LUNATP: ENABLING EFFICIENT AND SCALABLE TENSOR PRODUCT FOR EQUIVARIANT MLIPS

Anonymous authors
Paper under double-blind review

ABSTRACT

Equivariant machine learning interatomic potentials (MLIPs) promise atomistic simulations with accuracy approaching that of quantum-mechanical calculations. However, the high computational cost of their core Clebsch–Gordan tensor product (CGTP) convolutions severely limits model scalability and efficiency. In this paper, we rethink the parallel execution of CGTP convolutions and introduce LunaTP. Built on node-centric execution, LunaTP exploits path-group parallelism and multi-level reuse to accelerate CGTP convolutions and higher-order derivative computation while preserving the underlying mathematical formulation. On an NVIDIA H100 GPU, LunaTP achieves kernel speedups of up to 250.0× over e3nn and 17.1× over OpenEquivariance. It also delivers end-to-end training throughput gains of up to 28.7× over e3nn and 3.11× over OpenEquivariance. Finally, LunaTP demonstrates scalability and robust performance across models, workload scales, and GPUs, delivering substantial acceleration for training and atomistic simulation with high-accuracy equivariant MLIPs. Our full source code will be made publicly available.

1 INTRODUCTION

Scaling equivariant machine learning interatomic potentials (MLIPs) is challenging, since the Clebsch-Gordan tensor products (CGTP) at their heart have computational cost that scales as $O(L_{max}^6)$ in the maximum degree L_{max} (Passaro & Zitnick, 2023; Luo et al., 2024). Consequently, modern equivariant MLIPs are often restricted to $L_{\max}$ to small values (typically no greater than 3), limiting the use of high-degree equivariant features (Batatia et al., 2022; Liao & Smidt, 2023; Park et al., 2024). Ideally, we would like to use higher-degree features to better capture angular resolution and directional information, which are crucial for accurate potential energy surface (PES) modeling (Batzner et al., 2022; Liao et al., 2024; Duval et al., 2024; Xie et al., 2025).

In modern equivariant MLIPs (Batzner et al., 2022; Kim et al., 2026; Xu et al., 2026; Kavanagh et al., 2026), CGTP convolutions can account for over 70% of the training time, making them a computational bottleneck for model scaling. A series of pioneering works have explored ways to optimize CGTP. Their approaches fall into two categories (see Figure 1). Algorithm-level methods reformulate the tensor product or reduce its degrees of freedom, resulting in a trade-off between expressivity and efficiency (Unke & Maennel, 2024; Luo et al., 2024; Passaro & Zitnick, 2023; Xie et al., 2025). By contrast, operator-level methods improve the efficiency of CGTP on GPUs without changing its mathematical formulation or sacrificing expressivity through kernel optimizations (NVIDIA, 2024; Lee et al., 2025;

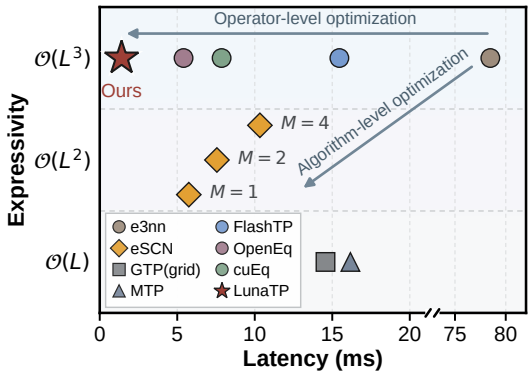


Figure 1: Expressivity–efficiency landscape of tensor product convolutions. Forward latency is measured on an NVIDIA H100 GPU with 768 atoms, $L_{\max} = 6$, and 64 channels per degree.

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.

Bharadwaj et al., 2025). Our work follows this direction and introduces LunaTP to further improve the efficiency of CGTP convolutions.

However, further accelerating convolution layers while preserving CGTP remains challenging. Kernel fusion and sparse CG computation cut computation and memory traffic (NVIDIA, 2024; Lee et al., 2025; Bharadwaj et al., 2025), yet edge-parallel execution still incurs atomic write contention when several edges update the same node during message aggregation. Organizing the reduction around nodes offers a way out, but it does not by itself make the kernel faster. Broader reuse across paths costs on-chip resources, which limits how much work can run in parallel. Differences in node degree and per-path cost then make the work hard to balance. Improving the efficiency of CGTP convolutions therefore means co-designing reduction, data reuse, and parallel execution.

To bridge this performance gap, we introduce LunaTP, which improves the execution of CGTP convolutions without sacrificing expressivity. Our main contributions are:

- We rethink the parallel execution of CGTP convolutions and introduce a node-centric execution scheme that accumulates edge messages on chip, reducing global memory updates.
- Building on this execution scheme, we develop path-group parallelism to balance input reuse and register usage, and reduce redundant computation through sparse expansion and multi-level reuse.
- We extend this execution scheme to higher-order backward passes, enabling efficient force- and Hessian-supervised training.
- We evaluate LunaTP through kernel benchmarks and end-to-end experiments on representative MLIP workloads. On an NVIDIA H100, LunaTP achieves up to a 250.0× kernel speedup and 28.7× higher training throughput than e3nn.

2 PRELIMINARIES

2.1 CLEBSCH–GORDAN TENSOR PRODUCT

The Clebsch–Gordan tensor product (CGTP) computes each degree-l component in the direct-sum decomposition. Given two irreducible representations (irreps) $\mathbf{x}^{(l_1)} \in \mathbb{R}^{2l_1+1}$ and $\mathbf{y}^{(l_2)} \in \mathbb{R}^{2l_2+1}$, the m-th component of the degree-l output is given by:

$$\left[\mathbf{x}^{(l_1)} \otimes \mathbf{y}^{(l_2)}\right]_m^{(l)} = \sum_{m_1=-l_1}^{l_1} \sum_{m_2=-l_2}^{l_2} C^{(l,m)}_{(l_1,m_1)(l_2,m_2)} x^{(l_1)}_{m_1} y^{(l_2)}_{m_2}, \tag{1}$$

where $C^{(l,m)}_{(l_1,m_1)(l_2,m_2)}$ are the CG coefficients and are nonzero only when $|l_1 - l_2| \le l \le |l_1 + l_2|$, making the coefficient matrices more than 70% sparse (Bharadwaj et al., 2025; Lee et al., 2025).

We refer to each coupling from input degrees l_1 and l_2 to an output degree l as a *path*, denoted by $(l_1, l_2) \to l$. Each path can be assigned a separate learnable weight while preserving equivariance. For irreps up to degree L, the full tensor product contains $\mathcal{O}(L^3)$ paths. In a naive implementation, each path requires $\mathcal{O}(L^3)$ multiply–add operations, yielding $\mathcal{O}(L^6)$ total complexity.

To understand how per-path CGTPs are combined in equivariant MLIPs, we next describe the tensor product convolution layer.

2.2 TENSOR PRODUCT CONVOLUTION

A tensor product convolution computes one CGTP for every combination of edge, channel, and path (Geiger & Smidt, 2022; Batzner et al., 2022; Batatia et al., 2022; Park et al., 2024). With E edges, C channels, and P paths, the layer performs a total of $E \times C \times P$ such computations. The resulting edge messages are then aggregated at their destination nodes.

Algorithm 1 presents the pseudocode for the tensor product convolution. For each edge ji, the CGTP couples the source-node irrep $\mathbf{x}_j^{(l_1,c)}$ with the angular edge irrep $\mathbf{y}_{ji}^{(l_2)}$. The result is a full degree-l output, whose m-th component is given by Eq. 1. This output is scaled by the corresponding path

Algorithm 1 Standard Clebsch–Gordan Tensor Product Convolution

Input: $\mathbf{x}_j^{(l_1,c)}$: node irreps
Input: $\mathbf{y}_{ji}^{(l_2)}$: angular edge irreps
Input: $w_{ji}^{(l_1,l_2,l,c)}$: path weights generated by the radial MLP
Input: edges ji, from source node j to destination node i
Output: $\mathbf{x'}_i^{(l,c)}$: updated node irreps

1: Initialize $\mathbf{x'}_i \leftarrow \mathbf{0}$ for all nodes i
2: **for** ji in the E edges **do** ▷ Compute edge messages
3: Initialize $\widetilde{\mathbf{m}}_{ji} \leftarrow \mathbf{0}$
4: **for** c in the C channels **do** ▷ Process each feature channel
5: **for** $(l_1, l_2) \rightarrow l$ in the P paths **do** ▷ Compute a CGTP for each path
6: $\widetilde{\mathbf{m}}_{ji}^{(l,c)} \leftarrow \widetilde{\mathbf{m}}_{ji}^{(l,c)} + w_{ji}^{(l_1,l_2,l,c)} \left[\mathbf{x}_j^{(l_1,c)} \otimes \mathbf{y}_{ji}^{(l_2)}\right]^{(l)}$
7: **end for**
8: **end for**
9: $\mathbf{x'}_i \leftarrow \mathbf{x'}_i + \widetilde{\mathbf{m}}_{ji}$ ▷ Aggregate at destination node i
10: **end for**

weight $w_{ji}^{(l_1,l_2,l,c)}$ and accumulated into the edge message (Line 6). The same operation is performed for every channel–path combination to construct the edge message $\widetilde{\mathbf{m}}_{ji}$ (Lines 2–8). The message is then aggregated at destination node i (Line 9). In particular, training conservative MLIPs typically requires higher-order derivatives, and we provide the corresponding pseudocode in Appendix F.

While the standard implementation is mathematically straightforward, its performance on modern GPUs depends on how the computation is organized, as we discuss next.

3 Rethinking Parallelism for CGTP Convolution

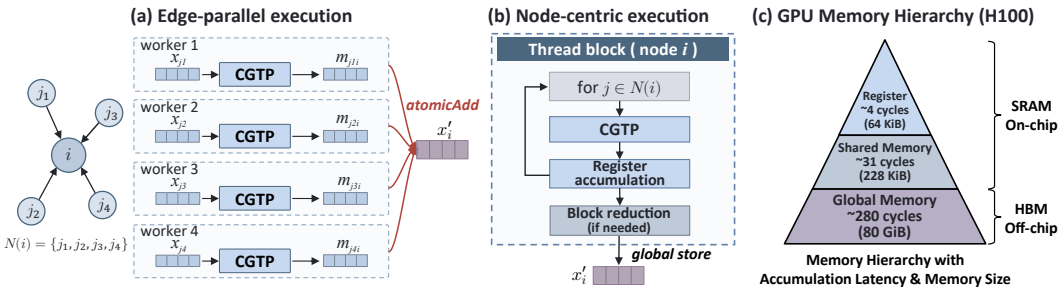


Figure 2: CGTP convolution parallelism and accumulation latencies across the memory hierarchy.

Parallelism across edges is a common execution scheme in CGTP convolutions (Figure 2(a)) (Lee et al., 2025; Bharadwaj et al., 2025). Since CGTP on each edge are independent, this scheme exploits data parallelism by assigning per-edge computations to different workers. However, during message aggregation (Algorithm 1, line 9), many workers attempt to update the same node concurrently, causing atomic contention and hardware-level serialization. Our profiling on an NVIDIA H100 GPU shows that aggregation can account for more than 30% of the total runtime. Thus, although kernel fusion and sparse CG evaluation reduce intermediate memory accesses and tensor product computation, aggregation still incurs substantial overhead.

On modern GPUs, the latency of accumulation depends on where the data resides. As Figure 2(c) shows, accumulation in registers and shared memory typically has lower latency than global atomic updates. This raises a natural question: can we use these low-latency on-chip memories to perform neighbor aggregation?

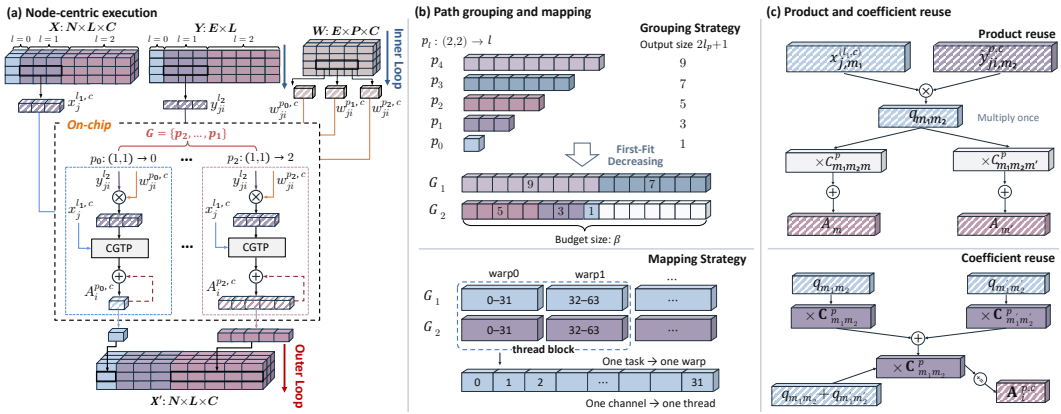


Figure 3: Overview of LunaTP. **(a)** Node-centric execution accumulates edge messages on chip before storing results. **(b)** Paths are grouped under an output component budget and mapped to warps by channel block. **(c)** Products are reused across outputs, and matching coefficients are factored.

