FLASHCART: FAST CARTESIAN TENSOR PRODUCTS FOR EQUIVARIANT INTERATOMIC POTENTIALS

Anonymous authors
Paper under double-blind review

ABSTRACT

Machine-learned interatomic potentials extend atomistic simulations beyond the length- and timescales accessible to electronic-structure methods. However, the computational cost of equivariant architectures limits the local correlations they can represent in practice and therefore their achievable accuracy. Here we introduce FlashCart, which makes higher-order correlations affordable by combining generated GPU kernels with an architecture that recursively builds equivariant features and compresses them to a fixed width at each step. We express tensor products in independent Cartesian components and symbolically simplify them and their derivatives, producing fused kernels that often outperform optimized spherical counterparts. We then show that increasing correlation order improves accuracy more efficiently than increasing width, depth, or tensor rank. On SPICE-MACE-OFF, FlashCart models advance the measured accuracy–efficiency frontier: a model with 5.6 million parameters achieves lower energy and force errors and $10\times$ faster inference than a transformer with 189 million parameters.

1 INTRODUCTION

Machine-learned interatomic potentials (MLIPs) enable molecular and materials simulations at length- and timescales inaccessible to electronic-structure methods, but their practical application depends on the cost of repeated energy and force evaluations. Equivariant models such as NequIP [1] and MACE [2] describe each atom's neighborhood using features that transform under rotations as scalars, vectors, or higher-rank tensors and are coupled through equivariant products [3–9]. These products can also be applied repeatedly to capture increasingly complex relationships among neighboring atoms, building higher-order correlations as in the atomic cluster expansion [10]. Higher orders can improve accuracy but increase evaluation cost. Methods like the Point Edge Transformer (PET) instead relax explicit rotational constraints [11], with recent models achieving state-of-the-art accuracy–efficiency tradeoffs [12, 13]. Making higher-order correlations affordable offers a complementary route to improved accuracy–efficiency tradeoffs while retaining explicit equivariance.

Higher-order correlations can be built by successively combining the same input features in different ways, called coupling paths. Retaining all these paths increases the number of intermediate features, so compressing them between successive correlation orders controls this growth [14]. The cost of individual products can also be reduced by fusing operations in GPU kernels to limit intermediate storage and memory traffic, as shown for spherical tensors [15–18]. Cartesian approaches provide an alternative to spherical tensor representations [8, 9], with Ref. 9 identifying dedicated GPU kernels as a promising direction. Such kernels can use the $2l + 1$ independent components of a rank-l irreducible Cartesian tensor, matching the dimension of its spherical counterpart [19]. Writing the products in these components gives polynomial expressions that can be simplified and differentiated symbolically before compilation. Further details on related work are provided in Appendix A.

We introduce FlashCart, a symbolic Cartesian tensor framework that combines generated GPU kernels with a recursive equivariant architecture. Specifically, (i) we express tensor products and their derivatives in independent Cartesian components and automatically generate fused GPU kernels for node-local products and neighbor aggregation (Sections 3 and 4). These kernels often outperform optimized spherical counterparts in our benchmarks. (ii) We build higher-order correlations through repeated products, mixing paths and channels back to fixed width at each step (Section 5). Each additional order requires one product and one linear map, so the correlation cost grows linearly at fixed

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.

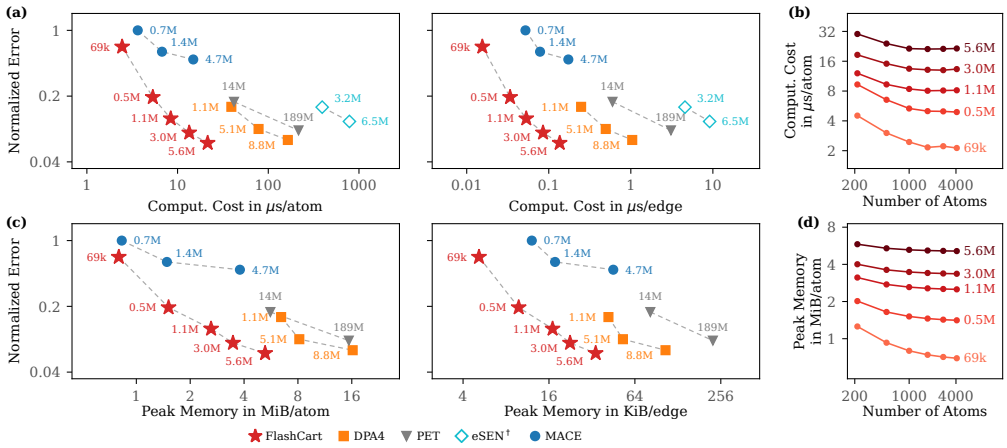

Figure 1: **Accuracy–efficiency tradeoff evaluated on SPICE-MACE-OFF.** (a,c) Normalized error as a function of evaluation time and peak GPU memory, with FlashCart models on the measured accuracy–efficiency frontier. Local evaluation costs in (a,c) use a 1000-atom diamond cell. (b,d) Evaluation time and peak GPU memory per atom as a function of system size, reaching a plateau at smaller system sizes for larger FlashCart models. †All timings use an NVIDIA RTX PRO 6000 Blackwell (96 GB), except for eSEN literature values measured on an NVIDIA A100 (80 GB) [23].

width and maximum tensor rank. (iii) On SPICE-MACE-OFF [20, 21], we show that increasing correlation order improves accuracy more efficiently than increasing width, depth, or tensor rank. The resulting FlashCart models lie on the measured accuracy–efficiency frontier (Figure 1), while results on MAD [22] support their use as competitive general-purpose interatomic potentials.

2 Background

Machine-learned interatomic potentials. MLIPs approximate electronic-structure solutions by a parametric map from an atomic configuration $\mathcal{S} = \{\boldsymbol{r}_u, Z_u\}_{u=1}^{N_{\text{at}}}$, defined by atomic positions and species, to its potential energy. Under periodic boundary conditions, it also includes a cell, and local neighborhoods include periodic images obtained by translating atoms by integer combinations of the cell vectors. The model writes the energy as a sum of atomic contributions, $E = \sum_u E_u$, built from finite-cutoff neighborhoods. In the absence of external fields, E is invariant under translations, global rotations and reflections $\boldsymbol{R} \in O(3)$ acting on the positions and cell, and permutations of atoms of the same species. Forces are obtained as $\boldsymbol{F}_u = -\nabla_{\boldsymbol{r}_u} E$ and transform as $\boldsymbol{F}_u \mapsto \boldsymbol{R}\boldsymbol{F}_u$. Deriving them from a scalar energy ensures a conservative force field and energy-conserving dynamics. MLIPs are usually trained on reference energies and forces, which requires differentiating the forces with respect to model parameters. Each operation in the energy computation must therefore support a vector–Jacobian product (VJP) for force evaluation that can itself be differentiated during training. We refer to these two derivative operations as the VJP and the VJP-of-VJP.

Equivariant convolution and many-body correlations. Architectures that build many-body correlations combine equivariant convolutions, which collect information from neighboring atoms, with repeated node-local products of the aggregated features [2, 5, 8]. For each atom u, $\mathcal{N}(u)$ contains all atoms and periodic images within the cutoff $r_{\max}$, excluding the central atom in the reference cell. The edge vector $\boldsymbol{r}_{uv}$ points from atom u to its neighbor v, including any periodic translation, with length r_{uv} and unit direction $\hat{\boldsymbol{r}}_{uv} = \boldsymbol{r}_{uv}/r_{uv}$. We write $\boldsymbol{T}_l(\hat{\boldsymbol{r}}_{uv})$ for the rank-l direction tensor of an edge and $\boldsymbol{h}_{vl}^{(t)}$ for the d feature channels at v. The edge directions, product intermediates, and propagated node states can retain different maximum ranks, denoted $l_{\max}^{\text{edge}}$, $l_{\max}^{\text{prod}}$, and $l_{\max}^{\text{prop}}$.

The convolution couples $\boldsymbol{h}_{vl_1}^{(t)}$ with $\boldsymbol{T}_{l_2}(\hat{\boldsymbol{r}}_{uv})$ along admissible paths $(l_1, l_2) \to l$, applies distance-dependent weights $\boldsymbol{W}_{l_1 l_2 l}^{(t)}(r_{uv})$, and sums over $v \in \mathcal{N}(u)$ to form the atomic basis $\boldsymbol{A}_u = \bigoplus_l \boldsymbol{A}_{ul}$. The node-local products are evaluated recursively as $\boldsymbol{\phi}_u^{(0)} = 1$ and $\boldsymbol{\phi}_u^{(q)} = \mathcal{W}^{(q)} \Pi_{\le l_q}(\boldsymbol{\phi}_u^{(q-1)} \hat{\otimes} \boldsymbol{A}_u)$

for $q = 1, \ldots, \nu$, where the projection $\Pi_{\leq l_q}$ discards ranks above l_q, the linear maps $\mathcal{W}^{(q)}$ are equivariant, and every step contributes to the layer output. The state $\boldsymbol{\phi}_u^{(q)}$ has polynomial degree q in $\boldsymbol{A}_u$. If each neighbor contribution to $\boldsymbol{A}_u$ depends only on the central atom and that neighbor, these products have body order at most $q+1$, including the central atom [2, 10]. Section 5 specifies the cutoffs l_q and maps $\mathcal{W}^{(q)}$, which define the fixed-width ladder in this work. Both operations use the same tensor products, but convolution sums contributions from neighbors, whereas the recurrence acts independently at each atom. This distinction motivates the two kernel designs in Section 4. Further details on the edge convolution and node-local products are provided in Appendix B.

Irreducible Cartesian tensors. For $l \geq 0$, $\mathcal{V}_l \subset (\mathbb{R}^3)^{\otimes l}$ denotes the space of symmetric trace-free (STF) Cartesian tensors of rank l. These tensors are invariant under every permutation of their indices and, for $l \geq 2$, have vanishing traces. Under $\boldsymbol{R} \in O(3)$, a tensor transforms as $\boldsymbol{x}_l \mapsto \boldsymbol{R}^{\otimes l}\boldsymbol{x}_l$, with $\boldsymbol{R}$ acting on each index. Inversion, i.e., $\boldsymbol{R} = -\boldsymbol{I}$, therefore contributes one factor of -1 per index, giving $\mathcal{V}_l$ natural parity of $(-1)^l$. A totally symmetric rank-l tensor has $(l+1)(l+2)/2$ independent components, tracelessness imposes $l(l-1)/2$ conditions, and the difference $2l+1$ is the dimension of $\mathcal{V}_l$. Real spherical tensors of degree l and rank-l STF Cartesian tensors represent the same $O(3)$ irrep and are related by a fixed invertible equivariant map (Appendix B).

For $|l_1 - l_2| \leq l_3 \leq l_1 + l_2$, the decomposition of $\mathcal{V}_{l_1} \otimes \mathcal{V}_{l_2}$ contains a single irreducible component labelled by l_3, with parity $(-1)^{l_1+l_2}$, which is natural exactly when $J = l_1 + l_2 + l_3$ is even. We call such paths $(l_1, l_2) \to l_3$ admissible. For an admissible path, the equivariant bilinear maps $\mathcal{V}_{l_1} \times \mathcal{V}_{l_2} \to \mathcal{V}_{l_3}$ form a one-dimensional space. Up to normalization, the spherical and Cartesian products therefore express the same coupling in different coordinates [24, 25]. The representation thus affects the arithmetic and memory accesses needed to evaluate the coupling, but not its expressive power. Following prior work [8, 26], we write the Cartesian product along an admissible path as $(\boldsymbol{x}_{l_1} \hat{\otimes}_{\mathrm{Cart}} \boldsymbol{y}_{l_2})_{l_3}$. It contracts $n = (l_1 + l_2 - l_3)/2$ index pairs between the inputs, then symmetrizes the remaining l_3 indices and removes traces to obtain a rank-l_3 STF tensor (Appendix B). The next section expresses the natural-parity couplings in the $2l+1$ independent components.

3 Generating Cartesian tensor products

We automatically generate GPU kernels for the Cartesian tensor products described in Section 2 and their derivatives. For a chosen maximum tensor rank, we derive explicit symbolic expressions in independent components and translate them into Triton code [27] for parallel execution on a GPU.

Symbolic construction. For each rank-l tensor, we choose the $2l+1$ independent components containing at most one z index, following Ref. 19 (Figure C1(a)). Because the tensor is symmetric, each component is determined by the number of x, y, and z indices. We denote these counts by p_x, p_y, and p_z, with $p_x + p_y + p_z = l$, and write the component as $(\boldsymbol{x}_l)_{p_x p_y p_z}$. The remaining components follow from the trace relation $(\boldsymbol{x}_l)_{p_x p_y p_z} = -(\boldsymbol{x}_l)_{p_x+2, p_y, p_z-2} - (\boldsymbol{x}_l)_{p_x, p_y+2, p_z-2}$ for $p_z \geq 2$. Repeated application of this trace relation expresses every component of the rank-l tensor through its $2l+1$ independent components. Our generator reconstructs all 3^l tensor components only symbolically, while model evaluation reads and writes the independent components.

For each admissible path $(l_1, l_2) \to l_3$, the generator reconstructs both inputs symbolically from their independent components and evaluates the Cartesian coupling described in Section 2. It computes their outer product, contracts $n = (l_1 + l_2 - l_3)/2$ index pairs between the inputs, and symmetrizes the remaining l_3 indices before removing traces. A path-dependent factor matches the normalization of the Cartesian product, accounting for the difference between averaging over index permutations and summing distinct index placements (Appendix C). For every admissible path up to the chosen maximum tensor rank, the generator selects the $2l_3+1$ independent components of the resulting tensor for code generation. Since these couplings are symmetric under exchange of the inputs, paths related by exchanging input ranks share one expression with exchanged operands.

Each independent output component is a bilinear polynomial in the independent input components with rational coefficients. For example, the $(1,1) \to 2$ path gives $\left[(\boldsymbol{x}_1 \hat{\otimes}_{\mathrm{Cart}} \boldsymbol{y}_1)_2\right]_{xx} = x_x y_x - (x_y y_y + x_z y_z)/2$. The generated code embeds the coefficients directly.

Symbolic derivatives. Training on forces requires differentiating through the VJP used to compute them. The polynomial expressions obtained above make both derivative operations accessible

symbolically. For one coupling path, let $\boldsymbol{f}(\boldsymbol{x}, \boldsymbol{y})$ denote the independent components of the unweighted output. We use SymPy to expand these expressions and differentiate them with respect to both inputs, obtaining explicit input VJPs. The tensor products used in the model also apply a scalar weight to each path and feature channel, so we require derivatives of the weighted contribution $\boldsymbol{c}(w, \boldsymbol{x}, \boldsymbol{y}) = w\boldsymbol{f}(\boldsymbol{x}, \boldsymbol{y})$. Its input VJPs are scaled by w, while its weight VJP is the inner product of the incoming gradient with $\boldsymbol{f}(\boldsymbol{x}, \boldsymbol{y})$. Because $\boldsymbol{f}$ is bilinear, differentiating this weighted VJP reuses the forward and input-VJP expressions, together with the terms arising from the weight dependence. Appendix C gives the resulting VJP and VJP-of-VJP formulas.

We embed these expressions in the Triton kernels described in Section 4. The kernels apply the scalar weights, optionally sum paths with the same output rank, and handle the input and output arrays for node-local products and convolutions over neighbors. All symbolic expansion and differentiation are completed before compilation. To verify the generated derivatives, we also generate a PyTorch forward implementation whose derivatives are computed by automatic differentiation.

Cost of the generated expressions. Expressing tensor products in independent components substantially reduces the arithmetic required compared with evaluating all 3^l Cartesian components. Counting additions and multiplications before compiler transformations, the operation count summed over all distinct couplings up to rank 3 decreases from 1,299 to 615, close to the 609 operations required by sparse Clebsch–Gordan coupling. Through ranks 4 and 5, the independent-component expressions require 1.3–1.5 times as many operations as their spherical counterparts (Figure C1(c)). The resulting Cartesian expressions therefore require a comparable number of arithmetic operations to sparse spherical coupling at the tested ranks. However, evaluation time also depends on memory access and kernel design, which we address in the next section.

