E3J: AN EFFICIENT AND OPEN-SOURCE BACKEND FOR EUCLIDEAN EQUIVARIANT OPERATIONS ON GPU AND TPU

Anonymous authors
Paper under double-blind review

ABSTRACT

We present `e3j`, a fast Euclid-equivariance backend for geometric deep learning applications with JAX bindings for GPU and TPU. Leveraging both optimized CUDA and Pallas kernels and algorithmic improvements, the library achieves state-of-the-art throughput and runtime on both forward and backward paths. On a machine learning interatomic potential (MLIP) use case, it outperforms established backends, measuring up to 34% speed-up over `cuEquivariance` on water box NPT simulation using MACE, while remaining fully open source. `e3j` achieves over 80 % efficiency over the H100 maximum memory bandwidth on tensor product operations, and in many cases more than doubles throughput of message passing convolutions forward compared to previously available backends. In addition, with the release of dedicated Pallas TPU kernel, `e3j` opens the possibility of large scale equivariant deep learning workloads on TPU architectures, which has so far been difficult to achieve. Our benchmarks show that `e3j` also achieves over 80% of a TPUv6e memory bandwidth, up to one order of magnitude more than `e3nn_jax`. The library is available on GitHub, PyPI and is released under an open source Apache 2.0 license.

1 INTRODUCTION

Euclid-equivariant operations are a core building block for many geometric deep learning applications. By providing exact equivariance guarantees, they have been shown to allow aggregation of high order geometric features in low data regime and without need for data augmentation (Thomas et al., 2018; Batzner et al., 2022). Their use has been explored across many fields that rely on geometric data, and in particular physical sciences, such as molecular dynamics (Batatia et al., 2023; Kovács et al., 2025; Batzner et al., 2022; Wood et al., 2026), computational fluid dynamics (CFD) (Kaszuba et al., 2025; Shankar et al., 2023), or electronic wavefunction and densities (Unke et al., 2021).

In this work, we present `e3j`, an open-source Euclid-equivariance backend providing all the core building blocks for E(3)-equivariant neural networks. The library is designed to facilitate constructions of these networks through a modular interface, but crucially is used as a means to integrate dedicated accelerated kernels into model architectures. To that end, the library includes kernels written both in Pallas and in CUDA, with the aim of deploying workloads across multiple types of architectures, including TPUs and GPUs. Namely, we present three sets of kernels backed by novel algorithms:

- **CUDA kernels**: Fully deterministic equivariant operations for GPU, focused on broad compatibility across JAX versions through ahead-of-time (AOT) compilation.
- **Pallas GPU kernels**: Focused on performance, setting state-of-the-art throughput at time of writing in most regimes tested through just-in-time (JIT) compilation by the Mosaic compiler. Sacrifices determinism on the backward pass, compatibility with some older JAX versions, and support for some operations (e.g. low number of channels).
- **Pallas TPU kernels**: Focused on delivering optimized throughput for TPUs.

We include and benchmark kernels specifically for both the equivariant tensor product, and for the full message passing convolution. In order to illustrate the capabilities of the library in realistic

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.

workloads, we focus on applications in the field of machine learning interatomic potentials (MLIPs) which are likely among the most relevant use case examples for the technology as (1) they require a complete absence of systematic bias from frame orientation to produce stable, long time molecular simulations, and (2) unlike other fields of application such as CFD, they rely on nearly perfectly equivariant training data.

The library achieves throughputs comparable to those of closed source alternatives, for example reaching 2.7 TB/s on H100 GPU (81% of memory bandwidth) on the forward tensor product operation, and 1.33 TB/s on TPUv6.2e (also 81% of memory bandwidth). In order to illustrate the practical benefits of `e3j`, we also benchmarked end-to-end integration within the MACE (Batatia et al., 2023; Kovács et al., 2025) and NequIP (Batzner et al., 2022) models, notably providing about 5x force inference speedup over `e3nn` on TPU. On GPU, `e3j` provides 2x speedup in simulations over the best open-source backend `OpenEquivariance`, and 34% speedup over the proprietary `cuEquivariance` backend, reaching up to 64 thousands atoms (with ~ 40 connectivity degree) on MACE before overflowing the H100 memory. Details of these results are presented in Section 3. For theoretical details about the operations included in the library, readers can refer to Appendix A.3. The details of our novel algorithmic and methodological developments are presented in Appendix E.

2 Related Work

Engineering work: There has been considerable effort in building parameterized equivariant transforms accessible to the machine learning (ML) and scientific computing ecosystem. Generally, this consists in extending standard ML frameworks such as JAX (Bradbury et al., 2018) and PyTorch (Paszke et al., 2017) to benefit from their automatic differentiation (AD) support, so that equivariant building blocks can be seamlessly integrated within larger workflows on accelerated hardware such as GPUs and TPUs. These extensions may be defined either within the AD framework itself (JAX / PyTorch), or via the lower-level definition of ad-hoc primitives in the CUDA language[1] (Corporation, 2025). The main examples of relevant libraries in this space are:

- `e3nn` (Geiger et al., 2022; Geiger and Smidt, 2022), one of the first and most feature-rich Euclid equivariance libraries, consisting of two sibling packages written in Torch and JAX, respectively. It has been used to construct widely used MLIP architectures such as MACE (Batatia et al., 2023; Kovács et al., 2025) and NequIP (Batzner et al., 2022), which can learn to predict energies and forces at quantum-levels of accuracy, thus disrupting the speed-accuracy compromise of MD simulations (Brunken et al., 2025).
- `e3x` (Unke and Maennel, 2024), an open source Euclid equivariance package written in JAX (Bradbury et al., 2018) which is slightly less flexible than `e3nn` due to its stricter data model, but offers efficient bilinear projections with cubic L scaling (Maennel et al., 2024).
- `OpenEquivariance` (Bharadwaj et al., 2025), an open source CUDA kernel generation package with Torch and JAX bindings. It delivers state-of-the-art performance on focused operations (tensor product, message-passing convolution) with dedicated double-backward kernels to optimize training and Hessian inference. It is meant as a drop-in replacement for specific `e3nn` operations on which it provides speedups of an order of magnitude.
- `cuEquivariance` (`cuEq`), a proprietary NVIDIA® package with a toolset already as complete as `e3nn`, providing efficient CUDA kernels for tensor products and polynomial evaluation along with JAX and Torch bindings. `cuEquivariance` is the most efficient backend, with an open-source API that however depends on closed-source kernels.

The above list includes the most feature-complete packages viable for the construction of larger equivariant neural networks such as MLIPs. Efficient, near-optimal (open-source or closed-source) solutions therefore exist to compute Clebsch-Gordan tensor products on NVIDIA GPUs. What best distinguishes `e3j` from recent work is that `e3j` provides a standalone and platform-agnostic API in JAX for GPU and TPU execution.

[1]While the CUDA language from NVIDIA® extends C++ to let users write general purpose programs on their GPUs, there is no low-level public kernel language for TPUs.

Theoretical work: Several alternative approaches to improve equivariant operations in geometric deep learning have come from the fundamental side. Naively a Clebsch–Gordan tensor product scales as $O(L^6)$, or as $O(L^5)$ if one exploits sparsity. Mathematical work has shown that cubic scaling with L can be obtained for equivariant tensor products, though it is worth noting that these always come with compromise in terms of applicability or expressivity (Xie et al., 2025).

Some methods demonstrate cubic scaling with L on arbitrary inputs. *Gaunt tensor product formulas* (Luo et al., 2024) are morally similar to performing element-wise products instead of discrete convolution via reciprocal Fourier transforms on the input and outputs. While the initial formulation of the Gaunt tensor product (GTP) however fails to fully reproduce the Clebsch-Gordan tensor product, as it does not incorporate anti-symmetric elements, limitations were remediated by a series of papers which formulate and incorporate an anti-symmetric counterpart called *Vector Signal Tensor Product* (VSTP) (Xie et al., 2025; 2026), and later (Heyraud et al., 2026; Bochkarev et al., 2026) providing a closed form formula for efficient complete simulation of Clebsch-Gordan tensor products. Let us also mention the matrix tensor product formulation of (Maennel et al., 2024) which also has $O(L^3)$ scaling and is implemented in `e3x`, providing order of magnitude speedups at $L = 10$ over the equivalent `e3nn` implementation. All of these methods however do result in automatic collapse of output multiplicity, a more efficient implementation which however does lead to a loss in expressivity (Xie et al., 2025).

Other methods specialize in delivering efficiency gains in the special case of a tensor product between an arbitrary feature vector of irreducible representations and equivariant features obtained by harmonic embeddings of an input vector. This particular case remains dominant in the literature, constituting one of the core building blocks of the original Tensor Field Network (Thomas et al., 2018), later used in NequIP and MACE. A first example is the $SO(2)$ convolution, presented in the equivariant Spherical Channel Network (eSCN) (Passaro and Zitnick, 2023) and notably used in eSEN (Fu et al., 2025) and UMA (Wood et al., 2026), which defines a frame a reference from each edge and rotates the corresponding tensor product operands using the Wigner-D matrices. This results in harmonic features collapsing to $m = 0$ across all L, removing the need to sum over m indices of the harmonic features when performing the tensor product. This, combined with the additional use of symmetries in the Clebsch-Gordan tensor product achieves a cubic scaling in L. A second example was presented in the E2former model (Li et al., 2025) (later combined with $SO(2)$ convolutions by the same team (Huang et al., 2026)) where the projected displacement vectors are replaced with the difference of projected input positions into harmonic features. Using a Binomial expansion, the authors show that one can construct a messaging passing block solely relying on node-wise tensor products. While message passing scaling remains unchanged, the number of tensor products (expected to be among the most costly operations) now scales with the number of nodes rather than the number of edges.

It is also worth noting that while theoretical work often focuses on tensor product scaling in L, most MLIP applications are in practice interested in the scaling in N but at fixed L (usually 2 or 3)[2]. Improving the scaling with L typically requires clever re-parameterizations of equivariant features, and incurs a practical overhead that may not prove beneficial over an efficient implementation of the full Clebsch-Gordan tensor product in the small L regime.

We have conducted benchmarks of our backends against methods based on Gaunt tensor products and its extensions, `e3x` methods and on $SO(2)$ convolution. These are presented in Sec. 3 and in more detail in Appendices B.1, B.2 and B.3 respectively.

3 RESULTS

The `e3j` package consists of a Python API targeting the JAX backend, alongside CUDA and Pallas kernel implementations for performance critical operations (tensor product, message-passing). The JAX framework (Bradbury et al., 2018) enables seamless integration within larger programs that can be *just-in-time* (JIT) compiled to XLA (for Accelerated Linear Algebra), as are typically all recent MLIP model implementations. In addition to atomic, module-wise benchmarks against reference E3 backends, we used the open-source `mlip` library (Brunken et al., 2025) as reference and starting point to estimate the end-to-end speedups `e3j` integration may provide in the MACE and NequIP

[2]Applications such as signal processing or meteorology are in contrast interested in high maximal degree L.

equivariant architectures (Batatia et al., 2023; Batzner et al., 2022). The details of the implementation method, algorithms, and differentiation rules can be found in Appendix E.

It is worth noting that our JAX primitives bound to custom kernels are all made infinitely differentiable via recursive AD rules, and compatible with other higher-order JAX transforms (`vmap`, `shard_map`, ...) to provide SPMD execution on GPU and TPU, an essential feature for MLIP training workflows on a large data scale. Architecture requirements for GPU and TPU however largely differ beyond that point, necessarily resulting in different algorithms. We have added a commentary on the GPU / TPU differences in appendix F.

In order to assess the performance of the library, we focus on two types of benchmarking (further discussion on the benchmark details can be found in Appendix D):

- **Module specific:** We benchmark the performance critical components of any Euclidean equivariant library, namely:
  - **Tensor product:** Bilinear coupling of latent equivariant features with static *Clebsch-Gordan coefficients*. In general these are performed per edge and per channel.
  - **Message passing:** Aggregates the edge-wise features, usually computed through combination of harmonics projection, tensor product and linear or scalar mixing. The message passing aggregation tends to be the operational bottleneck once efficient tensor product operations are implemented.
- **End-to-end:** To determine the overall relevance of the library in a complete workflow compared to alternatives, we also test two popular MLIP architectures, MACE (Batatia et al., 2023) and NequIP (Batzner et al., 2022). To maintain comparability, we connected the full benchmark in a fork of the open-source `mlip` library (Brunken et al., 2025), making the choice of backend the only variable differing in each run. Note that these benchmarks are for illustration only and that applicability of `e3j` is not restricted to these two architectures (nor to MLIPs in general), a faithful and exhaustive comparison across all possible models and fields of application is not in scope of this work, for obvious reasons.

Experiments were conducted running JAX (v0.11.1) on an NVIDIA (R) H100-HBM3 GPU with 3.35 TB/s HBM, and on a TPUv6e Trillium with 1.64 TB/s HBM. We benchmark seven different backends, five for GPU, and two for TPU. On GPU, we compare the following backends with our **CUDA** and **Pallas GPU** kernels:

- **cuEquivariance** (v0.11.1): Proprietary tensor product and message-passing convolution kernels of NVIDIA.
- **OpenEquivariance** (v0.7.0): We use JAX bindings to their equivalent tensor product and message-passing convolution kernels, which are JIT compiled from open-source CUDA kernels with NVRTC (Bharadwaj et al., 2025). Note that `OpenEquivariance` provides two distinct convolution binaries, a deterministic one for graphs with edges sorted by receiver node index, and a non-deterministic one.
- **e3nn-jax** (v0.21.1): The reference `e3nn-jax` library sets the performance threshold obtained by a pure JAX implementation, JIT compiled by XLA, but without any specific low level optimization. It should therefore only be considered as an illustrative reference. The `e3nn` label uses the default half-precision for `matmul`, while `e3nn_f32` enforces single-precision in unit benchmarks.

On TPU, we compare our **Pallas TPU** kernels with XLA compiled **e3nn-jax** equivalent implementations of the full tensor product and message-passing convolution.

Additional benchmark results can be found in appendix C, while comparisons with JAX implementations of cubically scaling algorithms (SO2, Gaunt/VSTP, `e3x`) are grouped in appendix B.

3.1 Unit benchmarks

This section provides efficiency comparisons of the currently implemented `e3j` kernels with available baselines in typical regimes. The tensor product operation is a full Clebsch-Gordan tensor product,

satisfying the so-called *universal property*[3] of tensor products. The message-passing convolution operation accumulates edge-wise tensor products on receiver nodes, and consists of a typical bottleneck in MLIP architectures (Liao et al., 2024; Passaro and Zitnick, 2023; Fu et al., 2025).

Runtimes were measured on XLA compiled, numerically equivalent implementations, averaging the fastest 20% of 100 runs, and disabling the Python garbage collector using the `timeit` module. The backward pass consists of the isolated, XLA compiled vector-jacobian product (VJP).

Our metric of interest is throughput, i.e. the total amount of input/output bytes processed in the operation per unit of time. The peak throughput can be directly compared with the GPU / TPU maximum bandwidth to provide a meaningful speed relative to the device, independent of I/O size. In the backward pass benchmarks, only input primals, output cotangents and input cotangents were considered in the VJP accounting, excluding any saved residuals.

3.1.1 Tensor Product

We developed dedicated tensor product kernels using CUDA, Pallas GPU and Pallas TPU, with results presented in Fig. 1. Our results show that the Clebsch-Gordan tensor product (CGTP) operation can reach up to 80% HBM in the forward and backward passes on both GPU and TPU. This means that CGTP in the small $L \leq 3$ regime is not a compute bottleneck in itself, while slightly larger degrees $L = 4, 5 \ldots$ remain amenable to further engineering optimizations that were not prioritized at this time. On GPU, our Pallas GPU kernel performs almost exactly on par with `cuEquivariance`, while our CUDA kernel performs similarly on the forward pass and slightly below Pallas GPU / `cuEquivariance` on the backward pass. All three outpace `OpenEquivariance`. On TPU our Pallas kernel achieves one order of magnitude higher throughput than the previously available `e3nn_jax`.

