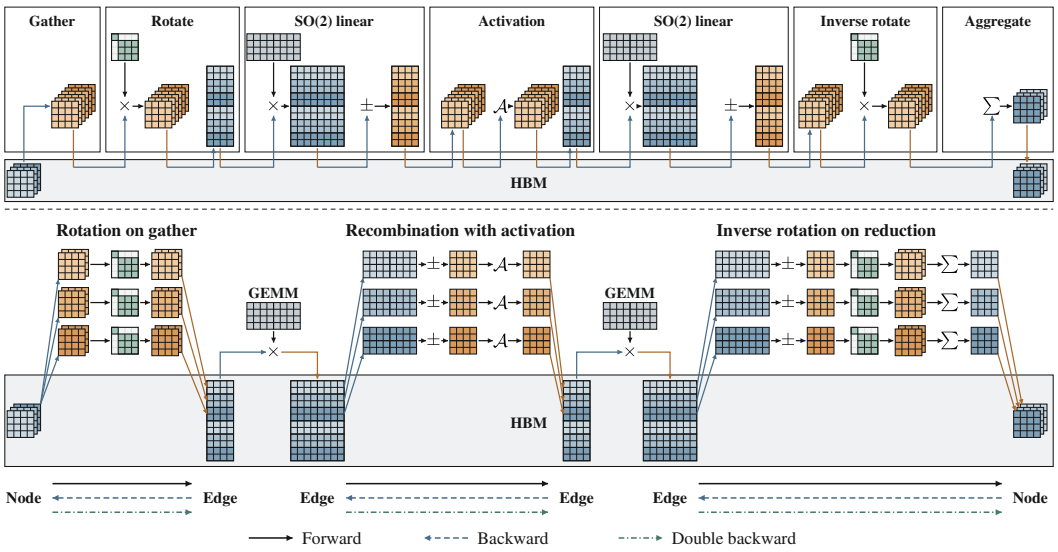


Figure 2: TrimSO(2) avoids intermediate HBM accesses by integrating representation transformations into existing dataflow. Top: separate operations materialize edge arrays. Bottom: rotation on gather, recombination with activation, and inverse rotation on reduction keep temporary values on chip. Blue and ochre transfer arrows indicate HBM reads and writes. The bottom arrows show the forward, backward, and double-backward feature dataflows.

tion to be combined with the operation that supplies or consumes its features, avoiding separate intermediate arrays while preserving the efficient dense matrix products.

3.2 TRIMSO(2)

TrimSO(2) combines these pairs of operations into three strategies, shown in the lower half of Figure 2. They follow Node $\rightarrow$ Edge, Edge $\rightarrow$ Edge, and Edge $\rightarrow$ Node dataflows. Two GEMMs connect them through input and output matrices in HBM. Algorithm 1 gives the full forward computation.

Rotation on gather. The first GEMM requires node features in local edge frames. We combine gathering and rotation to construct this input directly, giving the Node $\rightarrow$ Edge dataflow in Figure 2 (lower left). Each edge-feature tile loads features from atoms j and contracts them with the required rows of D_{ij} on chip, covering all angular components needed by the contraction. It writes the rotated coefficients directly into $T^{(m)}$ in the GEMM input layout, eliminating the gathered edge array and absorbing the conversion from grouping by degree to grouping by order. Compact degree blocks hold only the rotation rows for retained orders $|m| \leq M$.

Recombination with activation. Activation consumes the paired-order features obtained by recombining the first GEMM's outputs. We perform recombination as part of activation, so these features are consumed before they leave the chip. This gives the Edge $\rightarrow$ Edge dataflow in Figure 2 (lower center). Each tile loads matching entries from the four product blocks, forms $Q_{00} - Q_{11}$ and $Q_{10} + Q_{01}$, and applies the activation $\mathcal{A}$ locally. The tile includes components needed by reductions in $\mathcal{A}$. For example, a sigmoid gate scales the recombined features. Writing the activation output directly in the next GEMM's input layout eliminates the recombined array and absorbs the surrounding layout conversions while preserving the order of recombination and activation.

Inverse rotation on reduction. Aggregation sums inverse-rotated messages at each receiving atom. We combine inverse rotation with this reduction, accumulating each message immediately after it is transformed back to the global frame. This gives the Edge $\rightarrow$ Node dataflow in Figure 2 (lower right). We group incoming edges by receiver i and assign each node-output tile an on-chip accumulator. For each neighbor j, the tile recombines the second GEMM's product entries into a local message t_{ij} and

Algorithm 1 Forward SO(2) convolution with TrimSO(2)

Notation: $\mathcal{B}$ indexes an output tile and its required inputs. Loads and stores access HBM. Rotations act per edge and degree.

Rotation on gather: $(x, D) \mapsto T$
for all edge–feature tiles $\mathcal{B}$ in parallel **do**
  $X, R \leftarrow \mathrm{load}(x[\mathrm{src}(\mathcal{B})]), \mathrm{load}(D[\mathcal{B}])$
  $U \leftarrow RX$
  $\mathrm{store}(T[\mathcal{B}], U)$ ▶ *GEMM input layout*
end for
$Q \leftarrow \mathrm{GEMM}(T, W)$ ▶ *first SO(2) linear layer*
Recombination with activation: $(Q, \mathcal{A}) \mapsto z$
for all edge–feature tiles $\mathcal{B}$ in parallel **do**
  $q \leftarrow \mathrm{load}(Q[\mathcal{B}])$
  $Z \leftarrow \mathcal{A}(\mathrm{recombine}(q))$
  $\mathrm{store}(z[\mathcal{B}], Z)$ ▶ *next GEMM input layout*
end for
$Q \leftarrow \mathrm{GEMM}(z, V)$ ▶ *second SO(2) linear layer*
Inverse rotation on reduction: $(Q, D, \mathcal{N}) \mapsto x'$
for all node-output tiles $(i, \mathcal{B})$ in parallel **do**
  $a \leftarrow 0$
  for $j \in \mathcal{N}(i)$ **do**
    $q, R \leftarrow \mathrm{load}(Q_{ij}[\mathcal{B}]), \mathrm{load}(D_{ij}[\mathcal{B}])$
    $a \leftarrow a + R^{\mathsf{T}}\,\mathrm{recombine}(q)$
  end for
  $\mathrm{store}(x'_i[\mathcal{B}], a)$
end for

accumulates $D_{ij}^{\mathsf{T}} t_{ij}$ using the compact rotation blocks. After all incoming contributions have been accumulated, it writes the node output once. This removes separate local and global message arrays and absorbs the layout conversion before inverse rotation.

3.3 Differentiation for training

The feature gradients in backward and double backward follow the same three dataflow patterns. Backward reverses Node $\to$ Edge to Edge $\to$ Node and Edge $\to$ Node to Node $\to$ Edge, while Edge $\to$ Edge remains local to each edge, as shown by the arrows below HBM in Figure 2. Writing $\bar{v} = \partial E_\theta / \partial v$ for an energy derivative and treating omitted local components as zero, the feature gradients of rotation on gather and inverse rotation on reduction are

$$\bar{x}_j = \sum_{i:\, j \in \mathcal{N}(i)} D_{ij}^{\mathsf{T}} \bar{\tilde{x}}_{ij}, \qquad \bar{t}_{ij} = D_{ij} \bar{x}'_i \quad (j \in \mathcal{N}(i)). \tag{4}$$

The first expression accumulates rotated edge contributions into node gradients. The second gathers a node gradient and rotates it for each edge. They therefore follow the same patterns as inverse rotation on reduction and rotation on gather, respectively. Recombination differentiates locally, distributing $(\bar{u}_+, \bar{u}_-)$ to the four product blocks with the required signs.

Double backward differentiates these backward computations. Let p_j and q_{ij} be the incoming loss gradients with respect to $\bar{x}_j$ and $\bar{t}_{ij}$, respectively. Along these two feature paths, the loss gradients propagate as

$$\frac{\partial \mathcal{L}}{\partial \bar{\tilde{x}}_{ij}} = D_{ij} p_j, \qquad \frac{\partial \mathcal{L}}{\partial \bar{x}'_i} = \sum_{j \in \mathcal{N}(i)} D_{ij}^{\mathsf{T}} q_{ij}. \tag{5}$$

The first expression returns to Node $\to$ Edge by gathering and rotating node quantities. The second returns to Edge $\to$ Node by inverse-rotating and accumulating edge contributions. These directions match the double-backward arrows in Figure 2. Edge $\to$ Edge derivatives continue to act locally. Rotation and parameter derivatives are computed alongside these feature paths, with rotation derivatives propagated to atomic positions.

To evaluate these derivatives, we retain node features, compact rotation blocks, and the required layer inputs, outputs, and incoming gradients. Omitted values are reconstructed from this state within each tile when needed. For example, computing rotation derivatives gathers the saved node features again. This local reconstruction avoids retaining the intermediate edge arrays across training stages. Appendix B gives detailed algorithms and derivative formulas for the three dataflows.

3.4 Memory and I/O analysis

For the forward schedule in Figure 2, assume all orders are retained ($M = L$) and all feature widths equal C. Table 1 counts the eliminated intermediate arrays, including the four layout conversions. Each array contains EdC elements.

Table 1: Eliminated intermediate storage and logical I/O. Each array incurs one write and one read at b bytes per element. Shared GEMM inputs and outputs are excluded.

	HBM intermediates (elements)		Avoided I/O
Dataflow	Separate operations	TrimSO(2)	(bytes)
Node $\rightarrow$ Edge	$2EdC$	0	$4bEdC$
Edge $\rightarrow$ Edge	$3EdC$	0	$6bEdC$
Edge $\rightarrow$ Node	$3EdC$	0	$6bEdC$

Removing one write and one read per array avoids

$$B_{\text{eliminated}} = 2b(2+3+3)EdC = 16bEdC \tag{6}$$

of logical feature accesses per convolution. GEMM inputs and outputs remain in HBM, while temporary feature storage scales as $O(BdC)$ for a tile of B edges, independently of E.

During differentiation, retained node features occupy NdC elements and allow EdC gathered values to be reconstructed locally, an E/N-fold difference in representation size. Peak-memory savings depend on the lifetimes of all retained arrays, and physical HBM traffic also depends on cache reuse. Appendix B.5 details the counts and extends them to unequal widths and retained-order layouts.

4 Experiments

We evaluate TrimSO(2) in terms of training efficiency and numerical accuracy. We first measure its benefits for training speed, peak memory, and HBM traffic at the model level (Section 4.2). To understand these gains, we examine the contributions of the dataflow optimizations through cumulative ablations (Section 4.3) and analyze operator performance across computation stages and configurations (Section 4.4). Finally, we assess numerical accuracy by comparing outputs, gradients, and training trajectories with the official PyTorch implementations (Section 4.5). Appendix C provides detailed experimental settings and additional results.

4.1 Experimental setup

We compare TrimSO(2) with official PyTorch implementations of eSEN (Fu et al., 2025), UMA (Wood et al., 2026), and EquiformerV3 (Liao et al., 2026) on OMat24 (Barros-Luque et al., 2026), following their official architectures and training settings. Each pair shares random initialization, fixed hyperparameters, and batch sequence. The loss includes energy, forces, and stresses, with forces and stresses obtained by differentiation. We use one A100 80GB or H100 80GB GPU in FP32. Three independent runs each measure 100 post-warmup steps, reporting mean step time and the largest peak allocated memory. Separate Nsight Compute runs measure HBM traffic over energy and force/stress evaluation and loss backpropagation on a fixed batch. Timing runs are uninstrumented. Speedup and reduction factors divide PyTorch measurements by optimized measurements.