4 Executing Cartesian tensor products

The generated Triton kernels evaluate the symbolic expressions of Section 3 for node-local tensor products and edge convolutions. For node-local products, each independently scheduled kernel instance, called a program, processes a group of atoms. The same implementation supports edge convolutions by gathering inputs along edges and atomically adding their contributions to receivers. Our default receiver-sorted kernel instead assigns each program a receiver, a tile of feature channels, and a group of output ranks. The program accumulates contributions from all incoming edges before storing the result, avoiding atomic additions in the forward pass. Here we focus on this kernel and adapt it for derivative evaluation, using atomic additions when multiple programs contribute to the same gradient. Appendix D outlines the implementation of the node-local kernels.

Data layout. Each row of an equivariant array contains the features of a node or an edge, with a contiguous block for each tensor rank. Within the block for tensor rank l, entries are ordered by the independent Cartesian component and then by the coupling path, with the feature channels for each pair stored contiguously. The blocks follow increasing tensor rank, so retaining only lower ranks requires no rearrangement or copying of the features. Figure D1 illustrates this layout, and Appendix D specifies how its entries are addressed in memory.

Receiver-sorted forward. For an edge e from sender v_e to receiver u_e, the kernel combines the projected sender features $\boldsymbol{x}_{v_e k}$ with the direction tensor $\boldsymbol{y}_e$, shared across channels. With a scalar weight $w_{e\eta k}$ for each edge, coupling path η, and channel k, the output is

$$\boldsymbol{z}_{u\eta k} = \sum_{e:u_e=u} w_{e\eta k}\, \boldsymbol{f}_\eta(\boldsymbol{x}_{v_e k}, \boldsymbol{y}_e), \tag{1}$$

where $\boldsymbol{f}_\eta$ is the unweighted Cartesian product for the input and output ranks selected by η. This is the convolution of Section 2 with separate outputs for each path. The neighbor normalization and channel mixing with $\boldsymbol{W}^{(t)}$ remain outside the kernel.

We store edges in receiver order, with a row-pointer array locating each receiver's incoming edges and a sender array storing each edge's source-atom index. Each program evaluates the generated products over all incoming edges of one receiver for a channel tile $\mathcal{K}$ and a group of output ranks $\mathcal{G}$. It accumulates the sums locally and stores each once, without atomic additions or an intermediate tensor of edge messages. Separate kernel launches compute different groups of output ranks, limiting the number of sums accumulated simultaneously. Algorithm 1 describes the forward convolution, with $\mathcal{P}_\mathcal{G}$ denoting all coupling paths that produce tensor ranks in $\mathcal{G}$. Figure 2 compares

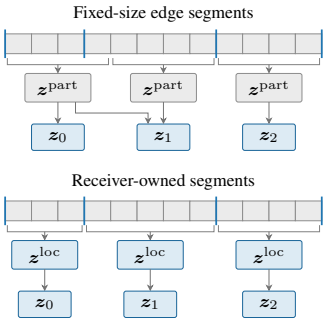

Figure 2: **Accumulation over receiver-sorted edges.** Blue lines separate edges of different receivers. Fixed-size segments can split a receiver's sum across programs, while assigning all its edges to one program allows the sum to be computed locally and stored once.

Algorithm 1 Receiver-sorted forward kernel

Require: Receiver-sorted arrays `rowptr` and `sender`
Require: Tensor inputs $\boldsymbol{x}$, $\boldsymbol{y}$ and scalar weights w
Require: Rank groups $\mathcal{G}$, paths $\mathcal{P}_{\mathcal{G}}$, and channel tiles $\mathcal{K}$

for each rank group $\mathcal{G}$ **do**
  for all receivers u and channel tiles $\mathcal{K}$ **do**
    Load $e_0, e_1 \leftarrow \texttt{rowptr}[u], \texttt{rowptr}[u+1]$
    Initialize $\boldsymbol{z}^{\text{loc}}_{\eta k} \leftarrow \mathbf{0}$ for $\eta \in \mathcal{P}_{\mathcal{G}}, k \in \mathcal{K}$
    for $e = e_0, \ldots, e_1 - 1$ **do**
      Load $v \leftarrow \texttt{sender}[e]$, $\boldsymbol{x}_{v\mathcal{K}}$, and $\boldsymbol{y}_e$
      for $\eta \in \mathcal{P}_{\mathcal{G}}, k \in \mathcal{K}$ **do**
        Load $w_{e\eta k}$
        $\boldsymbol{z}^{\text{loc}}_{\eta k} \mathrel{+}= w_{e\eta k} \boldsymbol{f}_\eta(\boldsymbol{x}_{vk}, \boldsymbol{y}_e)$
      end for
    end for
    Store $\boldsymbol{z}_{u\eta k} \leftarrow \boldsymbol{z}^{\text{loc}}_{\eta k}$ for $\eta \in \mathcal{P}_{\mathcal{G}}, k \in \mathcal{K}$
  end for
end for

assigning all incoming edges of a receiver to one program with the fixed-size edge segments used by OpenEquivariance's deterministic forward kernel [16].

Receiver-sorted derivatives. The VJP processes the same receiver-sorted edges and loads the receiver's incoming gradient $\boldsymbol{g}_{u\eta k}$ once per program. For each edge, it evaluates the generated input VJPs and computes the weight gradient by taking the inner product of $\boldsymbol{g}_{u\eta k}$ with the unweighted output $\boldsymbol{f}_\eta(\boldsymbol{x}_{v_e k}, \boldsymbol{y}_e)$. Several receivers may contribute to the same sender gradient, while the direction-tensor gradient receives contributions from multiple feature channels and coupling paths. We accumulate shared input gradients using atomic additions, but store each weight gradient directly because each weight belongs to a single edge, coupling path, and feature channel. This also allows the VJP to divide a receiver's incoming edges into chunks processed by several programs, increasing parallelism for receivers with many neighbors without a separate step to combine partial receiver sums.

The VJP-of-VJP processes the same receiver-sorted edges using the generated forward and input-VJP expressions. Each program accumulates contributions from all incoming edges of receiver u to compute $\widehat{\boldsymbol{g}}_{u\eta k}$, the gradient with respect to the incoming gradient $\boldsymbol{g}_{u\eta k}$, and stores the sum once. As in the VJP, we accumulate shared input gradients $\widehat{\boldsymbol{x}}$ and $\widehat{\boldsymbol{y}}$ using atomic additions, but store each weight gradient $\widehat{w}$ directly. Appendix D describes both derivative algorithms, including VJP edge chunking and additional kernel launches at higher tensor ranks.

Comparison with other kernels. In our benchmarks, we use OpenEquivariance's atomic mode [16], which accumulates contributions to each receiver atomically in the forward pass. Its backward pass loads the receiver gradient for each edge and, like FlashCart, accumulates sender-feature gradients atomically. FlashCart groups edges by receiver, accumulating contributions in each forward program before storing the result and reusing each loaded receiver gradient across all edges in a VJP program. In OpenEquivariance's deterministic mode, fixed-size forward edge segments can split a receiver's sum across warps, requiring temporary storage and a second kernel to combine partial sums (Figure 2). Its backward pass uses an additional permutation to order edges by sender, which must be updated when the neighbor list changes [16]. Sobek likewise orders edges by sender to accumulate sender-feature gradients deterministically [18]. FlashCart retains receiver ordering throughout, but its atomic gradient accumulation does not guarantee bitwise reproducibility.

5 Model Architecture

FlashCart consists of T message-passing blocks, each containing an equivariant convolution followed by a node-local product ladder. Here we describe the ladder and its channel mixing, with details of the convolution and the remaining architecture provided in Appendix E.

Factorized fixed-width correlations. The recursive construction of Section 2 leaves open which ranks are retained after each product and how the channels are mixed. We propose a fixed-width

ladder using channel-diagonal Cartesian products, which couple only input features with the same channel index. A learned linear map then mixes paths and channels to produce d channels per retained tensor rank. Each additional correlation order requires one pairwise product and one linear map, using the generated Cartesian kernels without a separate higher-order tensor-product operator.

We initialize the ladder from the atomic basis $\boldsymbol{A}_{ukl}$ as $\boldsymbol{\phi}^{(1)}_{ukl} = \sum_{k'} W^{(1)}_{l,kk'} \boldsymbol{A}_{uk'l}$, where the learned $d \times d$ map $W^{(1)}$ mixes channels independently at each tensor rank. For $q = 2, \ldots, \nu$, each step applies a Cartesian product followed by learned channel mixing,

$$\boldsymbol{B}^{(q)}_{u\eta_q kl} = \left(\boldsymbol{\phi}^{(q-1)}_{ukl_1} \hat{\otimes}_{\mathrm{Cart}} \boldsymbol{A}_{ukl_2}\right)_l, \quad \boldsymbol{\phi}^{(q)}_{ukl} = \sum\nolimits_{\eta_q, k'} W^{(q)}_{l,k(\eta_q k')} \boldsymbol{B}^{(q)}_{u\eta_q k'l}, \qquad l \le l_q. \tag{2}$$

Here $\eta_q = (l_1, l_2 \to l)$ is an admissible coupling path, and the cutoff l_q specifies the highest output rank retained at step q. For $q < \nu$, each step retains tensor ranks up to $l_q = l^{\mathrm{prod}}_{\max}$. The last step retains ranks up to $l_\nu = l^{\mathrm{out}}_{\max}$, which equals $l^{\mathrm{prop}}_{\max}$ in all blocks except the final one, where it is zero and only scalars are returned. Summing the contributions from all steps gives $\boldsymbol{z}_u = \sum_{q=1}^{\nu} \Pi_{\le l^{\mathrm{out}}_{\max}} \boldsymbol{\phi}^{(q)}_u$, where the projection retains only the tensor ranks required at the ladder output. Each product introduces another factor of $\boldsymbol{A}$, making $\boldsymbol{\phi}^{(q)}$ a polynomial of degree q in the basis. Because normalization and nonlinear gating make $\boldsymbol{A}$ depend nonlinearly on the neighborhood, q specifies the polynomial degree in $\boldsymbol{A}$, not a bound on the number of interacting neighbors.

In Equation 2, the product couples channel k of $\boldsymbol{\phi}^{(q-1)}$ with channel k of $\boldsymbol{A}$. The learned map $W^{(q)}$ then mixes the resulting paths and channels separately for each output tensor rank. With P_q retained paths, this gives $P_q d^2$ learned parameters per step, compared with $P_q d^3$ for a dense map combining products between all pairs of input channels. Each step returns d channels per retained tensor rank, allowing the ladder to build higher-order correlations without increasing its channel width. The ladder's learned mixing maps $W^{(q)}$ contain $d^2 \sum_{q=1}^{\nu} P_q$ parameters in total. For the initialization, P_1 is the number of retained tensor ranks, each with its own $d \times d$ map. With the maximum tensor ranks held fixed, both the parameter count and the computational cost per atom grow quadratically with the channel width d and linearly with the correlation degree ν.

Our node-local correlation ladder combines compression between successive correlation orders, as in NICE and TRACE [14, 28], with channel-diagonal products followed by dense learned mixing, as used in Cormorant and Allegro [3, 29]. Our ladder reuses the same atomic basis at every step. Expanding this recurrence gives a tensor-train factorization of the polynomial coefficients for each retained sequence of coupling paths, with the d channels passed between steps connecting its factors (Appendix E). We also evaluate path-reduced products that sum weighted paths at each step and mix channels only after the recurrence. The smallest model in Figure 1 uses these products, which are also evaluated alongside the full ladder in the node-local kernel benchmark in Section 6.

6 Results and discussion

We benchmark the generated kernels, examine how architectural choices affect the tradeoff between accuracy and computational cost, and compare FlashCart models with state-of-the-art potentials on the SPICE-MACE-OFF data set [21]. We also evaluate a FlashCart model trained on the MAD data set [22] across a wider range of elements and atomic environments, comparing its accuracy with PET-MAD [12], a state-of-the-art transformer-based potential. We measure runtime on an NVIDIA RTX PRO 6000 Blackwell GPU (96 GB) and report FlashCart prediction errors averaged over three independent runs. Appendix F provides further details on the datasets, training, and benchmarks.

Kernel microbenchmarks. Figure 3(a) compares edge convolutions with 1024 nodes, 128 feature channels, and 128 neighbors per node as the maximum tensor rank increases from 1 to 5. Over the full benchmark grid of 30 configurations, which also includes 64 and 256 channels and 64 neighbors per node, FlashCart is on average $1.6\times$, $1.1\times$, and $1.9\times$ faster than cuEquivariance for the forward, VJP, and VJP-of-VJP, respectively. Compared with OpenEquivariance, FlashCart is $1.9\times$, $2.1\times$, and $3.1\times$ faster on average, although OpenEquivariance is faster in some configurations at ranks $\{1, 2\}$. The largest average improvement occurs in the VJP-of-VJP used to differentiate the force loss. In this phase, FlashCart also uses about half the peak GPU memory of either baseline. These timings compare the complete kernel implementations, including memory access and the aggregation of

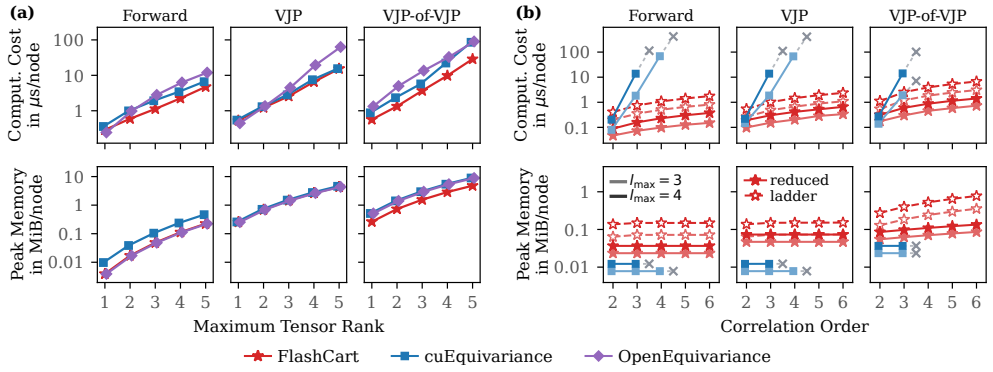

Figure 3: **Runtime and memory of edge convolutions and node-local tensor products.** (a) Results for edge convolutions with 1024 nodes and 128 edges per node as a function of the maximum tensor rank. (b) Results for node-local tensor products with 4096 nodes as a function of the correlation order. Solid FlashCart curves show path-reduced products that sum contributions within the kernel, with a scalar weight per path and feature channel. Dashed curves show the full ladder, including the linear maps that mix paths and channels between steps. cuEquivariance evaluates symmetric contractions with a learned weight for each contraction term and feature channel. Both implementations use the indicated maximum rank for their input and output tensors. Grey crosses denote cases where contraction construction was not completed, or a kernel failed to launch (Appendix G). Dashed extensions indicate trends beyond the available measurements. All results are averaged over three independent runs and use 128 feature channels and single precision.

edge contributions. The operation counts in Section 3 show no arithmetic advantage over sparse spherical coupling and therefore do not explain the observed runtime advantages.

Figure 3(b) compares node-local products with 4096 nodes, 128 feature channels, and maximum tensor ranks 3 and 4. At fixed tensor ranks and width, each increase in correlation order adds one pairwise product to the FlashCart recurrence. The full ladder also applies a learned linear map after each product, whose runtime and memory are included in the measurements. At order 2, path-reduced products have comparable runtime to cuEquivariance's symmetric contraction, while the full ladder is slower. At higher orders, both FlashCart variants become faster than cuEquivariance. At rank 3 and order 4, forward evaluation takes 0.40 ms for path-reduced products and 2.07 ms for the full ladder, compared with 280.61 ms for cuEquivariance, making them approximately $695\times$ and $136\times$ faster, respectively. Furthermore, FlashCart reaches order 6 at every tested rank, while some cuEquivariance contractions could not be constructed or their kernels failed to launch (Appendix G).

We note that reported runtime differences also reflect how the correlations are constructed. cuEquivariance assigns learned weights to complete symmetric contractions, while FlashCart applies weighted products successively, retaining only tensor ranks up to the chosen maximum after each step. Within FlashCart, the full ladder retains separate path contributions for learned mixing, which requires more memory than summing them within the product kernel. Its additional peak GPU allocation reaches 3.1 GiB for the VJP-of-VJP at rank 4 and order 6.