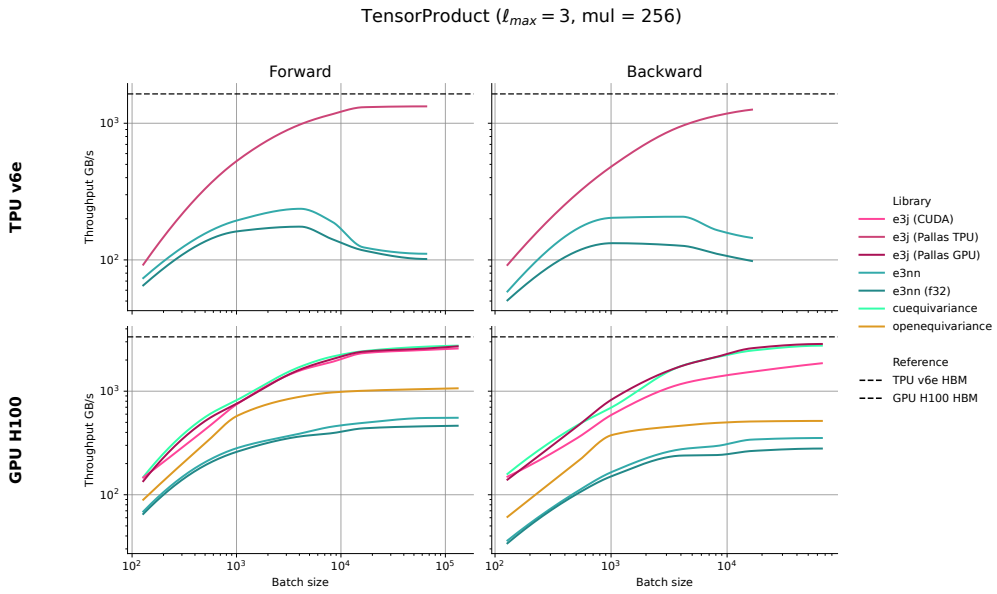

Figure 1: **Throughputs (GB/s) for the tensor product operation on TPU (top) and GPU (bottom)**. The `e3j` CUDA implementation is compared against `cuEq` (jax), and `e3nn_jax` (single precision and default half precision). The `e3j` Pallas implementation is compared against the only available baseline `e3nn_jax` for the universal ("full") Clebsch-Gordan tensor product. Results for $\ell_{max} = 2$ and $\ell_{max} = 3$ and multiplicities up to 256 channels, alongside the (a) theoretical maximum throughput of the NVIDIA H100 GPU ($\sim$ 3.35 TB/s) and (b) theoretical maximum throughput of a TPU v6e ($\sim$ 1.64 TB/s) on which the benchmarks were performed. When lines are not complete, it indicates that the library reached the memory bound.

[3]Any bilinear map $b : (X, Y) \to Z$ factors through the tensor product space $X \otimes Y$ as a linear map $\tilde{b} : X \otimes Y \to Z$.

Note that our tensor product primitive is exposed as a generic bilinear coupling of operands with an arbitrary sparse COO array of coefficients. As presented in appendix B.1, this allows us to further benchmark our kernels against Gaunt (Luo et al., 2024) and Vector Signal Tensor products (Xie et al., 2025; 2026; Heyraud et al., 2026; Bochkarev et al., 2026) on symmetric and skew-symmetric paths respectively. We find (see figure 5) that for $L \leq 3$, our kernels are 2 to 4 times faster than the Gaunt tensor product on symmetric paths, with crossing at $L = 6$ for the forward and $L = 5$ for the backward. For the skew-symmetric paths, our kernels outperform the VSTP by a factor of 4 to 6 for $L \leq 3$, and the crossing occurs one degree later due to the extra cost of the vector-valued operations in the VSTP. While we believe our implementations of GTP and VSTP to be efficient, dedicated kernel optimization of these operations could improve the relative results.

3.1.2 Message Passing Convolution

We present results for our three sets of Message Passing Convolution kernels: CUDA, Pallas GPU and Pallas TPU. On GPU, the Pallas GPU dominates the benchmarks, with over two times the throughput of `cuEquivariance` in the forward pass under most settings, and a slightly higher throughput in the backward pass. It is worth noting that our Pallas GPU kernel is not (yet) compatible with channel counts lower than 128. The CUDA kernels are broadly on par with `OpenEquivariance`, both having significantly lower throughput than Pallas GPU and `cuEquivariance`. These results are presented in Tab. 1. On TPU, our convolution kernels provide well over one order of magnitude speedups on the forward, and nearly one order of magnitude on the backward over `e3nn-jax`. An overview of the results is presented in Fig. 2.

Our convolution kernels further provide the ability to skip padding edges, which typically all point to the same padding node[4] in most JAX-based graph neural network frameworks (Godwin et al., 2020). This feature avoids the potentially significant overhead of aggregating messages on a padding node of unusually high valency, as illustrated by the simulation runtimes summarized in table 2. Implementation details are found in Appendix E.

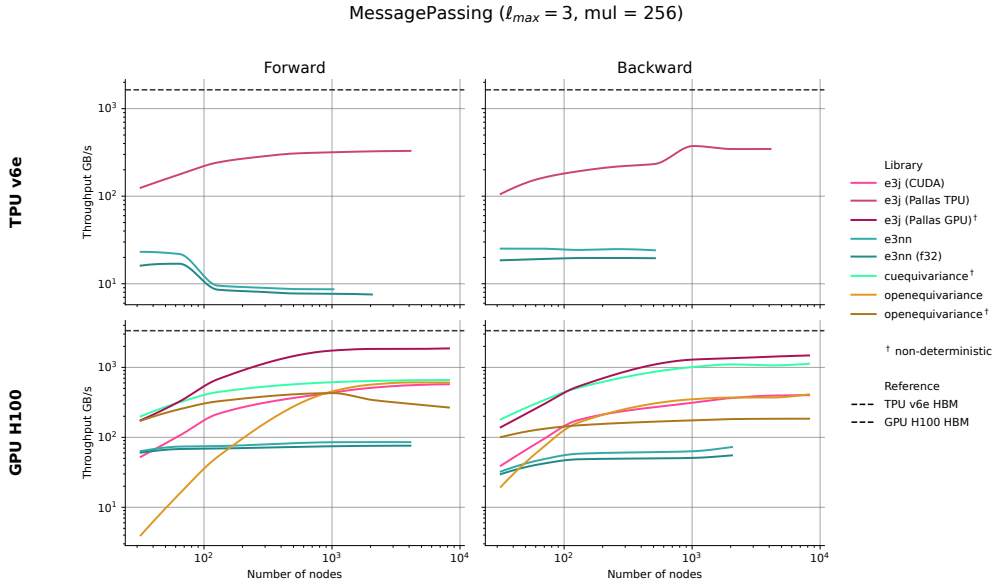

Figure 2: **Throughputs (GB/s) for the convolution operation on TPU (top) and GPU (bottom)**. Here we present results for $\ell_{max} = 3$ 256 channels, alongside the theoretical maximum throughput of a TPU v6e Trillium (1.64 TB/s) and the theoretical maximum throughput of the NVIDIA H100 HBM3 GPU ($\sim$ 3.35 TB/s) on which the benchmarks were performed. When lines are not complete, it indicates that the library reached the memory bound of the device.

[4]Given the graph topology is dynamic, while XLA compiled programs expect static shapes, simulations typically add a single padding node to the atom list while managing a fixed-size buffer of edges, only expanded when the real connecting edges overflow the neighbor list (a rare event triggering re-compilation).

Table 1: **Maximal convolution throughput (GB/s) by channel count on GPU.** End-to-end throughput is reported at the maximal power-of-two node count fitting on the NVIDIA H100 HBM3, with 45 average neighbors. This keeps the product of node count with channel count fixed to 2^{22} at $\ell_{max} = 2$, and to 2^{21} at $\ell_{max} = 3$ for fused kernels. The `e3nn` baseline is kept to illustrate the yield of a pure JAX implementation of the same operation, but message materialization overflows memory at 2^{18} and 2^{17} respectively. The Pallas GPU kernel does not support channel counts below 128 yet (minimal block size imposed by Pallas).

$\ell_{\max}$	Implementation	Det.	Channels 64	128	256	512	1024
			Forward pass				
2	`e3nn` (f32)†		89 ± 1	90 ± 1	90 ± 0	92 ± 0	90 ± 0
	`CuEquivariance`		779 ± 1	833 ±15	786 ± 4	807 ±11	755 ± 0
	`OpenEquivariance`	✓	**871** ± 3	1092 ± 8	760 ± 8	631 ± 5	652 ± 7
	`OpenEquivariance`		348 ± 1	343 ± 1	370 ± 1	354 ± 2	443 ± 1
	`e3j` (CUDA)	✓	625 ± 2	955 ± 4	913 ± 4	956 ± 2	880 ± 5
	`e3j` (Pallas GPU)	✓	–	**1621** ± 9	**1669** ±12	**1716** ±29	**1871** ±41
3	`e3nn` (f32)†		73 ± 0	72 ± 1	75 ± 0	75 ± 1	73 ± 1
	`CuEquivariance`		829 ±11	1072 ± 3	662 ± 1	632 ± 1	629 ± 0
	`OpenEquivariance`	✓	**856** ± 2	709 ± 1	612 ± 1	565 ± 2	347 ± 5
	`OpenEquivariance`		265 ± 1	268 ± 1	303 ± 1	323 ± 0	303 ± 4
	`e3j` (CUDA)	✓	389 ± 0	569 ± 2	573 ± 1	562 ± 2	538 ± 2
	`e3j` (Pallas GPU)	✓	–	**1818** ± 6	**1859** ±14	**2074** ±14	**2067** ±19
			Backward pass				
2	`e3nn` (f32)†		63 ± 1	65 ± 1	64 ± 0	62 ± 2	60 ± 1
	`CuEquivariance`		**1245** ± 3	**1287** ± 2	1263 ±18	1264 ±22	1248 ±15
	`OpenEquivariance`	✓	767 ± 7	627 ± 4	574 ± 3	573 ± 4	564 ± 4
	`OpenEquivariance`		806 ± 7	598 ± 3	504 ± 2	233 ± 1	205 ± 1
	`e3j` (CUDA)	✓	446 ± 1	709 ± 2	638 ± 5	560 ± 3	623 ± 4
	`e3j` (Pallas GPU)		–	1267 ±15	**1292** ± 9	**1319** ±10	**1328** ±30
3	`e3nn` (f32)†		56 ± 0	53 ± 1	52 ± 1	51 ± 1	27 ± 1
	`CuEquivariance`		**1059** ± 8	1104 ± 3	1119 ±12	1118 ±22	1128 ±29
	`OpenEquivariance`	✓	451 ± 2	458 ± 1	411 ± 9	324 ± 6	253 ± 2
	`OpenEquivariance`		470 ± 2	190 ± 0	185 ± 1	151 ± 0	154 ± 1
	`e3j` (CUDA)	✓	310 ± 0	423 ± 2	402 ± 2	317 ± 1	183 ± 0
	`e3j` (Pallas GPU)		–	**1407** ± 3	**1432** ±18	**1483** ±22	**1599** ±12

We also compare the performance of our CUDA and Pallas GPU kernels against the $SO(2)$ convolution (Passaro and Zitnick, 2023). Our convolution kernels are 4 to 6 times faster than a JAX-based $SO(2)$ convolution up to $L = 3$ (see (Brunken et al., 2025) for implementation details), while curves cross at $L = 4$ for the forward, and $L = 5$ for the backward (see figure 7) when force-collapsing the multiplicity to maintain an equivalent level of expressivity. Further details and plots can be found in Appendix B.3. It is worth noting that the authors of UMA (Wood et al., 2026), also produced a Triton based kernel optimization for the convolution which reduces runtime by a third though fixed for $L = 2$. Assuming this improvement was ported into the JAX version it would still remain ~ 6 times slower than our Pallas GPU convolution kernel.

3.2 END-TO-END BENCHMARKS

In order to assess the library in a practical setting, we performed end-to-end MLIP profiling and benchmarks of MACE and NequIP architectures. We compare our implementation with the `e3nn_jax` baseline of the `mlip` library (Brunken et al., 2025), chosen for its unified and fast integration in downstream workflows (batched inference and simulations or relaxations), and to make sure that the backends are directly comparable. For each architecture, we connected `e3j`, `cuEquivariance` and `OpenEquivariance` as numerically equivalent message-passing backends.

Although MLIP models have linear $O(N)$ complexity in the total number of atoms N, the message-passing step scales with the number of edges and the average connectivity of the graph is typically larger than 40. For our MACE model (a) variant, message materialization may effectively bound achievable system sizes to around 13 thousand atoms before reaching memory overflows, while fused message-passing makes it possible to process up to 64 thousand atoms on a single NVIDIA® H100 GPU with 80 GB of global memory.

Our results for MACE and NequIP are detailed in figures 3 and 4. It is worth noting that while not tested in this paper, the library can also be deployed across a number of alternative MLIP architectures such as GRACE (Lysogorskiy et al., 2026) and Equiformer / E2Former (Liao and Smidt, 2023; Liao et al., 2024; 2026; Li et al., 2025).

Table 2: **End-to-end NPT performance of a MACE model on a 25Å water box (GPU).** Runtimes are reported for 100ps long NPT simulations with a Monte Carlo barostat, Hyperparameters are from the MACE (a) variant of table 3, notably `correlation` $= 2$ and `node_symmetry` $= 2$. Only the convolution block is dispatched to dedicated kernels matching the reference implementation numerically. The initial structure (solvated 2-methyl-butane, equilibrated with a classical force field) consists of 1503 atoms and 10% initial edge padding (83,325 static edge count).

Backend	Deterministic	Open-source	ms/step	ns/day
`e3j` (CUDA)	✓	✓	**8.821** ± 0.045	**9.80** ± 0.05
`OpenEquivariance`	✓	✓	16.137 ± 1.063	5.38 ± 0.35
`cuEquivariance`			6.586 ± 0.013	13.12 ± 0.03
`e3j` (Pallas GPU)		✓	**4.906** ± 0.014	**17.61** ± 0.05
`OpenEquivariance`		✓	10.041 ± 0.040	8.60 ± 0.03
`e3nn`		✓	68.298 ± 0.063	1.27 ± 0.00

One point to note is that in all MACE benchmarks, the symmetric contraction relies on the CUDA tensor product kernel of `e3j` with channel-mixing mode `MAP`. This helps enforce numerical consistency and allowed us to run stable simulations from a single trained checkpoint. It also has a relatively small impact with the `correlation = 2` results presented here. See Appendix C.1 for experiments at `correlation = 3` involving the proprietary `cuEquivariance` kernel for the symmetric contraction step and more discussion.

Paradoxically, table 4 shows `OpenEquivariance` leads to slower simulations with the deterministic convolution kernel (see figure 2). This gap increases dramatically with the number of padding edges (above 30 ms/step with 25% padding), indicating their kernel hangs waiting for the slowest block accumulating messages on the single padding node. Our kernels flag padding edges so that work on these edges can be skipped, avoiding this overhead.

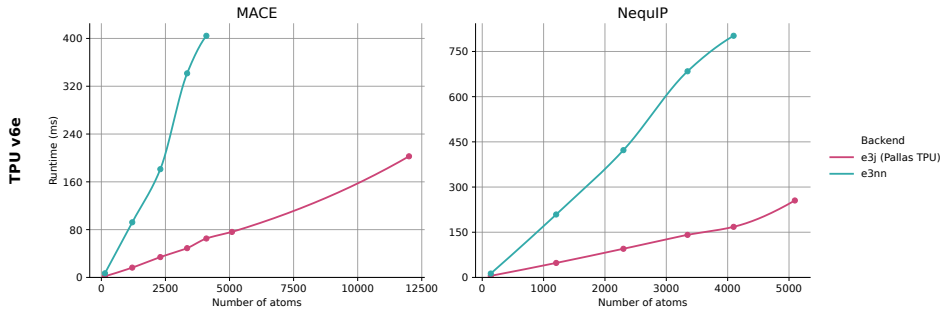

Figure 3: **Runtime (ms) for end-to-end force inference of MACE and NequIP on TPU**. Inference is performed through integration of additional convolution backends for the `mlip` library (Brunken et al., 2025) and run on real protein systems with 5Å cutoff. Hyperparameters can be found in table 3: MACE (a) (correlation 2) has 2 layers and NequIP has 5 layers, both have 128 channels.