Node-centric execution provides a natural way to do so (Figure 2(b)). This scheme aggregates messages for each node on chip and writes the aggregated result to global memory. A straightforward way to implement this is to assign one thread block to each node. However, our experiments show that this direct mapping eliminates per-edge atomic updates but does not improve runtime, as it does not fully exploit intra-node parallelism. Thus, efficient node-centric execution still faces the following primary challenges:

(1) Work partitioning limited by on-chip reduction. Node-centric execution requires each thread block to traverse all edges of a node and aggregate messages on chip. This makes work partitioning difficult, since the workload depends on both node degree and path computation. On the one hand, node degree determines how many edges a block traverses (e.g., degree distributions are often skewed in atomistic datasets (Barros-Luque et al., 2026; Levine et al., 2026; Gharakhanyan et al., 2026)). On the other hand, the amount of CG computation also varies considerably across paths. Thus, blocks computing the same number of output features may still have different workloads.

(2) Fusion scope limited by register capacity. In node-centric execution, we aim to exploit shared inputs across paths to reduce redundant memory accesses and computation. However, each fused path needs to keep its output accumulators live throughout neighbor reduction, so a larger fusion scope increases register demand and may lower occupancy or cause spilling. Under identical thread configuration and register constraints, fusing all compatible paths runs 35.4% slower than limiting the fusion scope and incurs register spilling.

4 LUNATP

In this section, we introduce LunaTP's node-centric execution (Section 4.1). Building on this execution scheme, we describe path-group parallelism (Section 4.2), followed by sparse expansion and multilevel reuse (Section 4.3). Finally, we present higher-order differentiation and GPU implementation details (Section 4.4).

4.1 NODE-CENTRIC EXECUTION

LunaTP organizes CGTP computation around destination nodes, accumulates edge messages on chip, and stores the results in global memory after the inner loop. Figure 3(a) shows this execution scheme. The outer loop runs in parallel across destination nodes, paths, and channels, while the inner loop traverses the edges of each destination node. To support this traversal, LunaTP stores the adjacency list in CSR format by destination node, with the edges of node i in the range $[\text{rowptr}[i], \text{rowptr}[i+1])$.

In each inner-loop iteration, a thread loads the source-node features $\mathbf{x}_j^{(l_1,c)}$, edge features $\mathbf{y}_{ji}^{(l_2)}$, and path weight $w_{ji}^{p,c}$ for edge ji. It computes CGTP between the source-node features and the weighted edge features $\widetilde{\mathbf{y}}_{ji}^{p,c} = w_{ji}^{p,c}\mathbf{y}_{ji}^{(l_2)}$. The thread computes only component products corresponding to nonzero CG terms, multiplies them by their CG coefficients, and accumulates the results directly in registers. For path $p = (l_1, l_2) \to l$, the accumulator $\mathbf{A}_{i,c}^{p}$ is defined as

$$\mathbf{A}_i^{p,c} = \sum_{j \in \mathcal{N}(i)} \left[\mathbf{x}_j^{(l_1,c)} \otimes \widetilde{\mathbf{y}}_{ji}^{p,c}\right]^{(l)}, \tag{2}$$

where $\mathcal{N}(i)$ denotes the neighbors of node i, $w_{ji}^{p,c}$ is the path weight generated by the radial MLP, and the CG tensor product is defined in Eq. 1.

After traversing all edges of the destination node, each thread stores its accumulated result once at the corresponding output location. Computation across channels and paths remains parallel, while global updates decrease from $O(E)$ to $O(N)$ for each fixed path and channel. However, the accumulators $\mathbf{A}_i^{p,c}$ need to remain in registers throughout the inner loop. Computing more paths that share inputs can improve reuse but also increases register demand. Work partitioning therefore needs to account for both input reuse and accumulator storage. The next section describes how path-group parallelism controls group size and organizes parallel computation.

4.2 Path-Group Parallelism

Multiple paths for the same destination node can share source features. Prior work aggregates these paths to reduce repeated loads (Lee et al., 2025). However, the aggregation scope affects both reuse and register usage. Aggregating all paths that share inputs keeps their output accumulators live until the inner loop ends, which may cause register spilling and limit parallelism. Processing each path separately reduces register usage but repeatedly loads the same features, limiting input reuse. LunaTP therefore uses a grouping strategy to balance input reuse and register usage. For a path group G, let l_p denote the output degree of path p. The number of output components retained per channel is

$$R_G = \sum_{p \in G} (2l_p + 1), \tag{3}$$

LunaTP uses $R_G \leq \beta$ as the grouping constraint, where β is the per-channel output component budget ($\beta = 16$ in our experiments). We use First-Fit Decreasing (FFD) to group paths in descending order of output size, as shown in Figure 3(b).

Since each path group can be computed independently, we use path groups as units of parallel work, which we call *path-group parallelism*. Specifically, LunaTP partitions work across destination nodes, path groups, and channels. Each task is identified by (i, G, B), where i is the destination node, G is the path group, and B is a contiguous channel block. In the mapping strategy shown in Figure 3(b), one warp processes one task covering 32 consecutive channels, with each thread handling one channel and computing all paths in the group to reuse the loaded source features. Each task traverses all edges of its destination node and completes its accumulation on chip, avoiding an additional reduction to combine partial results from splitting the edge loop.

Task ordering. This partitioning provides more parallel tasks, but differences in node degree and per-path cost can still cause workload imbalance within a thread block. When some warps finish early, the block must wait for the remaining warps before releasing all its resources. To reduce this tail effect, LunaTP sorts path groups by their total number of nonzero CG terms in descending order, then sorts nodes within each group by degree in descending order, placing heavier tasks first and helping assign tasks with similar workloads to the same thread block.

4.3 Sparse Expansion and Multi-Level Reuse

LunaTP exploits sparsity and reuse in the CG structure to reduce memory accesses and computation. Path grouping enables input reuse across paths, while product and coefficient reuse further reduce computation within each path.

Sparse expansion. CG coefficients and their component indices are determined by the path and do not depend on the graph or input features. Like OpenEq (Bharadwaj et al., 2025), LunaTP embeds normalized nonzero coefficients and their indices directly in the generated code, avoiding runtime table lookups and checks for zero terms.

Beyond CG sparsity, LunaTP uses path grouping to reuse loaded inputs across paths. In each inner-loop iteration, a thread loads each source-feature and edge-feature component needed by the group once and shares it across the paths that use it.

Product reuse. The same input product $q_{m_1 m_2} = x_{j,m_1}^{(l_1,c)} \widetilde{y}_{ji,m_2}^{p,c}$ may be used by multiple output components. LunaTP therefore organizes CG terms by input component pair, computes each product once, and reuses it for all terms with $C_{m_1 m_2 m}^{p} \neq 0$, as shown in the upper part of Figure 3(c). Here, m_1 and m_2 follow the input component indices in Eq. 1.

Coefficient reuse. Different input component pairs may have coefficient vectors that are identical up to an overall sign. Let $\mathbf{C}_{m_1 m_2}^{p} \in \mathbb{R}^{2l+1}$ collect the CG coefficients over all output components. When $\mathbf{C}_{m_1' m_2'}^{p} = \pm \mathbf{C}_{m_1 m_2}^{p}$, we combine the products before applying the shared coefficients,

$$q_{m_1 m_2} \mathbf{C}_{m_1 m_2}^{p} + q_{m_1' m_2'} \mathbf{C}_{m_1' m_2'}^{p} = (q_{m_1 m_2} \pm q_{m_1' m_2'}) \mathbf{C}_{m_1 m_2}^{p}. \tag{4}$$

This reduces coefficient multiplications, as shown in the lower part of Figure 3(c). In particular, we identify these matches only within each path at operator construction time.

4.4 Implementation Details

Extension to higher-order derivatives. LunaTP extends the node-centric execution scheme and reuse methods above to backward, double-backward, and triple-backward passes by adapting the reduction direction to the differentiation target. Computing the gradient of node features $\mathbf{x}_j$ requires accumulating results over all edges with source node j, so reduction shifts from destination nodes to source nodes. To support this traversal, LunaTP organizes a CSR by source node while retaining the partitioning across nodes, path groups, and channels and accumulating gradients on chip. Derivatives with respect to the incoming gradient instead reduce at destination nodes, as in the forward pass. Edge-feature and path-weight derivatives are computed per edge, with edge-feature derivatives also requiring a reduction across channels. Double-backward and triple-backward use the same approach, organizing computation around the reduction required by each output. The full pseudocode is provided in Appendix F.

Higher-order backward passes also allow reuse across derivative branches. In double-backward, for example, computing node-feature and edge-feature derivatives requires the same incoming gradient for a given edge. LunaTP combines branches that share inputs to avoid repeated loads and applies sparse expansion, product reuse, and coefficient reuse within each branch.

Low-level optimization and interface. LunaTP uses `__ldg` to load read-only features, weights, and adjacency indices, and assigns consecutive channels to threads within each warp to coalesce path-weight loads. We express accumulation as multiply-add operations so the compiler can generate fused multiply-add instructions. In higher-order backward passes, we assign independent derivative branches to different CUDA streams to overlap their execution. Meanwhile, we provide a programming interface for LunaTP to improve ease of use (see Appendix G).

5 Experiments

We evaluate LunaTP across four dimensions: computational efficiency, memory efficiency, scalability, and accuracy. Our baselines include e3nn (Geiger & Smidt, 2022), FlashTP (Lee et al., 2025), OpenEquivariance (OpenEq) (Bharadwaj et al., 2025), and cuEquivariance (cuEq) (NVIDIA, 2024). We report kernel microbenchmark results (Section 5.1), end-to-end training and inference performance (Section 5.2), and scalability analysis (Section 5.3). The experimental setup is detailed in Appendix A and ablation studies are described in Appendix C.

5.1 KERNEL MICROBENCHMARK

The microbenchmarks cover ten irrep configurations with $L_{\max}$ ranging from 2 to 6. Most are based on configurations used in existing equivariant MLIPs (Batzner et al., 2022; Batatia et al., 2022; Park et al., 2024; Kim et al., 2026; Kavanagh et al., 2026). The hidden, edge, and output irreps and their representative MLIPs are summarized in Appendix Table 1.

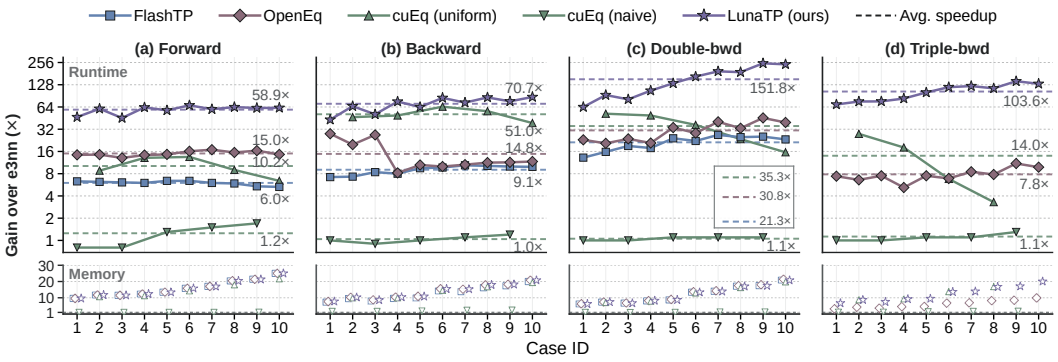


Figure 4: LunaTP performance in kernel microbenchmarks. Solid lines show runtime speedup over e3nn, while hollow markers show the reduction in peak memory relative to e3nn. Evaluations are performed in FP32 on NVIDIA H100 80GB GPUs. The raw data are provided in Appendix B.1.

Runtime and peak memory. Figure 4 summarizes the FP32 microbenchmark results on H100 (FP64 results are provided in Appendix D.1). LunaTP achieves average speedups of 58.9×, 70.7×, 151.8×, and 103.6× over e3nn for forward, backward, double-backward, and triple-backward, respectively. Its corresponding speedups over OpenEq are 3.93×, 5.93×, 4.78×, and 13.31×. Although FlashTP and OpenEq already use kernel fusion to reduce memory traffic, LunaTP further improves performance through node-centric execution, path-group parallelism, and multilevel reuse. The gains are most pronounced for double- and triple-backward, which offer more opportunities to share data and computation across derivative branches. cuEq also performs well with uniform channel counts across degrees, but uses a naive implementation with limited speedups otherwise.

In addition to runtime improvements, LunaTP uses only 4.0%–16.2% of e3nn's peak memory. Its memory footprint is comparable to that of FlashTP and OpenEq from forward through double-backward, since all three avoid materializing per-edge message tensors. For triple-backward, LunaTP uses approximately 48% of OpenEq's peak memory while also achieving lower runtime.