Efficiency of higher-order correlations. We next compare the accuracy and cost of increasing correlation order with those of increasing channel width, the number of message-passing blocks, or maximum tensor ranks (Figure 4). We quantify accuracy using normalized error, which averages the energy and force MAEs after dividing each by a fixed reference value (Appendix F). On the logarithmic axes, the fitted slope describes the relative change in error for a relative change in runtime or memory. For example, a slope of -3 in the runtime panel corresponds to approximately 3% lower error for 1% more runtime along the fitted curve. Across the tested configurations, runtime slopes range from -6.5 to -2.6 for correlation order and from -1.5 to -0.8 for the other architectural choices. Increasing correlation order therefore gives the largest error reduction relative to added runtime. Across configurations with the remaining architecture fixed, raising ν from 2 to 5 reduces normalized error by 19 to 60%, while runtime increases by 4 to 37% and peak GPU memory by at most 1.7%. The steep memory slopes reflect substantial error reductions with little change in mem-

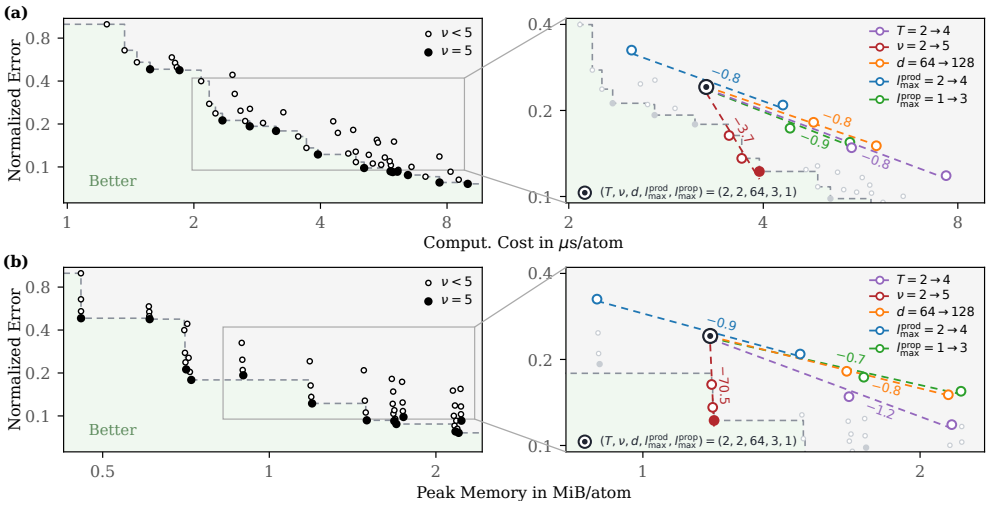

Figure 4: **Accuracy–efficiency tradeoffs across architectural choices.** Normalized error on SPICE-MACE-OFF is shown against runtime (a) and peak GPU memory (b). Filled symbols denote $\nu = 5$, while unfilled ones denote lower correlation orders. The insets show the effect of varying one architectural choice at a time, with the maximum edge and product ranks varied together. Larger negative fitted slopes indicate larger relative error reductions per relative increase in runtime or memory. We average errors over three independent runs and measure runtime and memory for energy and force evaluation on 4096 diamond atoms using one trained model per configuration.

ory use. Consistent with these results, most models on the observed accuracy–efficiency frontier have $\nu = 4$ and 5, for both runtime and memory.

Figure G1 separates the results of Figure 4 by tensor rank, channel width, and number of message-passing blocks, showing how energy error, force error, and runtime vary with correlation order. When both the direction tensors and the intermediate tensors in the correlation ladder are limited to rank 1, raising ν from 2 to 5 lowers energy error by 20% and increases runtime by 4%. At maximum ranks of 4, the corresponding changes are 58% and 37%. Higher tensor ranks thus allow larger relative error reductions from increasing order, at greater computational cost. The relative reductions in energy and force errors from increasing correlation order generally become smaller with more message-passing blocks or a higher maximum propagated rank, but vary little with channel width. Since increasing order also adds parameters, we additionally compare models with approximately 0.93M parameters, reducing channel width as ν increases. At maximum edge and product rank 3, an order-5 model with 64 channels reaches a force MAE of 8.29 meV/Å at 3.94 μs/atom, compared with 11.53 meV/Å at 5.49 μs/atom for order 2 with 108 channels. For maximum edge and product ranks 1 to 4, increasing order while reducing channel width lowers force error by 9 to 29% and runtime by 10 to 34% at approximately the same parameter count.

Model accuracy and computational cost. Figure 1 compares five FlashCart models with state-of-the-art MLIPs using a normalized error that combines energy and force MAEs averaged equally over the seven SPICE test subsets. Table G1 reports the separate energy and force errors for each subset. Among models benchmarked on the same GPU, the FlashCart models lie on the observed accuracy–efficiency frontier for both runtime and memory. The largest FlashCart model has 5.6M parameters and reaches energy and force MAEs of 0.072 ± 0.002 meV/atom and 1.99 ± 0.04 meV/Å. Evaluating energies and forces for 1000 diamond atoms takes 21.5 μs/atom and 5.2 MiB/atom of peak GPU memory. This model improves on PET-L's errors of 0.106 meV/atom and 2.52 meV/Å, despite having far fewer parameters and running approximately ten times faster. It also approaches DPA4-Plus' force MAE of 1.80 meV/Å while improving on its energy MAE of 0.091 meV/atom, with 7.7× faster evaluation and 3.1× less memory. The 5.6M model thus approaches the lowest reported force errors while achieving lower combined error with substantially less runtime and memory.

The timings use each model's own cutoff radius. In diamond, FlashCart processes 158 edges per atom compared with PET-L's 70, giving it an approximately 23× runtime advantage per edge. This

Table 1: **Energy and force errors on the PET-MAD benchmark.** Each energy/force pair gives MAEs in meV/atom and meV/Å, respectively. FlashCart and PET-MAD are trained on MAD for 400 and 1500 epochs with 5.0 and 4.5 Å cutoffs, respectively. Both use reference energies and forces computed at MAD's electronic-structure level. The other baselines use reference calculations consistent with their training data. FlashCart results show the mean and standard deviation across three random seeds with a fixed data split. The lowest energy and force MAEs are shown in bold.

Dataset	FlashCart	PET-MAD [12]	MACE-MP-0-L [30]	MatterSim-5M [31]	Orb-v2 [32]	SevenNet-l3i5 [33]
MAD	**10.2±0.4 / 52.7±1.8**	17.6 / 65.1	81.6 / 181.5	47.3 / 133.7	52.9 / 96.2	82.1 / 173.5
MPtrj	21.1±0.3 / 67.9±0.8	22.3 / 77.9	15.1 / 50.8	21.3 / 61.4	**5.6 / 21.9**	9.8 / 25.5
MatBench	**29.6±0.6** / –	31.3 / –	58.5 / –	38.2 / –	37.9 / –	47.5 / –
Alexandria	45.3±0.8 / 60.9±1.9	49.0 / 66.8	65.4 / 79.5	21.2 / 39.9	**13.2 / 10.5**	47.6 / 70.3
OC2020	**12.8±0.3** / 103.9±1.1	18.3 / 114.5	82.4 / 169.6	31.5 / 119.2	19.8 / **99.3**	45.7 / 162.7
SPICE	**2.9±0.3 / 43.3±1.9**	3.7 / 59.5	10.6 / 166.8	21.3 / 145.6	59.0 / 140.8	11.3 / 139.1
MD22	3.4±0.4 / **54.2±3.9**	**1.9** / 65.6	9.4 / 182.9	28.6 / 160.4	174.3 / 220.7	11.1 / 146.2

normalization accounts for the different numbers of neighbors, although the comparison still includes architectural differences. Runtime per atom also depends on system size, approaching a constant at smaller atom counts for the larger FlashCart models than for the smallest one (Figure 1(b)).

To measure the contribution of the generated kernels, Figure G2 compares them with the corresponding PyTorch implementations of tensor products and edge convolutions, using the same 5.6M model checkpoint. With both implementations compiled, the generated kernels reduce inference time from 175.5 to 21.5 μs/atom and runtime in the training benchmark from 442.6 to 82.2 μs/atom, making them 8.2× and 5.4× faster, respectively. Compilation also accelerates the generated implementation itself, while increasing its peak GPU memory during training by 24% (Appendix G).

Accuracy across materials and molecules. We evaluate a 2.3M-parameter FlashCart model trained on MAD [22], which contains fewer than 100,000 structures spanning 85 elements. Table 1 compares it with the 2.8M-parameter PET-MAD potential and other MLIPs on inorganic materials (MPtrj [34], Matbench Discovery [35], and Alexandria [36]), molecules adsorbed on catalyst surfaces (OC2020 [37]), molecular configurations (SPICE [20] and MD22 [38]), and 360 MAD test structures. FlashCart has lower force MAEs than PET-MAD on all six benchmarks reporting force errors and lower energy MAEs on six of seven benchmarks. Both MAD-trained models have lower energy and force errors on MAD and the molecular benchmarks than the other baselines. On the full MAD test set, FlashCart reaches energy and force MAEs of 11.4 ± 0.1 meV/atom and 63.9 ± 0.5 meV/Å, compared with PET-MAD's 14.7 meV/atom and 72.9 meV/Å (Table G2). For energy and force evaluation on 1000 diamond atoms, FlashCart and PET-MAD take 8.4 and 45.1 μs/atom and use 2.0 and 3.3 MiB/atom of peak GPU memory, respectively. These results establish FlashCart as a competitive architecture for general-purpose potentials, achieving accuracy comparable to state-of-the-art models across diverse materials and molecular environments.

7 Conclusions

FlashCart combines symbolic Cartesian tensor products with generated GPU kernels and a correlation ladder whose cost grows linearly with correlation order at fixed width and maximum tensor rank. These generated kernels often outperform optimized spherical counterparts in our benchmarks and reduce the time required for energy and force evaluation and differentiation of the training loss compared with a compiled PyTorch implementation of the same model. On SPICE-MACE-OFF, FlashCart models lie on the accuracy–efficiency frontier among models benchmarked on the same GPU. We also demonstrate that increasing correlation order improves the accuracy–runtime tradeoff more than increasing width, depth, or tensor rank in the tested configurations. Trading width for higher order also lowers force error and runtime at approximately fixed parameter counts. Finally, MAD results show competitive accuracy across diverse materials and molecular environments.

Limitations. FlashCart uses natural-parity tensor products, which can be extended to opposite parity through Levi–Civita contractions and parity labels. Fixed channel widths and truncated tensor ranks keep the ladder's intermediate features compact but limit the correlations it can represent. Retaining higher ranks also increases the size of the generated kernel code and its compilation cost.

Ethics statement

This work develops computational methods for atomistic simulations using existing datasets. It does not involve human participants or personal data.

Reproducibility statement

Appendix F describes the datasets and explains how we train and evaluate the models and benchmark the kernels. Source code is available at `https://anonymous.4open.science/r/flashcart-EBC7`. All trained FlashCart models will be uploaded to Zenodo upon acceptance.

Acknowledgments

The authors acknowledge the use of OpenAI Codex and Anthropic Claude Code for assistance with code generation and manuscript preparation. Responsibility for the final manuscript and code remains with the authors.

References

[1] S. Batzner, A. Musaelian, L. Sun, M. Geiger, J. P. Mailoa et al.: *E(3)-equivariant graph neural networks for data-efficient and accurate interatomic potentials*. Nat. Commun. **13**, 2453 (2022)

[2] I. Batatia, D. P. Kovács, G. N. C. Simm, C. Ortner, and G. Csányi: *MACE: Higher Order Equivariant Message Passing Neural Networks for Fast and Accurate Force Fields*. Adv. Neural Inf. Process. Syst. **35**, 11423–11436 (2022)

[3] A. Musaelian, S. Batzner, A. Johansson, L. Sun, C. J. Owen et al.: *Learning local equivariant representations for large-scale atomistic dynamics*. Nat. Commun. **14**, 579 (2023)

[4] Y.-L. Liao and T. Smidt: *Equiformer: Equivariant Graph Attention Transformer for 3D Atomistic Graphs*. Int. Conf. Learn. Represent. `https://arxiv.org/abs/2206.11990` (2023)

[5] A. Bochkarev, Y. Lysogorskiy, and R. Drautz: *Graph Atomic Cluster Expansion for Semilocal Interactions beyond Equivariant Message Passing*. Phys. Rev. X **14**, 021036 (2024)

[6] Y. Lysogorskiy, A. Bochkarev, and R. Drautz: *Graph atomic cluster expansion for foundational machine learning interatomic potentials*. npj Comput. Mater. **12**, 114 (2026)

[7] T. Li, W. Li, A. Peng, J. Xue, L. Zhang et al.: *DPA4: Pushing the Accuracy–Cost Frontier of Interatomic Potentials with EMFA SO(2) Convolution*. `https://arxiv.org/abs/2606.02419` (2026)

[8] V. Zaverkin, F. Alesiani, T. Maruyama, F. Errica, H. Christiansen et al.: *Higher-Rank Irreducible Cartesian Tensors for Equivariant Message Passing*. Adv. Neural Inf. Process. Syst. **37**, 124025–124068 (2024)

[9] Q. Chen, A. S. L. S. Pattamatta, B. Wang, D. J. Srolovitz, and M. Wen: *Atomistic machine learning with irreducible Cartesian natural tensors*. Nat. Commun. `https://www.nature.com/articles/s41467-026-77263-z` (2026)

[10] R. Drautz: *Atomic cluster expansion for accurate and transferable interatomic potentials*. Phys. Rev. B **99**, 014104 (2019)

[11] S. N. Pozdnyakov and M. Ceriotti: *Smooth, exact rotational symmetrization for deep learning on point clouds*. Adv. Neural Inf. Process. Syst. **36**, 79469–79501 (2023)

[12] A. Mazitov, F. Bigi, M. Kellner, P. Pegolo, D. Tisi et al.: *PET-MAD as a lightweight universal interatomic potential for advanced materials modeling*. Nat. Commun. **16**, 10653 (2025)

[13] F. Bigi, P. Pegolo, A. Mazitov, J. Schmidt, and M. Ceriotti: *Pushing the limits of unconstrained machine-learned interatomic potentials*. Mach. Learn.: Sci. Technol. **7**, 035051 (2026)

[14] J. Nigam, S. N. Pozdnyakov, and M. Ceriotti: *Recursive evaluation and iterative contraction of N-body equivariant features*. J. Chem. Phys. **153**, 121101 (2020)

[15] M. Geiger, E. Kucukbenli, B. Zandstein, and K. Tretina: *Accelerate Drug and Material Discovery with New Math Library NVIDIA cuEquivariance*. NVIDIA Technical Blog, https://developer.nvidia.com/blog/accelerate-drug-and-material-discovery-with-new-math-library-nvidia-cuequivariance/ (2024)

[16] V. Bharadwaj, A. Glover, A. Buluç, and J. Demmel: *An Efficient Sparse Kernel Generator for O(3)-Equivariant Deep Networks*. Proc. Conf. Appl. Comput. Discrete Algorithms `https://arxiv.org/abs/2501.13986` (2025)

[17] C. W. Tan, M. L. Descoteaux, M. Kotak, G. de Miranda Nascimento, S. R. Kavanagh et al.: *High-performance training and inference for deep equivariant interatomic potentials*. Digit. Discov. **5**, 1558–1567 (2026)

[18] V. Chorošajev and C. Bény: *Sobek: Streaming Equivariant Tensor Product Convolutions*. `https://arxiv.org/abs/2607.18074` (2026)

[19] J. Applequist: *Traceless Cartesian tensor forms for spherical harmonic functions: new theorems and applications to electrostatics of dielectric media*. J. Phys. A: Math. Gen. **22**, 4303 (1989)

[20] P. Eastman, P. K. Behara, D. L. Dotson, R. Galvelis, J. E. Herr et al.: *SPICE, A Dataset of Drug-like Molecules and Peptides for Training Machine Learning Potentials*. Sci. Data **10**, 11 (2023)

[21] D. P. Kovács, J. H. Moore, N. J. Browning, I. Batatia, J. T. Horton et al.: *MACE-OFF: Short-Range Transferable Machine Learning Force Fields for Organic Molecules*. J. Am. Chem. Soc. **147**, 17598–17611 (2025)

[22] A. Mazitov, S. Chorna, G. Fraux, M. Bercx, G. Pizzi et al.: *Massive Atomic Diversity: a compact universal dataset for atomistic machine learning*. Sci. Data **12**, 1857 (2025)

[23] X. Fu, B. M. Wood, L. Barroso-Luque, D. S. Levine, M. Gao et al.: *Learning Smooth and Expressive Interatomic Potentials for Physical Property Prediction*. Int. Conf. Mach. Learn. **267**, 17875–17893 (2025)