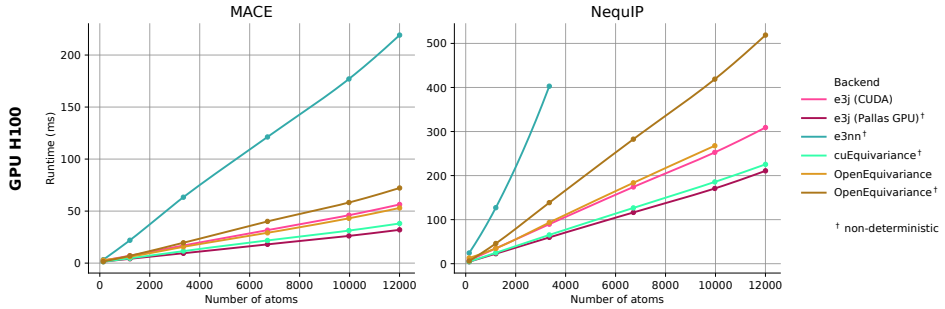

Figure 4: **Runtime (ms) for end-to-end force inference of MACE and NequIP on GPU**. Inference is performed through integration of additional convolution backends for the `mlip` library (Brunken et al., 2025) and run on real protein systems with 5Å cutoff. Hyperparameters can be found in table 3: MACE (a) (correlation 2) has 2 layers and NequIP has 5 layers, both have 128 channels.

4 Discussion

Equivariant architectures have often been criticized for being computationally heavy, with learned equivariance often being put forward as an efficient inference time alternative. Multiple architectures have recently moved away from the strict inductive bias in favor of transformer architectures (Qu et al., 2026; Elhag et al., 2026), using the advantageous engineering of the transformer architecture as motivation for the change. In this paper we show that with proper engineering, full Clebsch-Gordan tensor product and associated message passing convolutions can be efficiently implemented on both GPUs and TPUs, reaching over 80% of HBM throughput. This should provide a more even playing field when comparing highly engineered transformer architectures with equivariant networks.

It is worth noting that while dedicated kernels do improve the efficiency of the Clebsch-Gordan coefficients, and appear to be the best performing operation for $L < 5$, they remain at a disadvantage in terms of scaling in L compared to other methods, in particular the $SO(2)$ convolution. Full treatment of the tensor product however has the benefit of being applicable to arbitrary feature vector pairs, and not solely in combination with harmonic features. A fair comparison would also require for $SO(2)$ convolutions to also receive dedicated engineering, an effort that has begun with the second version of UMA (Wood et al., 2026) but that remains poorly explored by the community.

One key point to note however, is that the achieved throughput remains below maximum bandwidth of the H100 GPU and v6e TPU, suggesting further improvements may be possible. We believe the release of `e3j` as an open-source library will prove a valuable new starting point for the community to continue optimizing these operations on current and future hardware.

AI USE STATEMENT

In this work, we used generative AI tools to help implement methods. We have not used generative AI tools to help develop theoretical models or conceptual frameworks, formulate mathematical claims, propose or refine hypotheses, design or provide feedback on research methodology or experiments, assist with translation, support qualitative and thematic data analysis, interpret results, and the rest of the required disclosure tasks (generate synthetic data sets, help develop theoretical models or conceptual frameworks, provide critical ingredients for proving mathematical claims, assist in the writing of proofs, clean and reformat dataset) are not applicable to this work. Additionally, we used generative AI tools to modify scientific figures and tables, create or edit software code. We have reviewed all AI-assisted work. LLM-generated code was reviewed by more than 2 authors and tested for correctness. We take responsibility for the final content of this work, including text, claims or artifacts produced with the aid of generative AI.

ACKNOWLEDGMENTS

This work was supported by Cloud TPUs from Google's TPU Research Cloud (TRC).

We would like to express our gratitude to Sébastien B. and Oliver B. for encouragement and support in the early development of the library, and to warmly thank Marco C. for his continued assistance in the MLIP integration effort.

REFERENCES

Ilyes Batatia, Dávid Péter Kovács, Gregor N. C. Simm, Christoph Ortner, and Gábor Csányi. MACE: Higher Order Equivariant Message Passing Neural Networks for Fast and Accurate Force Fields, 2023. URL https://arxiv.org/abs/2206.07697.

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.

Anton Bochkarev, Yury Lysogorskiy, and Ralf Drautz. Fast contracted clebsch–gordan tensor products for equivariant graph neural networks, 2026. URL https://arxiv.org/abs/2605.15073.

James Bradbury, Roy Frostig, Peter Hawkins, Matthew James Johnson, Chris Leary, Dougal Maclaurin, George Necula, Adam Paszke, Jake VanderPlas, Skye Wanderman-Milne, and Qiao Zhang. JAX: composable transformations of Python+NumPy programs, 2018. URL http://github.com/jax-ml/jax.

Christoph Brunken, Olivier Peltre, Heloise Chomet, Lucien Walewski, Manus McAuliffe, Valentin Heyraud, Solal Attias, Martin Maarand, Yessine Khanfir, Edan Toledo, Fabio Falcioni, Marie Bluntzer, Silvia Acosta-Gutiérrez, and Jules Tilly. Machine learning interatomic potentials: library for efficient training, model development and simulation of molecular systems, 2025. URL https://arxiv.org/abs/2505.22397.

NVIDIA Corporation. *CUDA C++ Programming Guide*, 2025. URL https://docs.nvidia.com/cuda/archive/12.8.1/cuda-c-programming-guide/index.html.

Ahmed A. Elhag, Arun Raja, Alex Morehead, Samuel M. Blau, Hongtao Zhao, Christian Tyrchan, Eva Nittinger, Garrett M. Morris, and Michael M. Bronstein. Learning inter-atomic potentials without explicit equivariance, 2026. URL https://arxiv.org/abs/2510.00027.

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, 2025. URL https://arxiv.org/abs/2502.12147.

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

Mario Geiger, Tess Smidt, Alby M., Benjamin Kurt Miller, Wouter Boomsma, Bradley Dice, Kostiantyn Lapchevskyi, Maurice Weiler, Michał Tyszkiewicz, Simon Batzner, Dylan Madisetti, Martin Uhrin, Jes Frellsen, Nuri Jung, Sophia Sanborn, Mingjian Wen, Josh Rackers, Marcel Rød, and Michael Bailey. Euclidean neural networks: e3nn, April 2022. URL https://doi.org/10.5281/zenodo.6459381.

Jonathan Godwin, Thomas Keck, Peter Battaglia, Victor Bapst, Thomas Kipf, Yujia Li, Kimberly Stachenfeld, Petar Veličković, and Alvaro Sanchez-Gonzalez. Jraph: A library for graph neural networks in jax., 2020. URL http://github.com/deepmind/jraph.

Brian C. Hall. An Elementary Introduction to Groups and Representations, 2000. URL https://arxiv.org/abs/math-ph/0005032.

Valentin Heyraud, Zachary Weller-Davies, and Jules Tilly. Integral formulas for vector spherical tensor products, 2026. URL https://arxiv.org/abs/2603.08630.

Lin Huang, Chengxiang Huang, Ziang Wang, Yiyue Du, Chu Wang, Haocheng Lu, Yunyang Li, Xiaoli Liu, Arthur Jiang, and Jia Zhang. E2former-v2: On-the-fly equivariant attention with linear activation memory, 2026. URL https://arxiv.org/abs/2601.16622.

Norman P. Jouppi, Cliff Young, Nishant Patil, David Patterson, Gaurav Agrawal, Raminder Bajwa, Sarah Bates, Suresh Bhatia, Nan Boden, Al Borchers, Rick Boyle, Pierre-luc Cantin, Clifford Chao, Chris Clark, Jeremy Coriell, Mike Daley, Matt Dau, Jeffrey Dean, Ben Gelb, Tara Vazir Ghaemmaghami, Rajendra Gottipati, William Gulland, Robert Hagmann, C. Richard Ho, Doug Hogberg, John Hu, Robert Hundt, Dan Hurt, Julian Ibarz, Aaron Jaffey, Alek Jaworski, Alexander Kaplan, Harshit Khaitan, Daniel Killebrew, Andy Koch, Naveen Kumar, Steve Lacy, James Laudon, James Law, Diemthu Le, Chris Leary, Zhuyuan Liu, Kyle Lucke, Alan Lundin, Gordon MacKean, Adriana Maggiore, Maire Mahony, Kieran Miller, Rahul Nagarajan, Ravi Narayanaswami, Ray Ni, Kathy Nix, Thomas Norrie, Mark Omernick, Narayana Penukonda, Andy Phelps, Jonathan Ross, Matt Ross, Amir Salek, Emad Samadiani, Chris Severn, Gregory Sizikov, Matthew Snelham, Jed Souter, Dan Steinberg, Andy Swing, Mercedes Tan, Gregory Thorson, Bo Tian, Horia Toma, Erick Tuttle, Vijay Vasudevan, Richard Walter, Walter Wang, Eric Wilcox, and Doe Hyun Yoon. In-datacenter performance analysis of a tensor processing unit. *ACM SIGARCH Computer Architecture News*, 45(2):1–12, 2017. ISSN 0163-5964. doi: 10.1145/3140659.3080246. URL http://dx.doi.org/10.1145/3140659.3080246.

Grzegorz Kaszuba, Tomasz Krakowski, Bartosz Ziegler, Andrzej Jaszkiewicz, and Piotr Sankowski. Implicit modeling of equivariant tensor basis with Euclidean turbulence closure neural network. *Physics of Fluids*, 37, 02 2025. doi: 10.1063/5.0249490.

Dávid Péter Kovács, J. Harry Moore, Nicholas J. Browning, Ilyes Batatia, Joshua T. Horton, Yixuan Pu, Venkat Kapil, William C. Witt, Ioan-Bogdan Magdău, Daniel J. Cole, and Gábor Csányi. MACE-OFF: Transferable Short Range Machine Learning Force Fields for Organic Molecules, 2025. URL https://arxiv.org/abs/2312.15211.

Yunyang Li, Lin Huang, Zhihao Ding, Xinran Wei, Chu Wang, Han Yang, Zun Wang, Chang Liu, Yu Shi, Peiran Jin, Tao Qin, Mark Gerstein, and Jia Zhang. E2former: An efficient and equivariant transformer with linear-scaling tensor products. In *The Thirty-ninth Annual Conference on Neural Information Processing Systems*, 2025. URL https://openreview.net/forum?id=ls5L4IMEwt.

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.

Yi-Lun Liao, Alexander J. Hoffman, Sabrina C. Shen, Alexandre Duval, Sam Walton Norwood, and Tess Smidt. Equiformerv3: Scaling efficient, expressive, and general se(3)-equivariant graph attention transformers, 2026. URL https://arxiv.org/abs/2604.09130.

Shengjie Luo, Tianlang Chen, and Aditi S. Krishnapriyan. Enabling efficient equivariant operations in the fourier basis via gaunt tensor products, 2024. URL https://arxiv.org/abs/2401.10216.

Yury Lysogorskiy, Anton Bochkarev, and Ralf Drautz. Graph atomic cluster expansion for foundational machine learning interatomic potentials. *npj Computational Materials*, 12(1), February 2026. ISSN 2057-3960. doi: 10.1038/s41524-026-01979-1. URL http://dx.doi.org/10.1038/s41524-026-01979-1.

Hartmut Maennel, Oliver T. Unke, and Klaus-Robert Müller. Complete and efficient covariants for 3d point configurations with application to learning molecular quantum properties, 2024. URL https://arxiv.org/abs/2409.02730.

Saro Passaro and C. Lawrence Zitnick. Reducing SO(3) Convolutions to SO(2) for Efficient Equivariant GNNs. In *Proceedings of the 40th International Conference on Machine Learning*, ICML'23. JMLR.org, 2023.

Adam Paszke, Sam Gross, Soumith Chintala, Gregory Chanan, Edward Yang, Zachary DeVito, Zeming Lin, Alban Desmaison, Luca Antiga, and Adam Lerer. Automatic differentiation in pytorch. In *NIPS-W*, 2017.

Eric Qu, Brandon M. Wood, Aditi S. Krishnapriyan, and Zachary W. Ulissi. A recipe for scalable attention-based mlips: unlocking long-range accuracy with all-to-all node attention, 2026. URL https://arxiv.org/abs/2603.06567.

Varun Shankar, Shivam Barwey, Zico Kolter, Romit Maulik, and Venkatasubramanian Viswanathan. Importance of equivariant and invariant symmetries for fluid flow modeling, 2023. URL https://arxiv.org/abs/2307.05486.

Nathaniel Thomas, Tess Smidt, Steven Kearnes, Lusann Yang, Li Li, Kai Kohlhoff, and Patrick Riley. Tensor field networks: Rotation- and translation-equivariant neural networks for 3d point clouds, 2018. URL https://arxiv.org/abs/1802.08219.

Oliver T. Unke and Hartmut Maennel. E3x: $\mathrm{E}(3)$-equivariant deep learning made easy. *arXiv preprint arXiv:2401.07595*, 2024.

Oliver T. Unke, Mihail Bogojeski, Michael Gastegger, Mario Geiger, Tess Smidt, and Klaus-Robert Müller. Se(3)-equivariant prediction of molecular wavefunctions and electronic densities, 2021. URL https://arxiv.org/abs/2106.02347.

Eugene Wigner. *Group Theory And Its Application to the Quantum Mechanics of Atomic Spectra*. Elsevier Science, 1959.

Robert S. Womersley. *Efficient Spherical Designs with Good Geometric Properties*, page 1243–1285. Springer International Publishing, 2018. ISBN 9783319724560. doi: 10.1007/978-3-319-72456-0_57. URL http://dx.doi.org/10.1007/978-3-319-72456-0_57.

Brandon M. Wood, Misko Dzamba, Xiang Fu, Meng Gao, Muhammed Shuaibi, Luis Barroso-Luque, Kareem Abdelmaqsoud, Vahe Gharakhanyan, John R. Kitchin, Daniel S. Levine, Kyle Michel, Anuroop Sriram, Taco Cohen, Abhishek Das, Ammar Rizvi, Sushree Jagriti Sahoo, Zachary W. Ulissi, and C. Lawrence Zitnick. Uma: A family of universal models for atoms, 2026. URL https://arxiv.org/abs/2506.23971.

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.

YuQing Xie, Ameya Daigavane, Mit Kotak, and Tess Smidt. Asymptotically fast clebsch-gordan tensor products with vector spherical harmonics, 2026. URL https://arxiv.org/abs/2602.21466.

A Mathematical Background

In this appendix, we provide further theoretical details on the core components of `e3j`. Representation theory of the rotation and Euclid groups had ground-breaking applications in Quantum Mechanics, where they notably provided a first derivation of the Hydrogen energy levels and their degeneracy, see e.g. Wigner (Wigner, 1959). For more contemporary introductions to the subject, readers may refer to (Hall, 2000; Unke and Maennel, 2024).

A.1 Euclidean equivariance

The Euclid group $E_3 = O_3 \ltimes \mathbb{R}^3$ describes the possible changes of frames of reference over the Euclidean 3-space, i.e. the compositions of translations, rotations and reflections. Euclidean equivariance is the property by which a function on the Euclidean space transforms consistently with its inputs upon any *Euclidean transform* $g \in E(3)$, for instance, with a force function $\mathbf{F}(\mathbf{r}, \mathbf{z})$:

$$\mathbf{F}(g \cdot \mathbf{r}, \mathbf{z}) = g \cdot \mathbf{F}(\mathbf{r}, \mathbf{z}). \tag{1}$$

where $\mathbf{r} \in \mathbb{R}^{n\times 3}$ denotes a matrix of atomic positions, $\mathbf{z} \in \mathbb{N}^n$ denotes a vector of atomic numbers, and n is the number of atoms of the system or region of interest.

In general, the Euclidean group may not only act on (n copies of) $\mathbb{R}^3$, but also on general (real or complex) vector spaces called *representations* of E_3 (also called E_3-modules): they consist of pairs (V, ρ) where the vector space V is equipped with a smooth group morphism ρ mapping any Euclidean transform $g \in E_3$ to an invertible matrix $\rho_g \in GL(V)$. A function $\mathbf{F} : V \to V'$ is called *equivariant* if the following diagram is commutative:

$$\begin{array}{ccc} V & \xrightarrow{\mathbf{F}} & V' \\ {\scriptstyle \rho_g}\downarrow & & \downarrow{\scriptstyle \rho'_g} \\ V & \xrightarrow{\mathbf{F}} & V' \end{array} \tag{2}$$

While morphisms of E_3-representations are usually assumed linear, note that the above definition of equivariance applies to non-linear functionals just as well, in particular polynomial functionals which are of particular importance in the classification of E_3-representations.

A fundamental result is that any orthogonal representation V of O_3 can be decomposed into a direct sum of irreducible representations or *irreps*, each *irrep* being chosen among a well known classification of possible fundamental types (related to harmonic polynomials over $\mathbb{R}^3$). Note that the dimensions of irreducible representations translate into tangible observables in quantum chemistry (QC), where l and $2l + 1$ respectively determine the symmetry (S, P, D...) and degeneracy (half the maximal number of occupying electrons) of an energy level in a hydrogen-like atom.