4.2 End-to-end performance

To test whether avoiding intermediate arrays benefits complete model training, we measure HBM traffic, step time, and peak memory under matched training settings. Figure 3 compares TrimSO(2)

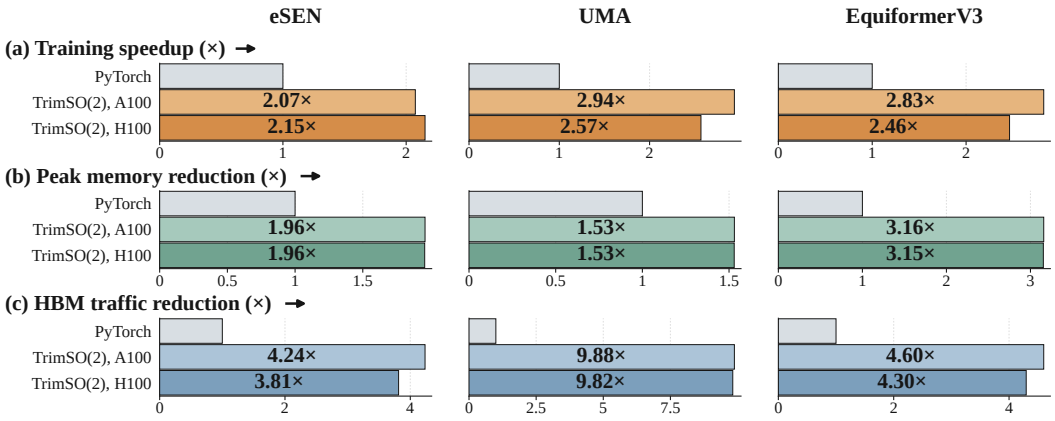


Figure 3: TrimSO(2) reduces HBM traffic, accelerates training, and lowers peak memory across all three models on A100 and H100.

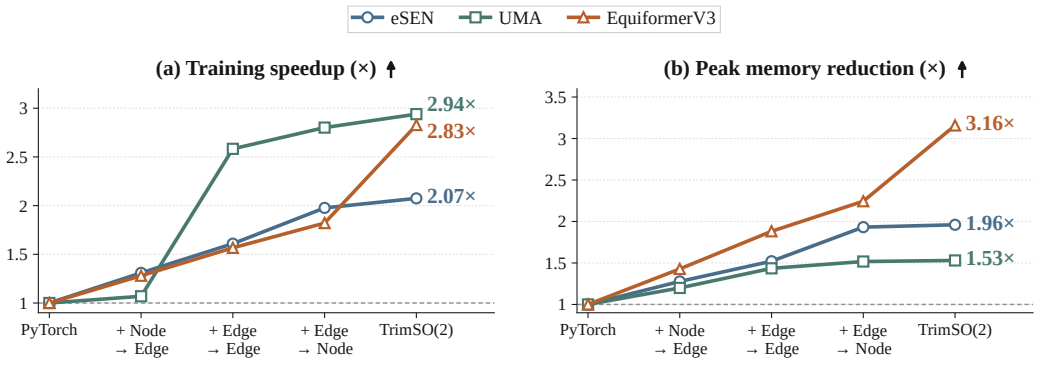


Figure 4: Cumulative benefits of TrimSO(2) optimizations on A100. Optimizations are added sequentially, showing their contributions to training speedup and peak memory reduction.

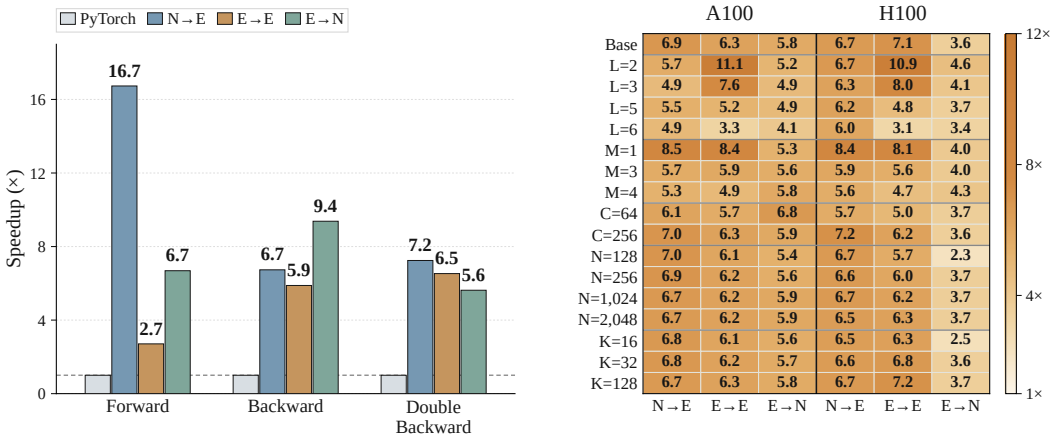
with the official implementations, labeled PyTorch. TrimSO(2) achieves 3.81–9.88× reductions in HBM traffic across all three models on both GPUs. Complete training steps, including data processing and parameter updates, are 2.07–2.94× faster, with 1.53–3.16× reductions in peak allocated memory at unchanged batch sizes. Optimizing the dataflow around dense SO(2) products thus improves both training speed and memory efficiency without changing the model architectures.

4.3 Ablation Study

We examine how training speed and memory savings accumulate as the dataflow optimizations are enabled. Starting from PyTorch, we add Node → Edge, Edge → Edge, and Edge → Node in sequence (Figure 4). These stages correspond to the dataflows in Section 3.2. We also extend the design to equivariant feed-forward networks to avoid intermediate materialization within each node. We denote these optimizations by Node → Node (Appendix B.4) and add them last to obtain full TrimSO(2). Figure 4 shows that each successive optimization improves training speed and reduces peak memory across all three models.

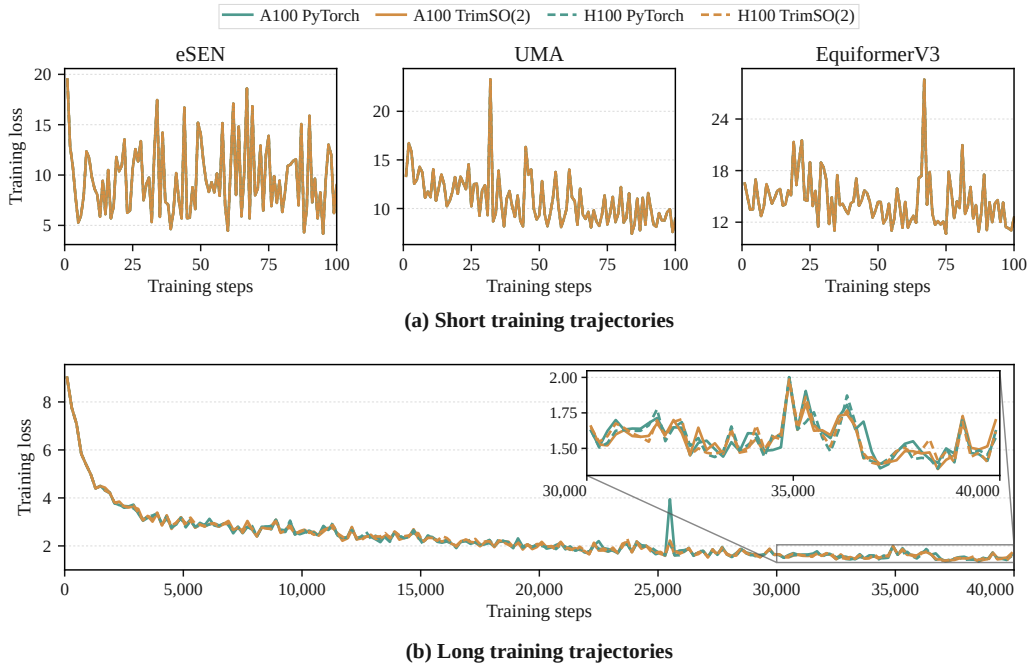
4.4 Operator Performance

We analyze the three dataflows in Section 3.2 across forward, backward, and double backward. Inputs are captured from an EquiformerV3 training batch with $L/M/C = 4/2/128$, 399 nodes, and 17,940 directed edges. The local PyTorch reference follows the official implementation's operation order and layouts. Both paths share inputs and differentiation directions. We average 30 CUDA-event



(a) Operator performance **(b) Configuration scaling**

Figure 5: All three operators run faster than PyTorch across the evaluated stages and configurations. (a) Stage-wise speedups on A100 using captured training inputs. (b) Double-backward speedups on A100 and H100 with synthetic inputs, varying L, M, C, N, or K.



(a) Short training trajectories

(b) Long training trajectories

Figure 6: TrimSO(2) and PyTorch follow closely aligned training loss trajectories on both GPUs. (a) Unsmoothed 100-step trajectories for all three models. (b) eSEN over 40,000 steps, shown as non-overlapping 200-step means, with the final 10,000 steps enlarged.

measurements per stage after warmup, excluding prerequisite computations. Figure 5(a) shows 2.71–16.73× forward speedups, 5.88–9.38× backward speedups, and 5.62–7.24× double-backward speedups. All three operators benefit across the three computation stages.

To assess the effectiveness of our optimizations across a broader range of configurations, we then sweep configurations for double backward, the most time-consuming stage in the measured operator workloads. Using synthetic inputs with the same computation structure, we vary L, M, C, node count N, and outgoing edges per node K individually. Figure 5(b) shows 2.34–11.06× speedups, demonstrating benefits across angular resolutions, channel widths, and graph workloads.

4.5 NUMERICAL ACCURACY

We compare outputs and gradients with PyTorch to assess numerical accuracy. With shared inputs, relative L_2 errors in operator outputs and first and second derivatives are at most 2.77×10^{-7} against the PyTorch reference. We then compare the three models at a shared random initialization. Relative L_2 errors in energy, force, and stress predictions are at most 1.47×10^{-6}, and parameter-gradient errors are at most 8.36×10^{-7}.

We next examine how these differences affect training. Loss trajectories remain closely aligned on both GPUs over 100 steps for all three models and 40,000 steps for eSEN (Figure 6), with one long run per implementation on each GPU. Over 40,000 steps, TrimSO(2) retains $2.06\times$ and $1.95\times$ training speedups on A100 and H100, respectively, and a $1.97\times$ reduction in peak memory on both GPUs.

5 RELATED WORK

Equivariant interatomic potentials. Equivariant interatomic potentials describe directional interactions through geometric message passing and many-body coupling (Batzner et al., 2022; Batatia et al., 2022), as well as equivariant attention (Thölke & Fabritiis, 2022; Frank et al., 2022; Liao & Smidt, 2023). Local-frame representations (Zitnick et al., 2022) lead to efficient SO(2) convolutions (Passaro & Zitnick, 2023) and higher-degree equivariant attention (Liao et al., 2024). Recent SO(2)-based models develop this foundation for reliable physical-property prediction (Fu et al., 2025), broad chemical coverage (Wood et al., 2026), and expressive, efficient attention (Liao et al., 2026; Li et al., 2026a). TrimSO(2) targets efficient training of these SO(2)-based potentials by optimizing their shared local-frame computations.

Accelerating equivariant models. Accelerating equivariant models has motivated improvements in tensor-product algorithms and GPU implementations. Alternative tensor-product formulations (Luo et al., 2024; Xie et al., 2025; Lin et al., 2026) and model architectures (Qu & Krishnapriyan, 2024; Li et al., 2026b) reduce computational costs. Specialized GPU kernels deliver model-level speedups through sparsity-aware tensor-product computation and fusion (Bharadwaj et al., 2025; Lee et al., 2025). Streaming strategies further reduce intermediate materialization in equivariant attention (Huang et al., 2026) and SO(3) tensor-product convolutions (Chorošajev & Bény, 2026). FAIRChem also provides Triton kernels for UMA inference.[1] TrimSO(2) targets training of existing SO(2)-based models, optimizing forward, backward, and double backward while preserving model architectures and mathematical operations.

Memory-efficient training. Memory-efficient training can avoid storing large intermediate tensors by computing them in tiles and reconstructing values needed for backpropagation, as demonstrated by fused attention algorithms (Dao et al., 2022; Dao, 2024; Shah et al., 2024). Related strategies reduce memory traffic in FFT convolutions (Fu et al., 2024) and avoid full intermediate arrays in pairwise reductions (Charlier et al., 2021) and cross-entropy computation (Wijmans et al., 2025). For graph neural networks, dataflow optimization combines message generation and aggregation to reduce redundant computation and storage (Xie et al., 2022). TrimSO(2) applies this principle to SO(2) representation transformations, reducing intermediate storage across energy evaluation, force differentiation, and loss backpropagation.