[24] J. A. R. Coope: *Irreducible Cartesian Tensors. III. Clebsch–Gordan Reduction*. J. Math. Phys. **11**, 1591–1612 (1970)

[25] A. J. Stone: *Properties of Cartesian–spherical transformation coefficients*. J. Phys. A: Math. Gen. **9**, 485 (1976)

[26] D. R. Lehman and W. C. Parke: *Angular reduction in multiparticle matrix elements*. J. Math. Phys. **30**, 2797–2806 (1989)

[27] P. Tillet, H. T. Kung, and D. Cox: *Triton: An Intermediate Language and Compiler for Tiled Neural Network Computations*. Proc. 3rd ACM SIGPLAN Int. Workshop Mach. Learn. Program. Lang. `https://dl.acm.org/doi/10.1145/3315508.3329973` (2019)

[28] P. Ahlawat: *Transformer Atomic Cluster Expansion: TRACE*. `https://arxiv.org/abs/2607.25652` (2026)

[29] B. Anderson, T. S. Hy, and R. Kondor: *Cormorant: Covariant Molecular Neural Networks*. Adv. Neural Inf. Process. Syst. `https://arxiv.org/abs/1906.04015` (2019)

[30] I. Batatia, P. Benner, Y. Chiang, A. M. Elena, D. P. Kovács et al.: *A foundation model for atomistic materials chemistry*. J. Chem. Phys. **163**, 184110 (2025)

[31] H. Yang, C. Hu, Y. Zhou, X. Liu, Y. Shi et al.: *MatterSim: A Deep Learning Atomistic Model Across Elements, Temperatures and Pressures*. `https://arxiv.org/abs/2405.04967` (2024)

[32] M. Neumann, J. Gin, B. Rhodes, S. Bennett, Z. Li et al.: *Orb: A Fast, Scalable Neural Network Potential*. https://arxiv.org/abs/2410.22570 (2024)

[33] Y. Park, J. Kim, S. Hwang, and S. Han: *Scalable Parallel Algorithm for Graph Neural Network Interatomic Potentials in Molecular Dynamics Simulations*. J. Chem. Theory Comput. **20**, 4857–4868 (2024)

[34] B. Deng, P. Zhong, K. Jun, J. Riebesell, K. Han et al.: *CHGNet as a pretrained universal neural network potential for charge-informed atomistic modelling*. Nat. Mach. Intell. **5**, 1031–1041 (2023)

[35] J. Riebesell, R. E. A. Goodall, P. Benner, Y. Chiang, B. Deng et al.: *A framework to evaluate machine learning crystal stability predictions*. Nat. Mach. Intell. **7**, 836–847 (2025)

[36] J. Schmidt, N. Hoffmann, H.-C. Wang, P. Borlido, P. J. M. A. Carriço et al.: *Machine-Learning-Assisted Determination of the Global Zero-Temperature Phase Diagram of Materials*. Adv. Mater. **35**, 2210788 (2023)

[37] L. Chanussot, A. Das, S. Goyal, T. Lavril, M. Shuaibi et al.: *Open Catalyst 2020 (OC20) Dataset and Community Challenges*. ACS Catal. **11**, 6059–6072 (2021)

[38] S. Chmiela, V. Vassilev-Galindo, O. T. Unke, A. Kabylda, H. E. Sauceda et al.: *Accurate global machine learning force fields for molecules with hundreds of atoms*. Sci. Adv. **9**, eadf0873 (2023)

[39] Y. Lysogorskiy, C. van der Oord, A. Bochkarev, S. Menon, M. Rinaldi et al.: *Performant implementation of the atomic cluster expansion (PACE) and application to copper and silicon*. npj Comput. Mater. **7**, 97 (2021)

[40] R. Kondor, Z. Lin, and S. Trivedi: *Clebsch–Gordan Nets: a Fully Fourier Space Spherical Convolutional Neural Network*. Adv. Neural Inf. Process. Syst. https://arxiv.org/abs/1806.09231 (2018)

[41] J. P. Darby, D. P. Kovács, I. Batatia, M. A. Caro, G. L. W. Hart et al.: *Tensor-reduced atomic density representations*. Phys. Rev. Lett. **131**, 028001 (2023)

[42] I. V. Oseledets: *Tensor-train decomposition*. SIAM J. Sci. Comput. **33**, 2295–2317 (2011)

[43] M. Hodapp and A. Shapeev: *Equivariant tensor network potentials*. Mach. Learn.: Sci. Technol. **5**, 035075 (2024)

[44] J. A. R. Coope, R. F. Snider, and F. R. McCourt: *Irreducible Cartesian Tensors*. J. Chem. Phys. **43**, 2269–2275 (1965)

[45] J. A. R. Coope and R. F. Snider: *Irreducible Cartesian Tensors. II. General Formulation*. J. Math. Phys. **11**, 1003–1017 (1970)

[46] R. F. Snider: *Irreducible Cartesian Tensors*. De Gruyter, Berlin, Boston (2018)

[47] G. Simeon and G. D. Fabritiis: *TensorNet: Cartesian Tensor Representations for Efficient Learning of Molecular Potentials*. Adv. Neural Inf. Process. Syst. **36**, 37334–37353 (2023)

[48] J. Wang, Y. Wang, H. Zhang, Z. Yang, Z. Liang et al.: *E(n)-Equivariant Cartesian Tensor Message Passing Interatomic Potential*. Nat. Commun. **15**, 7607 (2024)

[49] Z. Xu, C. Wu, W. Xie, and P. Hu: *A Cartesian-3j Framework for Machine Learning Interatomic Potentials*. Int. Conf. Mach. Learn. https://arxiv.org/abs/2512.16882 (2026)

[50] I. Batatia, S. Batzner, D. P. Kovács, A. Musaelian, G. N. C. Simm et al.: *The Design Space of E(3)-Equivariant Atom-Centered Interatomic Potentials*. Nat. Mach. Intell. **7**, 56–67 (2025)

[51] C. K. Joshi, C. Bodnar, S. V. Mathis, T. Cohen, and P. Liò: *On the Expressive Power of Geometric Graph Neural Networks*. Int. Conf. Mach. Learn. **202**, 15330–15355 (2023)

[52] Y.-L. Liao, A. J. Hoffman, S. C. Shen, A. Duval, S. W. Norwood et al.: *EquiformerV3: Scaling Efficient, Expressive, and General SE(3)-Equivariant Graph Attention Transformers*. https://arxiv.org/abs/2604.09130 (2026)

[53] M. Weiler, M. Geiger, M. Welling, W. Boomsma, and T. Cohen: *3D Steerable CNNs: Learning Rotationally Equivariant Features in Volumetric Data*. Adv. Neural Inf. Process. Syst. https://arxiv.org/abs/1807.02547 (2018)

[54] V. Zaverkin, D. Holzmüller, I. Steinwart, and J. Kästner: *Fast and Sample-Efficient Interatomic Neural Network Potentials for Molecules and Materials Based on Gaussian Moments*. J. Chem. Theory Comput. **17**, 6658–6670 (2021)

[55] J. Gasteiger, J. Groß, and S. Günnemann: *Directional Message Passing for Molecular Graphs*. Int. Conf. Learn. Represent. https://arxiv.org/abs/2003.03123 (2020)

[56] K. Gubaev, V. Zaverkin, P. Srinivasan, A. I. Duff, J. Kästner et al.: *Performance of two complementary machine-learned potentials in modelling chemically complex systems*. npj Comput. Mater. **9**, 129 (2023)

[57] J. Zeng, D. Zhang, A. Peng, X. Zhang, S. He et al.: *DeePMD-kit v3: A Multiple-Backend Framework for Machine Learning Potentials*. J. Chem. Theory Comput. **21**, 4375–4385 (2025)

[58] A. P. Thompson, H. M. Aktulga, R. Berger, D. S. Bolintineanu, W. M. Brown et al.: *LAMMPS - a flexible simulation tool for particle-based materials modeling at the atomic, meso, and continuum scales*. Comp. Phys. Comm. **271**, 108171 (2022)

[59] A. Johansson, E. Weinberg, C. Trott, M. McCarthy, and S. Moore: *LAMMPS-KOKKOS: Performance Portable Molecular Dynamics Across Exascale Architectures*. Proc. SC '25 Workshops Int. Conf. High Perform. Comput. Netw. Storage Anal. https://doi.org/10.1145/3731599.3767498 (2025)

[60] J. S. Smith, M. Bettencourt, F. Pellegrini, F. Glines, and E. Kucukbenli: *Enabling Scalable AI-Driven Molecular Dynamics Simulations*. NVIDIA Technical Blog, https://developer.nvidia.com/blog/enabling-scalable-ai-driven-molecular-dynamics-simulations/ (2025)

[61] G. J. Martyna, M. L. Klein, and M. Tuckerman: *Nosé–Hoover chains: The canonical ensemble via continuous dynamics*. J. Chem. Phys. **97**, 2635–2643 (1992)

[62] G. J. Martyna, D. J. Tobias, and M. L. Klein: *Constant pressure molecular dynamics algorithms*. J. Chem. Phys. **101**, 4177–4189 (1994)

[63] W. Shinoda, M. Shiga, and M. Mikami: *Rapid estimation of elastic constants by molecular dynamics simulation under constant stress*. Phys. Rev. B **69**, 134103 (2004)

[64] D. R. Lide (Editor): *CRC Handbook of Chemistry and Physics, Internet Version 2005*. CRC Press, Boca Raton, FL (2005)

[65] V. Zaverkin, M. Ferraz, F. Alesiani, and M. Niepert: *Performance of Universal Machine-Learned Potentials with Explicit Long-Range Interactions in Biomolecular Simulations*. J. Chem. Theory Comput. **22**, 2074–2088 (2026)

[66] I. Batatia, W. J. Baldwin, D. Kuryla, J. Hart, E. Kasoar et al.: *MACE-POLAR-1: A Polarisable Electrostatic Foundation Model for Molecular Chemistry*. https://arxiv.org/abs/2602.19411 (2026)

APPENDICES

A RELATED WORK

Product bases and recursive evaluation. The atomic cluster expansion provides a body-ordered polynomial basis. It uses the density trick to evaluate many-body features through products of an atomic basis constructed by one-particle neighbor sums [10]. MACE constructs equivariant many-body features by contracting products of the atomic basis with precomputed generalized Clebsch–Gordan tensors and learned weights [2]. Recursive evaluation builds higher-order products from lower-order intermediates, reusing them to reduce the cost of constructing a product basis [39]. ICTP uses a recursive construction for Cartesian correlations, combining the atomic basis through successive pairwise tensor products [8]. The number of intermediate features can further be limited by compressing the output of each recursive step. NICE uses principal component analysis to reduce the number of equivariant features between successive correlation orders [14], while TRACE incorporates learned mixing within each tensor product, with weights that combine pairs of input channels into the retained output channels [28].

Channel mixing and coefficient factorization. Learned mixing reduces tensor-product outputs to the desired number of channels for each tensor rank, as in Clebsch–Gordan networks [40]. Cormorant combines this mixing with channel-diagonal tensor products, which reduce the computational cost by coupling only matching input channels [29]. Allegro uses this combination in its edge-wise tensor-product layers, with dense mixing over both coupling paths and channels [3]. Channel-diagonal products also arise from factorizations of polynomial coefficients. Tensor-reduced ACE expresses these coefficients as sums of rank-one terms over chemical and radial indices, leading to products that share a channel index, as in MACE [2, 41]. GRACE shows how this shared-index factorization can be written as a tensor train whose intermediate cores preserve that index [5, 42]. Tensor trains also provide a direct parameterization of polynomial coefficients in equivariant tensor network potentials [43].

Irreducible Cartesian tensors. The description of irreducible Cartesian tensors in Section 2 and Appendix B follows classical work on irreducible tensors [19, 24–26, 44–46]. ICTP provides the framework for equivariant message passing with higher-rank irreducible Cartesian tensors that we follow, including its tensor-product conventions [8]. Other Cartesian potentials include TensorNet with rank-two tensors and HotPP with reducible higher-rank tensors [47, 48]. Cartesian-3j implements Cartesian couplings through transformed Wigner-$3j$ coefficients [49], while CarNet provides precomputed projectors for natural tensors, their products, and general tensorial observables [9].

Accelerated tensor-product kernels. For spherical tensor representations, cuEquivariance, OpenEquivariance, and Sobek accelerate tensor products and convolutions with specialized GPU kernels [15, 16, 18]. Optimizations across these libraries include exploiting sparse coupling coefficients and fusing tensor products with neighbor aggregation to reduce intermediate storage and memory traffic. cuEquivariance describes these operations through segmented tensor products, which specify blocks of tensor components and the contractions between them [15]. OpenEquivariance generates specialized GPU kernels for the requested Clebsch–Gordan couplings, incorporating their nonzero coefficients directly into the generated code [16]. The Triton implementation for Allegro evaluates tensor products using only the nonzero Wigner-$3j$ coefficients and their corresponding component indices [17]. Sobek supports dense and channel-wise mixing and reorders the convolution to perform receiver accumulation before channel matrix products, avoiding intermediate edge-weight tensors [18].

B BACKGROUND

Equivariant convolution and many-body correlations. Equivariant architectures differ in the body order represented within a local neighborhood, the angular ranks retained at different layers, the number of feature channels, and their depth [50, 51]. A radial envelope makes every edge contribution vanish smoothly at the cutoff, so the energy remains twice differentiable in the positions.

Suppressing channel indices, the convolution used in this work reads as

$$\boldsymbol{A}_{ul}^{(t)} = \sum_{v\in\mathcal{N}(u)} \sum_{\substack{(l_1,l_2)\to l \\ l_1\le l_{\max}^{\mathrm{prop}},\, l_2\le l_{\max}^{\mathrm{edge}}}} \boldsymbol{W}_{l_1 l_2 l}^{(t)}(r_{uv})\big(\boldsymbol{h}_{vl_1}^{(t)} \hat{\otimes} \boldsymbol{T}_{l_2}(\hat{\boldsymbol{r}}_{uv})\big)_l, \qquad l \le l_{\max}^{\mathrm{prod}}.$$

Here $(l_1, l_2) \to l$ is an admissible coupling path, and the distance-dependent maps $\boldsymbol{W}_{l_1 l_2 l}^{(t)}$ combine radial weighting with channel mixing for each tensor rank. The product $\hat{\otimes}$ denotes equivariant coupling, with $(\cdot)_l$ selecting the output rank. It is evaluated using Clebsch–Gordan coefficients in the spherical basis or the Cartesian contractions defined below [8].

Writing $\boldsymbol{A}_u = \bigoplus_l \boldsymbol{A}_{ul}$, the node-local products are evaluated recursively as

$$\boldsymbol{\phi}_u^{(0)} = 1, \qquad \boldsymbol{\phi}_u^{(q)} = \mathcal{W}^{(q)} \Pi_{\le l_q}\big(\boldsymbol{\phi}_u^{(q-1)} \hat{\otimes} \boldsymbol{A}_u\big), \qquad q = 1, \dots, \nu.$$

The projection $\Pi_{\le l_q}$ discards ranks above l_q, the maps $\mathcal{W}^{(q)}$ are equivariant and linear, and every step contributes to the layer output. Equivariance fixes neither the retained ranks nor the form of these maps. Since $\boldsymbol{\phi}_u^{(q)}$ is degree q in $\boldsymbol{A}_u$, expanding the neighbor sums gives local terms with at most q neighbors. Counting the central atom gives a maximum body order of $q+1$, with lower-order terms when neighbor indices repeat [2, 10]. Stacking T message-passing layers extends the effective interaction range to $Tr_{\max}$.

Irreducible Cartesian tensors. The real spherical tensor space $\mathcal{H}_l \cong \mathbb{R}^{2l+1}$ and the Cartesian STF space $\mathcal{V}_l$ represent the same $O(3)$ irrep. An invertible equivariant map $B_l : \mathcal{H}_l \to \mathcal{V}_l$ therefore exists and is unique up to a nonzero scalar. For each admissible path, the equivariant bilinear coupling is also unique up to normalization, so the two products satisfy [24, 25]

$$(\boldsymbol{x}_{l_1} \hat{\otimes}_{\mathrm{Cart}} \boldsymbol{y}_{l_2})_{l_3} = \kappa_{l_1 l_2 l_3}\, B_{l_3}\big[\big(B_{l_1}^{-1}\boldsymbol{x}_{l_1} \hat{\otimes}_{\mathrm{CG}} B_{l_2}^{-1}\boldsymbol{y}_{l_2}\big)_{l_3}\big],$$

where $\kappa_{l_1 l_2 l_3}$ accounts for the product normalizations and the scaling of B_l.