A.2 Harmonic polynomials

Definition. The representation theory of SO_3 is closely related to the harmonic polynomials of $\mathbb{C}[x, y, z]$. Homogeneous, degree-l polynomials are finite-dimensional vector spaces, naturally equipped with an SO_3 action. Because the Laplacian operator Δ (trace of the hessian) is also invariant under SO_3, the space $\mathbb{Y}_l \subset \mathbb{C}_l[x, y, z]$ of degree-l harmonic polynomials (satisfying $\Delta P = 0$) is a sub-representation, i.e. a subspace stable under SO_3.

$$\Delta Y_l^m = \frac{\partial^2 Y_l^m}{\partial x^2} + \frac{\partial^2 Y_l^m}{\partial y^2} + \frac{\partial^2 Y_l^m}{\partial z^2} = 0$$

Any irreducible representation of SO_3 (i.e. a "smallest" vector space with an SO_3-action, having no other stable strict subspace than 0) is isomorphic to some $\mathbb{Y}_l$, of odd dimension $2l+1$. Any larger representation of SO_3 can be decomposed as a direct sum of irreducible representations $\bigoplus_l (\mathbb{Y}_l)^{k_l}$.

Equivariance. For every degree l, the maps $\mathbf{Y}_l : \mathbb{R}^3 \to \mathbb{Y}_l \simeq \mathbb{C}^{2l+1}$ are *equivariant non-linear embeddings*, typically used as a basis for learning more complex non-linear equivariant representations $f_\theta : V \to V'$ within e.g. deep MLIP networks. This means that for every rotation $g \in SO_3$, one may construct the so-called *Wigner D-matrix* D_g, acting on the $2l+1$ space of polynomial activations $\mathbf{Y}_l$ in the following commutative diagram:

$$\begin{array}{ccc} \mathbb{R}^3 & \xrightarrow{\mathbf{Y}_l} & \mathbb{Y}_l \\ {\scriptstyle g}\downarrow & & \downarrow{\scriptstyle D_g} \\ \mathbb{R}^3 & \xrightarrow{\mathbf{Y}_l} & \mathbb{Y}_l \end{array} \tag{3}$$

The spaces $\mathbb{Y}_l$ consist of the basic pieces of any Euclidean representation V, as any irreducible representation of the group of rotations O_3 is isomorphic to some $\mathbb{Y}_l$ for some l. Additionally, irreducible E_3 representations carry a parity label $\pm$ (even/odd) dictating whether reflections act with a sign or as the identity. See Appendix G for further mathematical details.

A.3 Key Operations in Scope

Equivariant architectures typically enforce the equivariance constraint (1) by a few common design patterns and building blocks, the `e3j` package provides a harmonized and functional API around these:

- **Harmonics:** Restrict available geometric information to the edge vectors $(\mathbf{r}_{ab}) \in \mathbb{R}^{n_E \times 3}$. This already enforces translation invariance. Then expand edge vectors $\mathbf{r}_{ab} \in \mathbb{R}^3$ with *harmonic embeddings* $\mathbf{Y}_l(\mathbf{r}_{ab})$ where $\mathbf{Y}_l$ denotes one bank of $2l+1$ rotation-equivariant activation filters built from the $2l+1$ degree-l harmonic polynomials, typically concatenated over degrees $l = 0, \dots, l_{\max}$ with $l_{\max}$ rarely exceeding 3:
$$\mathbf{Y}_l(\mathbf{r}) = \left[\, Y_{lm}(\mathbf{r}) \,|\, m = -l \dots l \,\right] \tag{4}$$
- **Linear mixing:** Rescale or mix channels and multiplicities linearly between irreducible features of a same degree l. While rarely a bottleneck by itself, there is opportunity for scalar mixings to be fused e.g. within message-passing operations, where edge scalars representing chemical species and radial embeddings of interatomic distances are coupled with rotation-equivariant features.
$$\mathtt{L}_W(\mathbf{x})^{lm,k} = \sum_{k'm'} W^{km}_{k'm'} \mathbf{x}^{lm',k'} \tag{5}$$
- **Tensor product:** Couple latent equivariant features with *Clebsch-Gordan tensor products* $\mathbf{z} = \mathbf{x} \otimes_C \mathbf{y}$, where $\mathbf{x}$ may for instance denote latent node features from the previous layer and $\mathbf{y}$ the harmonic embedding of an edge vector, or a more general latent feature vectors organized in irreps. When $\mathbf{x}$ and $\mathbf{y}$ are irreducibles of degree l and l', their tensor product $\mathbf{z}$ is obtained from the $2\min(l,l')+1$ bilinear pairings (or "paths") of output degrees $L = |l-l'|, \dots, l+l'$, given by:
$$(\mathbf{x} \otimes_C \mathbf{y})^{LM} = \sum_{m+m'=M} C^{LM}_{lm,l'm'}\, \mathbf{x}^{lm}\, \mathbf{y}^{l'm'} \tag{6}$$
where $C^{LM}_{lm,l'm'}$ are the Clebsch-Gordan coefficients.
- **Message passing:** Aggregate the edge-wise features, usually computed through combination of harmonics projection, tensor product and linear or scalar mixing. The message passing aggregation tends to be the operational bottleneck once efficient tensor product operations

are implemented. A typical message passing layer in the context of equivariant GNNs can be written as:

$$\mathbf{m}\prime_b = \sum_{a \in \mathcal{N}(b)} \mathtt{L}_{\mathbf{s}_{ab}} \left(\mathbf{x}_a \otimes_C \mathbf{Y}(\mathbf{r}_{ab}) \right) \tag{7}$$

where $\mathbf{Y}(\mathbf{r}_{ab}) = \bigoplus_l \mathbf{Y}_l(\mathbf{r}_{ab})$ and $\mathtt{L}_{\mathbf{s}_{ab}}$ denotes a linear mixing with radial edge scalars $\mathbf{s}_{ab}$, which typically depend on interatomic distances $||\mathbf{r}_{ab}||$ via a radial basis function (RBF) embedding followed by a multi-layer perceptron (MLP).

While we have used MLIP as a testing ground for realistic workloads, we have not yet included operations specifically targeted at particular MLIP models. The main example is the Symmetric Contraction module in MACE. We did however include benchmarks with the existing `cuEquivariance` Symmetric Contraction kernel in appendix C.1.

B Comparison with Cubically Scaling Tensor Products

B.1 Gaunt and Vector Signal Tensor Products

In this section, we benchmark the `e3j` tensor product against the Gaunt tensor product (GTP) (Luo et al., 2024) and the vector signal tensor product (VSTP) (Xie et al., 2026; Heyraud et al., 2026; Bochkarev et al., 2026), which evaluate the tensor product as an integral over the sphere rather than a contraction over Clebsch-Gordan coefficients. The GTP and VSTP are complementary: the GTP covers only the symmetric paths, where the sum $l_1 + l_2 + l_3$ of all of the irreps entering into the tensor product is even, while the VSTP covers only the skew-symmetric paths where the sum is odd.

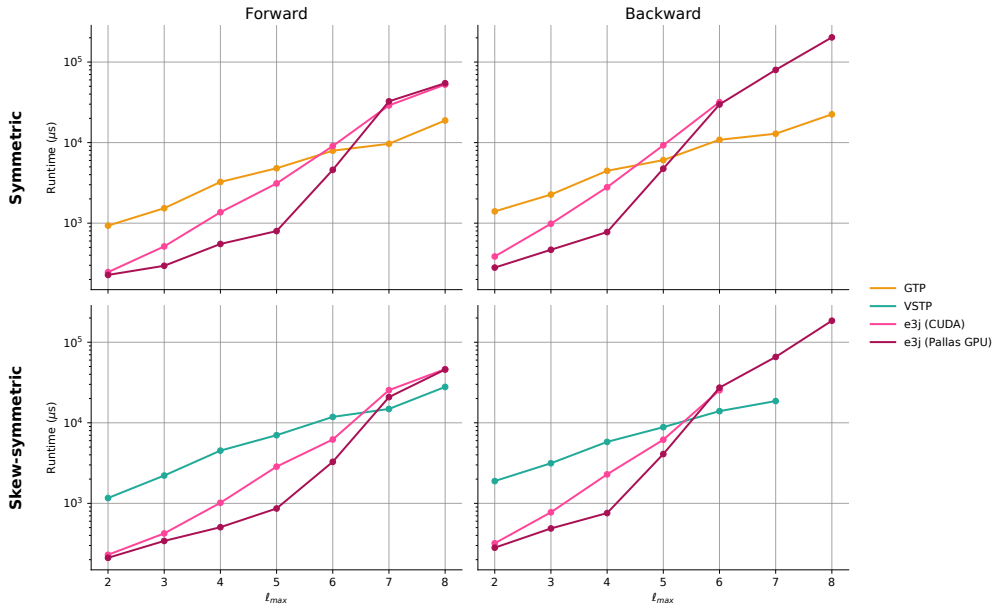

Figure 5: **Comparison with Gaunt and Vector Signal Tensor Products.** Benchmarks show the runtime scaling with $L = l_{max}$ with a fixed batch size $B = 32{,}768$ and channels $C = 128$. Every Clebsch-Gordan path (l_1, l_2, l_3) is symmetric or skew-symmetric, according to the parity of $l_1 + l_2 + l_3$. The symmetric panel, where the sum is even, benchmarks the `e3j` tensor product on symmetric paths against the Gaunt tensor product (GTP) (Luo et al., 2024), whilst the skew-symmetric panel benchmarks against the vector signal tensor product (VSTP) (Xie et al., 2026; Heyraud et al., 2026). Both the GTP and the VSTP are evaluated through integral formulas on a spherical t-design quadrature (Womersley, 2018). Runtimes are obtained on a single NVIDIA H100.

Efficient implementations of both the VSTP and GTP rely on the tensor product emitting a single copy of each output degree rather than one per path, and, relatedly, that a weighted tensor product has weights taking a factorised form $w^{l_3}_{l_1 l_2} = a_{l_1} b_{l_2} c_{l_3}$. In this benchmark, we therefore collapse the `e3j`

output by degree to ensure the output spaces match in all cases. We also choose to time the tensor product, since factorised weights act as linear maps on the input and output spaces that are identical in all three cases. In practice, we compute the GTP and VSTP from their integral formulas, which are evaluated on a spherical t-design quadrature (Womersley, 2018).

To match the paths of the GTP and VSTP, we utilize the parity selection rule $p_1 p_2 = p_3$ that the Clebsch-Gordan product already enforces. With natural-parity irreps $p_l = (-1)^l$ for both the inputs and the targets, the parity rule enforces $(-1)^{l_1+l_2+l_3} = 1$ and keeps exactly the symmetric paths. Conversely, with the anti-natural $p_l = (-1)^{l+1}$ irreps, only the skew-symmetric paths are allowed.

The results are shown in Figure 5. For $L \leq 3$, where the cost of the quadrature operations dominates, the `e3j` kernel is 2 to 4 times faster than the GTP and 4 to 6 times faster than the VSTP. As L grows, the theoretical scaling in L becomes relevant ($O(L^4)$ for quadrature methods vs $O(L^5)$ for sparse Clebsch-Gordan contraction). The forward pass curves cross at $L = 6$ and $L = 7$ for the GTP/VSTP respectively, and at $L = 5$ and $L = 6$ for the backward, showing that `e3j` remains faster in the regime practical for MLIP applications. It is worth noting that there are also other implementations of the GTP and VSTP that have better theoretical scaling in L (Xie et al., 2026), but that we find to be slower in practice over this range of degrees.

B.2 Matrix Tensor Products from `e3x`

Another tensor product operation with an efficient $O(L^4)$ scaling has been proposed by the authors of the `e3x` library (Maennel et al., 2024; Unke and Maennel, 2024). We follow Xie et al. (Xie et al., 2025) and refer to this operation as Matrix Tensor Product (MTP). The MTP is implemented in the `FusedTensor` module of `e3x`. This operation computes the tensor product between feature vectors in three steps. First, considering the operation couples two input vectors containing irreps features of order $0 \leq l \leq L$, each input vector is mapped to a $(2\tilde{l}+1) \times (2\tilde{l}+1)$ square matrix, with $\tilde{l} = \lceil L/2 \rceil$. This map corresponds to the isomorphism

$$\bigoplus_{l=0}^{L} \mathbb{Y}_l \cong \mathbb{Y}_{\tilde{l}} \otimes \mathbb{Y}_{\tilde{l}}, \tag{8}$$

where elements of the right hand-side space can be seen as square matrices produced by an outer product of two feature vectors in $\mathbb{Y}_{\tilde{l}}$. Then, the square matrices encoding the inputs are multiplied. Finally, the resulting square matrix is mapped back to the output feature vector. The matrix product exhibits an efficient $O(L^3)$ scaling, but the conversions between square matrices and feature vectors yield an overall scaling of $O(L^4)$.

In this section, we benchmark the `e3j` tensor product against the MTP. Note that for a given irrep path, the MTP is proportional to the GTP-VSTP operations, although the proportionality coefficient sometimes vanishes, so that the MTP is in general strictly less expressive than the GTP-VSTP. We compare the FusedTensor module of `e3j` against the same `e3j` tensor product used in the benchmark of Appendix B.1. As the `FusedTensor` implementation includes the path weights $w_{l_1 l_2}^{l_3}$, we include linear mixings on both the inputs and the output feature vectors of the `e3j` tensor product, so that both operations are strictly equivalent. Note that these additional linear mixings account for its higher runtime compared to the version of Appendix B.1.

The results are shown in Figure 6. For $L \leq 8$ the `e3j` tensor product is faster than the `e3x` implementation. As L grows, the efficient $O(L^4)$ scaling of the MTP reduces the gap with the `e3j` implementation, though `e3j` remains faster on this domain.

B.3 SO2 Convolution

In order to compare with SO2 convolution fairly we evaluated the `e3j` convolution implementations with a set of coefficients that collapses output multiplicities, i.e. sums all isomorphic copies of a given irreducible output space together. The reduced output feature dimension means a cost on expressivity, at the benefit of a smaller memory traffic.

Note that while `e3j` convolution kernels natively support any set of coefficients, they have not been optimized for these smaller problem shapes. We however notice that `e3j` is between 4x and 6x faster at $\ell_{max} \leq 3$, while matching SO2 convolution at $\ell_{max} = 4$, see figure 7. Although the

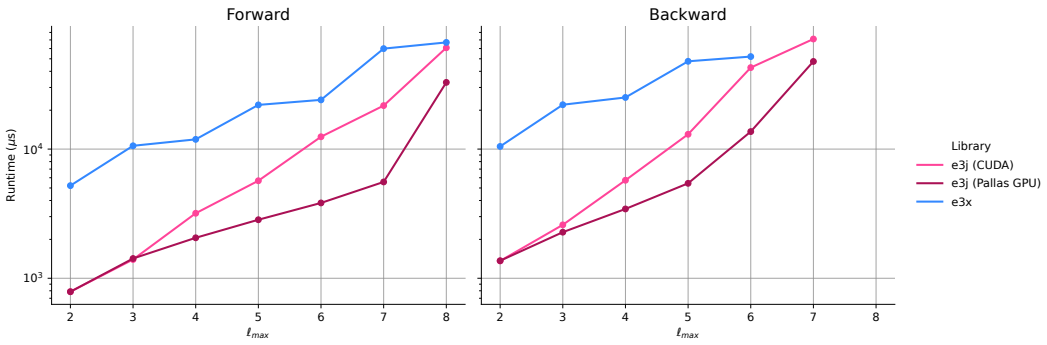

Figure 6: **Comparison of `e3j` and `e3x`'s `FusedTensor` tensor products.** Benchmarks show the runtime scaling with $L = l_{max}$ with a fixed batch size $B = 32{,}768$ and channels $C = 128$. Every Clebsch-Gordan path (l_1, l_2, l_3) is symmetric. Runtimes are obtained on a single NVIDIA H100.

XLA compilation of a plain JAX implementation of SO2 convolution already proves very efficient, a faithful comparison of the method would require similar low-level engineering on the SO2 convolution implementation.