Numerical stability and training convergence. Since kernel fusion and parallel reductions can alter floating-point operation order, LunaTP and e3nn may exhibit small numerical discrepancies. Using e3nn's FP64 results as the reference, we measure the RMSEs of their FP32 outputs and corresponding derivatives. From the forward pass through third-order derivatives, both implementations maintain FP32 RMSEs on the order of 10^{-7}–10^{-6}, while LunaTP's FP64 RMSEs remain between 10^{-16} and 10^{-14} (see Appendix E.1).

To assess training stability, we train NequIP-XL (Batzner et al., 2022; Kavanagh et al., 2026) on MPTrj-PFT (Deng et al., 2023; Koker et al., 2026) with the same settings. LunaTP's validation MAEs for E, F, S, and H differ from those of e3nn by less than 0.3% across all targets, while showing similar convergence trends. (see Appendix E.2).

Hardware portability. We evaluate LunaTP on NVIDIA A100 and RTX 3090 GPUs to assess its portability across data-center and consumer hardware. These GPUs differ in compute performance and memory bandwidth. LunaTP also has a performance advantage over the baselines on both GPUs. Full results are reported in Appendix D.2.

5.2 End-to-end Performance

We evaluate LunaTP's end-to-end performance in both training and inference. For training, we measure throughput on eight state-of-the-art (SOTA) equivariant MLIPs with EFS and EFSH objectives. For inference, we measure MD speed and the maximum system size each backend can run. Experimental details are provided in Appendix A.3.

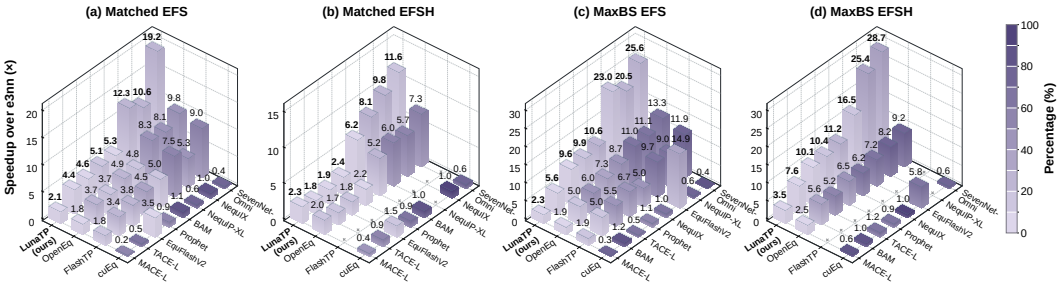


Figure 5: End-to-end training performance. We benchmark EFS (energy, forces, and stress) and EFSH (EFS plus a Hessian objective). Matched uses equal batch sizes across backends, and MaxBS selects each backend's maximum batch size. Bars show throughput speedup over e3nn, while bar color indicates the CGTP convolution time fraction. The raw data are provided in Appendix B.2.

Training. As shown in Figure 5(a) and (b), LunaTP achieves speedups of 2.10–19.24× over e3nn for EFS and 1.81–11.6× for EFSH at matched batch sizes (Matched). The speedup depends on the fraction of training time spent in CGTP convolutions. For example, MACE-L achieves a 2.10× speedup under Matched EFS. With e3nn, CGTP convolutions account for 53.5% of runtime, limiting the overall speedup achievable by optimizing this operator. In contrast, SevenNet-Omni spends a larger fraction of runtime in CGTP convolutions and achieves a 19.2× speedup. Under Matched EFSH, LunaTP achieves speedups of 1.06–1.72× over OpenEq, with further gains also limited by work outside CGTP convolutions.

To evaluate larger workloads, we increase the batch size to the maximum each backend can run on a single GPU (MaxBS). As shown in Figure 5(c) and (d), LunaTP achieves speedups of 2.30–25.6× over e3nn for EFS and 3.50–28.7× for EFSH. The corresponding speedups over OpenEq are 1.14–2.10× and 1.34–3.11×. These gains over OpenEq generally exceed those under Matched, showing that LunaTP provides greater end-to-end benefits on larger workloads.

Inference. We evaluate inference scalability on TACE-L and NequIP-XL, as shown in Figure 6. LunaTP achieves an 8.70× speedup on TACE-L and a 15.27× speedup on NequIP-XL at e3nn's maximum system size. Since kernel fusion avoids storing per-edge message tensors, LunaTP increases the maximum atom count by 4.63× and 5.04× for TACE-L and NequIP-XL, respectively. OpenEq and FlashTP also use kernel fusion, so all three backends reach the same maximum system size. At these workloads, LunaTP remains 1.8× faster on TACE-L and 2.1× faster on NequIP-XL than OpenEq. This shows that LunaTP's optimizations further improve inference performance even with kernel fusion already in place.

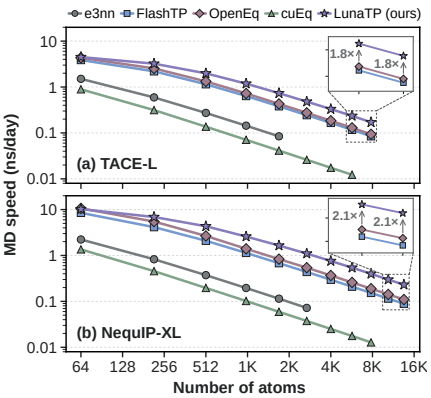


Figure 6: Inference scaling. MD speed on Si with a lattice of 5.43 Å and a 1-fs time step.

5.3 Scalability Analysis

We evaluate LunaTP's forward scalability across four dimensions: atom count, average in-degree, maximum degree, and channel size. As Figure 7

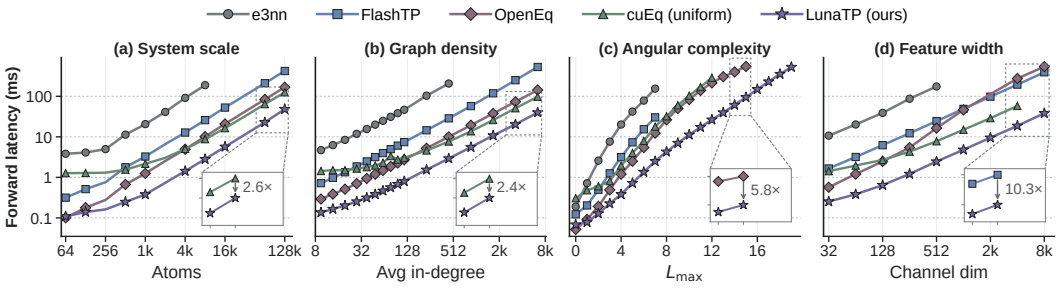


Figure 7: Forward scalability of CGTP convolution.

shows, latency generally increases with workload size, but LunaTP achieves the lowest latency at nearly all measured points. At the largest comparable point in each sweep, it is 2.4–10.3× faster than the fastest available baseline, with larger advantages in the maximum-degree and channel-size sweeps. As $L_{\max}$ increases, the number of coupling paths and CG terms grows. LunaTP controls accumulator storage through path grouping and reduces repeated computation through multilevel reuse. As channels become wider, LunaTP partitions the additional computation into more independent channel-block tasks that run across warps without increasing the per-thread accumulator count under the basic mapping. These results demonstrate LunaTP's scalability to large graphs, higher-degree features, and wider channels. The experimental setup is provided in Appendix A.2.

6 Related Work

Algorithm-level optimization. SO(2)-based methods align a local coordinate system with each edge direction to reduce SO(3) convolutions to simpler SO(2) operations (Passaro & Zitnick, 2023). Other approaches reformulate feature representations and coupling operations to exploit efficient computational primitives. Fused Tensor Product organizes computations across irreps into matrix multiplications (Unke & Maennel, 2024), while Gaunt Tensor Product uses a Fourier basis and FFTs to accelerate products of spherical functions (Luo et al., 2024). These approaches offer efficient alternatives for high-degree equivariant models, although their parameterizations can involve trade-offs between expressivity and efficiency (Xie et al., 2025). LunaTP instead improves execution efficiency while preserving the original CGTP formulation and expressivity.

Operator-level optimization. A complementary direction improves tensor product implementations on GPUs. e3nn provides general-purpose equivariant operators (Geiger & Smidt, 2022), while cuEquivariance offers specialized GPU kernels (NVIDIA, 2024). FlashTP exploits CG sparsity, aggregates paths sharing source features, and fuses convolution operations to reduce repeated loads and intermediate storage (Lee et al., 2025). OpenEq generates sparse kernels just in time for CG contractions and supports fused graph convolutions and higher-order derivatives. Its deterministic implementation also accumulates results locally over sorted edges to reduce global updates (Bharadwaj et al., 2025). Building on these advances, LunaTP focuses on parallel execution and reuse within node-centric computation. It groups paths under an accumulator budget to expose intra-node parallelism and reuses common products and coefficients to reduce computation. Our experiments demonstrate further performance gains over these strong baselines.

7 Conclusion

In this work, we rethink the parallelism of CGTP convolutions and present LunaTP, a GPU kernel for CGTP convolutions in equivariant MLIPs. By organizing node reduction, data reuse, and work partitioning, LunaTP improves computational efficiency without sacrificing expressivity, and extends to third-order derivatives, making efficient Hessian-based training practical. Experiments show that LunaTP accelerates kernels by up to $250.0\times$ over e3nn and $17.1\times$ over OpenEq, while delivering up to $28.7\times$ the end-to-end training throughput of e3nn. Future work will extend these optimizations to other operators in equivariant MLIPs.

AI USE STATEMENT

In this work, we did not use generative AI tools for any tasks requiring disclosure. We used generative AI tools only for language editing to improve the clarity and readability of the manuscript and for literature search, which are tasks with recommended disclosure. We reviewed all AI-assisted edits and independently verified the identified literature and references. We take responsibility for the final content of this work, including all AI-assisted language edits and literature-related content.

REPRODUCIBILITY STATEMENT

We describe our methods in Section 4. Appendix A provides the experimental setup, including the hardware and software environments, benchmarking protocols, and model configurations. The raw benchmark data underlying the results reported in the main text are provided in Appendix B. We will make the complete source code publicly available.

REFERENCES

Luis Barros-Luque, Muhammed Shuaibi, Xiang Fu, Brandon M. Wood, Misko Dzamba, Meng Gao, Ammar Rizvi, Matt Uyttendaele, C. Lawrence Zitnick, and Zachary W. Ulissi. The Open Materials 2024 (OMat24) inorganic materials dataset and models. *Nature Computational Science*, 6(6):642–652, June 2026. ISSN 2662-8457. doi: 10.1038/s43588-026-00996-w. URL https://doi.org/10.1038/s43588-026-00996-w.

Ilyes Batatia, David Peter Kovacs, Gregor N. C. Simm, Christoph Ortner, and Gabor Csanyi. MACE: Higher order equivariant message passing neural networks for fast and accurate force fields. In Alice H. Oh, Alekh Agarwal, Danielle Belgrave, and Kyunghyun Cho (eds.), *Advances in Neural Information Processing Systems*, 2022. URL https://openreview.net/forum?id=YPpSngE-ZU.

Ilyes Batatia, Philipp Benner, Yuan Chiang, Alin M. Elena, Dávid P. Kovács, Janosh Riebesell, Xavier R. Advincula, Mark Asta, Matthew Avaylon, William J. Baldwin, Fabian Berger, Noam Bernstein, Arghya Bhowmik, Filippo Bigi, Samuel M. Blau, Vlad Cărare, Michele Ceriotti, Sanggyu Chong, James P. Darby, Sandip De, Flaviano Della Pia, Volker L. Deringer, Rokas Elijošius, Zakariya El-Machachi, Edvin Fako, Fabio Falcioni, Andrea C. Ferrari, John L. A. Gardner, Mikołaj J. Gawkowski, Annalena Genreith-Schriever, Janine George, Rhys E. A. Goodall, Jonas Grandel, Clare P. Grey, Petr Grigorev, Shuang Han, Will Handley, Hendrik H. Heenen, Kersti Hermansson, Cheuk Hin Ho, Stephan Hofmann, Christian Holm, Jad Jaafar, Konstantin S. Jakob, Hyunwook Jung, Venkat Kapil, Aaron D. Kaplan, Nima Karimitari, James R. Kermode, Panagiotis Kourtis, Namu Kroupa, Jolla Kullgren, Matthew C. Kuner, Domantas Kuryla, Guoda Liepuoniute, Chen Lin, Johannes T. Margraf, Ioan-Bogdan Magdău, Angelos Michaelides, J. Harry Moore, Aakash A. Naik, Samuel P. Niblett, Sam Walton Norwood, Niamh O'Neill, Christoph Ortner, Kristin A. Persson, Karsten Reuter, Andrew S. Rosen, Louise A. M. Rosset, Lars L. Schaaf, Christoph Schran, Benjamin X. Shi, Eric Sivonxay, Tamás K. Stenczel, Christopher Sutton, Viktor Svahn, Thomas D. Swinburne, Jules Tilly, Cas van der Oord, Santiago Vargas, Eszter Varga-Umbrich, Tejs Vegge, Martin Vondrák, Yangshuai Wang, William C. Witt, Thomas Wolf, Fabian Zills, and Gábor Csányi. A foundation model for atomistic materials chemistry. *The Journal of Chemical Physics*, 163(18):184110, 11 2025. ISSN 0021-9606. doi: 10.1063/5.0297006. URL https://doi.org/10.1063/5.0297006.