The direction tensors used in the convolution are normalized STF projections of repeated outer products of the unit direction. Following Refs. 8, 26, we define

$$\boldsymbol{T}_l(\hat{\boldsymbol{r}}_{uv}) = \frac{(2l-1)!!}{l!} \sum_{m=0}^{\lfloor l/2 \rfloor} (-1)^m \frac{(2l-2m-1)!!}{(2l-1)!!} \{\hat{\boldsymbol{r}}_{uv}^{\otimes(l-2m)} \otimes \boldsymbol{I}^{\otimes m}\}.$$

Here $\boldsymbol{I}$ is the identity tensor, $(-1)!! = 1$, and the braces sum over distinct placements of the free indices, leading to $l!/[(l-2m)!\, 2^m\, m!]$ terms. For example, $\{\hat{\boldsymbol{r}} \otimes \boldsymbol{I}\}_{ijk} = \hat{r}_i\delta_{jk} + \hat{r}_j\delta_{ik} + \hat{r}_k\delta_{ij}$. The prefactor normalizes the full contraction of $\boldsymbol{T}_l$ with $\hat{\boldsymbol{r}}_{uv}$ to unity, and the $\boldsymbol{I}$ terms remove traces while preserving rank l. We collect all edge ranks as $\boldsymbol{T}(\hat{\boldsymbol{r}}_{uv}) = \bigoplus_{l=0}^{l_{\max}^{\mathrm{edge}}} \boldsymbol{T}_l(\hat{\boldsymbol{r}}_{uv})$.

The same symmetrization and trace removal define products between two STF tensors. For an admissible path $(l_1, l_2) \to l_3$, $J = l_1 + l_2 + l_3$ is even and contracting $n = (l_1 + l_2 - l_3)/2$ index pairs reduces the total rank to l_3. With $m_{\max} = \min(l_1, l_2) - n$, the bounds $n \ge 0$ and $m_{\max} \ge 0$ are equivalent to the triangle inequalities. For $\boldsymbol{x}_{l_1} \in \mathcal{V}_{l_1}$ and $\boldsymbol{y}_{l_2} \in \mathcal{V}_{l_2}$, the coupling reads [8, 26]

$$(\boldsymbol{x}_{l_1} \hat{\otimes}_{\mathrm{Cart}} \boldsymbol{y}_{l_2})_{l_3} = C_{l_1 l_2 l_3} \sum_{m=0}^{m_{\max}} (-2)^m \frac{(2l_3-2m-1)!!}{(2l_3-1)!!} \{(\boldsymbol{x}_{l_1} \cdot (n+m) \cdot \boldsymbol{y}_{l_2}) \otimes \boldsymbol{I}^{\otimes m}\}. \tag{B1}$$

Here $\boldsymbol{x}_{l_1} \cdot q \cdot \boldsymbol{y}_{l_2}$ contracts q index pairs by summing over each shared Cartesian index. The m-th term contracts $n+m$ pairs and introduces m identity tensors to restore rank l_3. The braces sum over distinct placements of the free indices among the remaining input indices and the identity factors. The constant $C_{l_1 l_2 l_3}$ fixes the path normalization and is defined elsewhere [8].

Paths with odd J require the Levi–Civita symbol and produce outputs with parity opposite to $(-1)^{l_3}$. We restrict this work to natural-parity outputs, but the same symbolic construction extends to odd-J couplings by including Levi–Civita contractions and explicitly tracking parity.

C Generating Cartesian tensor products

Symbolic construction. The generator averages over all $l_3!$ permutations, while the braces in Equation B1 sum over distinct placements. Let $\boldsymbol{Q} = \boldsymbol{x}_{l_1} \cdot n \cdot \boldsymbol{y}_{l_2}$ be the contracted product before trace

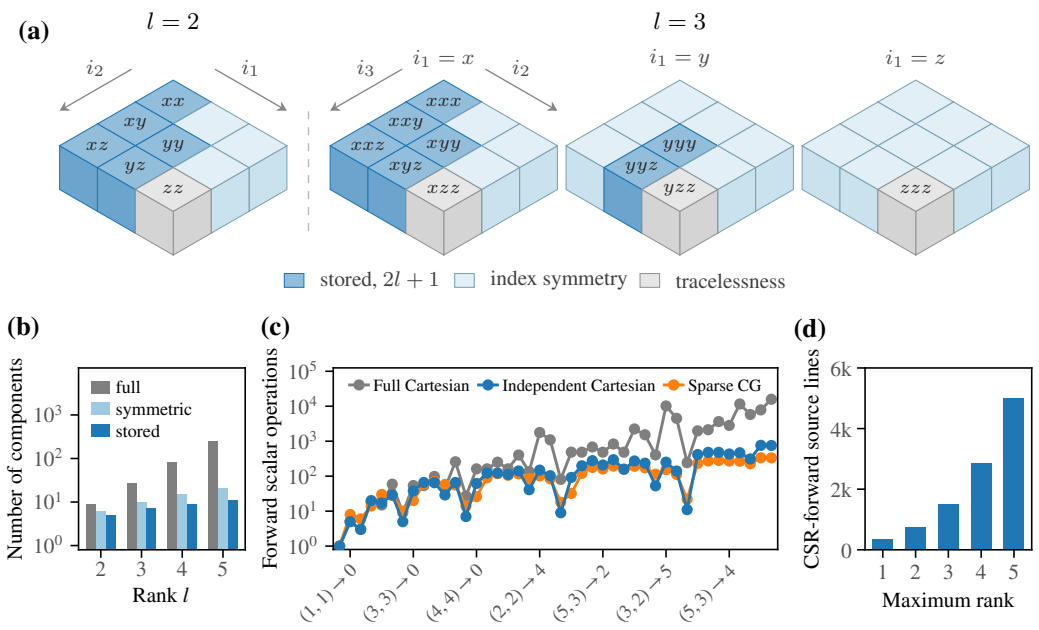

Figure C1: **Independent Cartesian components and the cost of generated tensor products.** (a) Stored components for ranks 2 and 3 are blue, with light blue entries obtained by index symmetry and grey entries from tracelessness. (b) Full, symmetric, and independent component counts as a function of tensor rank. (c) Additions and multiplications for distinct couplings up to rank 5, ordered by $J = l_1 + l_2 + l_3$, before compiler transformations. Independent components reduce Cartesian operation counts to values closer to sparse Clebsch–Gordan coupling. (d) Lines of generated source code for the forward convolution as a function of maximum tensor rank.

removal. Its $l_1 - n$ indices inherited from the first input and its $l_2 - n$ indices inherited from the second input are separately symmetric, so every distinct placement occurs $(l_1 - n)!\,(l_2 - n)!$ times in the full permutation sum

$$\frac{1}{l_3!} \sum_{\pi \in S_{l_3}} \pi \boldsymbol{Q} = \frac{(l_1 - n)!\,(l_2 - n)!}{l_3!} \{\boldsymbol{Q}\}.$$

Here S_{l_3} is the permutation group of the l_3 free indices, and $\pi \boldsymbol{Q}$ permutes these indices. Multiplication by $l_3!/[(l_1 - n)!\,(l_2 - n)!] = \binom{l_3}{l_1 - n}$, where $(l_1 - n) + (l_2 - n) = l_3$, therefore converts the permutation average to the distinct-placement convention, and the generator multiplies each path by $\binom{l_3}{l_1 - n} C_{l_1 l_2 l_3}$, where $C_{l_1 l_2 l_3}$ is the normalization constant in Equation B1.

In Equation B1, the $m > 0$ terms remove traces from $\{\boldsymbol{Q}\}$, with coefficients $(-2)^m (2l_3 - 2m - 1)!!/(2l_3 - 1)!!$ chosen so that the full sum is trace-free. The permutation average differs from $\{\boldsymbol{Q}\}$ only by a scalar factor, and removing traces preserves this relation. The generator therefore applies the factor derived above after trace removal, with no additional normalization.

Symbolic derivatives. For one path and feature channel, consider the weighted product $\boldsymbol{c}(w, \boldsymbol{x}, \boldsymbol{y}) = w \boldsymbol{f}(\boldsymbol{x}, \boldsymbol{y})$, where $\boldsymbol{f}$ contains the independent output components. The scalar w multiplies all $2l_3 + 1$ components. Given an incoming gradient $\boldsymbol{g}$ with respect to $\boldsymbol{c}$, the VJP computes the corresponding gradients $\bar{\boldsymbol{x}}$, $\bar{\boldsymbol{y}}$, and $\bar{w}$ with respect to the inputs and weight. Writing a for an output-component index and b_1, b_2 for the two input-component indices gives

$$\bar{x}_{b_1} = w \sum_a g_a \frac{\partial f_a}{\partial x_{b_1}}, \quad \bar{y}_{b_2} = w \sum_a g_a \frac{\partial f_a}{\partial y_{b_2}}, \quad \bar{w} = \sum_a g_a f_a(\boldsymbol{x}, \boldsymbol{y}). \tag{C1}$$

The weight gradient can also be written as $\bar{w} = \langle \boldsymbol{g}, \boldsymbol{f}(\boldsymbol{x}, \boldsymbol{y}) \rangle$, where $\langle \boldsymbol{g}, \boldsymbol{z} \rangle = \sum_a g_a z_a$ denotes the inner product over the independent output components.

To differentiate the VJP, we denote its unweighted input derivatives by $\mathrm{VJP}_1(\boldsymbol{g}, \boldsymbol{y})$ and $\mathrm{VJP}_2(\boldsymbol{g}, \boldsymbol{x})$, whose components are the sums in Equation C1 without the factor w. Because $\boldsymbol{f}$ is bilinear, the first

Table C1: Operation counts for Cartesian and spherical tensor products.

		Cartesian		Spherical
Maximum rank	Couplings	Full	Independent	Sparse CG
3	15	1,299	615	609
4	27	13,802	2,774	2,141
5	42	79,128	7,505	4,951

depends on $\boldsymbol{y}$ but not $\boldsymbol{x}$, and the second on $\boldsymbol{x}$ but not $\boldsymbol{y}$. Let $\boldsymbol{v}_x$, $\boldsymbol{v}_y$, and v_w be the incoming gradients with respect to $\bar{\boldsymbol{x}}$, $\bar{\boldsymbol{y}}$, and $\bar{w}$. Differentiating the VJP gives gradients with respect to $\boldsymbol{g}$, $\boldsymbol{x}$, $\boldsymbol{y}$, and w, denoted by hats,

$$\begin{aligned}
\widehat{\boldsymbol{g}} &= w\big[\boldsymbol{f}(\boldsymbol{v}_x, \boldsymbol{y}) + \boldsymbol{f}(\boldsymbol{x}, \boldsymbol{v}_y)\big] + v_w \boldsymbol{f}(\boldsymbol{x}, \boldsymbol{y}), \\
\widehat{\boldsymbol{x}} &= \mathrm{VJP}_1(\boldsymbol{g}, w\boldsymbol{v}_y + v_w \boldsymbol{y}), \\
\widehat{\boldsymbol{y}} &= \mathrm{VJP}_2(\boldsymbol{g}, w\boldsymbol{v}_x + v_w \boldsymbol{x}), \\
\widehat{w} &= \langle \boldsymbol{g}, \boldsymbol{f}(\boldsymbol{v}_x, \boldsymbol{y}) + \boldsymbol{f}(\boldsymbol{x}, \boldsymbol{v}_y)\rangle.
\end{aligned} \tag{C2}$$

The contributions to $\widehat{\boldsymbol{x}}$ from differentiating $\bar{\boldsymbol{y}}$ and $\bar{w}$ combine into one evaluation of VJP_1, since this map is linear in its second argument. The same applies to $\widehat{\boldsymbol{y}}$. No $\boldsymbol{v}_x$ term appears in $\widehat{\boldsymbol{x}}$ because $\bar{\boldsymbol{x}}$ is independent of $\boldsymbol{x}$, and likewise for $\widehat{\boldsymbol{y}}$. Finally, $\widehat{w}$ has no v_w term because $\bar{w}$ is independent of w.

To use the generated kernels with PyTorch automatic differentiation, we register the explicit VJP as the derivative of the forward operation and the VJP-of-VJP as the derivative of the VJP. Force training also requires derivatives of the direction tensors described in Appendix B. These tensors are polynomial functions of the direction vector, so a separate generator differentiates their independent components symbolically to provide the corresponding VJP and VJP-of-VJP.

Cost of the generated expressions. Table C1 compares the arithmetic required to evaluate the same set of couplings using full Cartesian components, independent Cartesian components, or sparse Clebsch–Gordan coefficients. For each maximum tensor rank, we sum the operation counts over all admissible paths, counting paths related by exchanging the inputs only once. Figure C1(c) gives the counts for individual couplings. These counts exclude memory accesses and subsequent compiler transformations and therefore do not directly predict evaluation time.

The generated source grows with the maximum tensor rank (Figure C1(d)). Symbolic generation takes seconds, while Triton compilation on first use took a few minutes at rank 4 on our device, after which the compiled kernels are cached and reused.

D Executing Cartesian tensor products

The kernels described in Section 4 evaluate tensor products independently for each feature channel. When both inputs have d feature channels, the kernel couples channel k of one input with channel k of the other. If one input has a single channel, as with direction tensors, it is reused for every channel of the other input. Separate linear maps perform dense mixing over channels and coupling paths. The same kernel implementation supports keeping coupling paths separate or summing their outputs, sharing weights, using symmetric inputs, and skipping unneeded derivative computations. These options are fixed at compile time.

Data layout. An equivariant array X has one row per node or edge and d_X feature channels. A minibatch concatenates the rows of its graphs and adjusts the edge indices. This allows the kernels to process the batch without a separate batch dimension. Each row contains contiguous blocks in increasing tensor rank. Within the block for rank l, entries are ordered by the independent Cartesian component, then by the coupling path, and finally by the feature channel (Figure D1). Here P_l counts the paths stored separately for output rank l, with $P_l = 1$ for node features and direction tensors. For component index $c \in \{0, \dots, 2l\}$, path index $\eta \in \{0, \dots, P_l - 1\}$, and channel index

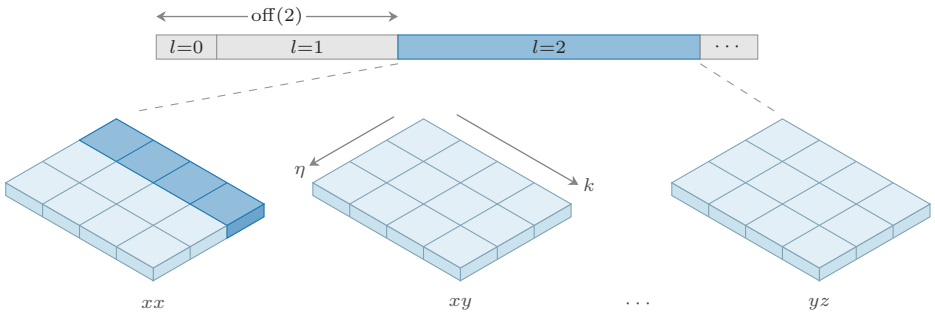

Figure D1: **Memory layout of Cartesian tensor features.** Each row $X_{r,:}$ stores tensor ranks in contiguous blocks, with $\mathrm{off}(l)$ giving the offset of the block for rank l within the row. The expanded rank-2 block shows how each stored Cartesian component contains P_2 coupling paths and d_X feature channels. The darker cells highlight the contiguous channels of one component–path pair.

$k \in \{0, \ldots, d_X - 1\}$, the entry's offset within the row is

$$\mathrm{off}(l) + (cP_l + \eta)d_X + k, \qquad \mathrm{off}(l) = \sum_{l'<l} (2l' + 1)P_{l'}d_X.$$

Keeping ranks up to L therefore selects the first $\mathrm{off}(L + 1)$ entries of each row without copying or rearranging the features. The product layer uses this selection when intermediate ranks exceed the block's output ranks, including when the last message-passing block returns only scalars.

Node-local and edge-parallel kernels. The shared implementation evaluates tensor products for a tile of nodes or edges. For node-local products, it reads the corresponding input rows and stores each coupling path's output separately. For edge convolutions, the edge indices identify the input rows and the receiver to which each contribution is added atomically. Summing coupling paths into a common output also uses atomic additions in this implementation. The receiver-sorted kernel instead sums both paths and incoming-edge contributions in local accumulators and stores each result once. Neither convolution kernel stores an intermediate tensor of edge messages.