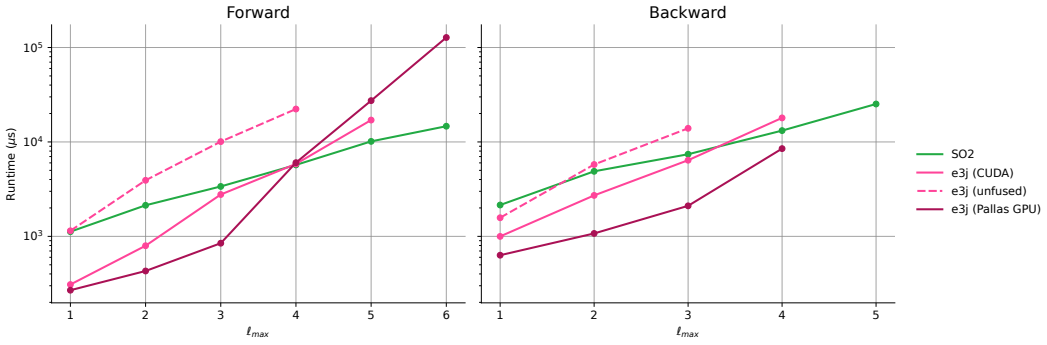

Figure 7: **Comparison of SO2 convolution with Clebsch-Gordan convolution.** Benchmarks show the runtime scaling with $L = l_{max}$ with fixed number of nodes $N = 2048$, number of edges $N_e = 45 \times N$, and fixed number of channels $C = 128$. In contrast to other convolution benchmarks with third-party baselines, output multiplicities are collapsed to one by providing an ad-hoc set of sparse COO coefficients to match a JAX implementation of the SO2 convolution algorithm described in (Passaro and Zitnick, 2023). Runtimes are obtained on a single NVIDIA H100.

C Additional End-to-end Benchmarks

C.1 Symmetric contraction benchmarks

Most of the MACE model benchmarks reported in the main text use the same implementation for the `SymmetricContraction` operation, which consists of:

- a `PowerExpansion` of node features (quadratic or cubic) using the tensor product kernels of `e3j` with channel mixing-mode `MAP`[5],
- a `LinearIndexwise` projection of the concatenated higher-order features using weights that depend on the atomic species.

While conceptually simple, this implementation is suboptimal as the power expansion generates large multiplicities that all collapse eventually after the linear projection steps. The GMEM materialization

[5]Note that not all CUDA backends expose this functionality, and that the batch and channel axes are not contiguous in the optimal trailing channels layout.

of the large intermediate array can be skipped by a dedicated `SymmetricContraction` fusing both operations, as reported in figure 8.

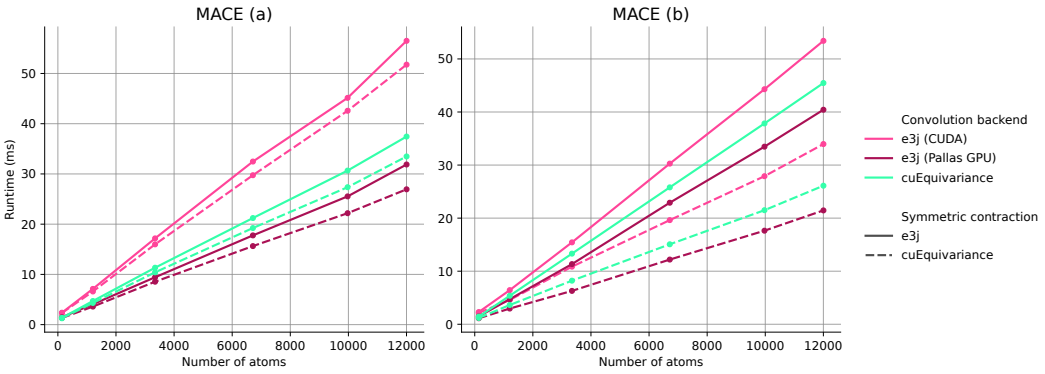

Figure 8: **Effect of the `SymmetricContraction` backend for the MACE model on GPU.** We compare the effect of replacing the naive `e3j`-based power expansion, followed by a species-wise linear projection of higher-order features, with the dedicated `SymmetricContraction` kernel of `cuEquivariance`. The two variants (a) and (b) of the MACE model are detailed in table 3.

An optimized implementation may follow the algorithm proposed in the original MACE paper (Batatia et al., 2023), in device code. The plain JAX implementation of this algorithm however performs worse than the naive power expansion and linear projection, once an efficient CGTP is available. The `e3nn` force-inference benchmarks, illustrating the best efficiency one may reach with a pure JAX implementation, rely on this implementation. However, we cannot currently change the algorithm significantly without breaking numerical consistency. Restoring consistency requires a complex 1-to-1 transformation of learnable parameters and CG coefficient normalization choices to be resolved.

All NPT simulation benchmarks use the same `e3j`-based `SymmetricContraction`. While isolating the effect of the convolution backend, this allows us to load a single model checkpoint to run stable NPT simulations.

At the time of writing, `e3j` does not provide a dedicated `SymmetricContraction` kernel. Our current investigations seemed to show that inlining coefficients with a JIT compiled kernel (using Pallas or NVRTC) seems necessary to reach the same performance as `cuEquivariance`, given that our AOT compiled CUDA kernels reach due to the large number of coefficients and feature sizes that occur during this operation.

C.2 Determinism and deviations of end-to-end predictions

Non-deterministic message aggregation may prove very efficient, since it a allows a kernel to directly loop and distribute work over edges (instead of nesting a potentially imbalanced loop over neighbors inside a loop over receiver nodes) and the memory cache hierarchy may efficiently hide the latency of memory-locked atomic operations. Valency imbalance for instance explains why the deterministic `OpenEquivariance` kernel leads to slower simulations than the non-deterministic, when enforcing a static edge count with all padding edges joining a single padding node, despite being faster in non-padded unique benchmarks.

In some downstream workflows (relaxation, geometry optimization, ...), deterministic predictions may however be a important requirement. When relaxing a periodic cell (as in NPT simulations, see table 2) with a Monte-Carlo barostat, deterministic energy predictions are for instance a hard constraint: small energy fluctuations enter an exponential Boltzmann factor used for a Metropolis-Hastings rejection criterion, and non-deterministic message aggregation would lead to exploding simulations.

In addition to the message-passing step, two sources of non-determinism or numerical noise may compound in the energy prediction:

- addition of atomic energies E_0, which are multiple orders of magnitude larger than the geometry-induced energy variations, and may truncate the constant relative precision of floating-points data types,
- aggregation of graph energies from node energy summands, which may scatter a very large number of contributions to a few scalar numbers with a high degree of concurrency and collisions.

Both of these effects are analyzed in figure 9. In contrast, the force prediction may eliminate the addition of constants and replace non-deterministic scatter operations by deterministic gather operations in the computational graph, as illustrated by figure 10.

We note that this mitigation of stochastic variations is not an automatic consequence of the differentiation process, since differentiating a mean-squared-error loss would presumably compound sources of non-determinism instead of eliminating most of them (through a product of the energy error with energy gradients). A rigorous analysis of the effect of non-determinism on model training behavior is however out of scope of the present work.

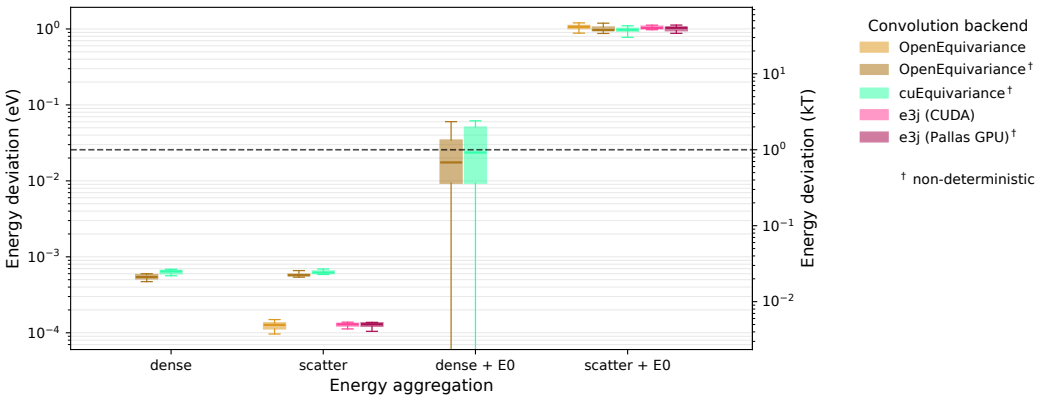

Figure 9: **Run-to-run energy deviation of a MACE model (a) on GPU.** The trained model used for NPT simulations (table 2) is evaluated on a batch of 8 water boxes totaling 21,016 atoms and 889,840 edges, over a 100 times. The experiment was repeated using different energy aggregation schemes: including or dropping the constant atomic energies (E_0); using `mlip`'s default scatter aggregation of node energies over graphs or a dense matrix-vector product for the energy head. The plot represents 5th/95th percentiles as whiskers and 25th/75th percentiles as boxes. When a box is missing, it means that all runs were rigorously deterministic. Hyperparameters are detailed in table 3.

D Benchmarks details

D.1 Throughput and HBM

Our module-specific benchmarks mostly focus on kernel *throughput*, commonly defined as:

$$\texttt{throughput} = \frac{\texttt{sizeof(inputs)} + \texttt{sizeof(outputs)}}{\texttt{runtime}} \tag{9}$$

In addition to being asymptotically independent of I/O size, throughput is also bounded by the so-called *global memory (GMEM) bandwidth*, corresponding to the ideal throughput of an optimal array copy. Global memory is the long-lived data bank used by the device processors to load/store I/O data. It is also called *high bandwidth memory* (HBM) in manufacturer specifications, given its practical importance in delivering the best I/O throughput possible.

The NVIDIA® H100 graphical processing units (GPUs), on which most of our experiments were performed, advertises about 3.35 TB/s HBM. The Google®tensor processing units (TPUs) we could experiment with advertizes 1.20 TB/s HBM for v4 and 1.64 TB/s HBM for v6e (Trillium). Note constant technical progress is made on those characteristics, and a fair comparison should compare devices from the same year, and weigh those metrics by affordability.

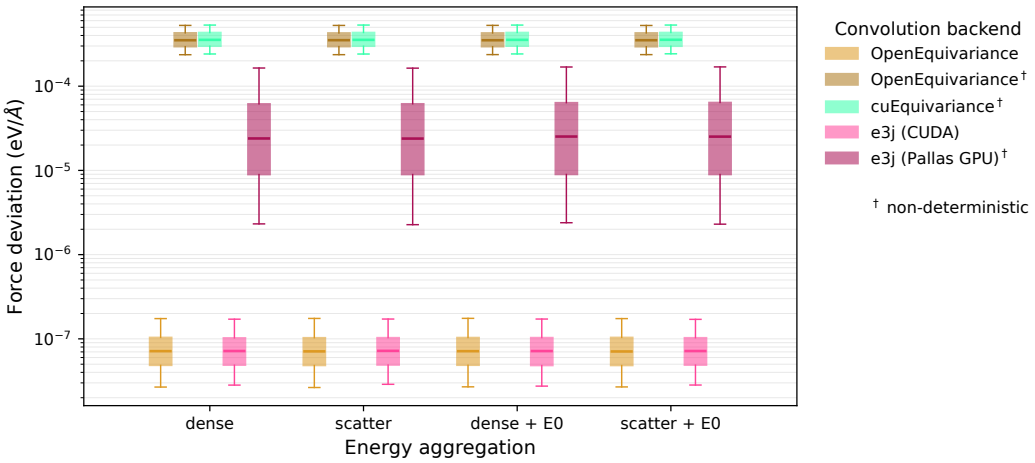

Figure 10: **Run-to-run force deviation of a MACE model (a) on GPU.** The trained model used for NPT simulations (table 2) is evaluated in the same conditions as figure 9. Because XLA can eliminate the addition of atomic energies, and because the VJP of the scatter operation is a deterministic gather operation, we see that the addition of E_0 and the aggregation method have no effect on the deviation of forces. Even with deterministic convolution kernels, forces have non-zero deviations due to some forward gather operations being transposed as non-deterministic scatter operations. Interestingly the Pallas GPU kernel, only deterministic in the forward pass, lies in between.

D.2 Precision

Although `e3j` also provides `float64` binaries, all benchmarks were carried with I/O arrays in single `float32` precision, which proves enough to run stable molecular dynamics simulations. Note however that JAX may internally resort to half-precision arithmetic in tensor contractions (`matmul`, `einsum`, ...) for faster execution, and caps to single-precision by default to avoid downsides incurred by undesired upcasts.

On normal random input, our experiments show that all equivariance backends, regardless of platform, lead to similar accuracies of order 4×10^{-8} and 1×10^{-7} for tensor products and message-passing operations respectively, with respect to a common, deterministic `float64` reference. The only exception is the `e3nn` backend which only yields about 4×10^{-4} elementwise accuracy when the environment variable `JAX_DEFAULT_MATMUL_PRECISION` is not set to highest. This significant gap in precision should therefore be considered when comparing `e3nn` with other backends in end-to-end benchmarks.

D.3 Considerations regarding Pallas and CUDA kernels

To yield speedups on the Google®tensor processing units (TPUs) which JAX targets as well, E3J also defines kernels written in Pallas, a domain-specific language (DSL) which is part of the JAX package and targets GPU and TPU compilation. While TPU benchmarks can only compare Pallas kernels of E3J with E3NN (Geiger et al., 2022), the GPU benchmarks may compare CUDA and Pallas implementations of E3J with other CUDA implementations such as NVIDIA CuEquivariance (TM) and OpenEquivariance (Bharadwaj et al., 2025). One advantage CUDA nonetheless brings over Pallas is the relative stability of compiled binaries and `nvcc` toolchain over JAX version dependencies.

However, the Pallas language and Mosaic GPU compiler streamline the just-in-time (JIT) compilation of device code, enabling the production of very specialized and efficient kernels. In addition to problem shapes and sizes, that cannot be trivially defined from static parameters with ahead-of-time (AOT) compilation of a traditional CUDA kernel, JIT compilation from Python source lets one seamlessly *inline* the Clebsch-Gordan coefficients inside the produced assembly code. This can significantly reduce memory traffic from the unified SMEM/L1 cache.

Note that other solutions such as NVIDIA's NVRTC compiler enable JIT compilation of CUDA/C++source, a solution leveraged by `OpenEquivariance`. Our CUDA kernels do not make use of JIT compilation at this time, and stream through coefficients as an actual array buffer loaded from global memory.

D.4 MLIP INTEGRATION

Hyperparameters defining the models used in the end-to-end benchmarks are detailed in table 3, they corresponding to the configuration fields of the `mlip` library (Brunken et al., 2025).

While end-to-end MLIP integration gives the most significant results for applications, it is also a complex task that needs to be carried carefully in order to preserve numerical predictions and faithfulness of comparisons. Furthermore, while MLIPs consist of a primary motivation for `e3j`, we view the library as an all-purpose low-level tool whose usage may not be limited to the two particular architectures benchmarked in the present work.

All the MLIP models compared were checked to match numerically regardless of the convolution or tensor product backend. Comparison with third-party implementations of models, or comparisons substituting additional blocks of the MACE model (such as `SymmetricContraction`) has not been performed at this time, due to the significant effort required to gain solid confidence in the faithfulness of final comparisons, and its orthogonality with the low-level engineering effort behind `e3j`.

Table 3: **Hyperparameters used in end-to-end MLIP benchmarks.**

MACE (a)	
num_layers	2
num_channels	128*
correlation	**2**
node_symmetry	**2**
l_max	3
cutoff_angstrom	5
num_rbf	8
node_gating	true
include_pseudotensors	false

MACE (b)	
num_layers	2
num_channels	128*
correlation	**3**
node_symmetry	**1**
l_max	3
cutoff_angstrom	5
num_rbf	8
node_gating	true
include_pseudotensors	false

NequIP	
num_layers	5
num_channels	32*
node_irreps	2x0e + 2x0o + 1o + 1e + 2e + 2o
l_max	2
cutoff_angstrom	5
num_rbf	8

E TENSOR PRODUCT AND MESSAGE PASSING CONVOLUTION KERNELS

In this appendix, we present algorithmic details of the three sets of kernels developed for `e3j`: CUDA, Pallas GPU, and Pallas TPU. The CUDA kernel offers the broadest applicability across various GPU architectures and is fully deterministic, while Pallas GPU focuses on performance for the latest GPU architectures and JAX versions. We first present the relevant notation and recall the operation that needs to be performed as part of the Clebsch–Gordan Tensor Product and associated message passing, and then present in order the CUDA, Pallas GPU and Pallas TPU kernels. In order to facilitate understanding for readers with different backgrounds, we have added a short section outlining the key hardware concepts for GPU and TPU in appendix F.