6 CONCLUSION

TrimSO(2) avoids intermediate materialization by integrating SO(2) representation transformations into the model's dataflow. Across eSEN, UMA, and EquiformerV3 on A100 and H100, it achieves 2.07–$2.94\times$ end-to-end training speedups and 1.53–$3.16\times$ reductions in peak training memory. Outputs, gradients, and training trajectories remain close to the official PyTorch implementations. These results demonstrate that TrimSO(2) improves training efficiency while preserving the mathematical operations of SO(2)-based equivariant models.

[1] https://github.com/facebookresearch/fairchem

AI USE STATEMENT

We used generative AI tools to assist with drafting and revising the manuscript, including improving paragraph organization, clarity of presentation, and language. The authors take responsibility for the final manuscript and its scientific claims.

REPRODUCIBILITY STATEMENT

Section 3 describes the dataflow transformations and their differentiation. Appendix B provides detailed algorithms, derivative identities, and memory and I/O calculations. Appendix C documents the model configurations, training data and settings, software versions, hardware, and measurement protocols, together with detailed performance results and numerical comparisons.

REFERENCES

Luis Barros-Luque, Muhammed Shuaibi, Xiang Fu, Brandon M Wood, Misko Dzamba, Meng Gao, Ammar Rizvi, Matt Uyttendaele, C Lawrence Zitnick, and Zachary W Ulissi. The Open Materials 2024 (OMat24) inorganic materials dataset and models. *Nature Computational Science*, pp. 1–11, 2026.

Ilyes Batatia, David P Kovacs, Gregor Simm, Christoph Ortner, and Gábor Csányi. MACE: Higher order equivariant message passing neural networks for fast and accurate force fields. *Advances in neural information processing systems*, 35:11423–11436, 2022.

Simon Batzner, Albert Musaelian, Lixin Sun, Mario Geiger, Jonathan P Mailoa, Mordechai Kornbluth, Nicola Molinari, Tess E Smidt, and Boris Kozinsky. E (3)-equivariant graph neural networks for data-efficient and accurate interatomic potentials. *Nature communications*, 13(1):2453, 2022.

Vivek Bharadwaj, Austin Glover, Aydın Buluç, and James Demmel. An efficient sparse kernel generator for O (3)-equivariant deep networks. In *2025 Proceedings of the Conference on Applied and Computational Discrete Algorithms (ACDA)*, pp. 32–46. SIAM, 2025.

Benjamin Charlier, Jean Feydy, Joan Alexis Glaunes, 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.

Vladimir Chorošajev and Cédric Bény. Sobek: Streaming Equivariant Tensor Product Convolutions. *arXiv preprint arXiv:2607.18074*, 2026.

Tri Dao. Flashattention-2: Faster attention with better parallelism and work partitioning. In *International Conference on Learning Representations*, volume 2024, pp. 35549–35562, 2024.

Tri Dao, Dan Fu, Stefano Ermon, Atri Rudra, and Christopher Ré. Flashattention: Fast and memory-efficient exact attention with io-awareness. *Advances in neural information processing systems*, 35: 16344–16359, 2022.

Thorben Frank, Oliver Unke, and Klaus-Robert Müller. So3krates: Equivariant attention for interactions on arbitrary length-scales in molecular systems. *Advances in Neural Information Processing Systems*, 35:29400–29413, 2022.

Dan Fu, Hermann Kumbong, Eric Nguyen, and Christopher Ré. Flashfftconv: Efficient convolutions for long sequences with tensor cores. In *International Conference on Learning Representations*, volume 2024, pp. 9455–9483, 2024.

Xiang Fu, Brandon M Wood, Luis Barroso-Luque, Daniel S. Levine, Meng Gao, Misko Dzamba, and C. Lawrence Zitnick. Learning Smooth and Expressive Interatomic Potentials for Physical Property Prediction. In *Forty-second International Conference on Machine Learning*, 2025. URL https://openreview.net/forum?id=R0PBjxIbgm.

Mario Geiger and Tess Smidt. e3nn: Euclidean neural networks. *arXiv preprint arXiv:2207.09453*, 2022.

Lin Huang, Chengxiang Huang, Ziang Wang, Yiyue Du, Chu Wang, Haocheng Lu, Yunyang Li, Xiaoli LIU, Arthur JIANG, and Jia Zhang. E2Former-V2: On-the-Fly Equivariant Attention with Linear Activation Memory. In *Forty-third International Conference on Machine Learning*, 2026. URL https://openreview.net/forum?id=gIUVdEAa3l.

Seung Yul Lee, Hojoon Kim, Yutack Park, Dawoon Jeong, Seungwu Han, Yeonhong Park, and Jae W Lee. FlashTP: Fused, Sparsity-Aware Tensor Product for Machine Learning Interatomic Potentials. In *ICML*, 2025.

Tiancheng Li, Wentao Li, Anyang Peng, Jianming Xue, Linfeng Zhang, Duo Zhang, and Han Wang. DPA4: Pushing the Accuracy-Cost Frontier of Interatomic Potentials with EMFA SO (2) Convolution. *arXiv preprint arXiv:2606.02419*, 2026a.

Yunyang Li, Lin Huang, Zhihao Ding, Xinran Wei, Chu Wang, Han Yang, Zun Wang, Chang Liu, Yu Shi, Peiran Jin, et al. E2former: An efficient and equivariant transformer with linear-scaling tensor products. *Advances in Neural Information Processing Systems*, 38:23593–23630, 2026b.

Yi-Lun Liao and Tess Smidt. Equiformer: Equivariant Graph Attention Transformer for 3D Atomistic Graphs. In *The Eleventh International Conference on Learning Representations*, 2023. URL https://openreview.net/forum?id=KwmPfARgOTD.

Yi-Lun Liao, Brandon Wood, Abhishek Das, and Tess Smidt. Equiformerv2: Improved equivariant transformer for scaling to higher-degree representations. In *International Conference on Learning Representations*, volume 2024, pp. 39282–39309, 2024.

Yi-Lun Liao, Alexander J Hoffman, Sabrina C Shen, Alexandre Duval, Sam Walton Norwood, and Tess Smidt. EquiformerV3: Scaling efficient, expressive, and general SE (3)-equivariant graph attention transformers. *arXiv preprint arXiv:2604.09130*, 2026.

Yuchao Lin, Cong Fu, Zachary Krueger, Haiyang Yu, Maho Nakata, Jianwen Xie, Emine Kucukbenli, Xiaofeng Qian, and Shuiwang Ji. Tensor Decomposition Networks for Fast Machine Learning Interatomic Potential Computations. *Advances in Neural Information Processing Systems*, 38: 88471–88495, 2026.

Shengjie Luo, Tianlang Chen, and Aditi Krishnapriyan. Enabling efficient equivariant operations in the fourier basis via gaunt tensor products. In *International Conference on Learning Representations*, volume 2024, pp. 24742–24777, 2024.

Saro Passaro and C Lawrence Zitnick. Reducing SO (3) convolutions to SO (2) for efficient equivariant GNNs. In *International conference on machine learning*, pp. 27420–27438. PMLR, 2023.

Eric Qu and Aditi S Krishnapriyan. The importance of being scalable: Improving the speed and accuracy of neural network interatomic potentials across chemical domains. *Advances in Neural Information Processing Systems*, 37:139030–139053, 2024.

Jay Shah, Ganesh Bikshandi, Ying Zhang, Vijay Thakkar, Pradeep Ramani, and Tri Dao. Flashattention-3: Fast and accurate attention with asynchrony and low-precision. *Advances in Neural Information Processing Systems*, 37:68658–68685, 2024.

Chuin Wei Tan, Marc L Descoteaux, Mit Kotak, Gabriel de Miranda Nascimento, Seán R Kavanagh, Laura Zichi, Menghang Wang, Aadit Saluja, Yizhong R Hu, Tess Smidt, et al. High-performance training and inference for deep equivariant interatomic potentials. *Digital Discovery*, 5(4):1558–1567, 2026.

Philipp Thölke and Gianni De Fabritiis. Equivariant Transformers for Neural Network based Molecular Potentials. In *International Conference on Learning Representations*, 2022. URL https://openreview.net/forum?id=zNHzqZ9wrRB.

Nathaniel Thomas, Tess Smidt, Steven Kearnes, Lusann Yang, Li Li, Kai Kohlhoff, and Patrick Riley. Tensor field networks: Rotation-and translation-equivariant neural networks for 3d point clouds. *arXiv preprint arXiv:1802.08219*, 2018.

Oliver T Unke, Stefan Chmiela, Huziel E Sauceda, Michael Gastegger, Igor Poltavsky, Kristof T Schutt, Alexandre Tkatchenko, and Klaus-Robert Muller. Machine learning force fields. *Chemical reviews*, 121(16):10142–10186, 2021.

Erik Wijmans, Brody Huval, Alexander Hertzberg, Vladlen Koltun, and Philipp Krähenbühl. Cut your losses in large-vocabulary language models. In *International Conference on Learning Representations*, volume 2025, pp. 68174–68193, 2025.

Brandon Wood, Misko Dzamba, Xiang Fu, Meng Gao, Muhammed Shuaibi, Luis Barroso-Luque, Kareem Abdelmaqsoud, Vahe Gharakhanyan, John Kitchin, Daniel Levine, et al. UMA: A family of universal models for atoms. *Advances in Neural Information Processing Systems*, 38: 129391–129427, 2026.

YuQing Xie, Ameya Daigavane, Mit Kotak, and Tess Smidt. The Price of Freedom: Exploring Expressivity and Runtime Tradeoffs in Equivariant Tensor Products. In *Forty-second International Conference on Machine Learning*, 2025. URL https://openreview.net/forum?id=EvIwwGYTLc.

Zhiqiang Xie, Minjie Wang, Zihao Ye, Zheng Zhang, and Rui Fan. Graphiler: Optimizing graph neural networks with message passing data flow graph. *Proceedings of Machine Learning and Systems*, 4:515–528, 2022.

Larry Zitnick, Abhishek Das, Adeesh Kolluru, Janice Lan, Muhammed Shuaibi, Anuroop Sriram, Zachary Ulissi, and Brandon Wood. Spherical channels for modeling atomic interactions. *Advances in Neural Information Processing Systems*, 35:8054–8067, 2022.

A SO(2)-BASED EQUIVARIANT MODELS

A.1 EQUIVARIANT REPRESENTATIONS

Equivariant interatomic potentials encode the directional structure of an atomic environment in features that transform with spatial rotations. Preserving this transformation rule through the network allows the model to process directional information while producing a rotation-invariant energy (Thomas et al., 2018; Batzner et al., 2022).

SO(3) representations organize these features by degree ℓ. Degree zero describes scalars, degree one describes vectors, and higher degrees describe higher-order angular patterns. Each degree has $2\ell + 1$ angular components, indexed by $m = -\ell, \ldots, \ell$. With C channels carrying independent copies of this representation, atom i has a feature block $x_i^{(\ell)} \in \mathbb{R}^{(2\ell+1)\times C}$. A spatial rotation acts as

$$x_i^{(\ell)} \longmapsto D^{(\ell)} x_i^{(\ell)}, \qquad D^{(\ell)} \in \mathbb{R}^{(2\ell+1)\times(2\ell+1)}. \tag{7}$$

The Wigner rotation matrix $D^{(\ell)}$ mixes angular components within the degree and acts identically on every channel. Collecting degrees $0, \ldots, L$ gives the full node feature $x_i \in \mathbb{R}^{d\times C}$ and its block-diagonal rotation matrix,

$$D = \operatorname{diag}\Big(D^{(0)}, \ldots, D^{(L)}\Big) \in \mathbb{R}^{d\times d}, \qquad d = \sum_{\ell=0}^{L}(2\ell+1) = (L+1)^2. \tag{8}$$

Each node therefore carries dC feature values. In the real spherical harmonic basis, the rotation blocks are orthogonal, so $D^{-1} = D^{\mathsf{T}}$ transforms the features back to their original frame.