Reusing an input or weight in the forward computation requires adding the corresponding gradient contributions during differentiation. When an input is reused across feature channels, its gradient sums the contributions from those channels. Similarly, when a weight is shared across rows, its gradient sums the contributions from those rows.

Receiver-sorted forward. Section 4 indexes edges by e, while Section 2 identifies them by the atom pair uv. All edges are sorted by receiver, and a row-pointer array identifies the incoming edges of receiver u by the range $[\texttt{rowptr}[u], \texttt{rowptr}[u + 1])$. The sender of edge e is provided by $\texttt{sender}[e]$. Each coupling path $\eta = (l_1, l_2 \to l)$ combines rank l_1 of the sender features with rank l_2 of the direction tensor $\boldsymbol{y}_e = \boldsymbol{T}(\hat{\boldsymbol{r}}_{u_e v_e})$ to produce output rank l. When both inputs have the same number of channels, we write the second input as $\boldsymbol{y}_{ek}$ and load it for each channel tile. Using these inputs and the precomputed scalar path weights, each program evaluates the tensor products over its receiver's incoming edges and stores the accumulated sums after processing all of them.

To limit the number of sums accumulated simultaneously, we group output ranks with a target of 100 stored components per channel, counting the output components of each coupling path separately. Outputs of the same tensor rank are kept in one group, even if their combined component count exceeds the target. For the fused VJP-of-VJP, we reduce the target number of output components per group to limit register use. We launch a separate kernel for each group of output ranks $\mathcal{G}$. Within each launch, we divide the feature channels into tiles $\mathcal{K}$, and each program processes one receiver and one tile. It evaluates all coupling paths that produce ranks in $\mathcal{G}$, denoted by $\mathcal{P}_\mathcal{G}$ in Algorithm 1. Each program stores the output components for its receiver and channel tile, without atomic additions. During code generation, the operations for every path are written out explicitly and applied independently to the channels processed by that program.

Receiver-sorted derivatives. Let $\boldsymbol{g}_{u\eta k}$ be the incoming gradient of $\boldsymbol{z}_{u\eta k}$, and let $\mathrm{VJP}_{1,\eta}$ and $\mathrm{VJP}_{2,\eta}$ denote the unweighted input VJPs defined in Appendix C. For each edge, the program sums

Algorithm 2 Receiver-sorted VJP

Require: Receiver-sorted arrays `rowptr`, `sender`
Require: Tensor inputs $\boldsymbol{x}, \boldsymbol{y}$ and scalar weights w
Require: Rank groups $\mathcal{G}$, paths $\mathcal{P}_{\mathcal{G}}$, and channel tiles $\mathcal{K}$
Require: Cotangent $\boldsymbol{g}$, chunk length C, and slots S
Initialize $\bar{\boldsymbol{x}}, \bar{\boldsymbol{y}} \leftarrow \mathbf{0}$
for each rank group $\mathcal{G}$ **do**
  for all receivers u, slots $\sigma < S$, and tiles $\mathcal{K}$ **do**
    Load $e_0, e_1 \leftarrow \texttt{rowptr}[u], \texttt{rowptr}[u+1]$
    Set $j \leftarrow e_0 + \sigma C$
    if $j \geq e_1$ **then**
      exit program
    end if
    Load $\boldsymbol{g}_{u\eta k}$ for $\eta \in \mathcal{P}_{\mathcal{G}}, k \in \mathcal{K}$
    while $j < e_1$ **do**
      for $e = j, \ldots, \min(j + C, e_1) - 1$ **do**
        Load $v \leftarrow \texttt{sender}[e]$, $\boldsymbol{x}_{v\mathcal{K}}$, and $\boldsymbol{y}_e$
        Initialize $\boldsymbol{a}^x_k \leftarrow \mathbf{0}$ for $k \in \mathcal{K}$ and $\boldsymbol{a}^y \leftarrow \mathbf{0}$
        for $\eta \in \mathcal{P}_{\mathcal{G}}, k \in \mathcal{K}$ **do**
          Load $w_{e\eta k}$
          $\boldsymbol{a}^x_k \mathrel{+}= w_{e\eta k}\, \mathrm{VJP}_{1,\eta}(\boldsymbol{g}_{u\eta k}, \boldsymbol{y}_e)$
          $\boldsymbol{a}^y \mathrel{+}= w_{e\eta k}\, \mathrm{VJP}_{2,\eta}(\boldsymbol{g}_{u\eta k}, \boldsymbol{x}_{vk})$
          Store $\bar{w}_{e\eta k} \leftarrow \langle \boldsymbol{g}_{u\eta k}, \boldsymbol{f}_\eta(\boldsymbol{x}_{vk}, \boldsymbol{y}_e) \rangle$
        end for
        $\mathrm{AtomicAdd}(\bar{\boldsymbol{x}}_{v\mathcal{K}}, \boldsymbol{a}^x_{\mathcal{K}})$
        $\mathrm{AtomicAdd}(\bar{\boldsymbol{y}}_e, \boldsymbol{a}^y)$
      end for
      Set $j \leftarrow j + SC$
    end while
  end for
end for

Algorithm 3 Receiver-sorted VJP-of-VJP

Require: Receiver-sorted arrays `rowptr`, `sender`
Require: Tensor inputs $\boldsymbol{x}, \boldsymbol{y}$ and scalar weights w
Require: Rank groups $\mathcal{G}$, paths $\mathcal{P}_{\mathcal{G}}$, and channel tiles $\mathcal{K}$
Require: Cotangents $\boldsymbol{g}$, $\boldsymbol{v}_x$, $\boldsymbol{v}_y$, v_w
Initialize $\widehat{\boldsymbol{x}}, \widehat{\boldsymbol{y}} \leftarrow \mathbf{0}$
for each rank group $\mathcal{G}$ **do**
  for all receivers u and channel tiles $\mathcal{K}$ **do**
    Load $e_0, e_1 \leftarrow \texttt{rowptr}[u], \texttt{rowptr}[u+1]$
    Load $\boldsymbol{g}_{u\eta k}$ for $\eta \in \mathcal{P}_{\mathcal{G}}, k \in \mathcal{K}$
    Initialize $\widehat{\boldsymbol{g}}^{\mathrm{loc}}_{\eta k} \leftarrow \mathbf{0}$ for $\eta \in \mathcal{P}_{\mathcal{G}}, k \in \mathcal{K}$
    for $e = e_0, \ldots, e_1 - 1$ **do**
      Load $v \leftarrow \texttt{sender}[e]$, $\boldsymbol{x}_{v\mathcal{K}}$, and $\boldsymbol{y}_e$
      Load $\boldsymbol{v}_{x,v\mathcal{K}}$, $\boldsymbol{v}_{y,e}$
      Initialize $\widehat{\boldsymbol{a}}^x_k \leftarrow \mathbf{0}$ for $k \in \mathcal{K}$ and $\widehat{\boldsymbol{a}}^y \leftarrow \mathbf{0}$
      for $\eta \in \mathcal{P}_{\mathcal{G}}, k \in \mathcal{K}$ **do**
        Load $w_{e\eta k}$, $v_{w,e\eta k}$
        $\boldsymbol{s}_{e\eta k} \leftarrow \boldsymbol{f}_\eta(\boldsymbol{v}_{x,vk}, \boldsymbol{y}_e) + \boldsymbol{f}_\eta(\boldsymbol{x}_{vk}, \boldsymbol{v}_{y,e})$
        $\widehat{\boldsymbol{g}}^{\mathrm{loc}}_{\eta k} \mathrel{+}= w_{e\eta k} \boldsymbol{s}_{e\eta k} + v_{w,e\eta k} \boldsymbol{f}_\eta(\boldsymbol{x}_{vk}, \boldsymbol{y}_e)$
        $\widehat{\boldsymbol{a}}^x_k \mathrel{+}= \mathrm{VJP}_{1,\eta}\big(\boldsymbol{g}_{u\eta k}, w_{e\eta k} \boldsymbol{v}_{y,e} + v_{w,e\eta k} \boldsymbol{y}_e\big)$
        $\widehat{\boldsymbol{a}}^y \mathrel{+}= \mathrm{VJP}_{2,\eta}\big(\boldsymbol{g}_{u\eta k}, w_{e\eta k} \boldsymbol{v}_{x,vk} + v_{w,e\eta k} \boldsymbol{x}_{vk}\big)$
        Store $\widehat{w}_{e\eta k} \leftarrow \langle \boldsymbol{g}_{u\eta k}, \boldsymbol{s}_{e\eta k} \rangle$
      end for
      $\mathrm{AtomicAdd}(\widehat{\boldsymbol{x}}_{v\mathcal{K}}, \widehat{\boldsymbol{a}}^x_{\mathcal{K}})$
      $\mathrm{AtomicAdd}(\widehat{\boldsymbol{y}}_e, \widehat{\boldsymbol{a}}^y)$
    end for
    Store $\widehat{\boldsymbol{g}}_{u\eta k} \leftarrow \widehat{\boldsymbol{g}}^{\mathrm{loc}}_{\eta k}$ for $\eta \in \mathcal{P}_{\mathcal{G}}, k \in \mathcal{K}$
  end for
end for

the contributions from its assigned coupling paths before atomically adding the resulting sums to the input gradients, as shown in Algorithm 2. Atomic additions combine contributions from different receivers to the gradient of the same sender features. They also combine contributions to the gradient of a direction tensor from programs processing different channel tiles or groups of output ranks. Each weight gradient belongs to one edge, path, and channel and is stored without atomic addition. The derivative kernels use the same indexing as the forward kernel. Edge index e identifies the direction tensor, scalar weights, and their gradients, while sender index v_e identifies the sender features and their gradients. The edges therefore remain sorted by receiver throughout differentiation.

To process receivers with many neighbors in parallel, we divide each receiver's incoming edges into chunks of at most C edges. For each rank group and channel tile, S programs share the work. Program σ starts with chunk σ and advances by S chunks, processing $\sigma, \sigma + S, \ldots$. Programs with no assigned edges skip the computation. Each active program loads the receiver gradient once and reuses it for all its edges. We select the number of programs S before launch based on the batch's average number of neighbors. Autotuning selects C together with the other kernel settings, including a large-chunk option in which program zero processes all incoming edges when they fit in one chunk.

The forward output and $\widehat{\boldsymbol{g}}$ in the VJP-of-VJP both require sums over all incoming edges of a receiver. Dividing these edges among programs would require combining their partial sums. We therefore restrict edge chunking to the VJP. In Algorithm 3, each program processes all incoming edges of one receiver for its assigned group of output ranks and channel tile. For each edge, it evaluates Equation C2 using the cotangents $\boldsymbol{v}_x$ of the sender's features in channel k, $\boldsymbol{v}_y$ of the edge's direction tensor, and v_w of the weight for that edge, coupling path, and channel. Each program computes $\widehat{\boldsymbol{g}}$, the gradient with respect to the incoming gradient $\boldsymbol{g}$, by summing contributions from all incoming

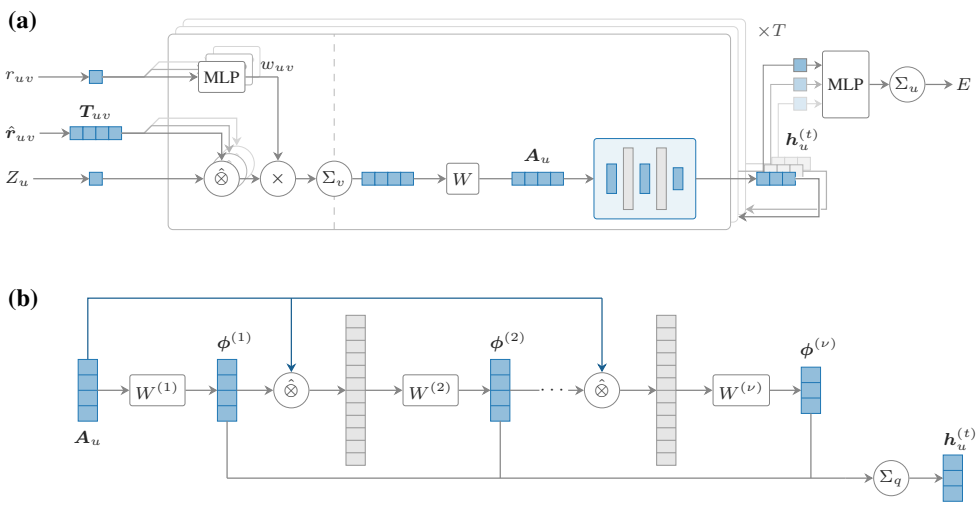

Figure E1: **Model architecture and fixed-width correlation ladder.** (a) Each message-passing block combines a convolution with the product ladder. Blue bars show tensor features grouped by rank l, starting at $l = 0$. Light grey lines show shared geometric inputs and blockwise scalar features combined in the readout to give E. The output of each block is also passed to the next block. (b) The same atomic basis $\boldsymbol{A}_u$ is reused at every tensor product. Grey bars show the intermediate tensor product outputs with the retained coupling paths, compressed by $W^{(q)}$ to d channels per retained rank, shown in blue. The output of each step enters the next product and also contributes to the final sum. Bar heights schematically indicate the number of tensor components. We skip residual updates, normalization, and nonlinear gates for simplicity.

edges of its receiver and stores the sum once. As in the VJP, programs atomically add contributions to shared input gradients $\widehat{\boldsymbol{x}}$ and $\widehat{\boldsymbol{y}}$ and store each weight gradient $\widehat{w}$ without atomic addition.

For maximum output ranks above four, we compute different derivative outputs in separate kernel launches to reduce the number of intermediate values held in registers. The VJP uses separate launches for $\bar{\boldsymbol{x}}$, $\bar{\boldsymbol{y}}$, and $\bar{w}$. For the VJP-of-VJP, one launch computes $\widehat{\boldsymbol{g}}$, another computes $\widehat{\boldsymbol{x}}$ and $\widehat{\boldsymbol{y}}$ together, and a third computes $\widehat{w}$. Each launch uses the generated component expressions and accumulates or stores its gradients as described above.

E Model Architecture

Each atom is represented by a learned scalar embedding of its species. For each neighboring pair, the distance r_{uv} determines the radial weights, while the unit direction $\hat{\boldsymbol{r}}_{uv}$ determines the Cartesian direction tensors used in the convolution. The features pass through T message-passing blocks, each combining an equivariant convolution with the node-local product ladder of Section 5 (Figure E1). All blocks use the independent Cartesian components and memory layout described in Sections 3–4.

A residual update and equivariant RMS normalization follow both the convolution and the product ladder. Each residual update adds a learned linear map of the corresponding input to the output. The normalization computes scalar invariants with the generated $(l, l) \to 0$ couplings and combines them across ranks and channels into one normalization scale per atom, without batch statistics [52]. A Euclidean norm of the stored components would not be rotationally invariant because the Cartesian coordinates are not orthonormal. After normalization, nonlinear functions of the scalar features gate the equivariant channels before the product ladder [53]. A learned output map transforms the gated features, and its output is added to the input of the gated layer. Initializing this map to zero makes the layer an identity at initialization. The resulting features define the atomic basis $\boldsymbol{A}_u$ supplied to the product ladder.

After summing contributions through degree ν, the product ladder is followed by a residual update and normalization to give the output of a message-passing block. Intermediate blocks retain ranks

up to $l_{\max}^{\text{prop}}$, while the final block outputs only scalars. Scalar channels from all T blocks are concatenated and passed through a readout multi-layer perceptron (MLP) to obtain the atomic energy E_u. The atomic energies are scaled and shifted using fixed values determined from the training set. The per-element shifts are obtained by ridge-regularized least squares [54], while the output scale is set to the root-mean-square (RMS) force component. Finally, the total energy is obtained by summing the atomic energies, $E = \sum_u E_u$.