Simon Batzner, Albert Musaelian, Lixin Sun, Mario Geiger, Jonathan P. Mailoa, Mordechai Kornbluth, Nicola Molinari, Tess E. Smidt, and Boris Kozinsky. E(3)-equivariant graph neural networks for data-efficient and accurate interatomic potentials. *Nature Communications*, 13(1), May 2022. ISSN 2041-1723. doi: 10.1038/s41467-022-29939-5. URL http://dx.doi.org/10.1038/s41467-022-29939-5.

Vivek Bharadwaj, Austin Glover, Aydin Buluc, and James Demmel. An efficient sparse kernel generator for o(3)-equivariant deep networks, 2025. URL https://arxiv.org/abs/2501.13986.

Bowen Deng, Peichen Zhong, KyuJung Jun, Janosh Riebesell, Kevin Han, Christopher J. Bartel, and Gerbrand Ceder. CHGNet as a pretrained universal neural network potential for charge-informed atomistic modelling. *Nature Machine Intelligence*, 5(9):1031–1041, September 2023. ISSN 2522-5839. doi: 10.1038/s42256-023-00716-3. URL https://doi.org/10.1038/s42256-023-00716-3.

Alexandre Duval, Simon V. Mathis, Chaitanya K. Joshi, Victor Schmidt, Santiago Miret, Fragkiskos D. Malliaros, Taco Cohen, Pietro Liò, Yoshua Bengio, and Michael Bronstein. A hitchhiker's guide to geometric gnns for 3d atomic systems, 2024. URL https://arxiv.org/abs/2312.07511.

Xiang Fu, Brandon M Wood, Luis Barroso-Luque, Daniel S. Levine, Meng Gao, Misko Dzamba, and C. Lawrence Zitnick. Learning smooth and expressive interatomic potentials for physical property prediction. In *Forty-second International Conference on Machine Learning*, 2025. URL https://openreview.net/forum?id=R0PBjxIbgm.

Johannes Gasteiger, Janek Groß, and Stephan Günnemann. Directional message passing for molecular graphs, 2022. URL https://arxiv.org/abs/2003.03123.

Mario Geiger and Tess Smidt. e3nn: Euclidean neural networks, 2022. URL https://arxiv.org/abs/2207.09453.

Vahe Gharakhanyan, Luis Barroso-Luque, Yi Yang, Muhammed Shuaibi, Kyle Michel, Daniel S. Levine, Misko Dzamba, Xiang Fu, Meng Gao, Xingyu Liu, Haoran Ni, Keian Noori, Brandon M. Wood, Matt Uyttendaele, Arman Boromand, C. Lawrence Zitnick, Noa Marom, Zachary W. Ulissi, and Anuroop Sriram. Open Molecular Crystals 2025 (OMC25) dataset and models. *Scientific Data*, 13(1):354, February 2026. ISSN 2052-4463. doi: 10.1038/s41597-026-06628-2. URL https://doi.org/10.1038/s41597-026-06628-2.

Seán R. Kavanagh, Chuin Wei Tan, Menghang Wang, Marc L. Descoteaux, Gabriel de Miranda Nascimento, Ulrik Unneberg, Laura Zichi, Francesco Libbi, Norma Rivano, Austin Glover, Vivek Bharadwaj, Anders Johansson, William C. Witt, Albert Musaelian, and Boris Kozinsky. Fast and accurate equivariant foundation models for atomistic simulation, 2026. URL https://arxiv.org/abs/2607.28461.

Jaesun Kim, Jinmu You, Yutack Park, Yunsung Lim, Yujin Kang, Jisu Kim, Haekwan Jeon, Suyeon Ju, Deokgi Hong, Seung Yul Lee, Saerom Choi, Yongdeok Kim, Jae W. Lee, and Seungwu Han. Optimizing cross-domain transfer for universal machine learning interatomic potentials. *Nature Communications*, 17(1):3432, March 2026. ISSN 2041-1723. doi: 10.1038/s41467-026-70195-8. URL https://doi.org/10.1038/s41467-026-70195-8.

Teddy Koker, Abhijeet Sadashiv Gangan, Mit Kotak, Jaime Marian, and Tess Smidt. PFT: Phonon fine-tuning for machine learned interatomic potentials. In *Forty-third International Conference on Machine Learning*, 2026. URL https://openreview.net/forum?id=QC9S8gUOYc.

Seung Yul Lee, Hojoon Kim, Yutack Park, Dawoon Jeong, Seungwu Han, Yeonhong Park, and Jae W. Lee. FlashTP: Fused, sparsity-aware tensor product for machine learning interatomic potentials. In *Forty-second International Conference on Machine Learning*, 2025. URL https://openreview.net/forum?id=wiQe95BPaB.

Daniel S. Levine, Muhammed Shuaibi, Evan Walter Clark Spotte-Smith, Michael G. Taylor, Muhammad R. Hasyim, Kyle Michel, Ilyes Batatia, Gábor Csányi, Misko Dzamba, Peter Eastman, Nathan C. Frey, Xiang Fu, Vahe Gharakhanyan, Aditi S. Krishnapriyan, Joshua A. Rackers, Sanjeev Raja, Ammar Rizvi, Andrew S. Rosen, Zachary Ulissi, Santiago Vargas, C. Lawrence Zitnick, Samuel M. Blau, and Brandon M. Wood. The open molecules 2025 (omol25) dataset, evaluations, and models, 2026. URL https://arxiv.org/abs/2505.08762.

Yi-Lun Liao and Tess Smidt. Equiformer: Equivariant graph attention transformer for 3d atomistic graphs. In *The Eleventh International Conference on Learning Representations*, 2023. URL https://openreview.net/forum?id=KwmPfARgOTD.

Yi-Lun Liao, Brandon M Wood, Abhishek Das, and Tess Smidt. Equiformerv2: Improved equivariant transformer for scaling to higher-degree representations. In *The Twelfth International Conference on Learning Representations*, 2024. URL https://openreview.net/forum?id=mCOBKZmrzD.

Shengjie Luo, Tianlang Chen, and Aditi S. Krishnapriyan. Enabling efficient equivariant operations in the fourier basis via gaunt tensor products. In *The Twelfth International Conference on Learning Representations*, 2024. URL https://openreview.net/forum?id=mhyQXJ6JsK.

NVIDIA. cuEquivariance. https://github.com/NVIDIA/cuEquivariance, 2024.

Yutack Park, Jaesun Kim, Seungwoo Hwang, and Seungwu Han. Scalable parallel algorithm for graph neural network interatomic potentials in molecular dynamics simulations. *Journal of Chemical Theory and Computation*, 20(11):4857–4868, May 2024. ISSN 1549-9626. doi: 10.1021/acs.jctc.4c00190. URL http://dx.doi.org/10.1021/acs.jctc.4c00190.

Saro Passaro and C. Lawrence Zitnick. Reducing so(3) convolutions to so(2) for efficient equivariant gnns. ICML'23. JMLR.org, 2023.

Jay Shah, Ganesh Bikshandi, Ying Zhang, Vijay Thakkar, Pradeep Ramani, and Tri Dao. Flashattention-3: Fast and accurate attention with asynchrony and low-precision. In *The Thirty-eighth Annual Conference on Neural Information Processing Systems*, 2024. URL https://openreview.net/forum?id=tVConYid20.

Oliver T. Unke and Hartmut Maennel. E3x: E(3)-equivariant deep learning made easy, 2024. URL https://arxiv.org/abs/2401.07595.

Soohaeng Yoo Willow, Tae Hyeon Park, Gi Beom Sim, Sung Wook Moon, Seung Kyu Min, D. ChangMo Yang, Hyun Woo Kim, Juho Lee, and Chang Woo Myung. Bayesian e(3)-equivariant interatomic potential with iterative restratification of many-body message passing, 2026. URL https://arxiv.org/abs/2510.03046.

YuQing Xie, Ameya Daigavane, Mit Kotak, and Tess Smidt. The price of freedom: Exploring expressivity and runtime tradeoffs in equivariant tensor products. In *Forty-second International Conference on Machine Learning*, 2025. URL https://openreview.net/forum?id=EvIwwGYTLc.

Zemin Xu, Wenbo Xie, and P. Hu. Spectral/spatial tensor atomic cluster expansion with universal embeddings in cartesian space, 2026. URL https://arxiv.org/abs/2509.14961.

APPENDIX

A Evaluation setup

A.1 Kernel microbenchmark settings

A.2 Scalability Analysis Settings

A.3 Training setup details

B Detailed Benchmark Data

B.1 Raw data for kernel microbenchmarks

B.2 Raw data for training benchmarks

C Ablation Studies

D Additional kernel microbenchmark results

D.1 Kernel microbenchmark (FP64)

D.2 Kernel microbenchmark (A100 & RTX 3090)

E Details on numerical stability and training convergence

E.1 Operator-level numerical stability

E.2 End-to-end training convergence

F Differentiation of CGTP convolution

G Programming interface

A EVALUATION SETUP

Software and hardware. All evaluations use FP32 with `PyTorch 2.11.0+cu128`, e3nn v0.6.0 (Geiger & Smidt, 2022), FlashTP v0.1.1 (Lee et al., 2025), OpenEquivariance v0.7.0 (Bharadwaj et al., 2025), and cuEquivariance v0.11.1 (NVIDIA, 2024). Notably, cuEquivariance uses `uniform_1d` when all segments have the same shape and `naive` otherwise, which we denote as cuEq(uniform) and cuEq(naive), respectively.

Unless otherwise specified, all experiments are conducted on NVIDIA H100-80GB SXM GPUs with Intel Xeon Platinum 8468 CPUs. NVLink provides up to 900 GB/s of bidirectional bandwidth per GPU.

Workload construction protocol. To construct workloads that reflect realistic atomic data distributions, we randomly shuffle the OMat24 dataset (Barros-Luque et al., 2026) and sample structures to form batches with specified atom counts. For each structure, we construct a neighbor graph under periodic boundary conditions with $r_{\mathrm{cut}} = 6.0$ Å, thereby preserving the neighbor distribution of the original dataset.

Measurement protocol. For all microbenchmarks, we warm up each configuration for 10 iterations and then measure GPU kernel execution time over the next 20 iterations using the PyTorch profiler. We report the mean runtime, excluding initial JIT compilation, graph construction, and host-to-device data transfers.

A.1 KERNEL MICROBENCHMARK SETTINGS

All kernel microbenchmarks in Section 5.1 use the same workload unless otherwise specified. The workload contains 768 nodes and 37304 edges, with an average of 48.6 neighbors per atom. To cover different angular-momentum orders and channel sizes, we use irrep configurations from several SOTA equivariant MLIPs, as summarized in Table 1.

Table 1: Irrep configurations used in the kernel microbenchmarks.

Case ID	$L_{\max}$	Hidden irreps	Edge irreps	Output irreps	Representative MLIP
1	2	128x0e+64x1e+32x2e	1x0e+1x1e +1x2e	224x0e+384x1e +352x2e	SevenNet-0 , GNoME
2	2	128x0e+128x1o+128x2e	1x0e+1x1o +1x2e+1x3o	384x0e+640x1o +640x2e+512x3o	MACE-MP-L
3	3	128x0e+64x1o+32x2e +32x3o	1x0e+1x1o +1x2e+1x3o	256x0e+352x1o +384x2e+320x3o	Nequix-MP/OAM , SevenNet-MF/Omni
4	3	256x0e+256x1o+256x2e +256x3o	1x0e+1x1o +1x2e+1x3o	1024x0e+1536x1o +1792x2e+1536x3o	EquFlash-OMat/OAM
5	4	320x0e+96x1o+64x2e +32x3o+32x4e	1x0e+1x1o+1x2e +1x3o+1x4e	544x0e+736x1o+832x2e +800x3o+704x4e	NequIP-XL series
6	4	128x0e+128x1o+128x2e +128x3o+128x4e	1x0e+1x1o+1x2e +1x3o+1x4e	640x0e+1024x1o+1280x2e +1280x3o+1152x4e	EquFlashV2-OMat/OAM
7	5	128x0e+64x1o+32x2e +32x3o+32x4e+32x5o	1x0e+1x1o+1x2e +1x3o+1x4e+1x5o	320x0e+480x1o+576x2e +608x3o+608x4e+512x5o	–
8*	5	96x0e+96x1o+96x2e +96x3o+96x4e+96x5o	1x0e+1x1o+1x2e +1x3o+1x4e+1x5o	576x0e+960x1o+1248x2e +1344x3o+1344x4e+1152x5o	–
9	6	128x0e+64x1o+32x2e+32x3o +32x4e+32x5o+32x6e	1x0e+1x1o+1x2e+1x3o +1x4e+1x5o+1x6e	352x0e+544x1o+672x2e+736x3o +768x4e+736x5o+640x6e	–
10*	6	64x0e+64x1o+64x2e+64x3o +64x4e+64x5o+64x6e	1x0e+1x1o+1x2e+1x3o +1x4e+1x5o+1x6e	448x0e+768x1o+1024x2e+1152x3o +1216x4e+1152x5o+1024x6e	–

"*" e3nn encounters OOM under the default workload; these cases therefore use 448 atoms and 20386 edges.
"–" Since no existing MLIP uses the corresponding settings, we constructed these configurations for evaluation.