The model updates these features by passing messages along an atomic graph. Let N and E denote the numbers of atoms and directed edges, and let $\mathcal{N}(i)$ contain the neighbors that send messages to atom i. For an edge from atom j to atom i, the relative geometry is

$$\mathbf{r}_{ij} = \mathbf{r}_j - \mathbf{r}_i + \mathbf{\Delta}_{ij}, \qquad \rho_{ij} = \|\mathbf{r}_{ij}\|, \qquad \widehat{\mathbf{r}}_{ij} = \mathbf{r}_{ij}/\rho_{ij}, \tag{9}$$

where $\mathbf{\Delta}_{ij}$ selects a periodic image and is zero for a nonperiodic edge. The distance ρ_{ij} is rotation-invariant and can determine scalar interaction weights. The direction $\widehat{\mathbf{r}}_{ij}$ carries angular information. Real spherical harmonics $Y_{\ell m}(\widehat{\mathbf{r}}_{ij})$ express this direction in components that transform under the same $D^{(\ell)}$ as the atomic features, allowing the two to be coupled equivariantly.

A.2 SO(2) CONVOLUTIONS

SO(2) convolutions perform this directional coupling in local edge frames. The reduction follows from the structure of an SO(3) tensor product, which combines a neighbor's features with directional harmonics to produce a message. For one input and output channel,

$$m_{ij,m_o}^{(\ell_o)} = \sum_{\ell_i,\ell_f} \sum_{m_i,m_f} \mathcal{C}_{(\ell_i,m_i),(\ell_f,m_f)}^{(\ell_o,m_o)} w_{ij}^{(\ell_i,\ell_f,\ell_o)} x_{j,m_i}^{(\ell_i)} Y_{\ell_f m_f}(\widehat{\mathbf{r}}_{ij}). \tag{10}$$

The indices ℓ_i, ℓ_f, and ℓ_o denote input, directional-filter, and output degrees. Clebsch–Gordan coefficients $\mathcal{C}$ determine how the angular components combine, while learned weights w_{ij} control each degree coupling and may depend on distance and atom types. The general tensor product can have an $O(L^6)$ computational cost as the maximum degree increases (Passaro & Zitnick, 2023).

A local frame simplifies the directional harmonics without changing the interaction. Equivariance ensures that rotating both tensor-product inputs rotates the output by the same spatial rotation. The model can therefore compute the message in an aligned frame and transform it back afterward. Let $D_{ij}^{(\ell)}$ rotate features into a frame where the edge direction lies on the polar axis $\widehat{\mathbf{e}}$. In this frame,

$$\tilde{x}_{ij}^{(\ell)} = D_{ij}^{(\ell)} x_j^{(\ell)}, \qquad Y_{\ell_f m_f}(\widehat{\mathbf{e}}) = \begin{cases} \alpha_{\ell_f}, & m_f = 0, \\ 0, & m_f \neq 0, \end{cases} \tag{11}$$

where α_{ℓ_f} is the order-zero harmonic evaluated at the polar axis. Only the $m_f = 0$ term remains in the directional filter, so the local message becomes

$$\tilde{m}_{ij,m_o}^{(\ell_o)} = \sum_{\ell_i,\ell_f} \sum_{m_i} \mathcal{C}_{(\ell_i,m_i),(\ell_f,0)}^{(\ell_o,m_o)} w_{ij}^{(\ell_i,\ell_f,\ell_o)} \alpha_{\ell_f} \tilde{x}_{ij,m_i}^{(\ell_i)}. \tag{12}$$

With $m_f = 0$, the real Clebsch–Gordan coefficients allow only $m_i = \pm m_o$. Positive and negative orders therefore form pairs that do not mix with other orders. After combining the filter-degree contributions, each nonzero-order pair can be transformed using two real weight maps (Passaro & Zitnick, 2023). The paired transformation applies one map to both diagonal components and the other with opposite signs to the cross components. This is the SO(2) structure induced by the edge frame: rotations about its axis mix each order pair in the same way across degrees, permitting learned transformations across degrees and channels within that order.

To express these transformations as matrix products, the model groups features by order. Each matrix has one row per edge and columns indexed by degree and channel. For input and output channel widths C_{in} and C_{out}, its widths are

$$k_m = (L - |m| + 1)C_{\text{in}}, \qquad h_m = (L - |m| + 1)C_{\text{out}}, \tag{13}$$

because order m occurs at degrees $|m|, \ldots, L$. Let $T^{(m)} \in \mathbb{R}^{E \times k_m}$ contain the inputs for order m. For features from atom j, the matrix entries are

$$T^{(m)}_{ij,(\ell,c)} = \sum_{m'=-\ell}^{\ell} D^{(\ell)}_{ij,mm'} x^{(\ell)}_{j,m',c}, \qquad |m| \le \ell. \tag{14}$$

Here ij identifies the edge from j to i. A layer using both endpoints also constructs the corresponding features from x_i and includes both atoms' features in its input channels.

For a positive order, write $T_+ = T^{(m)}$ and $T_- = T^{(-m)}$. The first SO(2) linear layer packs its two weight maps into $W_m = [W_{m,0}\ W_{m,1}] \in \mathbb{R}^{k_m \times 2h_m}$. A matrix product produces four $E \times h_m$ blocks, whose additions and subtractions form the output pair:

$$\begin{pmatrix} Q_{00} & Q_{01} \\ Q_{10} & Q_{11} \end{pmatrix} = \begin{pmatrix} T_+ \\ T_- \end{pmatrix} W_m, \qquad u_+ = Q_{00} - Q_{11}, \qquad u_- = Q_{10} + Q_{01}. \tag{15}$$

This *recombination* produces the features passed to the intervening activation. The second SO(2) linear layer uses weights $V_m = [V_{m,0}\ V_{m,1}]$ in the same form, with its own input and output widths. Its recombined outputs become the local messages for inverse rotation. For fixed channel widths, the degree-pair count across nonnegative orders is $\sum_{m=0}^{L}(L - m + 1)^2 = O(L^3)$, giving the lower complexity of the SO(2) formulation.

The model may retain only local orders $|m| \le M$, where $0 \le M \le L$. Keeping or omitting both signs together preserves the pair's transformation under rotations about the edge axis. At channel width C, the retained local feature has $Cq_{L,M}$ values, with

$$q_{L,M} = \sum_{\ell=0}^{L} \big(2\min(\ell, M) + 1\big) = (L + 1)(2M + 1) - M(M + 1). \tag{16}$$

Let Φ_θ denote the two SO(2) linear layers and the intervening activation. Their output is expressed in an edge-specific frame. Before aggregation, the model inserts zeros in omitted orders and applies inverse rotation to return every message to a common global frame:

$$m_{ij} = D_{ij}^{-1} \Phi_\theta(\tilde{x}_{ij}), \qquad x'_i = \sum_{j \in \mathcal{N}(i)} m_{ij}. \tag{17}$$

Since $D_{ij}^{-1} = D_{ij}^{\mathsf{T}}$, the inverse uses the transpose of the forward rotation matrix. The sum then updates atom i with information from its neighbors.

SO(2)-based potentials stack these convolutions in a Transformer-like architecture (Figure 1) (Fu et al., 2025; Wood et al., 2026; Liao et al., 2026). Each block combines an SO(2) convolution with an equivariant feed-forward network and residual connections. The convolution exchanges information between atoms, while the feed-forward network transforms each atom's features. Activations in both modules preserve the relevant rotation rules. Stacking blocks expands the atomic environment represented by the features, and an invariant readout maps the final features to the system energy.

A.3 Training of interatomic potentials

The resulting model predicts energy $E_\theta(\mathbf{r}, Z)$ from atomic positions $\mathbf{r}$ and atom types Z. Predicting forces as $F = -\nabla_{\mathbf{r}} E_\theta$ enforces their conservative relation to this energy, a physical property

important for downstream applications (Fu et al., 2025). An energy–force objective combines energy and force errors as $\mathcal{L} = \lambda_E \mathcal{L}_E + \lambda_F \mathcal{L}_F$.

Training on force labels requires differentiating through the energy gradient. As expressed in Equation (3), the force loss contributes a mixed second derivative with respect to model parameters and atomic positions. Energy evaluation first constructs the computation that predicts E_θ. Force differentiation propagates derivatives to the positions while retaining their dependence on the model parameters. Loss backpropagation then differentiates the energy and force errors with respect to those parameters. These three stages determine the forward, backward, and double-backward computations required for training.

B Method Details

B.1 Node to edge

Rotation on gather constructs the first SO(2) linear layer's inputs from node features. For each edge from atom j to atom i, it rotates x_j into the local frame and writes the retained components into the order-grouped matrices $T^{(m)}$. This combines the gather, rotation, and layout conversion in the upper-left part of Figure 2. The rotation representation and the output layout determine which entries each tile reads and writes.

Compact rotation blocks. The full matrix D_{ij} acts independently within each degree, and the local convolution retains only orders $|m| \le M$. For degree ℓ, define the retained order set $\mathcal{M}_\ell = \{m : -\ell \le m \le \ell,\ |m| \le M\}$, with $q_\ell = 2\min(\ell, M) + 1$ entries. Selecting these rows from the full $n_\ell \times n_\ell$ block, where $n_\ell = 2\ell + 1$, gives

$$R_{ij}^{(\ell)} = J_\ell D_{ij}^{(\ell)} \in \mathbb{R}^{q_\ell \times n_\ell}. \tag{18}$$

Here J_ℓ selects the rows indexed by $\mathcal{M}_\ell$. Thus $R_{ij}^{(\ell)}$ contains original entries of $D_{ij}^{(\ell)}$, with all global-frame columns but only the required local-frame rows. Figure 7 shows the omitted zeros between degrees and the unused rows within each block.

The blocks are stored consecutively across degrees, using

$$K_{L,M} = \sum_{\ell=0}^{L} q_\ell n_\ell = \sum_{\ell=0}^{L} \big(2\min(\ell, M) + 1\big)(2\ell + 1) \tag{19}$$

entries per edge, instead of d^2 for a dense rotation matrix. At $L = 3$ and $M = 1$, the block shapes are 1×1, 3×3, 3×5, and 3×7, totaling 46 entries instead of 256. The same stored blocks serve rotation on gather, inverse rotation on reduction, and their derivatives.

Writing the GEMM input. The retained rows are labeled by their signed order m, while columns retain the global order $m' = -\ell, \ldots, \ell$. At channel width C, the first linear layer's matrix entries are

$$T_{ij,(\ell,c)}^{(m)} = \tilde{x}_{ij,m,c}^{(\ell)} = \sum_{m'=-\ell}^{\ell} R_{ij,mm'}^{(\ell)} x_{j,m',c}^{(\ell)}, \qquad m \in \mathcal{M}_\ell. \tag{20}$$

For each edge tile, the kernel loads node features by the source atom indices, contracts them with $R_{ij}^{(\ell)}$, and writes the results directly to degree–channel columns of $T^{(m)}$. No gathered feature array or separate rotated layout is written to HBM. Algorithm 2 makes this indexing explicit.

Differentiation and retained state. Backward propagates energy derivatives to both the node features and the geometry-dependent rotation blocks. For one edge and degree, write the rotation as $\tilde{x} = Rx$. Its backward computation gives

$$\bar{x} = R^{\mathsf{T}} \bar{\tilde{x}}, \qquad \bar{R} = \bar{\tilde{x}} x^{\mathsf{T}}, \tag{21}$$

where bars denote derivatives of E_θ. The contributions to $\bar{x}_j$ are summed over edges that use atom j, giving the Edge $\to$ Node pattern. Computing $\bar{R}_{ij}$ gathers the saved x_j again, so the EdC gathered

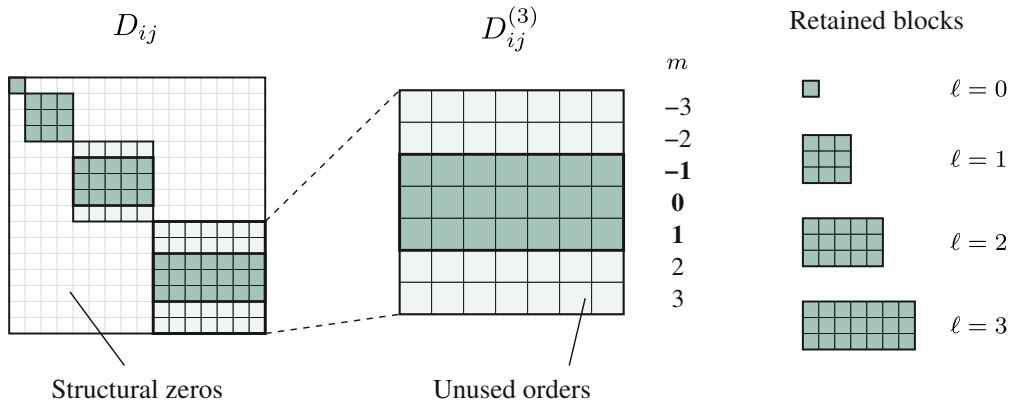


Figure 7: Compact rotation blocks omit structural zeros and unused angular orders. An enlarged diagonal block highlights the retained rows, which are stored separately for each degree.