Equivariant convolution. For message-passing block t, $\boldsymbol{h}^{(t)}_{vkl_1}$ denotes channel k of the rank-l_1 features at sender v, with d channels per retained rank. Before the convolution, a $d \times d$ map mixes the channels independently at each rank. Denoting the resulting features by $\bar{\boldsymbol{h}}^{(t)}$, the convolution aggregates their contributions from neighbors $v \in \mathcal{N}(u)$ at receiver u,

$$\boldsymbol{A}^{(t)}_{u\eta kl} = \frac{1}{\sqrt{\bar{N}}} \sum_{v \in \mathcal{N}(u)} w^{(t)}_{\eta k}(r_{uv}) \left(\bar{\boldsymbol{h}}^{(t)}_{vkl_1} \hat{\otimes}_{\text{Cart}} \boldsymbol{T}_{l_2}(\hat{\boldsymbol{r}}_{uv}) \right)_l , \tag{E1}$$

where $\eta = (l_1, l_2 \to l)$ is an admissible coupling path. The factor $1/\sqrt{\bar{N}}$ uses the mean number of neighbors in the training data. The receiver-sorted kernel of Section 4 evaluates Equation E1, retaining the contribution of each path until a rank-wise linear map mixes the P_l paths and d channels into d output channels. For $l^{\text{edge}}_{\max} = l^{\text{prod}}_{\max} = 3$ and $l^{\text{prop}}_{\max} = 2$, the first convolution has four paths because the sender features are scalar, while subsequent convolutions have 17.

The radial weights $w^{(t)}_{\eta k}(r_{uv})$ are produced by a bias-free MLP with one output per coupling path and channel. Its input is a Chebyshev expansion of the distance multiplied by a polynomial cutoff envelope whose value and first two derivatives vanish at $r_{\max}$ [55]. With a smooth activation satisfying $\sigma(0) = 0$, the radial weights and their first two derivatives also vanish at the cutoff. Edges can therefore enter or leave the cutoff without introducing discontinuities in the energy or its first two derivatives.

Factorized fixed-width correlations. A general bilinear equivariant map assigns a coefficient $\widetilde{W}^{(q)}_{l,k(\eta k_1 k_2)}$ to each coupling path η, pair of input channels (k_1, k_2), and output channel k. The product in Equation 2 couples only matching input channels, giving

$$\widetilde{W}^{(q)}_{l,k(\eta k_1 k_2)} = \delta_{k_1 k_2} W^{(q)}_{l,k(\eta k_1)}.$$

The coefficients therefore vanish when $k_1 \neq k_2$, while $W^{(q)}$ mixes all retained paths and channels densely within each output rank. The separate path contributions in $\boldsymbol{B}^{(q)}$ are stored until this map produces $\boldsymbol{\phi}^{(q)}$. When both inputs and the output retain tensor ranks $l = 0 \ldots 3$, $\boldsymbol{B}^{(q)}$ contains $99d$ independent Cartesian components across its coupling paths. Mixing reduces this to $16d$ components, corresponding to d channels at each retained rank (Figure E1(b)).

Tensor-train interpretation. Expanding the recurrence in Equation 2 expresses $\boldsymbol{\phi}^{(\nu)}$ as a polynomial in the atomic basis. For a fixed output rank l, let $\boldsymbol{\eta} = (\eta_2, \ldots, \eta_\nu)$ denote a retained sequence of coupling paths whose final product has rank l. We write $i_1, \ldots, i_\nu$ for the channels of the successive basis factors and j for the output channel. Suppressing the tensor ranks specified by $\boldsymbol{\eta}$, we obtain

$$\boldsymbol{\phi}^{(\nu)}_{uj} = \sum_{\boldsymbol{\eta}} \sum_{i_1, \ldots, i_\nu} \Theta^{j}_{i_1 \cdots i_\nu}(\boldsymbol{\eta}) \left[\boldsymbol{A}_{ui_1} \hat{\otimes}_{\text{Cart}} \cdots \hat{\otimes}_{\text{Cart}} \boldsymbol{A}_{ui_\nu} \right]_{\boldsymbol{\eta}}. \tag{E2}$$

The products in brackets are evaluated from left to right along the paths in $\boldsymbol{\eta}$. For each retained path sequence, the coefficient tensor factors into the learned maps of the successive ladder steps. This gives the tensor-train form [42]

$$\Theta^{j}_{i_1 \cdots i_\nu}(\boldsymbol{\eta}) = \sum_{k_1, \ldots, k_{\nu-1}} W^{(1)}_{k_1 i_1} \prod_{q=2}^{\nu} G^{(q)}_{k_{q-1}, i_q, k_q}(\eta_q), \qquad k_\nu = j, \tag{E3}$$

with

$$G^{(q)}_{k_{q-1}, i_q, k_q}(\eta_q) = \delta_{k_{q-1} i_q} W^{(q)}_{k_q(\eta_q i_q)}. \tag{E4}$$

Here $W^{(1)}$ mixes the channels of the first basis factor, while each $G^{(q)}$ describes one subsequent product step. The indices k_q connect successive steps, and each takes d values, set by the model

width. The Kronecker delta pairs each incoming channel with the same channel of the atomic basis, while $W^{(q)}$ mixes the resulting path and channel contributions into the outgoing channels. The recurrence uses this factorization implicitly and evaluates it directly without constructing Θ. Contributions from earlier ladder steps follow the same factorization truncated after q basis factors, and the ladder output sums contributions of degrees $1, \dots, \nu$.

Path-reduced products. For the path-reduced products, each path and channel has a learned weight $a^{(q)}_{\eta k} = 1 + \delta^{(q)}_{\eta k}$, shared across atoms and initialized with $\delta^{(q)}_{\eta k} = 0$. At the first step, there is one weight per rank and channel, $\boldsymbol{\phi}^{(1)}_{ukl} = a^{(1)}_{lk} \boldsymbol{A}_{ukl}$. For $q = 2, \dots, \nu$, the recurrence becomes

$$\boldsymbol{\phi}^{(q)}_{ukl} = \frac{1}{\sqrt{P_{q,l}}} \sum_{\eta=(l_1,l_2\to l)} a^{(q)}_{\eta k} \left(\boldsymbol{\phi}^{(q-1)}_{ukl_1} \hat{\otimes}_{\mathrm{Cart}} \boldsymbol{A}_{ukl_2} \right)_l, \qquad l \le l_q,$$

where $P_{q,l}$ is the number of coupling paths producing rank l. The kernel accumulates these paths as it evaluates the products, without mixing channels. The step outputs are then combined and passed through a rank-wise linear map W that mixes the channels to produce

$$\boldsymbol{z}_{ukl} = \sum_{k'} W_{l,kk'} \left(\sum_{q=1}^{\nu} \boldsymbol{\phi}^{(q)}_{uk'l} \right), \qquad l \le l^{\mathrm{out}}_{\max}.$$

F Experimental details

Datasets and splits. SPICE-MACE-OFF is a neutral-molecule dataset based on SPICE [20] that was originally used to train and evaluate MACE-OFF23 [21]. It covers H, C, N, O, F, P, S, Cl, Br, and I, with 951,005 structures in the training set and 50,195 in the test set. We train three models, each using a randomly selected validation set of 47,550 structures and the remaining 903,455 structures for training. The test set comprises seven subsets: PubChem, DES370K monomers and dimers, dipeptides, solvated amino acids, water, and QMugs.

MAD [22] contains 95,595 structures spanning 85 elements, including bulk and two-dimensional crystals, molecular crystals and fragments, surfaces, clusters, and structures with distorted geometries or randomized chemical compositions. We use the released split of 76,483 training, 9,566 validation, and 9,546 test structures for all three runs, varying model initialization. We further evaluate performance across materials and molecular systems using the PET-MAD benchmark [12]. Its 1,562 structures comprise 360 MAD [22], 136 MPtrj [34], 555 MatBench [35], 200 Alexandria [36], 78 OC2020-S2EF [37], 99 SPICE [20], and 134 MD22 [38] structures.

Model architecture and training. All FlashCart models use eight Chebyshev radial functions evaluated at cosine-transformed interatomic distances and a polynomial cutoff envelope with $p = 6$ [55]. The radial and readout MLPs each have two hidden layers of width 64. Within each message-passing block, we apply equivariant RMS normalization and scalar SiLU gates as described in Appendix E. The initial embedding depends on the chemical species, while the convolution and product weights are shared across all species. The architectural comparisons in Figures 4 and G1 examine changes in correlation order, channel width, message-passing depth, and maximum tensor ranks. Table F1 summarizes these configurations, the five FlashCart models in Figure 1, and the model trained on MAD. We also compare correlation orders at approximately fixed model size by reducing channel width as order increases, keeping the parameter count between 0.913M and 0.945M. This experiment uses configurations with two message-passing blocks, maximum propagated rank 1, and maximum edge and product ranks from 1 to 4.

Before training, we determine the per-element energy shifts, output scale, and convolution normalization, which remain fixed throughout training. We fit the energy shifts by ridge regression [54], with isolated-atom energies as reference values for SPICE-MACE-OFF. The output scale is set to the root mean square of the force components and the convolution normalization to the average number of neighbors per atom in the training set. We use single precision and the training hyperparameters listed in Table F2, with Muon for matrix weights and AdamW for the remaining parameters, excluding embeddings and biases from weight decay. The learning rate increases linearly from 10% of its peak value during warmup, then follows a cosine decay to 1% of the peak. For a batch with B

Table F1: **Model configurations.** T, ν, and d denote the number of message-passing blocks, correlation order, and channel width, respectively, while $l_{\max}^{\text{edge}} = l_{\max}^{\text{prod}}$ gives the shared maximum rank of the edge and product tensors. For the models in Figures 4 and G1, default values apply to parameters that are held fixed. The labels "Ladder" and "Reduced" denote the correlation ladder and path-reduced products described in Appendix E.

Configuration	T	ν	d	$l_{\max}^{\text{prop}}$	$l_{\max}^{\text{edge}} = l_{\max}^{\text{prod}}$	Product
Models shown in Figures 4 and G1						
Default values	2	2	64	1	3	Ladder
Explored values	1–4	2–5	$32, 64, 96, 128$	0–3	1–4	Ladder
Models shown in Figure 1 and Table G1						
FlashCart-69k	2	3	20	2	3	Reduced
FlashCart-0.5M	3	5	32	2	3	Ladder
FlashCart-1.1M	3	3	64	2	3	Ladder
FlashCart-3.0M	3	5	88	2	3	Ladder
FlashCart-5.6M	4	5	104	2	3	Ladder
Models shown in Tables 1 and G2						
FlashCart-2.3M	3	3	96	2	3	Ladder

Table F2: **Training parameters.** We use the target batch size and training-set averages to determine how many atoms and edges each batch can contain, so the number of structures per batch can vary. For distributed training, we divide the target batch size across GPUs. EMA denotes the exponential moving average of model parameters.

Hyperparameter	Figures 4 and G1	Figure 1 and Table G1	Tables 1 and G2
Cutoff radius	6.0	6.0	5.0
Maximum epochs	150	400	400
Warmup epochs	20	20	20
Cosine-decay epochs	130	380	380
Target batch size	128	128	192
Number of random seeds	3	3	3
Peak learning rate	5×10^{-3}	5×10^{-3}	2.5×10^{-3}
Weight decay	3×10^{-4}	3×10^{-4}	1.2×10^{-2}
EMA decay	0.999	0.999	0.999
Force weight	1.0	1.0	0.1

structures and $N = \sum_b N_b$ atoms, the training loss reads

$$\mathcal{L} = \frac{1}{B} \sum_{b=1}^{B} \frac{|E_b - E_b^{\text{ref}}|}{N_b} + w_F \frac{1}{N} \sum_{i=1}^{N} \|\boldsymbol{F}_i - \boldsymbol{F}_i^{\text{ref}}\|_2, \tag{F1}$$

where E_b and $\boldsymbol{F}_i$ are the predicted energies and forces, while E_b^{ref} and $\boldsymbol{F}_i^{\text{ref}}$ denote the corresponding reference values. Energies and forces are given in eV and eV/Å, and Table F2 provides the force weight w_F in Å. We evaluate the model using an exponential moving average of its parameters and select the epoch with the lowest sum of validation energy MAE (eV/atom) and force MAE (eV/Å).

Normalized error. We combine energy and force MAEs into normalized error (NE), following Ref. 56,

$$\text{NE} = \frac{1}{2} \left(\frac{\text{F-MAE}}{\max \text{F-MAE}} + \frac{\text{E-MAE}}{\max \text{E-MAE}} \right). \tag{F2}$$

Here E-MAE and F-MAE are the energy and force MAEs, and the denominators are their respective maxima over the models included in each figure. For Figure 1, the MAEs are averaged over the

seven SPICE-MACE-OFF test subsets reported in Table G1. For Figure 4, the MAEs are evaluated over the entire test set.

Runtime and memory measurements. We measure runtime and memory on an NVIDIA RTX PRO 6000 Blackwell Server Edition using PyTorch 2.11.0 and CUDA 12.8. All computations use single precision with TF32 disabled. We use periodic diamond cells with lattice constant 3.567 Å and precompute neighbor lists using each model's cutoff radius. Energy and force evaluation uses 1000 atoms for the models in Figure 1 and Tables 1, G1, and G2, and 4096 atoms for those in Figures 4 and G1. We also vary the system size from 216 to 4096 atoms to examine runtime and memory scaling in Figure 1. For Figure G2, we use 1000 atoms for inference and 512 for training because the uncompiled PyTorch implementation runs out of GPU memory when training on 1000 atoms. Training timings additionally include the loss and its parameter gradients, but exclude optimizer updates and data loading.

For each SPICE-MACE-OFF model benchmarked on our GPUs, we select the implementation with the lowest measured runtime. We enable compilation for FlashCart, MACE, PET-S, and DPA4. FlashCart uses generated Triton kernels, MACE uses cuEquivariance [15], and DPA4 uses DeePMD-kit's Triton kernels [57], while PET-L runs with TorchScript. For MAD, FlashCart also uses compilation, while PET-MAD runs in PyTorch without it. We estimate the eSEN runtimes from published results for an NVIDIA A100 GPU (80 GB) [23].

Kernel microbenchmarks. We benchmark individual kernels using the same hardware and numerical precision as for the model benchmarks above, comparing FlashCart with cuEquivariance 0.11.0 and OpenEquivariance 0.6.8. For edge convolutions, we use OpenEquivariance's fused tensor-product convolution with deterministic evaluation disabled and cuEquivariance's channel-wise tensor product with fused gather and scatter operations. For node-local correlations, we compare FlashCart's path-reduced products and correlation ladder with cuEquivariance's symmetric contraction. The edge-convolution benchmark covers 30 configurations with 1024 nodes, 64 or 128 neighbors per node, 64, 128, or 256 channels, and maximum tensor ranks from 1 to 5. Figure 3(a) shows the cases with 128 neighbors and 128 channels. To obtain the average runtime advantages reported in the main text, we divide each baseline's runtime by FlashCart's for each configuration and average these ratios over all 30 configurations. The node-local benchmark uses 4096 nodes, 64, 128, or 256 channels, maximum tensor ranks from 1 to 5, and correlation orders from 2 to 6. Figure 3(b) shows the configurations with 128 channels and maximum tensor ranks 3 and 4.

We measure the runtime of the forward, VJP, and VJP-of-VJP separately, excluding the preceding forward pass for the VJP and both the forward pass and first VJP for the VJP-of-VJP. Memory is measured as the peak increase in GPU allocation during each operation, excluding memory already used by its inputs and tensors saved from preceding computations. We report runtime and memory averaged over three independent runs. The VJP benchmark computes gradients with respect to the inputs and weights. The VJP-of-VJP benchmark differentiates the input gradients with respect to the inputs, weights, and incoming output gradient, following the derivative sequence needed for force training. It excludes contributions from differentiating the first VJP's weight gradients, corresponding to a zero weight-gradient cotangent in Equation C2.

G Results and discussion

Kernel microbenchmarks. The grey crosses in Figure 3(b) show cases in which we could not construct the cuEquivariance contraction or launch its kernel. For example, at rank 4, order 4, CPU RAM usage reached 767 GiB before construction failed after approximately 2.7 hours under the 1000 GiB memory limit we set. At rank 3, order 4, we measured forward and VJP runtimes, but the VJP-of-VJP kernel failed to launch with a CUDA out-of-memory error on our NVIDIA RTX PRO 6000 Blackwell GPU (96 GB). Preparing the first VJP call also took about 20 hours on the CPU at rank 4, order 3, and 37 hours at rank 3, order 4.