E.1 Mathematical Details

In this section we detail the notation used and provide a mathematical description for the operation performed by the kernel. For further details on the motivation for the construction of these operations from a theoretical standpoint, please refer to the original literature.

Notation.

- $\mathbf{x}$: Left-hand-side features. In the Tensor Product, it is an arbitrary feature tensor, the Message Passing operation, it represents the sender node features. It is indexed along three axes: batch elements (usually nodes), equivariant features (see below for indexing notations), and channels
- $\mathbf{y}$: Right-hand-side features. In the Tensor Product, it is an arbitrary feature tensor, in the Message Passing operation, it usually represents (broadcasted) spherical harmonics embeddings. It is indexed along three axes: batch elements (usually edges), equivariant features (see below for indexing notations), and channels.
- $\mathbf{s}$: Edge scalars. Edge specific weighting, usually computed using a MLP on radial embedding projecting to the number of channels, for each edge. It is indexed along edges, equivariant features, and channels.
- $\mathbf{m}_b$: Aggregated messages on receiver node. The output of the message passing convolution for each node. It is worth noting that it does not int principle, have the same feature dimension as $\mathbf{x}$ as multiplicities arise during the tensor product (these are in general contracted back to the number of channels in channel mixing). It is likewise indexed by nodes, equivariant features, and channels.
- C: Clebsch–Gordan coefficients. Its values are indexed by outputs ($i0$), l.h.s input ($i1$) and r.h.s. input ($i2$) feature indexing (each $i0$, $i1$, $i2$ represents a ℓ, m combination) .
- $\mathbf{m}_{ab}$: Edge level message between a sender node and a receiver node.
- a: Sender indexing
- b: Receiver indexing
- q: Channel indexing
- p: sparse Clebsch–Gordan record index (when unrolled)
- N_q: The number of channels
- N_p: The number of non-zero Clebsch–Gordan coefficients
- $c_p = C^{i0_p}_{i1_p, i2_p}$: Coefficient value of record p

Mathematical operations. Following the notation above, we can outline the operations performed by the kernels:

- **Tensor product of geometric features:** This formula generalises to bilinear operation on arbitrary feature vectors, it includes the Clebsch–Gordan tensor products when appropriate weights are selected. Here we treat the sender / receiver indices as implicit as this operation can be performed arbitrarily on any batch element (i.e. nodes or edge features). $\mathbf{z}$ is an arbitrary notation to represents the output feature tensor following a tensor product.

$$\mathbf{z}^{i0,q} = \sum_{i1,i2} C^{i0}_{i1,i2} * \mathbf{x}^{i1,q} * \mathbf{y}^{i2,q} \tag{10}$$

- **Message weighting with edge scalars:** From the computed Tensor Product features on a given edge, one can weight the edge specific message before aggregation. This is usually done through a MLP mapping radial embeddings to a set number of channels. In the notation below, the index $i3$ maps the feature coordinate $i0$ to its irreducible block (piecewise-constant on irreducible subspaces), we write this dependency as $i3(i0)$ for simplicity. The mapping from feature coordinates to scalar indices is performed at coefficient construction time from I/O representations. This operation can be viewed as a Tensor Product with the r.h.s. input being scalars instead of arbitrary geometric features.

$$\mathbf{m}^{i0,q}_{ab} = \mathbf{z}^{i0,q}_{ab} * \mathbf{s}^{i3(i0),q}_{ab} \tag{11}$$

- **Message Passing Convolution:** The complete operation for the message passing convolution, per node, can be written concisely as this common particular version of (7):

$$\mathbf{m}_b = \sum_{a \sim b} \tilde{\mathbf{m}}_{ab} \quad \text{where} \quad \tilde{\mathbf{m}}_{ab} = \mathbf{s}_{ab} \cdot (\mathbf{x}_a \otimes \mathbf{y}_{ab}) \tag{12}$$

given node features $\mathbf{x}_a$, edge features $\mathbf{y}_{ab}$, scalar embeddings $\mathbf{s}_{ab}$ and letting the dot denote the scalar mixing operation. Expanding equations (10) and (11) yields

$$\mathbf{m}_b^{\mathtt{i0,q}} = \sum_{a \sim b} \sum_{i1,i2} C_{i1,i2}^{i0}\, \mathbf{x}_a^{\mathtt{i1,q}}\, \mathbf{y}_{ab}^{\mathtt{i2,q}}\, \mathbf{s}_{ab}^{\mathtt{i3(i0),q}}. \tag{13}$$

The equivalence between (12) and (13), reflecting the associativity of bilinear couplings, leads to different computation graphs and implementations, as depicted in figure 12.

Differentiation. All of our kernels and associated primitives are infinitely differentiable. We outline below the mathematical derivation of their so-called reverse-mode AD primitives, or vector-jacobian products (VJP) rules, obtained by first differentiating the smooth function above a set of inputs, called *primals*, before transposing the linearized map. The transposed differential acts on *cotangents* (linear forms on tangent vectors) in the reversed direction, i.e. it maps *output cotangents* to *primal cotangents* so as to enable back-propagation of gradients in a complete neural network architecture.

- **Tensor product backward:** By bilinearity of the tensor product, the Leibniz rule gives the forward-mode AD rule for $\mathbf{z} = \mathbf{x} \otimes \mathbf{y} \in X \otimes Y$ as:

$$\delta\mathbf{z} = \delta\mathbf{x} \otimes \mathbf{y} + \mathbf{x} \otimes \delta\mathbf{y} \tag{14}$$

where $\delta\mathbf{x}, \delta\mathbf{y}$ denote *tangent vectors* of $T_{\mathbf{x}}X$ and $T_{\mathbf{y}}Y$ respectively, mapped to an output tangent $\delta\mathbf{z} \in T_{\mathbf{z}}(X \otimes Y)$ by the linearized tensor product operation. Equation (14) thus defines a *linear map* $L_{\mathbf{x},\mathbf{y}} : T_{\mathbf{x}}X \oplus T_{\mathbf{y}}Y \to T_{\mathbf{z}}(X \otimes Y)$ such that $\delta\mathbf{z} = L_{\mathbf{x},\mathbf{y}}(\delta\mathbf{x}, \delta\mathbf{y})$.

The backward tensor product primitive is the adjoint (a.k.a. transpose)[6] of $L_{\mathbf{x},\mathbf{y}}$, which maps an output cotangent $d\mathbf{z} \in T^*_{\mathbf{z}}(X \otimes Y)$ to primal cotangents $d\mathbf{x} \in T^*_{\mathbf{x}}X$ and $d\mathbf{y} \in T^*_{\mathbf{y}}Y$. In practice, this means transposing the partially applied tensor product maps $\mathbf{x} \otimes -$ and $- \otimes \mathbf{y}$, respectively consuming $\delta\mathbf{y}$ and $\delta\mathbf{x}$ in (14). The transposition amounts to applying a permutation of indices. For $d\mathbf{x}$, each record is permuted as $(i0, i1, i2) \mapsto (i1, i2, i0)$ and sorted by the new output index $i1$; for $d\mathbf{y}$ we use $(i0, i1, i2) \mapsto (i2, i1, i0)$ and sort by the new output index $i2$. The coefficient values are reordered identically.

Writing $\mathbf{z} = \mathtt{tensor_product}(\mathtt{C}^{\mathtt{i0}}_{\mathtt{i1i2}}, \mathbf{x}, \mathbf{y})$ in reference to algorithm 1, the source cotangents can be evaluated with the same sparse kernel after permuting the coefficient records:

$$\begin{cases} d\mathbf{x} &= \mathtt{tensor_product}(\mathtt{C}^{\mathtt{i1}}_{\mathtt{i2i0}}, \mathbf{y}, d\mathbf{z}) \\ d\mathbf{y} &= \mathtt{tensor_product}(\mathtt{C}^{\mathtt{i2}}_{\mathtt{i1i0}}, \mathbf{x}, d\mathbf{z}) \end{cases} \tag{15}$$

The operation is also illustrated in Fig. 11.

Although this simple backward algorithm provides infinite differentiability, it has the disadvantage of loading output cotangents $\delta\mathbf{z}$ twice in on-device memory, which can be significantly expensive as the output feature dimensions scales as the product of input feature dimensions.

- **Message-passing backward:** By duality, gather and scatter-add operations are swapped in reverse-mode differentiation and the backward message-passing rule can be expressed as a message-passing operation on the transposed graph, with multiple bilinear couplings being performed. Letting $\mathbf{z}_{ab} = \mathbf{x}_a \otimes \mathbf{y}_{ab}$ in (12), input cotangents may be computed from receiver cotangents $d\mathbf{m}_b$ as:

$$\mathtt{conv_bwd}(\mathbf{x}, \mathbf{y}, \mathbf{s}, \mathbf{dm}) = \begin{cases} d\mathbf{x}_a &= \sum_{b \sim a} \mathbf{s}_{ab} \cdot (\mathbf{y}_{ab} \otimes d\mathbf{m}_b) \\ d\mathbf{y}_{ab} &= \mathbf{s}_{ab} \cdot (\mathbf{x}_a \otimes d\mathbf{m}_b) \\ d\mathbf{s}_{ab}^{(l)} &= \sum_{m=-l}^{+l} \mathbf{z}_{ab}^{(l,m)} \cdot d\mathbf{m}_b^{(l,m)} \end{cases} \tag{16}$$

[6] Formally, the adjoint $L^*_{\mathbf{x},\mathbf{y}}$ is the composition or *pull-back* $d\mathbf{z} \circ L_{\mathbf{x},\mathbf{y}}$. Composing the linear form $d\mathbf{z}$ with the linear map $L_{\mathbf{x},\mathbf{y}}$ defines a linear form on $T_{\mathbf{x}}X \oplus T_{\mathbf{y}}Y$, or two linear forms on $T_{\mathbf{x}}X$ and $T_{\mathbf{y}}Y$, a.k.a. *cotangents*.

Because each cotangent is computed by an ad-hoc operation, (16) may hardly reuse device code from the forward pass yet specialized backward device code can prove more efficient[7]. Instead of (16), if one views the message-passing as a trilinear coupling – see figure 12 and equation (12) – the VJP can be expressed as three distinct trilinear coupling calls to a single `bigotimes` routine, via a straightforward generalization of (15) to three operands.

- **Message-passing double backward:** Deriving the second order rule from (16) is best viewed through the diagram 12 back-propagated twice, each differentiation of `bigotimes` yielding three transposed `bigotimes` calls writing to each of the input leaves. Denoting by $\delta d\mathbf{x}, \delta d\mathbf{y}, \delta d\mathbf{s}$ the second-order variations of the primal inputs (first-order variations of cotangents returned by the backward pass), the associated variations $\delta\mathbf{x}, \delta\mathbf{y}, \delta\mathbf{s}, \delta d\mathbf{m}$ returned by the backward rule are defined in terms of lower-order primitives as:

$$\delta d\mathbf{m} = \texttt{conv}(\delta d\mathbf{x}, \mathbf{y}, \mathbf{s}) + \texttt{conv}(\mathbf{x}, \delta d\mathbf{y}, \mathbf{s}) + \texttt{conv}(\mathbf{x}, \mathbf{y}, \delta d\mathbf{s}) \tag{17}$$

$$\begin{cases} \delta\mathbf{x} = \delta_y\mathbf{x} + \delta_s\mathbf{x} \\ \delta\mathbf{y} = \delta_x\mathbf{y} + \delta_s\mathbf{y} \\ \delta\mathbf{s} = \delta_x\mathbf{s} + \delta_y\mathbf{s} \end{cases} \quad \text{where} \quad \begin{cases} -, \delta_x\mathbf{y}, \delta_x\mathbf{s} &= \texttt{conv_bwd}(\delta d\mathbf{x}, \mathbf{y}, \mathbf{s}, d\mathbf{m}) \\ \delta_y\mathbf{x}, -, \delta_y\mathbf{s} &= \texttt{conv_bwd}(\mathbf{x}, \delta d\mathbf{y}, \mathbf{s}, d\mathbf{m}) \\ \delta_s\mathbf{x}, \delta_s\mathbf{y}, - &= \texttt{conv_bwd}(\mathbf{x}, \mathbf{y}, \delta d\mathbf{s}, d\mathbf{m}) \end{cases} . \tag{18}$$

Note that defining AD rules via cyclic references to lower-order differentials, as (15) and (17-18) do, automatically yields infinitely differentiable JAX primitives. At this time `e3j` doesn't define double-backward kernels, in contrast with `OpenEquivariance` (Bharadwaj et al., 2025). Double-backward kernels could save unused cotangents from being computed in (18), or save memory traffic by streaming through the primals in a single loop. Due to the increased number of I/O arrays, a GPU implementation would likely rely on implicit L1/L2 caching over shared memory buffering, yet these constraints do not apply on TPU whose VMEM slice is much larger. Double-backward optimization is left for future work.

E.2 CUDA KERNELS

A significant design choice of `e3j` is to rely on a sparse and agnostic representation of total Clebsch-Gordan coefficients, in which equivariant coordinates (traditionally a set of triplets (κ, ℓ, m) for the output and each of the two inputs, hence a total of 9 integers) are mapped to unfolded feature coordinates $i0, i1, i2$, for output, l.h.s input and r.h.s input, running over the whole I/O dimensions. Melding the equivariant coordinates together into opaque indices means we can rely on a very generic sparse tensor product algorithm, suitable for different applications.

The number of irreducible blocks in the sum scales as $O(l_{\max}^3)$ – number of input degree pairs multiplied by the number of possibilities for the output degree – while the number of non-zero coefficients within each block scales as $l_{\max}^2$, due to the spin constraints $M = m + m'$. Since the feature dimension D scales as $l_{\max}^2$, and our algorithm 1 scales as $O(l_{\max}^5) = O(D^{5/2})$ in terms of floating point operations (FLOPs). In practical situations we targeted, $l_{\max}$ is a fixed hyperparameter of the MLIP model and the number of non-zero coefficients is a few hundreds or less.

E.2.1 CUDA TENSOR PRODUCT OPERATION

The main bottleneck of the sparse, JAX-based implementation of the Tensor Product is the final scatter-reduction step, which scaled poorly to large input sizes. Our CUDA algorithm bypasses this bottleneck and implements a number of focused optimization, the key differentiating implementation factors of algorithm 1 are explained below:

- **Trailing channels:** we avoid any inter-thread communication by sequentially reducing the feature axis, and parallelizing across the trailing channel dimension instead. The 32 threads of a warp (simultaneously scheduled to execute the same instruction) can thus process distinct channels in parallel, while doing the same work: i.e. warp lanes process distinct channels while traversing the same ordered sequence of CG coefficient records.

[7] Our CUDA kernels prioritize code reuse while our Pallas kernels use specialized code in backward operations, see below.

- **Vectorization:** processing 2 or 4 channels simultaneously inside the inner coefficient loop, so as to amortize by the same factor the cost of coefficient loads. Ablation studies reveal using wider `float2` to `float4` datatypes in the coefficient loop nearly double the throughput compared to the non-vectorized kernel, as can be expected from the shared memory traffic per operand coupling: $(2 \times 4 + 16) = 24$ bytes loaded per batch, versus $(4 \times 2 \times 4 + 16)/4 = 12$ bytes loaded per batch with 4-fold vectorization, with 16B coefficients.
- **Coefficient packing:** using narrow index data types so as to halve the size of each coefficient load when dimensions are small enough. The array of coefficients `C` is passed as a pointer to `Coef` structures holding one 32-bit `float` value and three indices of type `Idx`, where:
  - `Idx` = `uint8` if all I/O dimensions are bounded by 255. `Coef` is then a 56 bit type, aligned to 64 bits, which can be loaded in a single `LDS.64` instruction.
  - `Idx` = `int32` by default. `Coef` is then a 128 bit type, which can be loaded in a single `LDS.128` instruction.

  In practice, narrow indices have to be expanded to 32 bits in CUDA registers, and the upside of index narrowing is mostly that of a reducing shared memory pressure.
- **Coefficient distribution and occupancy:** our kernels buffer input rows in shared memory to avoid long-scoreboard stalls when loading operands during the inner loop of algorithm 1. Since SMEM footprint limits the number of resident blocks on each SM, maximizing the number of resident threads requires increasing the block sizes above the given channel count. To that end, the coefficient loop is parallelized along the y-axis of a block, by first grouping coefficients by output indices before splitting them in evenly-sized groups.