Algorithm 2 Rotation on gather into order-grouped GEMM inputs

Input: Node features x, compact rotation blocks R, directed edges (i, j) from j to i, L, M, C.
Output: $T^{(m)} \in \mathbb{R}^{E\times(L-|m|+1)C}$ for $-M \le m \le M$.

for all edge tiles $\mathcal{B}$, degrees ℓ, and channel tiles $\mathcal{C}$ in parallel **do**
  $X \leftarrow \mathrm{load}\big((x_j^{(\ell)}[:, \mathcal{C}])_{(i,j)\in\mathcal{B}}\big)$
  for $m \in \mathcal{M}_\ell$ **do**
    $r \leftarrow \mathrm{load}\big((R_{ij}^{(\ell)}[m, :])_{(i,j)\in\mathcal{B}}\big)$
    $Z_{ij,c} \leftarrow \sum_{m'=-\ell}^{\ell} r_{ij,m'} X_{ij,m',c}$
    $p \leftarrow (\ell - |m|)C + \mathcal{C}$ ▶ *degree–channel columns*
    $\mathrm{store}(T^{(m)}[\mathcal{B}, p], Z)$
  end for
end for

values need not be retained in addition to the NdC node values. Double backward differentiates these same local products. Its feature path gathers node quantities and rotates them back to edges, as in Equation (5). Since row selection does not depend on atomic positions, derivatives of $R_{ij}^{(\ell)}$ select the same rows of the corresponding derivatives of $D_{ij}^{(\ell)}$. Rotation-gradient contributions therefore use the compact block dimensions before differentiation continues to positions.

B.2 Edge to edge

Recombination with activation connects the two SO(2) linear layers. The first GEMM writes product blocks Q to HBM. The activation requires the paired-order features formed from these blocks, while the second GEMM requires its inputs grouped by order. For an edge ij, the combined computation is

$$z_{ij} = \mathcal{A}(\mathrm{recombine}(Q_{ij}))\,, \qquad u_+ = Q_{00} - Q_{11}, \quad u_- = Q_{10} + Q_{01}. \tag{22}$$

Here $\mathcal{A}$ is the model's activation, as in Algorithm 1, and z is the next GEMM's input. Recombination remains before activation. A tile reads the matching entries of Q, forms the required features on chip, applies $\mathcal{A}$, and writes z directly in the next GEMM layout. This removes the recombined array and the activation input and output layouts identified in Figure 2.

An activation can depend on several angular components of the same edge. The tile must include those components or complete the required local reduction before applying the dependent operation. This determines the tile's extent without introducing a dependency on another edge. Gating provides a concrete instance of this dataflow.

Recombination with a gate. For a nonzero degree, let $\gamma_{ij,\ell,c}$ be a gate preactivation shared by the angular components of channel c, and let σ be the sigmoid function. At one positive order m, suppressing the edge, degree, and channel indices gives

$$z_+ = \sigma(\gamma) \odot (Q_{00} - Q_{11}), \qquad z_- = \sigma(\gamma) \odot (Q_{10} + Q_{01}). \tag{23}$$

Each product block has shape $E \times h_m$, where $h_m = (L - m + 1)C$. Algorithm 3 loads its degree–channel columns together with the gate and writes only the activated pair. The gate inputs retain their dependence on the model parameters and features.

Algorithm 3 Recombination with gating for a positive order m

Input: $Q_{00}, Q_{01}, Q_{10}, Q_{11} \in \mathbb{R}^{E \times h_m}$, gate preactivations γ, $1 \le m \le M$, L, C.
Output: $z_+, z_- \in \mathbb{R}^{E \times h_m}$ in the next GEMM's input layout.

1: **for all** edge tiles $\mathcal{B}$, degrees $\ell = m, \dots, L$, and channel tiles $\mathcal{C}$ in parallel **do**
2: $\quad p \leftarrow (\ell - m)C + \mathcal{C}$
3: $\quad q_{\alpha\beta} \leftarrow \mathrm{load}(Q_{\alpha\beta}[\mathcal{B}, p])$ for $\alpha, \beta \in \{0, 1\}$
4: $\quad s \leftarrow \sigma(\mathrm{load}(\gamma[\mathcal{B}, \ell, \mathcal{C}]))$
5: $\quad u_+ \leftarrow q_{00} - q_{11}, \quad u_- \leftarrow q_{10} + q_{01}$
6: $\quad \mathrm{store}(z_+[\mathcal{B}, p], s \odot u_+)$
7: $\quad \mathrm{store}(z_-[\mathcal{B}, p], s \odot u_-)$
8: **end for**

Differentiation and local reconstruction. Backward first differentiates the activation and then distributes the resulting derivatives to the product blocks. For one gate shared across the retained orders of a degree and channel,

$$\bar{u}_m = \sigma(\gamma)\bar{z}_m, \qquad \bar{\gamma} = \sigma'(\gamma) \sum_{m \in \mathcal{M}_\ell} \bar{z}_m u_m. \tag{24}$$

The angular sum produces one gate derivative for that degree and channel. For each positive-order pair, recombination then gives

$$(\bar{Q}_{00}, \bar{Q}_{01}, \bar{Q}_{10}, \bar{Q}_{11}) = (\bar{u}_+, \bar{u}_-, \bar{u}_-, -\bar{u}_+). \tag{25}$$

The same local calculation regenerates u from the saved product blocks when needed. Double backward differentiates the gate products and their angular sum, using $\sigma'(\gamma)$ and $\sigma''(\gamma)$ together with the reconstructed features. Recombination only redistributes entries with the signs above. Both derivative stages therefore remain Edge $\to$ Edge, without retaining the full recombined feature array.

B.3 Edge to node

Inverse rotation on reduction completes the convolution by turning the second SO(2) linear layer's outputs into node updates. Its local messages t_{ij} are formed by recombining that layer's product blocks. Reusing the compact rotation blocks defined in Section B.1, the output for degree ℓ is

$$x_i'^{(\ell)} = \sum_{j \in \mathcal{N}(i)} (R_{ij}^{(\ell)})^\mathsf{T} t_{ij}^{(\ell)}, \qquad x_{i,m',c}'^{(\ell)} = \sum_{j \in \mathcal{N}(i)} \sum_{m \in \mathcal{M}_\ell} R_{ij,mm'}^{(\ell)} t_{ij,m,c}^{(\ell)}. \tag{26}$$

The transpose maps q_ℓ retained local components to all n_ℓ global components. It equals $(D_{ij}^{(\ell)})^\mathsf{T} J_\ell^\mathsf{T}$, where J_ℓ^T inserts zeros in unused orders. The compact contraction gives the same result without forming the zero-padded local message or a separate inverse rotation matrix.

Accumulating node outputs. A parallel task owns a node-output tile, keeps its accumulator on chip, and traverses the node's incoming edges. For each neighbor, it loads the required second-layer product entries, recombines them, and accumulates their inverse-rotated contribution. Only the completed node tile is written to HBM. Thus neither the local messages nor the per-edge global messages are stored, and the layout conversion before inverse rotation is absorbed into the indexed reads of Q.

Algorithm 4 uses Q for the second GEMM's outputs, following Algorithm 1. Its order-zero output is $Q^{(0)}$, and positive orders have the four blocks $Q_{\alpha\beta}^{(m)}$. With output channel width C, their widths are

$(L+1)C$ and $(L-m+1)C$, respectively. Iterating $j \in \mathcal{N}(i)$ reads an incoming-edge list grouped by receiver. All rotation-row indices below are signed local orders, while m' indexes a global output component.

Algorithm 4 Inverse rotation on reduction from product blocks

Input: Second-layer blocks $Q^{(0)}$ and $\{Q^{(m)}_{\alpha\beta}\}_{m=1}^{M}$, compact rotations R, incoming neighbors $\mathcal{N}(i)$, N, L, M, C.
Output: $x'^{(\ell)} \in \mathbb{R}^{N\times(2\ell+1)\times C}$ for $\ell = 0, \ldots, L$.

for all nodes i, degrees ℓ, global components m', and channel tiles $\mathcal{C}$ in parallel **do**
  $a \leftarrow 0_{|\mathcal{C}|}$
  for $j \in \mathcal{N}(i)$ **do**
    $t_0 \leftarrow \mathrm{load}(Q^{(0)}[ij, \ell C + \mathcal{C}])$
    $r_0 \leftarrow \mathrm{load}(R^{(\ell)}_{ij}[0, m']), \quad a \leftarrow a + r_0 t_0$
    for $m = 1, \ldots, \min(\ell, M)$ **do**
      $p \leftarrow (\ell - m)C + \mathcal{C}$
      $q_{\alpha\beta} \leftarrow \mathrm{load}(Q^{(m)}_{\alpha\beta}[ij, p])$ for $\alpha, \beta \in \{0, 1\}$
      $r_{\pm} \leftarrow \mathrm{load}(R^{(\ell)}_{ij}[\pm m, m'])$
      $a \leftarrow a + r_{+}(q_{00} - q_{11}) + r_{-}(q_{10} + q_{01})$
    end for
  end for
  $\mathrm{store}(x'^{(\ell)}_i[m', \mathcal{C}], a)$
end for

Differentiation and retained state. Backward gathers the receiving node's energy derivative and rotates it to each local edge frame. For one edge and degree, the local derivatives are

$$\bar{t}_{ij} = R_{ij}\bar{x}'_i, \qquad \bar{R}_{ij} = t_{ij}(\bar{x}'_i)^{\mathsf{T}}. \tag{27}$$

The derivative $\bar{t}_{ij}$ is distributed to Q with the recombination signs in Equation (25). The rotation derivative uses a locally reconstructed t_{ij}, so the complete local-message array is unnecessary. Rotation-gradient contributions from the input and output of the convolution are accumulated in the same compact representation. The feature path in double backward reverses gathering back into a receiver reduction, while derivatives through R_{ij} continue to the edge geometry. Retaining the required product blocks, rotation blocks, and incoming gradients supports these computations without storing global edge messages.

B.4 Node to node

The same principle of avoiding intermediate materialization extends to Node → Node computations. Equivariant feed-forward networks combine channel transformations with nonlinear operations on each atom's features. These computations can involve rearranging feature layouts, expanding scalar gates across angular components, or converting harmonic coefficients to and from spherical-grid values. Applying these transformations within the operations that use their results avoids separately materializing the corresponding intermediate arrays. Although node tensors are typically smaller than edge tensors, wide hidden representations can make these savings substantial.

Spherical activations. EquiformerV3's equivariant feed-forward network provides one such example. Its spherical activations expand each atom's harmonic coefficients onto a grid before applying nonlinear operations. This expansion can produce large intermediate tensors when the hidden channels are wide. We examine the linear transformations around this expansion to identify which grid intermediates can be avoided.

For this example, write the features of one atom as $X^{(\ell)} = x_i^{(\ell)} \in \mathbb{R}^{n_\ell \times C_{\mathrm{in}}}$. Let $G_\ell \in \mathbb{R}^{n_g \times n_\ell}$ evaluate the degree-ℓ basis at n_g grid points, $W_\ell \in \mathbb{R}^{C_{\mathrm{in}} \times C_h}$ transform the coefficient channels, and $A \in \mathbb{R}^{C_h \times C_{\mathrm{out}}}$ transform the grid channels before the nonlinearity.