A.2 SCALABILITY ANALYSIS SETTINGS

We evaluate LunaTP's FP32 forward scalability along four dimensions: atom count, graph density, maximum degree, and channel size. In each test, we vary one dimension while holding the others fixed, and continue until OOM or compilation timeout.

Scaling with atom count. We combine distinct OMat structures and double the atom count starting from 16. We use a 6.0 Å neighbor cutoff, $L_{\max} = 4$, and 64 channels per degree.

Scaling with graph density. Using a fixed 768-atom batch, we increase the neighbor cutoff from 3.5 Å to 30.5 Å while keeping $L_{\max} = 4$ and 64 channels per degree.

Scaling with degree. For this test and the channel-size test, we use the same fixed 768-atom graph. We hold the channel count at 64 per degree and increase $L_{\max}$ from 0 in steps of one.

Scaling with channel size. We fix the 768-atom graph and $L_{\max} = 4$, and increase the number of channels per degree from 32 by successive doubling.

A.3 Training Setup Details

We benchmark training in FP32 on NVIDIA H100 GPUs, using OMat-1M (Barros-Luque et al., 2026) for EFS and MPTrj-PFT (Deng et al., 2023; Koker et al., 2026) for EFSH. All models are trained using their official PyTorch implementations (Batatia et al., 2025; Kim et al., 2026; Koker et al., 2026; Xu et al., 2026; Kavanagh et al., 2026; Willow et al., 2026). For EFSH, we follow the training protocol of Koker et al. (2026) and sample Hessian columns during training. Tables 2 and 3 summarize the model configurations and per-GPU batch sizes, respectively.

Table 2: MLIP configurations for training.

(a) MACE-L (10.10M)

Hyper-parameter	Value
Layers	2
Cutoff radius	5.0 Å
Stage	**Irreps**
Edge	1x0e+1x1o +1x2e+1x3o
Input / Output(L2)	128x0e / 128x0e
Hidden (L1)	128x0e +128x1o +128x2e

(b) NequIP-XL (32.14M)

Hyper-parameter	Value
Layers	6
Cutoff radius	6.0 Å
Stage	**Irreps**
Edge	1x0e+1x1o+1x2e +1x3o+1x4e
Input / Output(L6)	32x0e / 320x0e
Hidden (L1–5)	320x0e+96x1o +64x2e+32x3o +32x4e

(c) EquFlashV2 (44.89M)

Hyper-parameter	Value
Layers	10
Cutoff radius	6.0 Å
Stage	**Irreps**
Edge	1x0e+1x1o+1x2e +1x3o+1x4e
Input / Output(L10)	128x0e / 128x0e
Hidden (L1–9)	128x0e+128x1o +128x2e+128x3o +128x4e

(d) TACE-L (82.90M)

Hyper-parameter	Value
Layers	5
Cutoff radius	6.0 Å
Stage	**Irreps**
Edge	1x0e+1x1o+1x2e +1x3o+1x4e+1x5o
Input / Output (L5)	64x0e / 64x0e
Hidden (L1–4)	64x0e+64x1o +64x2e+64x3o +64x4e+64x5o

(e) NequIX (0.71M)

Hyper-parameter	Value
Layers	4
Cutoff radius	6.0 Å
Stage	**Irreps**
Edge	1x0e+1x1o +1x2e+1x3o
Input / Output (L4)	89x0e / 128x0e
Hidden (L1–3)	128x0e+64x1o +32x2e +32x3o

(f) Prophet (62.30M)

Hyper-parameter	Value
Layers	10
Cutoff radii	3.0, 7.0 Å
Stage	**Irreps**
Edge	1x0e+1x1o+1x2e +1x3o+1x4e
Input / Output(L10)	256x0e / 256x0e
Hidden (L1–9)	256x0e+96x1o +64x2e+32x3o +32x4e

(g) SevenNet-Omni (25.76M)

Hyper-parameter	Value
Layers	5
Cutoff radius	6.0 Å
Stage	**Irreps**
Edge	1x0e+1x1e+1x2e+1x3e
Input / Output (L5)	128x0e / 128x0e
Hidden (L1, L4)	128x0e+64x1o+32x2e+32x3o
Hidden (L2)	128x0e+64x1o+64x1e +32x2+32x2e+32x3o+32x3e
Hidden (L3)	128x0e+128x0e+64x1o+64x1e +32x2o+32x2e+32x3o+32x3e

(h) BAM (40.15M)

Hyper-parameter	Value
Layers	5
Cutoff radius	5.0 Å
Stage	**Irreps**
Edge	1x0e+1x1o +1x2e+1x3o
Input	128x0e
Hidden (L1–5)	64x0o+64x0e +64x1o+64x1e +64x2o+64x2e +64x3o+64x3e

Table 3: Training batch sizes (atoms per GPU). Matched uses the same batch size across all backends, while MaxBS uses the maximum batch size supported by each backend.

Atoms	SevenNet-Omni	MACE-L	NequIP-XL	EquFlashV2	TACE-L	NequIX	Prophet	BAM
				Matched				
EFS	2340	2450	1630	290	1030	8900	430	1280
EFSH	870	800	600	90	300	3200	120	580
				MaxBS: EFS				
e3nn	2340	2450	1630	290	1030	8900	430	1280
FlashTP	14600	3850	9550	2550	3940	30400	2220	3440
OpenEq	14600	3850	9550	2550	3940	37600	2220	3440
cuEq	14600	3850	3300	2550	2340	14500	510	1600
LunaTP	14600	3850	9550	2550	3940	37600	2220	3440
				MaxBS: EFSH				
e3nn	870	800	600	90	300	3200	120	580
FlashTP	–	–	–	–	–	–	–	–
OpenEq	3830	2600	2750	660	1380	9870	600	1700
cuEq	3830	3140	–	660	1340	3060	120	580
LunaTP	5130	6200	3730	860	1650	13200	680	1700

B Detailed Benchmark Data

B.1 Raw Data for Kernel Microbenchmarks

We evaluate LunaTP using the experimental setup described in Section A. Table 4 reports the raw results of our kernel microbenchmarks.

Table 4: Kernel measurements from forward through third-order differentiation. Parentheses report runtime speedup and peak-memory reduction relative to e3nn.

Case ID	Forward					Backward				
	e3nn	FlashTP	OpenEq	cuEq	LunaTP (ours)	e3nn	FlashTP	OpenEq	cuEq	LunaTP (ours)
					Runtime (ms)					
C1	3.56 (1.00x)	0.563 (6.32x)	0.248 (14.4x)	4.40 (0.809x)	0.0763 (46.7x)	9.21 (1.00x)	1.29 (7.16x)	0.331 (27.8x)	9.59 (0.960x)	0.214 (43.1x)
C2	9.85 (1.00x)	1.59 (6.19x)	0.681 (14.5x)	1.12 (8.81x)	0.163 (60.6x)	24.0 (1.00x)	3.28 (7.32x)	1.21 (19.8x)	0.514 (46.7x)	0.366 (65.6x)
C3	5.77 (1.00x)	0.947 (6.09x)	0.439 (13.1x)	6.80 (0.849x)	0.127 (45.4x)	16.3 (1.00x)	1.95 (8.35x)	0.610 (26.8x)	17.3 (0.944x)	0.320 (50.9x)
C4	28.3 (1.00x)	4.70 (6.03x)	1.99 (14.3x)	2.16 (13.1x)	0.449 (63.2x)	76.9 (1.00x)	9.59 (8.03x)	9.40 (8.18x)	1.56 (49.3x)	1.02 (75.8x)
C5	19.8 (1.00x)	3.10 (6.39x)	1.36 (14.6x)	14.7 (1.35x)	0.344 (57.6x)	67.3 (1.00x)	7.08 (9.51x)	6.40 (10.5x)	64.6 (1.04x)	1.05 (63.8x)
C6	39.5 (1.00x)	6.13 (6.45x)	2.46 (16.1x)	2.93 (13.5x)	0.586 (67.4x)	130 (1.00x)	13.4 (9.74x)	13.0 (10.0x)	2.02 (64.5x)	1.53 (85.0x)
C7	26.0 (1.00x)	4.33 (6.00x)	1.54 (16.9x)	17.0 (1.53x)	0.435 (59.7x)	101 (1.00x)	9.76 (10.4x)	9.53 (10.6x)	92.2 (1.10x)	1.38 (73.3x)
C8	34.7 (1.00x)	5.90 (5.88x)	2.23 (15.5x)	3.84 (9.05x)	0.544 (63.8x)	133 (1.00x)	13.1 (10.2x)	11.8 (11.3x)	2.37 (56.1x)	1.54 (86.2x)
C9	46.8 (1.00x)	8.67 (5.40x)	2.85 (16.4x)	27.3 (1.71x)	0.755 (62.0x)	212 (1.00x)	21.3 (9.99x)	18.6 (11.4x)	181 (1.17x)	2.79 (76.3x)
C10	44.5 (1.00x)	8.42 (5.29x)	3.05 (14.6x)	6.97 (6.39x)	0.713 (62.4x)	199 (1.00x)	20.2 (9.87x)	17.0 (11.7x)	5.14 (38.8x)	2.28 (87.4x)
					Peak memory (GB)					
C1	1.50 (1.00x)	0.157 (9.60x)	0.156 (9.62x)	1.51 (0.997x)	0.157 (9.59x)	2.41 (1.00x)	0.331 (7.27x)	0.312 (7.72x)	1.52 (1.59x)	0.313 (7.70x)
C2	4.19 (1.00x)	0.359 (11.7x)	0.359 (11.7x)	0.391 (10.7x)	0.360 (11.7x)	7.38 (1.00x)	0.775 (9.53x)	0.718 (10.3x)	0.749 (9.85x)	0.718 (10.3x)
C3	2.50 (1.00x)	0.218 (11.5x)	0.218 (11.5x)	2.47 (1.01x)	0.218 (11.5x)	3.76 (1.00x)	0.471 (7.99x)	0.435 (8.65x)	2.31 (1.63x)	0.436 (8.64x)
C4	12.0 (1.00x)	0.972 (12.3x)	0.972 (12.3x)	1.06 (11.3x)	0.972 (12.3x)	20.2 (1.00x)	2.10 (9.63x)	1.94 (10.4x)	2.03 (9.93x)	1.94 (10.4x)
C5	8.15 (1.00x)	0.607 (13.4x)	0.606 (13.5x)	8.00 (1.02x)	0.611 (13.4x)	13.2 (1.00x)	1.33 (9.87x)	1.21 (10.9x)	7.00 (1.88x)	1.22 (10.8x)
C6	14.3 (1.00x)	0.907 (15.8x)	0.907 (15.8x)	1.01 (14.2x)	0.907 (15.8x)	28.0 (1.00x)	1.98 (14.2x)	1.81 (15.5x)	1.91 (14.6x)	1.81 (15.4x)
C7	9.09 (1.00x)	0.535 (17.0x)	0.535 (17.0x)	8.86 (1.03x)	0.539 (16.9x)	16.4 (1.00x)	1.18 (13.8x)	1.07 (15.3x)	7.18 (2.28x)	1.07 (15.3x)
C8	12.8 (1.00x)	0.628 (20.4x)	0.628 (20.5x)	0.713 (18.0x)	0.629 (20.4x)	22.5 (1.00x)	1.39 (16.2x)	1.26 (17.9x)	1.34 (16.8x)	1.26 (17.9x)
C9	16.7 (1.00x)	0.781 (21.4x)	0.781 (21.4x)	15.1 (1.11x)	0.787 (21.3x)	28.5 (1.00x)	1.64 (17.4x)	1.56 (18.3x)	11.9 (2.40x)	1.57 (18.2x)
C10	16.4 (1.00x)	0.658 (25.0x)	0.658 (25.0x)	0.758 (21.7x)	0.659 (25.0x)	27.4 (1.00x)	1.37 (20.0x)	1.32 (20.8x)	1.42 (19.4x)	1.32 (20.8x)