Algorithm 1: Tensor product evaluation: parallel trailing channels, sequential aggregation

Data: non-zero COO coefficients $\mathtt{C} = (\mathtt{i0}, \mathtt{i1}, \mathtt{i2}, \mathtt{val}) \in (\mathbb{N}^3 \times \mathbb{R})^{N_c}$ sorted by output index, l.h.s. $\mathbf{x} \in \mathbb{R}^{d \times N_q}$, r.h.s. $\mathbf{y} \in \mathbb{C}^{d' \times N_q}$, thread index $t \in \mathbb{N}$
Result: output $\mathbf{z} = \mathbf{x} \otimes \mathbf{y} \in \mathbb{R}^D$

```
i0 ← 0 ;
z_i0 ← 0;
for p = 0 ... N_c − 1 do
    coef ← LOAD C[p] ;
    i1, i2 ← coef.i1, coef.i2 ;
    x_i1 ← LOAD x[i1, t] ;
    y_i2 ← LOAD y[i2, t] ;
    if coef.i0 == i0 then
        z_i0 += coef.val * x_i1 * y_i2 ;
    end
    else
        z[i0, t] ← STORE z_i0 ;
        z_i0 ← coef.val * x_i1 * y_i2 ;
        i0 ← coef.i0
    end
end
```

Backward pass. Back-propagating through the `tensor_product` primitive can be implemented as two additional calls to the same `tensor_product` kernel: hence yielding $3n$ kernel calls for a model with n tensor products. This is made possible by the fact that algorithm 1 is generic with respect to the sparse coefficient array, see equation (15).

Although this simple backward algorithm provides infinite differentiability, it has the disadvantage of loading output cotangents $\delta\mathbf{z}$ twice in on-device memory, which can be significantly expensive as the output feature dimensions scales as the product of input feature dimensions. Our backward tensor product kernel streams through batches of $(\mathbf{x}, \mathbf{y}, d\mathbf{z})$ once and computes the two cotangents as (15), by reusing the same device code for the bilinear coupling (algorithm 1).

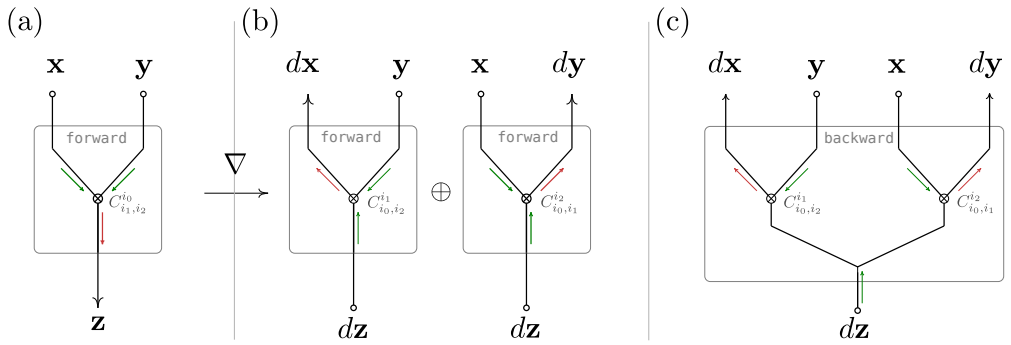

Figure 11: **Computation graphs for the tensor product forward pass (a) and backward pass (b-c).** Back-propagating through a tensor product operation yields two transposed tensor product operations, by the Leibniz rule. Instead of calling two tensor product kernels, a dedicated backward kernel reuses device code for batch-wise bilinear coupling, while streaming through batches of the $d\mathbf{z}$ cotangents only once and thus saving memory traffic.

E.2.2 CUDA Message Passing Convolution

It is worth noting that our CUDA message passing kernel is not constrained to edge based spherical harmonic embedding as second operand. Many Euclid equivariant MLIP architectures use this message-passing update of node features, such as MACE (Batatia et al., 2023) and NequIP (Batzner et al., 2022), and message formation is a known bottleneck of most MLIP architectures.[8] For practical improvements, it may however be more important to consider hardware behavior and constraints at typical values of $l_{\max}$ rather than complexity arguments on hyperparameters.

Even with an ideal memory-bound tensor product kernel, the message-passing operation may imply materializing edge features $\tilde{\mathbf{m}}_{ab} = \mathbf{x}_a \otimes \mathbf{Y}_l(\mathbf{r}_{ab})$ in global memory. Their leading axis, the number of edges, is more than one order of magnitude larger than the number of atoms N (average number of neighbors of around 45 in organic molecules at 5Å cutoff).

Our CUDA convolution kernel streams through edges and computes messages via a straightforward overload of algorithm 1 to three operands given sparse 4D coefficients C^{i0}_{i1i2i3}, which we refer to as the `bigotimes` device routine. See equation (13) and figure 12.

The trilinear mixing is embedded within an outer loop over receiver nodes, and an inner loop over sender nodes (neighbors) which relies on a Compressed Sparse Row (CSR) representation of the adjacency matrix. During each neighbor loop, messages $\tilde{\mathbf{m}}_{ab}$ are accumulated in a shared memory buffer. While this accumulation strategy and CSR adjacency format enables efficient and deterministic message aggregation (free of compare-and-swap operations, a.k.a. *atomics*), the additional operand and message buffers increase the shared memory pressure compared to the raw tensor product kernel.

Backward pass. The current backward kernel processes each of the trilinear mixing operations of Eq. (16) with a common `bigotimes()` routine (see Fig. 12). While postponing the scalar mixing step could save a few FMUL/FMA operations in the forward pass, treating scalar mixing as a distinct operation in the backward pass leads to more complex device code and accumulation patterns (inner product reduction on channels of $d\mathbf{y}_{ab}$) and increased register pressure. While backward convolution still shows a good margin for optimization, relative algorithmic simplicity remains a constraint for high enough occupancy on the device.

[8] MACE can be a notable exception due to its expensive multi-body atomic cluster expansion. However, with correlation 3 and maximal degree 1 for output node features, about 80% of runtime can be spent on the message-passing operation.

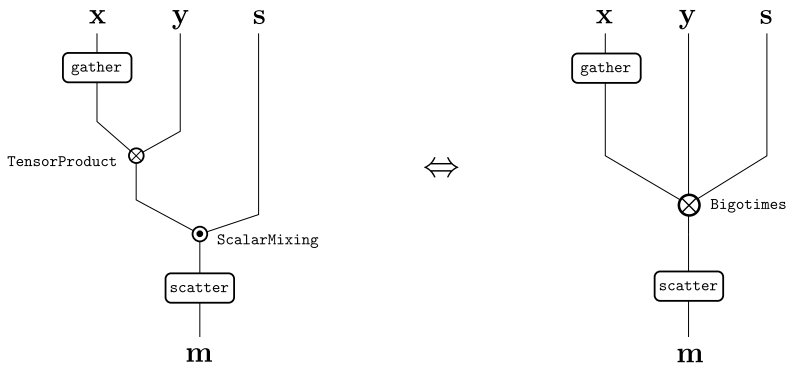

Figure 12: **Equivalent computation graphs for the convolution operation:** the edge-wise composition of a scalar mixing on top of a bilinear tensor product (left) is equivalent to an edge-wise trilinear coupling (right) by associativity.

E.3 PALLAS GPU CONVOLUTION KERNEL

The Pallas GPU kernel implements the same message-passing operation as Eq. 12, but differs from the CUDA implementation in how the sparse Clebsch–Gordan contraction and edge traversal are scheduled. Its main distinguishing feature is trace-time specialization of the Clebsch–Gordan coefficients. Edges are additionally traversed in receiver order, which trades receiver-local, atomics-free forward aggregation against locality and cache reuse in computations where atomic accumulation is required. Algorithm 2 summarizes the forward kernel. Note that buffering of $\mathbf{x}_a$, $\mathbf{y}_{ab}$ and $\mathbf{s}_{ab}$ through SMEM is kept implicit for conciseness, and the `LOAD` directive refers to a shared memory load (`LDS`) as in algorithm 1.

Trace-time Clebsch–Gordan specialization. Unlike the AOT-compiled CUDA Tensor Product implementation of Algorithm 1, the Pallas kernel does not loop through an array of coefficients at runtime. The non-zero coefficients and their indices depend only on the selected I/O feature spaces, and are therefore statically available for the Mosaic compiler[9]. Writing the sparse records as

$$(i0_p, i1_p, i2_p, c_p),$$

the loop over p is unrolled at trace time and the corresponding contraction paths are embedded directly in the generated program. There is therefore no `LOAD` of coefficients and their indices at runtime, in contrast with Algorithm 1, and the operand values can be loaded at each iteration without waiting for the coefficient and its indices to become available in registers.

The records are grouped by output index $i0$, and passed as a Python dictionary – which forces inlining by the upstream HLO compiler – mapping $i0$ to a sequence $\mathcal{C}[\texttt{i0}]$ of triplets $(i1_p, i2_p, c_p)$ in Algorithm 2. For a fixed edge, output feature $i0$, and channel q, the contraction

$$\sum_{i1,i2} C^{i0}_{i1,i2}\, \mathbf{x}^{i1,q}_a\, \mathbf{y}^{i2}_{ab}$$

can consequently be accumulated in registers before applying the corresponding edge scalar $\mathbf{s}^{i3(i0),q}_{ab}$. Trace-time specialization removes the coefficient loads and run-time loop and indexing overhead associated with traversing the sparse representation. This leads to having mostly `FMA` and `LDS` instructions that are all visible to the compiler, and can be reordered opportunistically to hide the latency of these instructions.

Receiver-ordered aggregation. Edges are stored in compressed sparse row (CSR) format ordered by receiver b, such that $[\texttt{rowptr}[b], \texttt{rowptr}[b+1])$ contains all edges incident on b. In the forward

[9] Clebsch-Gordan coefficients grouped by indices are actually passed as nested Python dictionaries in the kernel source, which forces inlining by the upstream HLO compiler.

Algorithm 2: Pallas GPU message-passing convolution: receiver-ordered aggregation and trace-time Clebsch–Gordan specialization.

Data: static coefficients grouped by output index $\mathcal{C}[\texttt{i0}] = (\texttt{i1}_p, \texttt{i2}_p, c_p)_{p \in \mathcal{P}_{\texttt{i0}}}$; l.h.s. node features $\mathbf{x} \in \mathbb{R}^{N \times d \times N_q}$; r.h.s. edge features $\mathbf{y} \in \mathbb{R}^{E \times d'}$; edge scalars $\mathbf{s} \in \mathbb{R}^{E \times d'' \times N_q}$; sender indices `sender`; receiver CSR pointers `rowptr`; channel q
Result: aggregated messages $\mathbf{m} \in \mathbb{R}^{N \times D \times N_c}$
foreach *receiver* b **do**
  $e_{\text{begin}}, e_{\text{end}} \leftarrow \texttt{rowptr}[b], \texttt{rowptr}[b+1]$
  $\texttt{acc} \leftarrow 0$
  for $e = e_{\text{begin}}, \ldots, e_{\text{end}} - 1$ **do**
    $a \leftarrow \texttt{sender}[e]$
    // UNROLL
    for $\texttt{i0} = 0, \ldots, D-1$ **do**
      $\texttt{z} \leftarrow 0$
      for $(\texttt{i1}_p, \texttt{i2}_p, c_p) \in \mathcal{C}[\texttt{i0}]$ **do**
        $\texttt{x} \leftarrow \texttt{LOAD}\ \mathbf{x}_a[\texttt{i1}_p, q]$
        $\texttt{y} \leftarrow \texttt{LOAD}\ \mathbf{y}_{ab}[\texttt{i2}_p]$
        $\texttt{z} \mathrel{+}= c_p * \texttt{x} * \texttt{y}$
      end
      $\texttt{s} \leftarrow \texttt{LOAD}\ \mathbf{s}_{ab}[i3(\texttt{i0}), q]$
      $\texttt{acc}[\texttt{i0}, q] \mathrel{+}= \texttt{s} * \texttt{z}$
    end
  end
  $\mathbf{m}_b \leftarrow \texttt{STORE}\ \texttt{acc}$
end
return $\mathbf{m}$

pass, one cooperative thread array (CTA) processes the complete reduction for a receiver. Its accumulator is initialized in registers, updated while traversing the incoming edges, and written to GMEM after the final edge.

This avoids inter-CTA atomic aggregation of $\mathbf{m}_b$. In an edge-parallel implementation, several CTAs may contribute simultaneously to

$$\mathbf{m}_b \mathrel{+}= \tilde{\mathbf{m}}_{ab},$$

requiring atomic read–modify–write operations. Receiver-stationary aggregation instead keeps the partial sum private to one CTA, providing a fixed receiver-local accumulation order and avoiding repeated stores of partial results.

The same property does not hold generally in the backward kernel. Receiver ordering makes consecutive edges reuse the receiver cotangent $d\mathbf{m}_b$, improving temporal locality and potentially L2-cache reuse, but gradient contributions need not remain local to one receiver. The implementation therefore uses atomic accumulation where multiple CTAs contribute to the same $d\mathbf{x}$, $d\mathbf{y}$, or $d\mathbf{s}$ output. Receiver ordering thus reflects a trade-off between atomics-free, deterministic receiver aggregation in the forward pass and improved locality in computations whose output aggregation may require atomics.

Edge traversal. The receiver CSR interval is traversed sequentially, or in small contiguous tiles. Pallas's `emit_pipeline` can stage receiver-ordered edge-local operands from GMEM to SMEM using asynchronous transfers, including TMA, while the current tile is processed. The sender access $\mathbf{x}_a$ remains an indirect gather.

This pipelining is a scheduling optimization rather than a defining feature of the kernel: its benefit depends on the balance between memory traffic and tensor-product computation. The central consequences of receiver ordering are instead the forward aggregation strategy and the locality obtained when repeatedly accessing receiver-associated quantities.

Algorithm 3: Pallas GPU message-passing convolution, backward pass: receiver-ordered traversal and trace-time Clebsch-Gordan specialization.

Data: static coefficients grouped by output index $\mathcal{C}[\texttt{i0}] = (\texttt{i1}_p, \texttt{i2}_p, c_p)_{p \in \mathcal{P}_{\texttt{i0}}}$; l.h.s. node features $\mathbf{x} \in \mathbb{R}^{N \times d \times N_q}$; r.h.s. edge features $\mathbf{y} \in \mathbb{R}^{E \times d'}$; edge scalars $\mathbf{s} \in \mathbb{R}^{E \times d'' \times N_q}$; output cotangent $d\mathbf{m} \in \mathbb{R}^{N \times D \times N_q}$; sender indices `sender`; receiver CSR pointers `rowptr`; channel q

Result: input cotangents $d\mathbf{x} \in \mathbb{R}^{N \times d \times N_q}$, $d\mathbf{y} \in \mathbb{R}^{E \times d'}$, $d\mathbf{s} \in \mathbb{R}^{E \times d'' \times N_q}$

$d\mathbf{x} \leftarrow 0$
foreach *receiver* b **do**
  $e_{\text{begin}}, e_{\text{end}} \leftarrow \texttt{rowptr}[b], \texttt{rowptr}[b+1]$
  for $e = e_{\text{begin}}, \ldots, e_{\text{end}} - 1$ **do**
    $a \leftarrow \texttt{sender}[e]$
    $\texttt{dx}, \texttt{dy}, \texttt{ds} \leftarrow 0$
    // UNROLL
    for $\texttt{i0} = 0, \ldots, D-1$ **do**
      $\texttt{s} \leftarrow \texttt{LOAD}\ \mathbf{s}_{ab}[i3(\texttt{i0}), q]$
      $\texttt{dm} \leftarrow \texttt{LOAD}\ d\mathbf{m}_b[\texttt{i0}, q]$
      $\texttt{dz} \leftarrow \texttt{s} * \texttt{dm}, \quad \texttt{z} \leftarrow 0$
      for $(\texttt{i1}_p, \texttt{i2}_p, c_p) \in \mathcal{C}[\texttt{i0}]$ **do**
        $\texttt{x} \leftarrow \texttt{LOAD}\ \mathbf{x}_a[\texttt{i1}_p, q]$
        $\texttt{y} \leftarrow \texttt{LOAD}\ \mathbf{y}_{ab}[\texttt{i2}_p]$
        $\texttt{z} \mathrel{+}= c_p * \texttt{x} * \texttt{y}$
        $\texttt{dx}[\texttt{i1}_p] \mathrel{+}= c_p * \texttt{y} * \texttt{dz}$
        $\texttt{dy}[\texttt{i2}_p] \mathrel{+}= c_p * \texttt{x} * \texttt{dz}$
      end
      $\texttt{ds}[i3(\texttt{i0})] \mathrel{+}= \texttt{dm} * \texttt{z}$
    end
    $d\mathbf{s}_{ab}[:, q] \leftarrow \texttt{STORE}\ \texttt{ds}$
    $d\mathbf{y}_{ab} \leftarrow \texttt{STORE}\ \sum_q \texttt{dy}$ // reduction over channels
    $d\mathbf{x}_a \leftarrow \texttt{ATOMIC_ADD}\ \texttt{dx}$
  end
end
return $d\mathbf{x}, d\mathbf{y}, d\mathbf{s}$

E.4 PALLAS TPU KERNEL

Notation The following notation is specific to the TPU implementation:

- B: number of edges in an edge tile
- t: edge-tile index
- $\mathbf{a}_t, \mathbf{b}_t$: sender and receiver index vectors for tile t
- $\mathbf{y}_t, \mathbf{s}_t$: edge-feature and edge-scalar tiles
- $\hat{\mathbf{x}}_t = \mathbf{x}[\mathbf{a}_t]$: sender features gathered for tile t
- $k = i3(i0)$: scalar-mixing block of output coordinate $i0$, which selects the edge scalar $\mathbf{s}[k]$
- acc, $current$ and $stage$: state used by the segmented receiver or sender reduction.