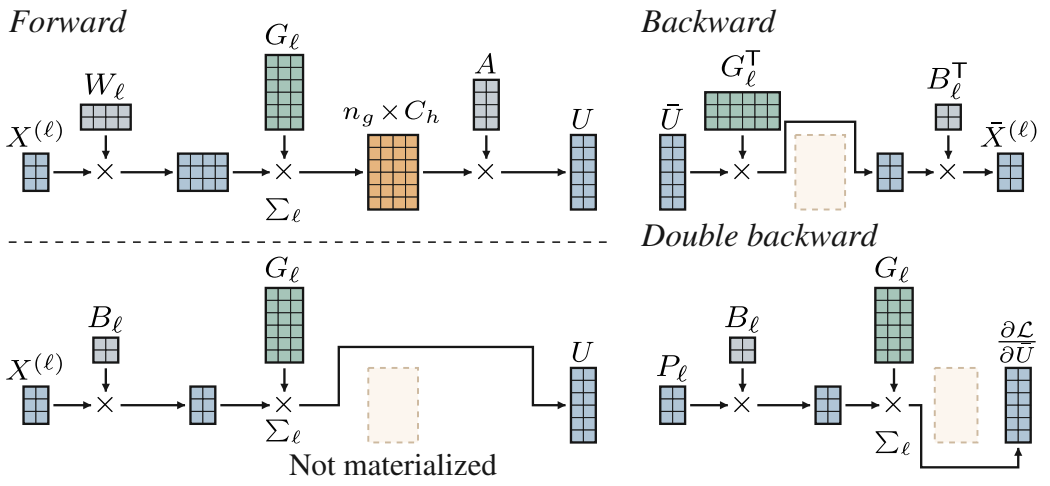


Figure 8: Algebraic reordering avoids expanded grid intermediates in the spherical-activation example. The forward comparison is shown on the left, with reordered backward and double-backward computations on the right.

Forward computation. In this computation, the grid transform acts on angular components and the channel maps act on channels. Associativity and distributivity therefore allow the channel maps to be composed before grid expansion:

$$U = \left[\sum_{\ell=0}^{L} G_\ell (X^{(\ell)} W_\ell)\right] A = \sum_{\ell=0}^{L} G_\ell (X^{(\ell)} B_\ell), \qquad B_\ell = W_\ell A. \tag{28}$$

The original order first forms grid features with C_h channels and then projects them to C_{out}. The reordered computation applies B_ℓ to the coefficients and expands only the resulting C_{out} channels. It produces the same $U \in \mathbb{R}^{n_g \times C_{\text{out}}}$ without the $n_g \times C_h$ grid intermediate in Figure 8. The composed maps are shared across atoms and remain functions of the original trainable parameters.

The output side permits the corresponding reordering. Let Y contain the nonlinear grid values and let $H_\ell \in \mathbb{R}^{n_\ell \times n_g}$ project them back to coefficients, including quadrature weights. When a channel map A_{out} precedes this projection,

$$H_\ell (Y A_{\text{out}}) = (H_\ell Y) A_{\text{out}}. \tag{29}$$

The channel map can then be composed with a subsequent coefficient-channel map. Both transformations keep the nonlinearity in its original position.

Backward. The energy derivative arriving at U is $\bar{U} = \partial E_\theta / \partial U$. Propagating it to the coefficients gives

$$\bar{X}^{(\ell)} = G_\ell^{\mathsf{T}} (\bar{U} A^{\mathsf{T}}) W_\ell^{\mathsf{T}} = (G_\ell^{\mathsf{T}} \bar{U}) B_\ell^{\mathsf{T}}. \tag{30}$$

The original order constructs $\bar{U} A^{\mathsf{T}}$ on the grid. The reordered path first returns $\bar{U}$ to coefficient space and then applies B_ℓ^{T}, avoiding the $n_g \times C_h$ derivative intermediate. The result continues to the preceding node features.

Double backward. Let $P_\ell = \partial \mathcal{L} / \partial \bar{X}^{(\ell)}$ be the incoming loss gradient along this feature path. Differentiating Equation (30) contributes

$$\frac{\partial \mathcal{L}}{\partial \bar{U}} = \sum_\ell G_\ell (P_\ell B_\ell). \tag{31}$$

Each P_ℓ is transformed in coefficient space and then expanded to the grid, following the reordered forward sequence. This again avoids an $n_g \times C_h$ intermediate. The resulting gradient continues through the computation of $\bar{U}$, including the derivative of the nonlinearity.

Gradients of the channel maps. Loss backpropagation also reaches B_ℓ through its uses in forward and backward. For the local computation above, the two contributions are

$$\frac{\partial \mathcal{L}}{\partial B_\ell} = (X^{(\ell)})^\mathsf{T} G_\ell^\mathsf{T} \frac{\partial \mathcal{L}}{\partial U} + P_\ell^\mathsf{T} G_\ell^\mathsf{T} \bar{U}. \tag{32}$$

The first term differentiates the forward map and the second differentiates the coefficient-gradient computation. Their contributions to the original parameters follow by the chain rule:

$$\frac{\partial \mathcal{L}}{\partial W_\ell} = \frac{\partial \mathcal{L}}{\partial B_\ell} A^\mathsf{T}, \qquad \frac{\partial \mathcal{L}}{\partial A} = \sum_\ell W_\ell^\mathsf{T} \frac{\partial \mathcal{L}}{\partial B_\ell}. \tag{33}$$

The node dimension remains independent in all these computations. Feature derivatives return to the same atom, while shared channel-map gradients are summed over atoms.

B.5 Storage and I/O counts

The separate-operation schedule stores the output of each transformation before the next operation reads it. We count these allocations individually, including the copies introduced by changes between degree and order layouts. The node input, node output, and the four GEMM input and output matrices are common to both schedules and are excluded from the eliminated intermediates. We count one logical write and one logical read for each eliminated array. Cache hits and repeated operand loads within a kernel are outside this count.

Full-order layouts. Let C_0, C_1, C_2, C_3 denote the channel widths at the convolution input, activation input, activation output, and convolution output, respectively. For $M = L$, the eliminated arrays have the following sizes:

$$\begin{aligned}
&\text{Node} \to \text{Edge:} && \underbrace{EdC_0}_{\text{gathered}} + \underbrace{EdC_0}_{\text{rotated before layout conversion}}, \\
&\text{Edge} \to \text{Edge:} && \underbrace{EdC_1}_{\text{recombined}} + \underbrace{EdC_1}_{\text{activation input layout}} + \underbrace{EdC_2}_{\text{activation output layout}}, \\
&\text{Edge} \to \text{Node:} && \underbrace{EdC_3}_{\text{recombined}} + \underbrace{EdC_3}_{\text{local message layout}} + \underbrace{EdC_3}_{\text{global messages}}.
\end{aligned} \tag{34}$$

The first row contains the gathered node features and the rotated coefficients before they are written in the first GEMM's input layout. In the second row, recombination writes one array, its conversion to the activation input layout writes another, and activation writes a third before conversion to the next GEMM layout. The final row contains the second layer's recombination output, its conversion to the local message layout, and the global messages before aggregation. The four layout copies are thus accounted for once, rather than added again to the eight intermediate allocations.

Writing and reading these arrays gives

$$B_{\text{eliminated}} = 2bEd(2C_0 + 2C_1 + C_2 + 3C_3). \tag{35}$$

At equal widths this becomes $16bEdC$. For $E = 32{,}768$, $L = M = 4$, $C = 128$, and FP32, this equals 6.25 GiB of logical feature accesses per convolution. The sum of allocated intermediate elements is $8EdC$.

Retained-order layouts. For $M < L$, the number of retained angular components is $q_{L,M}$ in Equation (16). Gathered node features and global messages still have d angular components. If all local-frame intermediate layouts store only the retained coefficients, the six local arrays use $q_{L,M}$ in place of d, giving

$$B_{\text{eliminated}} = 2bE\left[d(C_0 + C_3) + q_{L,M}(C_0 + 2C_1 + C_2 + 2C_3)\right]. \tag{36}$$

This count assumes that the activation consumes and produces retained-order features. If an intermediate instead uses a full degree layout with zero entries for omitted orders, its term uses d. This substitution accounts for the representation actually stored without changing the list of eliminated arrays.

Training state and reconstruction. For rotation on gather, node features occupy NdC_0 elements, whereas their gathered copies occupy EdC_0. Keeping the node features and edge indices suffices to regenerate the gathered values, removing the need to retain an additional EdC_0-element copy. Each reconstruction replaces a read of that copy with indexed node-feature reads and local computation. Rotations are retained in the compact blocks described in Section B.1, whose storage is $EK_{L,M}$ elements instead of Ed^2 for dense matrices.

The feature-gradient paths in Equations (4) and (5) admit the same local treatment. With all orders retained, separately gathering a node gradient before rotation or separately rotating edge contributions before reduction produces an EdC-element array. Combining these operations avoids its allocation and one write–read pair, or $2bEdC$ logical bytes for that intermediate.

C Experimental Details and Additional Results

C.1 Experimental setup

Models and training data. We evaluate eSEN (Fu et al., 2025), UMA (Wood et al., 2026), and EquiformerV3 (Liao et al., 2026) on OMat24 (Barros-Luque et al., 2026), using their official OMat24 architectures and training settings. Tables 2 and 3 list the hyperparameters used in our experiments, following the officially released configurations. We use eSEN-30M-OMat, UMA-S-1.2, and the $L = 4$ EquiformerV3 configuration. Trainable parameters are randomly initialized according to each model's initialization rules. Official checkpoints provide the fixed hyperparameters and non-trainable quantities, including reference energies, and are used to verify architecture consistency. The training objective includes energy, forces, and stresses, with forces and stresses obtained by differentiating the predicted energy.

Implementation. Our custom operators are implemented in Triton and exposed through PyTorch interfaces supporting backward and double backward. Dense matrix products use PyTorch's GPU matrix-multiplication routines. For Node $\rightarrow$ Edge, all three models combine gathering, rotation, and radial scaling into the input layout of the first SO(2) linear layer. For Edge $\rightarrow$ Edge, eSEN and UMA combine recombination with gated activation, while EquiformerV3 uses its spherical-grid activation. UMA additionally uses system-dependent mixtures of linear experts for its edge projections. For Edge $\rightarrow$ Node, eSEN and UMA sum messages with their cutoff envelopes, while EquiformerV3 applies attention weights before aggregation. Each path incorporates inverse rotation into the reduction. For Node $\rightarrow$ Node, eSEN and UMA fuse operations around their coefficient-space gated FFNs, including normalization and residual updates. EquiformerV3 uses spherical-grid FFNs, whose linear maps are reordered and composed to reduce intermediate expansion. These adaptations preserve each model's activation and weighting rules.

Hardware and software. PyTorch denotes the official implementation in model experiments and local PyTorch operations following the official operation order and layouts in operator experiments. TrimSO(2) replaces the corresponding computations within the same training pipeline. Each pair shares its initialization, batch sequence, optimizer, exponential moving average (EMA), and learning-rate schedule. All comparisons use a single A100 80GB or H100 80GB GPU in FP32, with TF32 and mixed precision disabled. Table 4 gives the upstream revisions and software versions used for each model.

Baselines. Our model-level baselines follow the official configurations for training on energy-derived forces, which do not enable `torch.compile`. Such training requires differentiating through the force computation, but PyTorch's AOTAutograd does not support double backward through the backward graphs it generates (PyTorch issue #91469). NequIP and Allegro address this limitation by tracing the force computation into the forward graph before compilation (Tan et al., 2026). The official implementations used here do not take this route: EquiformerV3 enables compilation for direct-force training but disables it in its official gradient-force configuration on OMat24, UMA compiles only in its inference path, and eSEN trains without compilation. A limitation is the absence of a compiled baseline using NequIP-style tracing. Such a baseline would require model-specific reimplementations beyond the scope of this work, and could be combined with TrimSO(2).