Case ID	Double-bwd					Triple-bwd				
	e3nn	FlashTP	OpenEq	cuEq	LunaTP (ours)	e3nn	FlashTP	OpenEq	cuEq	LunaTP (ours)
					Runtime (ms)					
C1	25.7 (1.00x)	1.95 (13.2x)	1.12 (23.0x)	25.3 (1.02x)	0.403 (63.9x)	47.2 (1.00x)	–	6.38 (7.41x)	45.5 (1.04x)	0.683 (69.2x)
C2	74.4 (1.00x)	4.71 (15.8x)	3.61 (20.6x)	1.43 (52.1x)	0.802 (92.8x)	132 (1.00x)	–	19.9 (6.64x)	4.77 (27.7x)	1.74 (75.9x)
C3	52.8 (1.00x)	2.78 (19.0x)	2.23 (23.7x)	52.0 (1.01x)	0.651 (81.2x)	89.5 (1.00x)	–	12.0 (7.48x)	86.7 (1.03x)	1.17 (76.2x)
C4	251 (1.00x)	14.1 (17.8x)	12.1 (20.8x)	5.11 (49.2x)	2.36 (106x)	429 (1.00x)	–	82.1 (5.22x)	23.7 (18.1x)	5.14 (83.3x)
C5	262 (1.00x)	10.9 (24.1x)	7.77 (33.7x)	245 (1.07x)	1.96 (134x)	401 (1.00x)	–	53.5 (7.49x)	362 (1.11x)	3.98 (101x)
C6	467 (1.00x)	21.0 (22.2x)	16.5 (28.4x)	12.9 (36.3x)	2.83 (165x)	758 (1.00x)	–	109 (6.93x)	112 (6.77x)	6.39 (119x)
C7	427 (1.00x)	15.9 (26.8x)	10.6 (40.3x)	392 (1.09x)	2.20 (194x)	629 (1.00x)	–	73.8 (8.53x)	555 (1.13x)	5.13 (123x)
C8	537 (1.00x)	21.5 (25.0x)	16.3 (33.0x)	22.9 (23.5x)	2.84 (189x)	813 (1.00x)	–	104 (7.85x)	243 (3.35x)	7.10 (114x)
C9	1010 (1.00x)	39.8 (25.4x)	22.2 (45.5x)	926 (1.09x)	4.04 (250x)	1630 (1.00x)	–	148 (11.0x)	1230 (1.33x)	11.4 (143x)
C10	895 (1.00x)	38.4 (23.3x)	22.6 (39.5x)	57.5 (15.6x)	3.70 (242x)	1420 (1.00x)	–	145 (9.80x)	OOM	10.7 (132x)
					Peak memory (GB)					
C1	3.80 (1.00x)	0.614 (6.19x)	0.604 (6.29x)	3.92 (0.969x)	0.614 (6.18x)	6.33 (1.00x)	–	1.98 (3.20x)	5.27 (1.20x)	0.947 (6.69x)
C2	10.2 (1.00x)	1.41 (7.25x)	1.38 (7.41x)	1.44 (7.10x)	1.39 (7.35x)	19.0 (1.00x)	–	4.52 (4.20x)	2.16 (8.79x)	2.18 (8.71x)
C3	5.71 (1.00x)	0.855 (6.68x)	0.836 (6.84x)	6.35 (0.900x)	0.855 (6.68x)	9.94 (1.00x)	–	2.74 (3.63x)	8.24 (1.21x)	1.33 (7.49x)
C4	30.7 (1.00x)	3.81 (8.07x)	3.73 (8.24x)	3.90 (7.88x)	3.75 (8.20x)	54.8 (1.00x)	–	12.2 (4.48x)	5.87 (9.34x)	5.93 (9.24x)
C5	20.7 (1.00x)	2.37 (8.75x)	2.31 (8.99x)	20.2 (1.03x)	2.38 (8.72x)	34.9 (1.00x)	–	7.58 (4.60x)	25.0 (1.39x)	3.72 (9.38x)
C6	47.1 (1.00x)	3.52 (13.4x)	3.44 (13.7x)	3.64 (13.0x)	3.52 (13.4x)	75.6 (1.00x)	–	11.3 (6.68x)	5.47 (13.8x)	5.56 (13.6x)
C7	28.9 (1.00x)	2.07 (14.0x)	2.01 (14.3x)	22.8 (1.27x)	2.08 (13.9x)	45.6 (1.00x)	–	6.64 (6.87x)	29.0 (1.57x)	3.28 (13.9x)
C8	41.1 (1.00x)	2.41 (17.1x)	2.35 (17.5x)	2.52 (16.3x)	2.42 (17.0x)	63.9 (1.00x)	–	7.77 (8.23x)	3.79 (16.9x)	3.84 (16.7x)
C9	52.5 (1.00x)	2.99 (17.5x)	2.91 (18.0x)	38.9 (1.35x)	3.02 (17.4x)	79.8 (1.00x)	–	9.39 (8.50x)	48.3 (1.65x)	4.67 (17.1x)
C10	52.3 (1.00x)	2.50 (20.9x)	2.44 (21.4x)	2.64 (19.8x)	2.51 (20.8x)	79.2 (1.00x)	–	7.97 (9.93x)	OOM	3.97 (20.0x)

"–" denotes an unsupported setting.

B.2 Raw Data for Training Benchmarks

Tables 5 and 6 report throughput and the CGTP convolution time fraction for all eight models under EFS and EFSH training.

Table 5: EFS training performance. Each entry reports throughput in "atoms/s", with the CGTP convolution time fraction in parentheses.

Setting	Model	Throughput (CGTP time fraction)				
		e3nn	FlashTP	OpenEq	cuEq	LunaTP (ours)
Matched	SevenNet-Omni	936.6 (95.9%)	8470.3 (51.2%)	9160.7 (47.5%)	358.7 (76.5%)	**18024.6** (21.3%)
	MACE-L	4023.5 (53.5%)	7084.6 (10.7%)	7435.6 (7.9%)	774.6 (59.7%)	**8455.9** (1.4%)
	NequIP-XL	625.8 (93.8%)	4696.5 (57.5%)	5225.1 (48.9%)	370.7 (70.3%)	**7703.4** (20.2%)
	NequIX	4929.0 (92.8%)	26052.0 (46.1%)	40017.6 (41.0%)	4989.0 (88.9%)	**52397.8** (20.9%)
	EquFlashV2	144.0 (78.4%)	547.1 (26.1%)	538.0 (19.8%)	504.6 (9.5%)	**659.7** (6.0%)
	TACE-L	477.8 (79.0%)	1625.5 (28.6%)	1748.3 (23.3%)	245.5 (69.5%)	**2099.9** (4.6%)
	Prophet	160.2 (84.3%)	724.5 (31.5%)	782.7 (24.8%)	149.2 (50.3%)	**824.7** (11.0%)
	BAM	531.4 (82.3%)	2654.6 (16.3%)	2538.7 (15.7%)	601.7 (80.0%)	**2824.5** (3.1%)
MaxBS	SevenNet-Omni	936.6 (95.1%)	11136.3 (71.6%)	12469.2 (61.6%)	361.1 (82.8%)	**23933.1** (26.1%)
	MACE-L	4023.5 (53.5%)	7521.5 (11.3%)	7759.8 (8.6%)	1019.9 (78.0%)	**9289.7** (1.5%)
	NequIP-XL	625.8 (93.8%)	5648.9 (69.1%)	6961.3 (64.0%)	368.8 (71.1%)	**12844.3** (26.2%)
	NequIX	4929.0 (92.8%)	24715.1 (43.4%)	42825.2 (44.1%)	5046.9 (66.8%)	**52424.9** (20.9%)
	EquFlashV2	144.0 (78.4%)	1390.4 (65.6%)	1577.2 (60.5%)	2147.8 (38.1%)	**3311.9** (19.5%)
	TACE-L	477.8 (79.0%)	2614.6 (46.6%)	2858.3 (40.5%)	242.6 (69.5%)	**4563.7** (9.1%)
	Prophet	160.2 (84.3%)	1076.0 (43.9%)	1174.9 (38.6%)	169.0 (53.0%)	**1583.3** (14.2%)
	BAM	531.4 (82.3%)	2659.0 (15.7%)	2640.0 (15.9%)	618.5 (80.0%)	**2998.9** (3.2%)

Table 6: EFSH training performance. Each entry reports throughput in "atoms/s", with the tensor product convolution time fraction in parentheses.

Setting	Model	Throughput (CGTP time fraction)				
		e3nn	FlashTP	OpenEq	cuEq	LunaTP (ours)
Matched	SevenNet-Omni	277.5 (93.4%)	–	2036.4 (52.4%)	156.1 (71.7%)	**3232.3** (18.0%)
	MACE-L	1204.6 (56.1%)	–	2371.2 (13.5%)	486.3 (48.7%)	**2775.5** (2.3%)
	NequIP-XL	182.5 (92.0%)	–	1100.4 (52.9%)	–	**1480.7** (22.6%)
	NequIX	1465.7 (89.9%)	–	8352.2 (56.3%)	1506.9 (91.8%)	**14369.5** (25.0%)
	EquFlashV2	47.0 (55.4%)	–	85.6 (11.2%)	72.5 (11.6%)	**91.4** (9.2%)
	TACE-L	122.6 (47.0%)	–	202.5 (12.5%)	116.3 (52.2%)	**221.5** (3.5%)
	Prophet	34.0 (53.9%)	–	76.0 (6.3%)	29.8 (64.7%)	**80.7** (4.2%)
	BAM	182.9 (83.7%)	–	949.9 (26.7%)	179.7 (86.1%)	**1138.2** (6.2%)
MaxBS	SevenNet-Omni	277.5 (93.4%)	–	2562.0 (72.2%)	166.9 (77.8%)	**7956.0** (30.2%)
	MACE-L	1204.6 (56.1%)	–	2979.7 (9.5%)	755.92 (87.7%)	**4248.1** (2.7%)
	NequIP-XL	182.5 (92.0%)	–	1490.5 (78.0%)	–	**4627.4** (33.4%)
	NequIX	1465.7 (89.9%)	–	9082.6 (60.0%)	1401.0 (92.9%)	**16450.9** (27.0%)
	EquFlashV2	42.7 (55.4%)	–	307.1 (64.3%)	247.5 (64.1%)	**704.6** (17.6%)
	TACE-L	122.6 (46.8%)	–	635.7 (50.6%)	150.3 (69.3%)	**1233.1** (9.8%)
	Prophet	39.1 (89.3%)	–	256.0 (43.3%)	36.0 (64.4%)	**408.5** (15.8%)
	BAM	182.9 (83.7%)	–	1029.6 (30.1%)	179.7 (86.1%)	**1386.7** (5.4%)

"–" denotes an unsupported setting.

C Ablation Studies

We evaluate the effects of path fusion scope, cross-path input reuse, task ordering, and compile-time reuse on LunaTP's forward performance. All experiments use the C5 configuration in Table 1 with FP32 precision. This configuration includes channel counts of 320, 96, 64, and 32.

C.1 Path Grouping

Path grouping needs to balance input reuse and accumulator storage. We first vary the per-channel output component budget β to examine the effect of fusion scope (Table 7). Among the tested settings, the default budget $\beta = 16$ gives the lowest latency. Fusing all compatible paths increases latency by 56.3%, while reducing the budget to 8 and 4 increases latency by 11.6% and 20.8%, respectively. Smaller groups require more repeated edge traversals, while full fusion keeps more accumulators live at once and is the only tested setting with reported register spilling.

Since the default policy automatically adjusts the number of warps as the grouping changes, we also compare the fusion scopes with fixed warp counts. With 8 and 16 warps, full fusion is still 44.3% (0.5001 vs. 0.3467 ms) and 53.0% (0.5423 vs. 0.3545 ms) slower, respectively. In both cases, the reported spill stores are 2976 B for full fusion and zero for the default budget.

Table 7: Output component budget with automatic warp selection (C5, FP32, forward). Latency is the median of two independent measurements. Spill stores are kernel-reported statistics, not total runtime traffic.

Configuration	Path groups	Execution units	Registers	Spill stores (B)	Latency (ms)
All compatible paths	5	17	64	2976	0.5393
Default ($\beta = 16$)	17	45	64	0	0.3449
$\beta = 8$	29	73	56	0	0.3851
$\beta = 4$	37	96	56	0	0.4167

Performance also depends on which paths can share inputs within a group. Compared with the baseline that groups paths by edge irrep, broader cross-path input reuse reduces the number of path groups from 25 to 17 and lowers latency by about 6.6% (Table 8). This change both expands input reuse and reduces the number of tasks, so the measured gain reflects the overall effect of the grouping change. With this grouping, enabling the work partitioning option shown in the table leaves latency nearly unchanged, with no additional benefit observed on C5.

Table 8: Cross-path input reuse and work partitioning (C5, FP32, forward). All configurations use node-centric execution.

Configuration	Path groups	Execution units	Latency (ms)
Baseline (grouped by edge irrep)	25	85	0.3688
+ Cross-path input reuse	17	45	0.3446
+ Work partitioning (full)	17	45	0.3447

C.2 Task Ordering

We next vary the task order while keeping the fusion scope and number of warps fixed. We independently reverse the ordering by nonzero CG-term count and by node degree, while keeping the task set unchanged (Table 9). Compared with descending order on both axes, changing only the node order to ascending increases latency by 13.0%. Changing only the CG-term count order to ascending increases latency by 5.8%, while changing both increases it by 17.1%. Sorting both in descending order, as used in the main text, gives the lowest latency among these settings.

Table 9: Task ordering with a fixed fusion scope and warp count (C5, FP32, forward). Each ordering axis is reversed independently; the task set, instruction set, and output are unchanged.