Efficiency of higher-order correlations. The 52 configurations in Figure 4 cover correlation orders $\nu = 2$ to 5 for 13 otherwise fixed architectures. These same configurations also let us compare channel widths, numbers of message-passing blocks, maximum propagated ranks, and jointly varied edge and product tensor ranks at each fixed correlation order, keeping the remaining architectural choices unchanged. For each comparison, we fit $\log \mathrm{NE} = a + b \log x$ by ordinary least squares, where NE

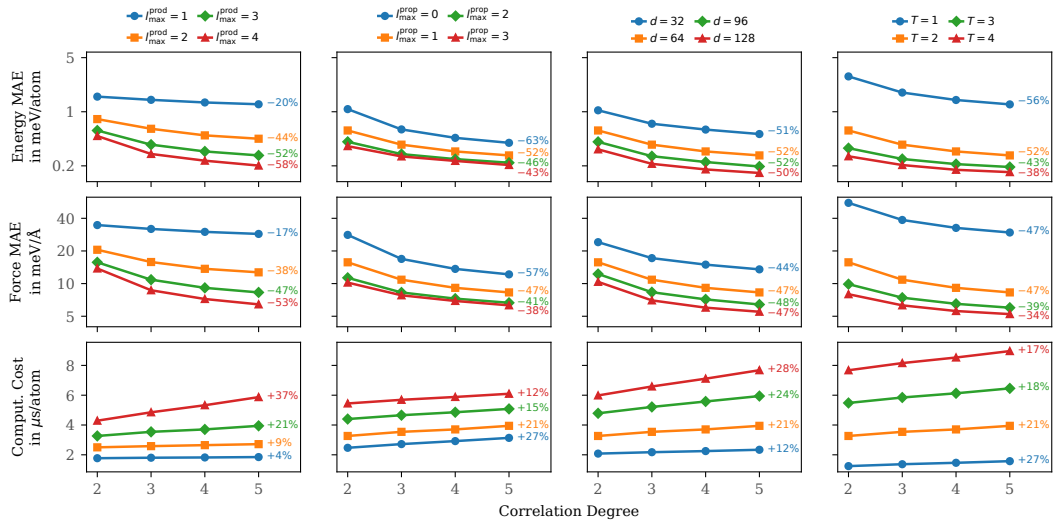

Figure G1: **Effect of correlation order across model architectures.** Results for the models trained on SPICE-MACE-OFF and shown in Figure 4 are presented as energy MAE (top), force MAE (middle), and runtime (bottom) against correlation order ν. From left to right, the columns show results for different maximum edge and product ranks (varied together), maximum propagated ranks, channel widths d, and numbers of message-passing blocks T. Unless varied in a column, models use maximum edge and product ranks of $l_{\max}^{\text{prod}} = l_{\max}^{\text{edge}} = 3$, maximum propagated rank $l_{\max}^{\text{prop}} = 1$, $d = 64$, and $T = 2$. Percentages show the relative changes in error or runtime as ν increases from 2 to 5. Prediction errors are averaged over three independent training runs. Shaded bands show one standard deviation but are not visible at the plotted line width. Runtimes are measured for energy and force evaluation on 4096 diamond atoms using one trained model per configuration.

is normalized error and x is runtime or peak GPU memory per atom. The insets in Figure 4 show how normalized error changes with runtime or peak GPU memory when correlation order, channel width, the number of message-passing blocks, or tensor rank is varied. The fitted slopes quantify the relative change in error for a relative change in runtime or memory. Figure G1 presents the same model results as Figure 4, showing energy errors, force errors, and runtimes separately as correlation order increases. Its first column compares models with two message-passing blocks, 64 channels, and maximum propagated rank 1 at four fixed maximum edge and product ranks. Using the errors and runtimes in this column, we fit normalized error against runtime on logarithmic axes across correlation orders $\nu = 2$ to 5. The resulting slopes are -5.0, -6.5, -3.7, and -2.6 for maximum edge and product ranks 1, 2, 3, and 4, respectively.

Model accuracy and computational cost. Table G1 provides the subset errors underlying the accuracy–efficiency comparison in Figure 1. The largest FlashCart model uses 5.6M parameters compared with DPA4-Plus' 8.8M and evaluates energies and forces 7.7× faster with 3.1× less peak GPU memory. It achieves a lower mean energy MAE and a force MAE of 1.99 ± 0.04 meV/Å compared with 1.80 meV/Å, as well as a lower combined normalized error. FlashCart has lower force errors than DPA4-Plus on DES370K monomers and dimers and the same reported force MAE on water. Compared with the similarly sized 5.1M-parameter DPA4-Air, it achieves lower energy and force MAEs on all seven subsets.

Without model compilation, replacing PyTorch tensor products and edge convolutions with the generated Triton kernels makes the same model 26.8× faster for inference and 23.8× faster in the training benchmark (Figure G2). Compiling the model with Triton kernels further reduces runtime but increases peak GPU memory during training. To locate this memory increase, we compare memory allocation before and during loss differentiation on 512 atoms, with CUDA graphs disabled in the compiled version. During training we first evaluate the energy and differentiate it with respect to atomic positions to obtain forces. After the energy and force errors have been combined into the loss, GPU allocation has increased by 5.21 GiB with compilation and 5.72 GiB without compilation. The loss is then differentiated with respect to model parameters. During this step, allocation peaks

Table G1: **Energy and force errors on SPICE-MACE-OFF.** Results are shown separately for the seven test subsets for the models in Figure 1. Rows give energy (E) MAEs in meV/atom and force (F) MAEs in meV/Å. We also report the arithmetic mean of the seven subset MAEs, separately for energy and forces. Normalized error (NE) combines these means as defined in Appendix F. FlashCart results show the mean and standard deviation across three independent training runs. We measure runtime and peak GPU memory for energy and force evaluation on a periodic diamond cell containing 1000 atoms using an NVIDIA RTX PRO 6000 Blackwell GPU (96 GB). The lowest energy and force MAEs and normalized errors are shown in bold, taking the reported standard deviations into account.

Model		Params.	PubChem	DES Mon.	DES Dim.	Dipept.	Solv. AA	Water	QMugs	Mean	NE	μs/atom	MiB/atom
MACE-OFF23-S [21]	E	694k	1.41	1.04	0.98	0.84	1.60	1.67	1.03	1.224	1.000	3.6	0.83
	F		35.68	17.63	16.31	25.07	38.56	28.53	41.45	29.03			
MACE-OFF23-M [21]	E	1.4M	0.91	0.63	0.58	0.52	1.21	0.76	0.69	0.757	0.593	6.7	1.48
	F		20.57	9.36	9.02	14.27	23.26	15.27	23.58	16.48			
MACE-OFF23-L [21]	E	4.7M	0.88	0.59	0.54	0.42	0.98	0.83	0.45	0.670	0.490	14.9	3.79
	F		14.75	6.58	6.62	10.19	19.43	13.57	16.93	12.58			
eSEN-3.2M [23]	E	3.2M	0.22	0.17	0.20	0.10	0.30	0.24	0.16	0.199	0.153	393†	–
	F		6.10	1.85	2.77	3.04	5.76	3.88	5.70	4.16			
eSEN-6.5M [23]	E	6.5M	0.15	0.13	0.15	0.07	0.25	0.15	0.12	0.146	0.108	785†	–
	F		4.21	1.24	2.12	2.00	3.68	2.50	3.78	2.79			
DPA4-Neo [7]	E	1.1M	0.24	0.17	0.18	0.10	0.27	0.30	0.12	0.197	0.154	39.2	6.44
	F		5.92	2.13	2.16	3.10	6.69	4.46	5.36	4.26			
DPA4-Air [7]	E	5.1M	0.13	0.17	0.11	0.06	0.13	0.16	0.07	0.119	0.089	78.2	8.14
	F		3.59	1.22	1.23	1.68	3.59	2.47	2.91	2.38			
DPA4-Plus [7]	E	8.8M	0.12	0.12	0.11	**0.04**	0.09	0.11	**0.05**	0.091	0.068	164.4	16.18
	F		**2.98**	0.91	0.96	**1.25**	**2.59**	**1.78**	**2.14**	**1.80**			
PET-S [13]	E	14.0M	0.19	0.19	0.18	0.11	0.36	0.27	0.15	0.207	0.174	41.9	5.61
	F		6.79	2.23	2.17	3.58	8.78	6.11	6.71	5.20			
PET-L [13]	E	188.9M	**0.09**	0.10	0.12	0.05	0.17	0.13	0.08	0.106	0.086	215.7	15.40
	F		3.53	1.00	1.20	1.55	4.37	3.05	2.91	2.52			
FlashCart-69k	E	69k	1.39±0.07	0.67±0.09	0.72±0.04	0.63±0.03	1.42±0.26	0.71±0.05	0.67±0.02	0.89±0.07	0.669	2.4	0.80
	F		26.45±0.03	10.38±0.02	10.05±0.13	15.26±0.13	22.87±0.41	14.49±0.41	24.94±0.18	17.78±0.05			
FlashCart-0.5M	E	464k	0.30±0.06	0.21±0.07	0.20±0.03	0.17±0.02	0.44±0.12	0.24±0.01	0.21±0.04	0.25±0.05	0.195	5.3	1.51
	F		7.71±0.19	2.62±0.11	2.82±0.08	4.54±0.13	7.33±0.12	4.63±0.18	7.66±0.20	5.33±0.14			
FlashCart-1.1M	E	1.1M	0.16±0.00	0.11±0.01	0.12±0.00	0.09±0.00	0.18±0.01	0.14±0.01	0.12±0.01	0.13±0.01	0.115	8.4	2.62
	F		5.41±0.04	1.64±0.02	1.77±0.02	2.95±0.03	4.82±0.09	3.08±0.08	5.26±0.03	3.56±0.04			
FlashCart-3.0M	E	3.0M	0.11±0.00	0.09±0.00	0.08±0.00	0.06±0.00	0.12±0.01	0.10±0.02	0.09±0.00	0.09±0.00	0.082	13.4	3.47
	F		3.97±0.02	1.10±0.01	1.16±0.01	2.00±0.01	3.53±0.11	2.33±0.11	3.67±0.03	2.54±0.03			
FlashCart-5.6M	E	5.6M	**0.09±0.00**	**0.07±0.01**	**0.06±0.00**	**0.04±0.00**	**0.08±0.00**	**0.08±0.00**	0.07±0.00	**0.07±0.00**	**0.063**	21.5	5.24
	F		3.26±0.03	**0.87±0.02**	**0.89±0.01**	1.53±0.02	2.73±0.06	**1.78±0.07**	2.86±0.06	1.99±0.04			

† Runtimes for eSEN are estimated from published results for an NVIDIA A100 GPU (80 GB) [23].

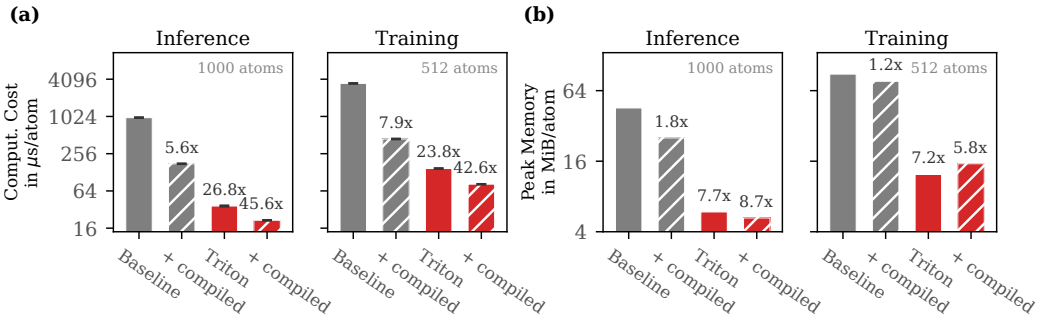

Figure G2: **Runtime and memory with generated kernels and model compilation.** Runtime (a) and peak GPU memory (b) per atom are compared for the same 5.6M-parameter FlashCart model during inference (left) and training (right). Baseline denotes the PyTorch implementation of tensor products and edge convolutions, while Triton denotes the generated kernels. We evaluate both implementations with and without model compilation, using the fastest measured compilation mode. The factors above the bars show the runtime and memory improvements relative to the uncompiled PyTorch baseline. Inference evaluates energies and forces on 1000 diamond atoms. Training evaluates the energy and force loss and its parameter gradients on 512 atoms, excluding optimizer updates. Runtime error bars show standard deviations over repeated evaluations.

Table G2: **Energy and force errors on the full MAD test set.** Each energy/force pair gives the energy MAE in meV/atom followed by the force MAE in meV/Å. FlashCart results show the mean and standard deviation across three models trained with different random seeds on a fixed data split. The overall MAEs are computed over the full test set. The lowest energy and force MAEs for each dataset are shown in bold.

Subset	FlashCart	PET-MAD
MC2D	**12.56±0.30** / **57.2±1.2**	15.54 / 72.7
MC3D	**7.02±0.07** / **35.1±0.4**	9.74 / 46.3
MC3D Clusters	**28.41±0.54** / **127.6±1.0**	43.06 / 148.3
MC3D Random	68.09±0.71 / 316.2±2.0	**63.87** / **249.6**
MC3D Rattled	**10.04±0.20** / 142.2±1.1	11.73 / **140.3**
MC3D Surfaces	**10.58±0.07** / **88.7±1.0**	13.61 / 102.6
SHIFTML Molecular Crystals	**0.90±0.01** / **21.4±0.2**	2.53 / 33.8
SHIFTML Molecular Fragments	**1.18±0.05** / **20.8±0.4**	6.34 / 57.1
Overall	**11.38±0.13** / **63.9±0.5**	14.72 / 72.9

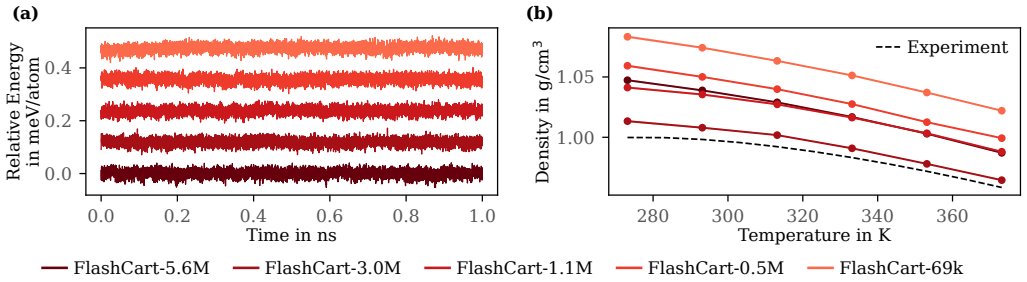

Figure G3: **Energy conservation and density of bulk liquid water.** (a) Total energy per atom during NVE dynamics, shown relative to its mean. We add a constant vertical offset to keep the curves distinguishable. (b) Density from NPT simulations at 1 bar, with shaded bands showing the standard error estimated by block averaging over 50 ps intervals. The bands are narrower than the plotted lines and are therefore not visible. The dashed curve shows experimental densities at 1 bar [64]. All results use one trained model per configuration and the final 1 ns of each trajectory.

a further 2.26 GiB above the preceding value with compilation, compared with 0.17 GiB without it. Thus, the additional peak memory arises during parameter-gradient computation when the model is compiled, with the same generated Triton kernels used in both cases.

Accuracy across materials and molecules. Table G2 compares the energy and force errors of FlashCart and PET-MAD, covering all 9,546 MAD test structures and complementing Table 1.

Molecular dynamics of bulk liquid water. To demonstrate the applicability of FlashCart models to atomistic simulations, we study bulk liquid water using molecular dynamics. We use the five models in Figure 1 to simulate 256 water molecules in LAMMPS [58] with Kokkos acceleration [59] and the ML-IAP-Kokkos interface [60]. All simulations begin with energy minimization and use periodic boundary conditions and a 0.5 fs integration timestep. For the NVE runs, we first equilibrate each system for 50 ps at 300 K in the NVT ensemble using a Nosé–Hoover chain thermostat [61] with a time constant of 0.2 ps, followed by 1.2 ns of NVE dynamics. The NPT simulations are performed over 1.2 ns after energy minimization at temperatures from 273.15 to 373.15 K and a pressure of 1 bar, using a Nosé–Hoover chain thermostat and an isotropic Martyna–Tobias–Klein barostat with a time constant of 1 ps [62, 63].

We analyze the final 1 ns of each trajectory to assess energy conservation in the NVE simulations and determine equilibrium densities in the NPT simulations. All models show energy drifts between −0.002 and 0.008 meV/atom/ns, with energy fluctuations around the mean having standard deviations of 0.012–0.013 meV/atom and maximum absolute deviations below 0.056 meV/atom

(Figure G3(a)). In the NPT simulations, the models predict water densities that decrease with temperature and remain above the experimental values (Figure G3(b)), consistent with the overestimation reported for MACE-OFF and some ICTP variants trained at the same electronic-structure level [21, 65]. Similar overestimation is also reported for MACE-POLAR-1, trained at a related electronic-structure level [66].