Table 2: Architecture hyperparameters of the evaluated official models.

Hyperparameter	eSEN	UMA	EquiformerV3
Number of layer blocks	10	4	7
Maximum degree L	3	2	4
Maximum order M	2	2	2
Number of channels C	128	128	128
FFN hidden channels	128	128	512
Radial hidden channels	128	128	128
Number of radial basis functions	64	32	64
Cutoff radius (Å)	6	6	6
Maximum number of neighbors	300	300	300
Number of linear experts	—	64	—
Number of attention heads	—	—	8
Attention hidden channels	—	—	32
Attention weight channels	—	—	64
Value channels per head	—	—	16
Activation	Gated	Gated	SwiGLU-S^2

Table 3: Training hyperparameters shared by PyTorch and TrimSO(2).

Hyperparameter	eSEN	UMA	EquiformerV3
Batch size per GPU	8 structures	At most 700 atoms	16 structures
Optimizer	AdamW	AdamW	AdamW
Maximum learning rate	3×10^{-4}	4×10^{-4}	10^{-4}
Adam (β_1, β_2)	$(0.9, 0.999)$	$(0.9, 0.999)$	$(0.9, 0.98)$
Adam ϵ	10^{-8}	10^{-8}	10^{-6}
Weight decay	10^{-3}	10^{-3}	10^{-3}
Learning-rate scheduling	Cosine	Cosine	Cosine
Schedule horizon	2 epochs	10^6 steps	2 epochs
Warmup steps	12,623	1	6,311
Warmup factor	0.2	0.2	0
Minimum learning-rate factor	0.05	0.01	0.01
Gradient clipping norm	100	100	100
Model EMA decay	0.999	0.999	0.999
Energy loss coefficient	20	15	20
Force loss coefficient	20	5	20
Stress loss coefficient	5	1	5
Precision	FP32	FP32	FP32

Table 4: Software versions for the official implementations.

Model	Official revision	PyTorch	Triton
eSEN	fairchem, 977a8032	2.11.0+cu126	3.6.0
UMA	fairchem, 802979b9	2.8.0+cu128	3.4.0
EquiformerV3	EquiformerV3, a7300c58	2.7.1+cu118	3.3.1

C.2 End-to-end performance

Training time and peak memory. We measure the complete training loop, including data loading, device transfers, graph construction, energy and force/stress evaluation, loss backpropagation, gradient clipping, optimizer updates, EMA, and learning-rate updates. After warmup, each run measures 100 consecutive steps, with synchronization only at the beginning and end of the interval. Dividing the elapsed time by 100 gives the average step time for that run. Table 5 reports the mean of these averages, using three independent runs for both implementations.

Each implementation runs in a separate process. After warmup, we release temporary outputs and computation graphs, synchronize, and reset the peak-memory counter before the measured interval. Peak allocated memory includes model parameters, inputs, optimizer states, and live temporary tensors, but excludes unused allocator cache. We report the largest peak across the three runs. At the same batch sizes, TrimSO(2) lowers both step time and peak memory for all three models on both GPUs, yielding the 2.07–2.94× speedups and 1.53–3.16× memory reductions in Figure 3.

Table 5: End-to-end training time and peak allocated memory. Time aggregates three independent runs. Memory is the largest peak across these runs. Speedup and memory reduction are relative to PyTorch.

		Time (ms/step)		Memory (GiB)			
GPU	Model	PyTorch	TrimSO(2)	PyTorch	TrimSO(2)	Speedup	Memory reduction
A100	eSEN	653.55	315.05	34.62	17.66	**2.07**×	**1.96**×
A100	UMA	1077.19	366.49	27.04	17.67	**2.94**×	**1.53**×
A100	EquiformerV3	1249.30	441.90	70.98	22.48	**2.83**×	**3.16**×
H100	eSEN	370.71	172.17	34.66	17.71	**2.15**×	**1.96**×
H100	UMA	629.05	244.86	27.09	17.71	**2.57**×	**1.53**×
H100	EquiformerV3	606.86	246.38	71.02	22.52	**2.46**×	**3.15**×

HBM traffic. For each model, we profile one fixed batch from the shared initial state with Nsight Compute. The capture covers energy evaluation, force/stress differentiation, and loss backpropagation. We sum the DRAM read and write counters over all kernels in this interval to obtain HBM traffic. Data preparation and optimizer updates are outside this capture. CUDA Graph is disabled, and segmented captures are combined only after matching their kernel sequence to an unsegmented trace. All timing results come from separate, uninstrumented runs.

Table 6 gives the absolute traffic underlying Figure 3. 3.81–9.88× reductions in HBM traffic accompany the training-time and memory benefits across the three models. Reduction factors in the figure and tables are the PyTorch measurement divided by the corresponding optimized measurement for the same model and GPU.

Table 6: Total HBM reads and writes during energy evaluation, force/stress differentiation, and loss backpropagation for one batch. Traffic is in GiB, and reductions are relative to PyTorch.

GPU	Model	PyTorch	TrimSO(2)	Reduction (×)
A100	eSEN	160.29	37.84	**4.24**×
A100	UMA	1071.02	108.37	**9.88**×
A100	EquiformerV3	913.75	198.59	**4.60**×
H100	eSEN	153.23	40.18	**3.81**×
H100	UMA	1066.33	108.56	**9.82**×
H100	EquiformerV3	1019.30	237.19	**4.30**×

C.3 Ablation study

We start from PyTorch and successively enable Node → Edge, Edge → Edge, and Edge → Node optimizations. The final stage adds Node → Node optimizations to obtain full TrimSO(2). Each stage retains the preceding optimizations and uses the same initialization, batch sequence, and training settings. Timing and memory follow the protocol above. The PyTorch and full TrimSO(2) rows reuse Table 5, so the endpoints are identical to the end-to-end comparison.

Table 7 provides the absolute measurements behind the A100 results in Figure 4 and the corresponding H100 results. On A100, the three edge stages together give 1.82–2.80× training speedups and 1.52–2.24× reductions in peak memory before the final node optimizations are added. Peak memory decreases at every stage on both GPUs. These measurements show how the reductions accumulate within the same model and training workload.

Table 7: Training results as optimizations are added cumulatively. Time is in ms/step and memory is in GiB. Speedup and peak-memory reduction are relative to PyTorch for the same model and GPU.

A100					
Model	Cumulative stage	Time	Memory	Speedup	Memory reduction
eSEN	PyTorch	653.55	34.62		
eSEN	+Node → Edge	499.08	27.10	**1.31**×	**1.28**×
eSEN	+Edge → Edge	406.14	22.76	**1.61**×	**1.52**×
eSEN	+Edge → Node	330.67	17.92	**1.98**×	**1.93**×
eSEN	TrimSO(2)	315.05	17.66	**2.07**×	**1.96**×
UMA	PyTorch	1077.19	27.04		
UMA	+Node → Edge	1007.46	22.55	**1.07**×	**1.20**×
UMA	+Edge → Edge	417.04	18.84	**2.58**×	**1.44**×
UMA	+Edge → Node	384.55	17.82	**2.80**×	**1.52**×
UMA	TrimSO(2)	366.49	17.67	**2.94**×	**1.53**×
EquiformerV3	PyTorch	1249.30	70.98		
EquiformerV3	+Node → Edge	976.16	49.70	**1.28**×	**1.43**×
EquiformerV3	+Edge → Edge	797.50	37.72	**1.57**×	**1.88**×
EquiformerV3	+Edge → Node	686.16	31.63	**1.82**×	**2.24**×
EquiformerV3	TrimSO(2)	441.90	22.48	**2.83**×	**3.16**×
H100					
Model	Cumulative stage	Time	Memory	Speedup	Memory reduction
eSEN	PyTorch	370.71	34.66		
eSEN	+Node → Edge	289.92	27.15	**1.28**×	**1.28**×
eSEN	+Edge → Edge	265.17	22.80	**1.40**×	**1.52**×
eSEN	+Edge → Node	196.76	17.97	**1.88**×	**1.93**×
eSEN	TrimSO(2)	172.17	17.71	**2.15**×	**1.96**×
UMA	PyTorch	629.05	27.09		
UMA	+Node → Edge	600.91	22.59	**1.05**×	**1.20**×
UMA	+Edge → Edge	326.17	18.89	**1.93**×	**1.43**×
UMA	+Edge → Node	238.37	17.86	**2.64**×	**1.52**×
UMA	TrimSO(2)	244.86	17.71	**2.57**×	**1.53**×
EquiformerV3	PyTorch	606.86	71.02		
EquiformerV3	+Node → Edge	487.96	49.74	**1.24**×	**1.43**×
EquiformerV3	+Edge → Edge	376.88	37.77	**1.61**×	**1.88**×
EquiformerV3	+Edge → Node	346.79	31.67	**1.75**×	**2.24**×
EquiformerV3	TrimSO(2)	246.38	22.52	**2.46**×	**3.15**×

C.4 Operator performance

C.4.1 Workloads and measurement

The three operator boundaries isolate the dataflows around the dense matrix products. Node → Edge includes indexed node reads, rotation, edge-coefficient application, and generation of the GEMM input layout. Edge → Edge begins at the first GEMM's output and includes recombination, activation, and layout conversion. Edge → Node begins at the second GEMM's output and includes recombination, inverse rotation, and aggregation at receiving nodes.

The stage-wise measurements use inputs captured from an EquiformerV3 training batch with $L/M/C = 4/2/128$, $N = 399$, and $E = 17{,}940$. Both implementations receive the same captured inputs and differentiation directions. The PyTorch reference reproduces the official operation order and layouts at each boundary.

We measure forward, backward, and double backward separately. Forward computes the outputs and retains the state needed for differentiation. Backward computes derivatives with respect to differentiable boundary inputs, including captured parameter-dependent tensors, and retains the graph needed for double backward. Double backward differentiates these first derivatives with respect to the boundary inputs and incoming output gradients, using shared differentiation directions. Each

timed stage starts with its prerequisite computations and graphs already prepared. After warmup, we take 30 CUDA-event measurements per stage and report their mean. Required layout generation and storage are included in the measured computation.

Table 8 supplements Figure 5(a) with absolute latencies and H100 measurements. All three operators are faster than PyTorch in forward, backward, and double backward on both GPUs. The gains therefore cover the computations needed for training, including its higher-order derivatives.

Table 8: Operator latency (ms) and speedup. Each stage is measured separately. Speedups compare PyTorch and TrimSO(2) on the same GPU. Dataflow labels abbreviate nodes and edges as N and E.

		A100			H100		
Dataflow	Stage	PyTorch	TrimSO(2)	Speedup	PyTorch	TrimSO(2)	Speedup
N → E	Forward	6.467	0.386	**16.73**×	2.898	0.226	**12.83**×
	Backward	17.644	2.620	**6.73**×	5.346	1.135	**4.71**×
	Double backward	31.908	4.406	**7.24**×	12.118	1.799	**6.74**×
E → E	Forward	1.851	0.684	**2.71**×	1.266	0.404	**3.14**×
	Backward	6.184	1.051	**5.88**×	3.638	0.639	**5.69**×
	Double backward	11.770	1.803	**6.53**×	7.041	1.015	**6.94**×
E → N	Forward	2.513	0.376	**6.69**×	1.188	0.241	**4.93**×
	Backward	9.099	0.970	**9.38**×	2.706	0.917	**2.95**×
	Double backward	16.368	2.910	**5.62**×	6.454	1.840	**3.51**×

C.4.2 Configuration and graph size

The synthetic sweep uses the same operator paths and PyTorch computation structure as the captured-input comparison. Its central configuration is $L/M/C = 4/2/128$, $N = 512$, and $K = 64$ outgoing edges per node, giving $E = NK = 32{,}768$. Each node selects K distinct targets uniformly from the other nodes. Both implementations share the generated graph, input tensors, and differentiation directions.

Table 9 varies L, M, and C at fixed N and K. Table 10 varies N and K at fixed $L/M/C$. Each configuration changes one factor, and the two tables reuse the same central measurements. These synthetic central measurements are distinct from the captured-input workload in Table 8. The reported double-backward speedups range from $2.34\times$ to $11.06\times$ across the two GPUs.