CG-term count order	Node degree order	Latency (ms)	vs. both descending
Descending	Descending	0.3446	—
Descending	Ascending	0.3895	+13.0%
Ascending	Descending	0.3645	+5.8%
Ascending	Ascending	0.4036	+17.1%

C.3 Compile-Time Reuse

Finally, we compare all four combinations of product reuse and coefficient reuse with a fixed fusion scope and number of warps (Table 10). Enabling both gives the lowest latency of 0.3449 ms. Disabling coefficient reuse increases latency by 3.7%, while disabling both increases it by 3.4%.

These two switches are not fully independent. The code generation path for coefficient reuse retains shared products, so disabling the product reuse switch does not necessarily remove product sharing from these computations. When coefficient reuse is disabled, disabling product reuse also does not increase latency.

Table 10: Compile-time reuse switches with fixed fusion scope and warp count (C5, FP32, forward). Coefficient reuse retains shared products even when the product-reuse switch is disabled.

Product reuse	Coefficient reuse	Latency (ms)	vs. both enabled
Enabled	Enabled	0.3449	—
Enabled	Disabled	0.3578	+3.7%
Disabled	Enabled	0.3526	+2.2%
Disabled	Disabled	0.3565	+3.4%

D Additional Kernel Microbenchmark Results

D.1 Kernel microbenchmark (FP64)

We evaluate FP64 performance using the same irrep configurations as in the FP32 benchmarks. Since FP64 requires more computation and memory, we reduce the atom count by half to 384 for most cases and further reduce it to 192 for C8 and C10. All runtime speedups and peak-memory reduction factors are relative to FP64 e3nn on the same hardware. Figure 8 summarizes the results.

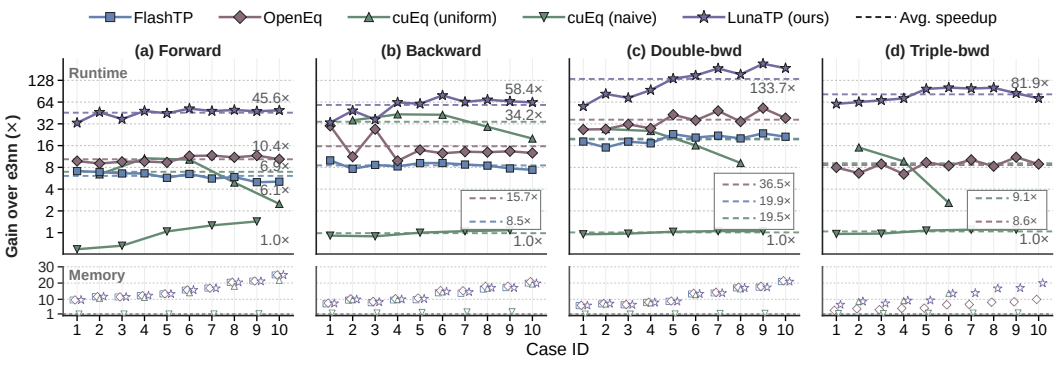


Figure 8: LunaTP performance in kernel microbenchmarks (FP64, NVIDIA H100-80GB).

LunaTP achieves arithmetic-mean speedups of 45.6×, 58.4×, 133.7×, and 81.9× over FP64 e3nn

for forward, backward, double backward, and triple backward, respectively. The corresponding speedups over OpenEq are 4.42×, 4.40×, 3.56×, and 9.61×.

D.2 KERNEL MICROBENCHMARK (A100 & RTX 3090)

We evaluate LunaTP on NVIDIA A100-80GB and RTX 3090-24GB GPUs using FP32 microbenchmarks. On A100, we use the same workloads as in the main evaluation. To fit within the RTX 3090's memory capacity, we use 192 atoms by default and reduce C8 and C10 to 122 and 97 atoms, respectively. For each case, we keep the workload fixed across all backends and all four stages.

On A100, LunaTP achieves average speedups of 49.7×, 62.4×, 122.7×, and 90.5× over e3nn for forward, backward, double backward, and triple backward, respectively. On RTX 3090, the corresponding speedups are 75.3×, 83.5×, 203.5×, and 151.2×. Figures 9 and 10 show the full results.

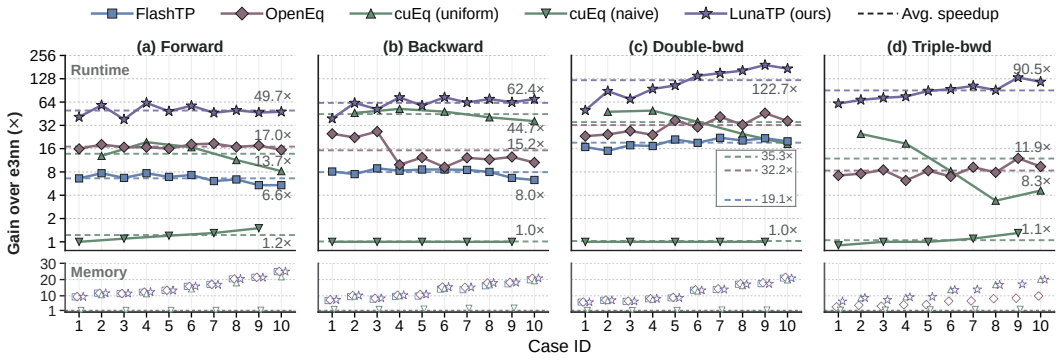


Figure 9: LunaTP performance in kernel microbenchmarks (NVIDIA A100-80GB PCIe).

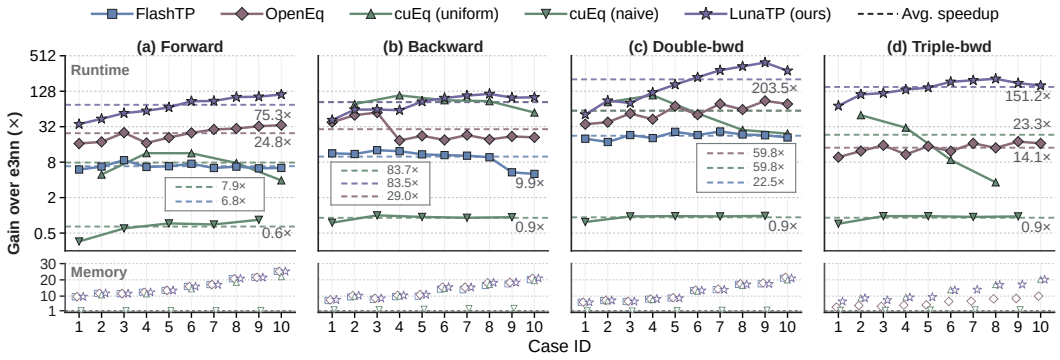


Figure 10: LunaTP performance in kernel microbenchmarks (RTX 3090-24GB).

LunaTP delivers performance gains on both data-center and consumer GPUs, although the speedups vary across hardware. Across all four stages, its average speedups over e3nn are highest on RTX 3090, followed by H100 and A100. This ordering matches that of the GPUs' peak FP32 compute-to-memory-bandwidth ratios. Since e3nn's tensor-product computations are primarily limited by memory bandwidth (Lee et al., 2025), LunaTP's reductions in intermediate memory traffic through kernel fusion and data reuse yield larger gains on GPUs with greater compute capacity relative to memory bandwidth.

E Details on Numerical Stability and Training Convergence

E.1 Operator-Level Numerical Stability

Following the numerical validation protocol of Shah et al. (2024), we use the e3nn FP64 as the reference. We run e3nn in FP32 and LunaTP in both FP32 and FP64 on the same inputs, and report RMSE for the forward and each derivative stage. The inputs are constructed as described in Appendix A. Table 11 reports the full results, averaged across all cases in Table 1.

Table 11: Numerical error comparison between e3nn and LunaTP.

Stage	Output	e3nn (FP32)	LunaTP (FP32)	LunaTP (FP64)
Forward	$\mathbf{x}'$	9.910429e-07	9.978108e-07	9.517594e-16
Backward	$\bar{\mathbf{x}}$	3.582043e-06	6.545904e-06	6.437515e-15
	$\bar{\mathbf{y}}$	3.516555e-06	2.934879e-06	8.966707e-15
	$\bar{\mathbf{w}}$	1.696597e-07	1.712254e-07	4.281635e-16
Double bwd	$\mathbf{h}_x$	5.083432e-06	7.955895e-06	1.274438e-14
	$\mathbf{h}_y$	5.002968e-06	4.300956e-06	5.290353e-15
	$\mathbf{h}_w$	2.546925e-07	2.582805e-07	6.404447e-16
	$\mathbf{h}_g$	1.769849e-06	2.792170e-06	7.214677e-15
Triple bwd	$\mathbf{t}_x$	5.818900e-06	6.511971e-06	9.811956e-15
	$\mathbf{t}_y$	6.808703e-06	6.200321e-06	9.310777e-15
	$\mathbf{t}_w$	3.152390e-07	2.198522e-07	6.591994e-16
	$\mathbf{t}_g$	2.530354e-06	3.285050e-06	5.920629e-15
	$\mathbf{t}_{v_x}$	5.368344e-06	5.646177e-06	8.875174e-15
	$\mathbf{t}_{v_y}$	5.896732e-06	5.317266e-06	9.918370e-15
	$\mathbf{t}_{v_w}$	2.677879e-07	2.514537e-07	6.436796e-16

E.2 End-to-End Training Convergence

We evaluate end-to-end training convergence by training NequIP-XL on MPTrj-PFT with either e3nn or LunaTP as the CGTP backend. Both runs use FP32 precision with the same model initialization, data, and training settings. The loss weights for energy, forces, stress, and Hessian are set to 20, 20, 100, and 100, respectively. We train for 50 epochs with a cosine learning-rate schedule, starting at 5×10^{-4} and ending at 1×10^{-6}. Figure 11 shows the training and validation MAE curves for both backends.

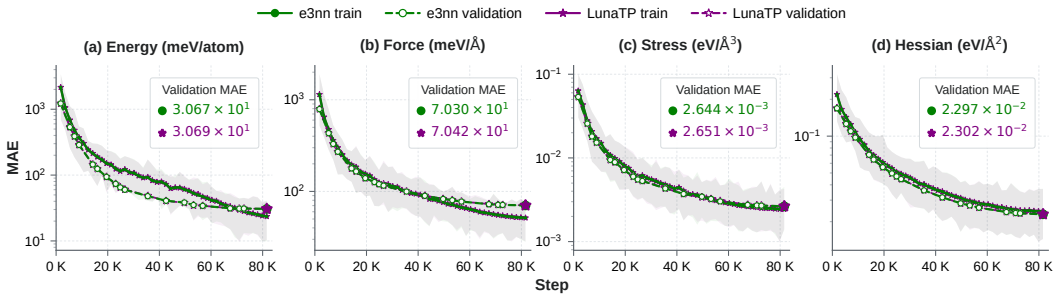


Figure 11: Loss convergence on MPTrj-PFT dataset.

F Differentiation of CGTP convolution

In this section, we derive the first-, second-, and third-order derivatives of tensor product convolution. The resulting formulas evaluate the gradients needed at each order without forming full

Jacobian, Hessian, or third-order derivative tensors, and provide the basis for our derivative kernels. Throughout, we present the derivatives componentwise, following Eq. 1. For each channel–path combination, $\mathbf{x}_j^{(l_1,c)} \in \mathbb{R}^{2l_1+1}$, $\mathbf{y}_{ji}^{(l_2)} \in \mathbb{R}^{2l_2+1}$, and $w_{ji}^{(l_1,l_2,l,c)} \in \mathbb{R}$.

The first-order derivative. For the convolution output $\mathbf{x'}_i^{(l,c)}$, let $\mathbf{g}_i^{(l,c)} = \partial\mathcal{L}/\partial\mathbf{x'}_i^{(l,c)}$ denote its gradient. Applying the chain rule to the weighted CGTP gives the first-order derivatives in Algorithm 2.

Algorithm 2 First-Order Derivative of Clebsch–Gordan Tensor Product Convolution

Input: $\mathbf{x}_j^{(l_1,c)}$: node irreps
Input: $\mathbf{y}_{ji}^{(l_2)}$: angular edge irreps
Input: $w_{ji}^{(l_1,l_2,l,c)}$: path weights
Input: $\mathbf{g}_i^{(l,c)}$: gradients with respect to the convolution output
Input: edges ji, from source node j to destination node i
Output: $\bar{\mathbf{x}}$, $\bar{\mathbf{y}}$, and $\bar{\mathbf{w}}$

Initialize all three outputs to zero
for ji in the E edges **do**
  for c in the C channels **do**
    for $(l_1, l_2) \to l$ in the P paths **do**

$$\bar{x}_{j,m_1}^{(l_1,c)} \mathrel{+}= w_{ji}^{(l_1,l_2,l,c)} \sum_{m=-l}^{l} \sum_{m_2=-l_2}^{l_2} C_{(l_1,m_1)(l_2,m_2)}^{(l,m)} y_{ji,m_2}^{(l_2)} g_{i,m}^{(l,c)}$$

$$\bar{y}_{ji,m_2}^{(l_2)} \mathrel{+}= w_{ji}^{(l_1,l_2,l,c)} \sum_{m=-l}^{l} \sum_{m_1=-l_1}^{l_1} C_{(l_1,m_1)(l_2,m_2)}^{(l,m)} x_{j,m_1}^{(l_1,c)} g_{i,m}^{(l,c)}$$

$$\bar{w}_{ji}^{(l_1,l_2,l,c)} \mathrel{+}= \sum_{m=-l}^{l} g_{i,m}^{(l,c)} \left[\mathbf{x}_j^{(l_1,c)} \otimes \mathbf{y}_{ji}^{(l_2)}\right]_m^{(l)}$$

    end for
  end for
end for

The second-order derivative. Let $\mathbf{v}_x$, $\mathbf{v}_y$, and v_w denote the gradients with respect to $\bar{\mathbf{x}}$, $\bar{\mathbf{y}}$, and $\bar{w}$, respectively. Applying the product rule to the first-order formulas gives the second-order derivatives in Algorithm 3.