Table 9: Double-backward latency (ms) and speedup across model dimensions, with $N = 512$ and $K = 64$. N and E in dataflow labels denote nodes and edges.

		A100			H100		
$L/M/C$	Dataflow	PyTorch	TrimSO(2)	Speedup	PyTorch	TrimSO(2)	Speedup
4/2/128	N → E	58.386	8.499	**6.87**×	21.994	3.297	**6.67**×
4/2/128	E → E	31.435	5.006	**6.28**×	18.587	2.629	**7.07**×
4/2/128	E → N	29.415	5.088	**5.78**×	11.300	3.098	**3.65**×
2/2/128	N → E	17.090	3.017	**5.67**×	8.638	1.284	**6.72**×
2/2/128	E → E	27.549	2.490	**11.06**×	17.616	1.620	**10.87**×
2/2/128	E → N	10.779	2.069	**5.21**×	5.865	1.271	**4.61**×
3/2/128	N → E	26.077	5.361	**4.86**×	13.321	2.127	**6.26**×
3/2/128	E → E	29.342	3.867	**7.59**×	18.034	2.252	**8.01**×
3/2/128	E → N	16.771	3.454	**4.86**×	8.328	2.007	**4.15**×
5/2/128	N → E	75.669	13.851	**5.46**×	32.783	5.250	**6.24**×
5/2/128	E → E	33.119	6.319	**5.24**×	20.509	4.318	**4.75**×
5/2/128	E → N	34.727	7.117	**4.88**×	15.176	4.144	**3.66**×
6/2/128	N → E	96.752	19.558	**4.95**×	43.519	7.306	**5.96**×
6/2/128	E → E	35.552	10.673	**3.33**×	21.773	6.968	**3.12**×
6/2/128	E → N	41.129	10.020	**4.10**×	18.626	5.493	**3.39**×
4/1/128	N → E	52.842	6.233	**8.48**×	21.389	2.533	**8.44**×
4/1/128	E → E	28.235	3.357	**8.41**×	17.372	2.143	**8.11**×

		A100			H100		
$L/M/C$	Dataflow	PyTorch	TrimSO(2)	Speedup	PyTorch	TrimSO(2)	Speedup
4/1/128	E → N	16.547	3.100	**5.34**×	7.635	1.916	**3.98**×
4/3/128	N → E	58.965	10.355	**5.69**×	22.351	3.759	**5.95**×
4/3/128	E → E	34.018	5.779	**5.89**×	20.973	3.760	**5.58**×
4/3/128	E → N	34.677	6.164	**5.63**×	14.399	3.634	**3.96**×
4/4/128	N → E	59.917	11.251	**5.33**×	22.646	4.045	**5.60**×
4/4/128	E → E	36.430	7.369	**4.94**×	22.293	4.792	**4.65**×
4/4/128	E → N	39.312	6.752	**5.82**×	16.744	3.936	**4.25**×
4/2/64	N → E	29.233	4.762	**6.14**×	11.064	1.957	**5.65**×
4/2/64	E → E	16.350	2.893	**5.65**×	10.107	2.008	**5.03**×
4/2/64	E → N	18.373	2.707	**6.79**×	6.018	1.638	**3.67**×
4/2/256	N → E	117.145	16.780	**6.98**×	43.615	6.035	**7.23**×
4/2/256	E → E	63.044	10.032	**6.28**×	38.382	6.166	**6.23**×
4/2/256	E → N	57.674	9.839	**5.86**×	21.660	5.947	**3.64**×

Table 10: Double-backward latency (ms) and speedup across graph sizes and densities, with $L/M/C = 4/2/128$ and $E = NK$. N and E in dataflow labels denote nodes and edges.

		A100			H100		
N/K	Dataflow	PyTorch	TrimSO(2)	Speedup	PyTorch	TrimSO(2)	Speedup
512/64	N → E	58.386	8.499	**6.87**×	21.994	3.297	**6.67**×
512/64	E → E	31.435	5.006	**6.28**×	18.587	2.629	**7.07**×
512/64	E → N	29.415	5.088	**5.78**×	11.300	3.098	**3.65**×
128/64	N → E	14.612	2.097	**6.97**×	5.625	0.841	**6.69**×
128/64	E → E	8.122	1.341	**6.06**×	5.132	0.907	**5.66**×
128/64	E → N	7.747	1.431	**5.42**×	3.161	1.353	**2.34**×
256/64	N → E	29.108	4.191	**6.94**×	11.083	1.670	**6.64**×
256/64	E → E	15.960	2.559	**6.24**×	9.956	1.669	**5.96**×
256/64	E → N	14.978	2.685	**5.58**×	5.931	1.608	**3.69**×
1,024/64	N → E	117.972	17.497	**6.74**×	43.974	6.587	**6.68**×
1,024/64	E → E	62.808	10.065	**6.24**×	38.485	6.169	**6.24**×
1,024/64	E → N	58.452	9.902	**5.90**×	22.166	6.015	**3.69**×
2,048/64	N → E	238.393	35.766	**6.67**×	88.241	13.541	**6.52**×
2,048/64	E → E	125.770	20.180	**6.23**×	76.779	12.150	**6.32**×
2,048/64	E → N	116.596	19.630	**5.94**×	43.852	11.830	**3.71**×
4,096/64	E → E	248.266	40.425	**6.14**×	146.699	19.811	**7.40**×
4,096/64	E → N	234.986	39.570	**5.94**×	87.409	23.430	**3.73**×
512/16	N → E	14.660	2.169	**6.76**×	5.636	0.869	**6.48**×
512/16	E → E	8.103	1.334	**6.07**×	4.891	0.770	**6.35**×
512/16	E → N	7.754	1.384	**5.60**×	3.178	1.257	**2.53**×
512/32	N → E	29.158	4.284	**6.81**×	11.072	1.672	**6.62**×
512/32	E → E	15.933	2.555	**6.23**×	9.482	1.398	**6.78**×
512/32	E → N	14.998	2.616	**5.73**×	5.950	1.647	**3.61**×
512/128	N → E	117.768	17.568	**6.70**×	43.884	6.553	**6.70**×
512/128	E → E	62.704	9.979	**6.28**×	36.765	5.124	**7.18**×
512/128	E → N	58.383	10.134	**5.76**×	22.166	6.067	**3.65**×

C.5 Numerical accuracy

C.5.1 Operator outputs and derivatives

We compare the two implementations on the captured inputs used for stage-wise operator timing, with identical inputs and differentiation directions. Relative L_2 error is the norm of the difference divided by the norm of the PyTorch reference. For outputs or derivatives containing multiple tensors, squared differences and squared reference values are summed across the tensors before taking the ratio of

their square roots. Table 11 gives the errors for forward, backward, and double backward. Relative L_2 errors are at most 2.77×10^{-7}, showing close agreement throughout the three computation stages.

Table 11: Relative L_2 errors in operator outputs and derivatives on the captured training inputs.

GPU	Dataflow	Forward	Backward	Double backward
A100	Edge → Edge	4.33e-09	1.96e-08	1.09e-07
A100	Edge → Node	2.76e-07	1.60e-07	2.06e-07
A100	Node → Edge	4.57e-09	1.43e-07	1.98e-07
H100	Edge → Edge	4.33e-09	1.96e-08	1.09e-07
H100	Edge → Node	2.75e-07	1.59e-07	2.08e-07
H100	Node → Edge	4.57e-09	1.42e-07	1.98e-07

C.5.2 Model outputs and parameter gradients

For each model and GPU, we compare one shared random initialization on a fixed batch of the size in Table 3. We hold the graph and weights fixed and compare energy, forces, stresses, and gradients of the energy–force–stress loss with respect to all trainable parameters. Parameter-gradient error is aggregated across parameters using the same relative L_2 definition. This comparison includes differentiation through atomic positions and the mixed second derivatives needed to train on forces and stresses.

Table 12 shows that the local agreement carries through to complete models. Output errors are at most 1.47×10^{-6}, and parameter-gradient errors are at most 8.36×10^{-7} across all three models and both GPUs.

Table 12: Relative L_2 errors in model outputs and loss gradients at a shared random initialization.

GPU	Model	Energy	Forces	Stress	Parameter gradients
A100	eSEN	1.95e-07	9.89e-07	9.92e-07	6.80e-07
A100	UMA	6.66e-08	6.09e-07	4.90e-07	4.12e-07
A100	EquiformerV3	1.24e-07	7.14e-07	1.19e-06	3.98e-07
H100	eSEN	2.25e-07	9.54e-07	1.12e-06	8.36e-07
H100	UMA	1.05e-07	6.20e-07	4.46e-07	4.19e-07
H100	EquiformerV3	1.68e-07	7.80e-07	1.46e-06	4.10e-07

C.5.3 Short training trajectories

We next compare 100 consecutive training steps for all three models on both GPUs, starting from shared initial parameters and following the same batch sequence, optimizer, EMA, and learning-rate schedule. Figure 6(a) plots the losses without smoothing. Table 13 complements those trajectories with the final parameters, optimizer state, and EMA, as well as the parameter updates relative to initialization.

For the update comparison, we subtract the initial parameters from each implementation's final parameters and normalize their difference by the norm of the PyTorch updates. This measures agreement in the updates themselves without letting the scale of the initial parameters dominate the denominator. Relative update differences are at most 3.35×10^{-5}, and final parameter differences are at most 4.54×10^{-6}. The training-state comparisons thus support the closely aligned loss trajectories across the three models.

Table 13: Relative L_2 differences in training states after 100 steps.

GPU	Model	Parameters	Updates	Optimizer state	EMA
A100	eSEN	3.13e-07	1.70e-05	1.51e-08	3.16e-08
A100	UMA	3.39e-06	2.50e-05	4.34e-09	1.49e-07
A100	EquiformerV3	5.05e-09	1.16e-05	1.64e-08	8.77e-10
H100	eSEN	2.92e-07	1.59e-05	1.51e-08	2.98e-08
H100	UMA	4.53e-06	3.34e-05	6.28e-09	1.54e-07

GPU	Model	Parameters	Updates	Optimizer state	EMA
H100	EquiformerV3	5.15e-09	1.18e-05	1.57e-08	8.92e-10

C.5.4 LONG TRAINING TRAJECTORIES

To examine agreement beyond the 100-step comparison, we train eSEN for 40,000 steps on each GPU, using one run per implementation, FP32, and eight structures per batch. Each pair shares its initial state and batch sequence. Figure 6(b) shows non-overlapping 200-step loss means and enlarges the final 10,000 steps. Table 14 reports the initial loss before the first parameter update and the final loss averaged over steps 35,001–40,000.

Training time covers the full 40,000-step loop, including per-step trajectory recording, but excludes initialization and final result saving. Peak allocated memory is measured over all 40,000 steps. The longer run encounters larger samples, leading to higher peak memory than the short benchmark. Nevertheless, the approximately $1.97\times$ memory reduction remains consistent with the short-run reduction of approximately $1.96\times$. The final losses remain close on both GPUs, consistent with the trajectories in Figure 6(b). The final-loss differences between TrimSO(2) and PyTorch are 1.06% on A100 and 0.69% on H100, both smaller than the approximately 2% difference between the PyTorch runs on the two GPUs (1.5584 versus 1.5282). Over these same long runs, TrimSO(2) achieves $2.06\times$ and $1.95\times$ training speedups on A100 and H100, with $1.97\times$ and $1.97\times$ reductions in peak memory, respectively. The long comparison therefore retains both numerical agreement and the training-efficiency benefits.

Table 14: Loss, training time, and peak allocated memory for 40,000 eSEN steps. Final loss is the mean over the last 5,000 steps. Time is in hours and memory is in GiB.

GPU	Implementation	Initial loss	Final loss	Time	Memory	Speedup	Memory reduction
A100	PyTorch	19.5021	1.5584	7.08	58.21		
A100	TrimSO(2)	19.5021	1.5419	3.44	29.54	**2.06**×	**1.97**×
H100	PyTorch	19.5021	1.5282	4.56	58.25		
H100	TrimSO(2)	19.5021	1.5387	2.34	29.59	**1.95**×	**1.97**×