Algorithm 3 Second-Order Derivative of Clebsch–Gordan Tensor Product Convolution

Input: $\mathbf{x}_j^{(l_1,c)}$: node irreps
Input: $\mathbf{y}_{ji}^{(l_2)}$: angular edge irreps
Input: $w_{ji}^{(l_1,l_2,l,c)}$: path weights
Input: $\mathbf{g}_i^{(l,c)}$: gradients with respect to the convolution output
Input: $\mathbf{v}_{x,j}^{(l_1,c)}$, $\mathbf{v}_{y,ji}^{(l_2)}$, and $v_{w,ji}^{(l_1,l_2,l,c)}$: gradients with respect to the first-order outputs
Input: edges ji, from source node j to destination node i
Output: $\mathbf{h}_x$, $\mathbf{h}_y$, $\mathbf{h}_w$, and $\mathbf{h}_g$

Initialize all four outputs to zero
for ji in the E edges **do**
  for c in the C channels **do**
    for $(l_1, l_2) \to l$ in the P paths **do**

$$h_{x,j,m_1}^{(l_1,c)} \mathrel{+}= \sum_{m=-l}^{l} \sum_{m_2=-l_2}^{l_2} C_{(l_1,m_1)(l_2,m_2)}^{(l,m)} g_{i,m}^{(l,c)} \left(v_{y,ji,m_2}^{(l_2)} w_{ji}^{(l_1,l_2,l,c)} + y_{ji,m_2}^{(l_2)} v_{w,ji}^{(l_1,l_2,l,c)} \right)$$

$$h_{y,ji,m_2}^{(l_2)} \mathrel{+}= \sum_{m=-l}^{l} \sum_{m_1=-l_1}^{l_1} C_{(l_1,m_1)(l_2,m_2)}^{(l,m)} g_{i,m}^{(l,c)} \left(v_{x,j,m_1}^{(l_1,c)} w_{ji}^{(l_1,l_2,l,c)} + x_{j,m_1}^{(l_1,c)} v_{w,ji}^{(l_1,l_2,l,c)} \right)$$

$$h_{w,ji}^{(l_1,l_2,l,c)} \mathrel{+}= \sum_{m=-l}^{l} g_{i,m}^{(l,c)} \left(\left[\mathbf{v}_{x,j}^{(l_1,c)} \otimes \mathbf{y}_{ji}^{(l_2)} \right]_m^{(l)} + \left[\mathbf{x}_j^{(l_1,c)} \otimes \mathbf{v}_{y,ji}^{(l_2)} \right]_m^{(l)} \right)$$

$$\mathbf{h}_{g,i}^{(l,c)} \mathrel{+}= w_{ji}^{(l_1,l_2,l,c)} \left(\left[\mathbf{v}_{x,j}^{(l_1,c)} \otimes \mathbf{y}_{ji}^{(l_2)} \right]^{(l)} + \left[\mathbf{x}_j^{(l_1,c)} \otimes \mathbf{v}_{y,ji}^{(l_2)} \right]^{(l)} \right) + v_{w,ji}^{(l_1,l_2,l,c)} \left[\mathbf{x}_j^{(l_1,c)} \otimes \mathbf{y}_{ji}^{(l_2)} \right]^{(l)}$$

    end for
  end for
end for

The third-order derivative. Let $\mathbf{q}_x$, $\mathbf{q}_y$, q_w, and $\mathbf{q}_g$ denote the gradients with respect to $\mathbf{h}_x$, $\mathbf{h}_y$, $\mathbf{h}_w$, and $\mathbf{h}_g$, respectively. Differentiating the second-order formulas gives the third-order derivatives in Algorithm 4.

Algorithm 4 Third-Order Derivative of Clebsch–Gordan Tensor Product Convolution

Input: $\mathbf{x}_j^{(l_1,c)}$: node irreps
Input: $\mathbf{y}_{ji}^{(l_2)}$: angular edge irreps
Input: $w_{ji}^{(l_1,l_2,l,c)}$: path weights
Input: $\mathbf{g}_i^{(l,c)}$: gradients with respect to the convolution output
Input: $\mathbf{v}_{x,j}^{(l_1,c)}$, $\mathbf{v}_{y,ji}^{(l_2)}$, and $v_{w,ji}^{(l_1,l_2,l,c)}$: gradients with respect to the first-order outputs
Input: $\mathbf{q}_{x,j}^{(l_1,c)}$, $\mathbf{q}_{y,ji}^{(l_2)}$, $q_{w,ji}^{(l_1,l_2,l,c)}$, and $\mathbf{q}_{g,i}^{(l,c)}$: gradients with respect to the second-order outputs
Input: edges ji, from source node j to destination node i
Output: $\mathbf{t}_x$, $\mathbf{t}_y$, $\mathbf{t}_w$, $\mathbf{t}_g$, $\mathbf{t}_{v_x}$, $\mathbf{t}_{v_y}$, and $\mathbf{t}_{v_w}$

Initialize all seven outputs to zero
for ji in the E edges **do**
  for c in the C channels **do**
    for $(l_1, l_2) \to l$ in the P paths **do**

$$t_{x,j,m_1}^{(l_1,c)} \mathrel{+}= \sum_{m=-l}^{l} \sum_{m_2=-l_2}^{l_2} C_{(l_1,m_1)(l_2,m_2)}^{(l,m)} \Big[g_{i,m}^{(l,c)} \Big(q_{y,ji,m_2}^{(l_2)} v_{w,ji}^{(l_1,l_2,l,c)} + v_{y,ji,m_2}^{(l_2)} q_{w,ji}^{(l_1,l_2,l,c)} \Big) + q_{g,i,m}^{(l,c)} \Big(v_{y,ji,m_2}^{(l_2)} w_{ji}^{(l_1,l_2,l,c)} + y_{ji,m_2}^{(l_2)} v_{w,ji}^{(l_1,l_2,l,c)} \Big) \Big]$$

$$t_{y,ji,m_2}^{(l_2)} \mathrel{+}= \sum_{m=-l}^{l} \sum_{m_1=-l_1}^{l_1} C_{(l_1,m_1)(l_2,m_2)}^{(l,m)} \Big[g_{i,m}^{(l,c)} \Big(q_{x,j,m_1}^{(l_1,c)} v_{w,ji}^{(l_1,l_2,l,c)} + v_{x,j,m_1}^{(l_1,c)} q_{w,ji}^{(l_1,l_2,l,c)} \Big) + q_{g,i,m}^{(l,c)} \Big(v_{x,j,m_1}^{(l_1,c)} w_{ji}^{(l_1,l_2,l,c)} + x_{j,m_1}^{(l_1,c)} v_{w,ji}^{(l_1,l_2,l,c)} \Big) \Big]$$

$$t_{w,ji}^{(l_1,l_2,l,c)} \mathrel{+}= \sum_{m=-l}^{l} \left(g_{i,m}^{(l,c)} \left(\left[\mathbf{q}_{x,j}^{(l_1,c)} \otimes \mathbf{v}_{y,ji}^{(l_2)} \right]_m^{(l)} + \left[\mathbf{v}_{x,j}^{(l_1,c)} \otimes \mathbf{q}_{y,ji}^{(l_2)} \right]_m^{(l)} \right) + q_{g,i,m}^{(l,c)} \left(\left[\mathbf{v}_{x,j}^{(l_1,c)} \otimes \mathbf{y}_{ji}^{(l_2)} \right]_m^{(l)} + \left[\mathbf{x}_j^{(l_1,c)} \otimes \mathbf{v}_{y,ji}^{(l_2)} \right]_m^{(l)} \right) \right)$$

$$\mathbf{t}_{g,i}^{(l,c)} \mathrel{+}= w_{ji}^{(l_1,l_2,l,c)} \left(\left[\mathbf{q}_{x,j}^{(l_1,c)} \otimes \mathbf{v}_{y,ji}^{(l_2)} \right]^{(l)} + \left[\mathbf{v}_{x,j}^{(l_1,c)} \otimes \mathbf{q}_{y,ji}^{(l_2)} \right]^{(l)} \right) + v_{w,ji}^{(l_1,l_2,l,c)} \left(\left[\mathbf{q}_{x,j}^{(l_1,c)} \otimes \mathbf{y}_{ji}^{(l_2)} \right]^{(l)} + \left[\mathbf{x}_j^{(l_1,c)} \otimes \mathbf{q}_{y,ji}^{(l_2)} \right]^{(l)} \right) + q_{w,ji}^{(l_1,l_2,l,c)} \left(\left[\mathbf{v}_{x,j}^{(l_1,c)} \otimes \mathbf{y}_{ji}^{(l_2)} \right]^{(l)} + \left[\mathbf{x}_j^{(l_1,c)} \otimes \mathbf{v}_{y,ji}^{(l_2)} \right]^{(l)} \right)$$

$$t_{v_x,j,m_1}^{(l_1,c)} \mathrel{+}= \sum_{m=-l}^{l} \sum_{m_2=-l_2}^{l_2} C_{(l_1,m_1)(l_2,m_2)}^{(l,m)} \Big[g_{i,m}^{(l,c)} \Big(w_{ji}^{(l_1,l_2,l,c)} q_{y,ji,m_2}^{(l_2)} + y_{ji,m_2}^{(l_2)} q_{w,ji}^{(l_1,l_2,l,c)} \Big) + q_{g,i,m}^{(l,c)} w_{ji}^{(l_1,l_2,l,c)} y_{ji,m_2}^{(l_2)} \Big]$$

$$t_{v_y,ji,m_2}^{(l_2)} \mathrel{+}= \sum_{m=-l}^{l} \sum_{m_1=-l_1}^{l_1} C_{(l_1,m_1)(l_2,m_2)}^{(l,m)} \Big[g_{i,m}^{(l,c)} \Big(w_{ji}^{(l_1,l_2,l,c)} q_{x,j,m_1}^{(l_1,c)} + x_{j,m_1}^{(l_1,c)} q_{w,ji}^{(l_1,l_2,l,c)} \Big) + q_{g,i,m}^{(l,c)} w_{ji}^{(l_1,l_2,l,c)} x_{j,m_1}^{(l_1,c)} \Big]$$

$$t_{v_w,ji}^{(l_1,l_2,l,c)} \mathrel{+}= \sum_{m=-l}^{l} \left(g_{i,m}^{(l,c)} \left(\left[\mathbf{q}_{x,j}^{(l_1,c)} \otimes \mathbf{y}_{ji}^{(l_2)} \right]_m^{(l)} + \left[\mathbf{x}_j^{(l_1,c)} \otimes \mathbf{q}_{y,ji}^{(l_2)} \right]_m^{(l)} \right) + q_{g,i,m}^{(l,c)} \left[\mathbf{x}_j^{(l_1,c)} \otimes \mathbf{y}_{ji}^{(l_2)} \right]_m^{(l)} \right)$$

    end for
  end for
end for

G PROGRAMMING INTERFACE

LunaTP replaces the gather–TP–reduce sequence with a single convolution call, using the same e3nn irreps, instructions, and per-edge weights. It also supports precision selection and optional envelope smoothing of the messages (Gasteiger et al., 2022; Fu et al., 2025). The code below illustrates the change in usage.

e3nn

```
from e3nn import o3

# 1.Initialize
e3nn_tp = o3.TensorProduct(
    irreps_in, irreps_edge, irreps_out, instructions,
    internal_weights=False, shared_weights=False,
)

# 2.Forward
x_edge = node.index_select(0, src)              # Gather
messages = e3nn_tp(x_edge, edge_attr, weight)   # Tensor product
if use_envelope:
    messages = messages * envelope              # Apply envelope
out = node.new_zeros((node.shape[0], e3nn_tp.irreps_out.dim))
out.index_add_(0, dst, messages)                # Reduce
```



LunaTP

```
import LunaTP

# 1.Initialize from an e3nn TP or custom instructions.
lunatp_conv = LunaTP.initialize(e3nn_tp)

# 2.Forward
out = lunatp_conv(
    node,
    edge_attr,
    weight,
    env=envelope, # Default: 'none' (no envelope)
    dtype='fp32'
)
```