The following acronyms can also be useful, though for more details we refer readers to Appendix F: HBM denotes the TPU's high-bandwidth memory, VMEM its vector memory, and SMEM its scalar memory.

Tiling over edges rather than receivers. In contrast to the receiver-stationary Pallas GPU kernel, the TPU kernel tiles receiver-ordered edges into blocks of B. For each tile, VMEM holds the edge-local operands and messages, while the TensorCore's SMEM holds the corresponding sender and receiver indices. The kernel copies $\mathbf{y}_t$ and $\mathbf{s}_t$ from HBM to VMEM, loads $\mathbf{a}_t$ and $\mathbf{b}_t$ into SMEM,

gathers the sender features $\hat{\mathbf{x}}_t = \mathbf{x}[\mathbf{a}_t]$ into VMEM, and computes the edge-level weighted messages $\tilde{\mathbf{m}}_{ab}$ of Eq. 12. These messages are then reduced over receivers to form $\mathbf{m}_b = \sum_{a \sim b} \tilde{\mathbf{m}}_{ab}$.

The reduction state is retained across consecutive tiles, so a receiver spanning a tile boundary is accumulated before being written to HBM (Algorithm 4). Like the CUDA and Pallas GPU convolution kernels, this fuses message formation with aggregation and therefore avoids materializing the complete edge-leading message tensor in HBM. The TPU-specific distinction is the edge-tiled schedule: VMEM holds a block of edge messages, while SMEM carries the edge indices used for the gathers and segmented receiver reduction.

Algorithm 4: Fused message-passing convolution on TPU.

$acc \leftarrow 0$, $current \leftarrow -1$
for each tile t of B receiver-ordered edges **do**
  $\mathbf{y}_t, \mathbf{s}_t \leftarrow$ HBM to VMEM, $\mathbf{a}_t, \mathbf{b}_t \leftarrow$ HBM to SMEM
  $\hat{\mathbf{x}}_t \leftarrow$ HBM to VMEM $\mathbf{x}[\mathbf{a}_t[k]]$, $k = 0, \ldots, B-1$
  $\tilde{\mathbf{m}}_t \leftarrow$ MESSAGETILE$(\hat{\mathbf{x}}_t, \mathbf{y}_t, \mathbf{s}_t)$
  REDUCEFLUSH$(\mathbf{b}_t, \tilde{\mathbf{m}}_t, acc, current)$
end
FLUSH$(current)$

Coefficient packing and Clebsch–Gordan contraction. MESSAGETILE evaluates the contraction path by path. The static records $(i0_p, i1_p, i2_p, c_p)$ are grouped on the host first by output coordinate $i0_p$ and then by l.h.s. input coordinate $i1_p$. This allows each $B \times N_q$ tile $\hat{\mathbf{x}}^{i1_p}$ to be loaded once and reused across the corresponding $i2_p$ paths (Algorithm 5).

As in the Pallas GPU kernel, the sparse coefficient loops are unrolled at trace time and c_p is embedded directly in the generated computation. Unlike Algorithm 1 for CUDA, the TPU kernel therefore does not load and interpret coefficient records at run time.

Factoring out the edge scalar. The edge scalar $s_{ab}^{i3(i0),q}$ in Eq. 13 is the same for every path of output coordinate $i0$, so the kernel pulls it out of the path sum, as in Eq. 12:

$$\tilde{m}_{ab}^{i0,q} = s_{ab}^{i3(i0),q} \underbrace{\sum_{i1,i2} C_{i1,i2}^{i0}\, x_a^{i1,q}\, y_{ab}^{i2,q}}_{z_{ab}^{i0,q}}.$$

It accumulates z first, then multiplies by the edge scalar once per output coordinate instead of once per path.

Algorithm 5: MESSAGETILE

for $i0$ output coordinate **do**
  $z \leftarrow 0$
  for $i1$ l.h.s. input coordinate of $i0$ **do**
    for $(i2_p, c_p)$ path of $(i0, i1)$ **do**
      $z \mathrel{+}= c_p\, \hat{\mathbf{x}}[i1]\, \mathbf{y}[i2_p]$ `// contraction before mixing`
    end
  end
  $\tilde{\mathbf{m}}[i0] \leftarrow z\, \mathbf{s}[i3(i0)]$ `// scalar mixing`
end

Reducing over receivers. REDUCEFLUSH performs a segmented sum of the edge-message tile $\tilde{\mathbf{m}}_t$, of shape (D, B, N_q). Because the edges are ordered by receiver, equal receiver indices form contiguous segments. The routine walks the B edges eight at a time, accumulates messages belonging to $current$, and flushes the completed receiver sum to $\mathbf{m}_{current}$ when the receiver changes (Algorithm 6). The variables acc and $current$ persist across tiles.

Algorithm 6: REDUCEFLUSH

Function `ReduceFlush(`$\mathbf{b}_t$, $\tilde{\mathbf{m}}$, *acc, current*`)`**:**
  for each chunk of 8 edges, receivers $b[0,\dots,7]$ **do**
    $start \leftarrow 0$
    `// entered when the chunk crosses a receiver boundary`
    if $b[0] \neq current$ **or** $b[0] \neq b[7]$ **then**
      for $j = 0,\dots,7$ with $b[j] \neq b[j-1]$, $b[-1] \equiv current$ **do**
        $acc \mathrel{+}= \tilde{\mathbf{m}}[\, start \le \text{edge} < j\,]$ `// complete current receiver`
        if $current \ge 0$ **then** `Flush(`*current*`)`
        $current \leftarrow b[j]$, $start \leftarrow j$
      end
    end
    $acc \mathrel{+}= \tilde{\mathbf{m}}[\,\text{edge} \ge start\,]$
  end

Function `Flush(`b`)`**:**
  $stage \leftarrow \sum_{\text{sublanes}} acc$, $acc \leftarrow 0$
  VMEM to HBM $stage \rightarrow \mathbf{m}_b$

Backward. The backward follows the same edge-tiled organization as Algorithm 4. In addition to the edge-local $\mathbf{y}_t$ and $\mathbf{s}_t$, it gathers $\mathbf{x}_a$ and the receiver cotangent $d\mathbf{m}_b$ for each edge, replaces MESSAGETILE by the sweep in Algorithm 7, and reduces the contributions to $d\mathbf{x}_a$ using REDUCEFLUSH keyed on senders. This segmented reduction therefore uses a sender-ordered edge traversal. The cotangents $d\mathbf{y}_{ab}$ and $d\mathbf{s}_{ab}$ remain edge-local.

The sweep makes one pass over the same static coefficient records to compute the three cotangents of Eq. 16. Since $\tilde{m}_{ab}^{i0,q} = s_{ab}^{k,q}\, z_{ab}^{i0,q}$ with $k = i3(i0)$, the edge scalar and the contraction also separate in the backward:

$$w_{ab}^{i0,q} = s_{ab}^{k,q}\, dm_b^{i0,q}, \qquad ds_{ab}^{k,q} = \sum_{i0\,:\,i3(i0)=k} z_{ab}^{i0,q}\, dm_b^{i0,q}.$$

The sweep computes w once per output coordinate and uses it in both the $d\mathbf{x}$ and $d\mathbf{y}$ updates. The edge-scalar cotangent instead needs z, which the sweep recomputes from the same paths. Because the records are grouped by block k, each $d\mathbf{s}[k]$ is accumulated in registers and written once. The r.h.s. $\mathbf{y}_{ab}$ is shared across channels, so its cotangent is summed over q at the end.

Algorithm 7: Backward sweep of one edge tile.

$d\hat{\mathbf{x}}, d\mathbf{y} \leftarrow 0$
for k scalar-mixing block **do**
  $\sigma \leftarrow 0$
  for $i0$ output coordinate of block k **do**
    $w \leftarrow \mathbf{s}[k]\, d\mathbf{m}[i0]$ `// mix the cotangent once`
    $z \leftarrow 0$
    for $(i1_p, i2_p, c_p)$ path of $i0$ **do**
      $z \mathrel{+}= c_p\, \mathbf{y}[i2_p]\, \hat{\mathbf{x}}[i1_p]$ `// as in the forward`
      $d\hat{\mathbf{x}}[i1_p] \mathrel{+}= c_p\, \mathbf{y}[i2_p]\, w$
      $d\mathbf{y}[i2_p] \mathrel{+}= c_p\, \hat{\mathbf{x}}[i1_p]\, w$
    end
    $\sigma \mathrel{+}= z\, d\mathbf{m}[i0]$
  end
  $d\mathbf{s}[k] \leftarrow \sigma$ `// one write per block`
end
$d\mathbf{y} \leftarrow \sum_q d\mathbf{y}$

VMEM budget. Each buffer has at most three axes: the tile's B edges, the N_q channels, and the feature components of a node, edge, or message value, whose count grows with $l_{\max}$. The

forward holds the gathered node features and message tile, one edge-scalar array per irreducible-representation block, and the segmented sum accumulator together with the row used for flushing; sender and receiver indices are held in the TPU TensorCore's SMEM. The backward additionally holds the three cotangents and two copies of $\mathbf{x}$ and $d\mathbf{m}$ for pipelining.

Together these buffers occupy 4.3 MiB in the forward and 5.4 MiB in the backward at $l_{\max} = 3$, $N_q = 128$, and fp32, and roughly half as much at $l_{\max} = 2$ for the same block sizes.

In addition to these buffers, the unrolled contraction of Algorithm 5 uses VMEM for compiler spills from vector registers, taking the peak to 13.1 and 10.1 MiB of the TPU TensorCore's 16 MiB budget (Table 4).

Table 4: VMEM usage on a TPU v4 TensorCore of the fused TPU kernel at $l_{\max} = 3$, $C = 128$, fp32, in MiB against the 16 MiB per-core budget; *buffers* counts the kernel's own arrays, *peak VMEM* adds the compiler's spill.

pass	B	buffers	peak VMEM
fwd	128	4.3	13.1 (82%)
bwd	32	5.4	10.1 (63%)

F Relevant GPU and TPU architectural concepts

In order to provide a more comprehensive description of our kernels, we begin with an overview of the relevant hardware concepts for both GPU and TPU, as well as their differences. This section is not intended as an exhaustive or even fully accurate description of these respective devices, but instead as a structured glossary of concepts useful to understand later sections.

F.1 GPU and TPU accelerators comparison

GPUs are massively parallel architectures exposing a single-instruction, multiple-threads (SIMT) programming model that gives fine-grained control over every level of parallelism via the CUDA C++ language extension (Corporation, 2025). A typical server-grade GPU embeds about a hundred streaming multi-processors (SMs) on the same die, each SM being capable of scheduling 2048 threads and carrying a low-latency 256 KB memory bank (L1 cache) alongside a 256 KB register file.

In contrast, TPUs rely on the Mosaic compiler to lower abstract array operations onto dedicated processing units, including a systolic matrix multiply unit (MXU) and single-instruction, multiple-data (SIMD) vector processing unit (VPU) (Jouppi et al., 2017). A TPU tray typically consists of 4 TPU chips, each containing one or two cores, each backed by large (tens of MB) dedicated memory banks.

All these architectural differences translate into significant algorithmic differences between the optimized kernels on each platform, on which more details can be found in the sections below. In particular, since end-to-end performance can depend strongly on reducing memory traffic and avoiding the materialization of intermediate results through operation fusion, different hardware architectures may opt for different fusion strategies and very different sizes for temporary buffers.

The algorithmic development of `e3j`, as outlined in appendix E reflects these differentiations.

Furthermore, significant engineering effort has been recently made to optimize or diversify the compiler stack. Just-in-time (JIT) compilation, from either CUDA or domain-specific languages embedded in Python such as Pallas, allows the production of specialized device assembly code from static problem parameters. It is now an ubiquitous alternative to traditional ahead-of-time (AOT) compilation of CUDA source into SASS or PTX. Further details on these distinctions can be found in appendix D.3.

F.2 Overview of relevant GPU concepts

Computational units. A GPU exposes a hierarchical execution model that maps a large number of software threads onto parallel hardware resources. It is useful to distinguish the programming hierarchy: threads, warps and cooperative thread arrays (CTAs), from the underlying hardware hierarchy of streaming multiprocessors (SMs) and their execution units.

- At the hardware level, the GPU is composed of many **streaming multiprocessors** (SMs). Each SM contains several execution pipelines for floating-point and integer arithmetic, load/store operations, specialized functions, and matrix operations such as those implemented by Tensor Cores.
- At the software level, a **thread** is one logical execution instance of the kernel program. Each thread maintains its own execution state, including registers and, on modern NVIDIA GPUs, its own program counter.
- Threads are organized in CTA, which are partitioned into groups of 32 threads called **warps**. An SM schedules and issues instructions at warp granularity: typically, one instruction is issued to the active threads of a warp, which apply that instruction to their respective data. This execution model is referred to as **Single Instruction, Multiple Threads** (SIMT). Threads may follow different control-flow paths, although divergence within a warp generally reduces execution efficiency.
- A **cooperative thread array** (CTA), also called a CUDA thread block, groups threads that cooperate on the same unit of work. All threads of a CTA are scheduled on the same SM, where they may synchronise and exchange data through shared memory. The SM partitions the CTA into warps and interleaves the execution of its ready warps. Multiple CTAs may reside concurrently on one SM when register and shared memory capacity permit.

Memory hierarchy. GPU performance is strongly influenced by where data reside in the memory hierarchy. Storage closer to the execution units provides high bandwidth and low latency but limited capacity, motivating kernels to move data through successively smaller on-chip memories and maximize reuse before returning to global memory.

- **Registers** provide the storage closest to arithmetic execution. Registers hold thread-local values such as operands, intermediate results and accumulators, and are allocated from an SM-wide register file among the resident threads. On H100, each SM provides $65\,536$ 32-bit registers, corresponding to 256 KiB of register-file capacity. Their limited capacity makes register usage an important constraint: using more registers per thread or CTA can reduce the number of CTAs that can reside concurrently on an SM.
- **Shared memory (SMEM)** is an explicitly managed on-chip scratchpad shared by the threads of a CTA. It enables fast data reuse and communication between cooperating threads. The H100 provides a combined 256 KiB L1/texture-cache and shared-memory pool per SM, of which up to 228 KiB can be configured as shared memory. Although L1 cache and SMEM use the same underlying capacity, they have different programming semantics: L1 is hardware-managed cache, whereas placement and access in SMEM are controlled explicitly by the kernel.
- The **L2 cache** is a larger cache shared across the GPU and forms the last on-chip caching level before device memory. H100 GPUs provide a 50 MB L2 cache, allowing frequently accessed data to be retained on-chip and reducing repeated accesses to the substantially larger off-chip memory.
- **Global memory (GMEM)** is the GPU-wide memory address space used to store the large tensors processed by a kernel. Its backing storage is off-chip high-bandwidth DRAM (HBM3 on the H100 SXM5, for example) and consequently has much greater capacity but higher access cost than registers, SMEM or the on-chip caches. Efficient kernels therefore aim to minimize global-memory traffic and to reuse data after bringing it on-chip.
- On Hopper GPUs, the **Tensor Memory Accelerator (TMA)** provides a specialized mechanism for asynchronously transferring multidimensional blocks of data between GMEM and SMEM. Once a transfer has been initiated, computation may proceed independently while TMA moves the next block of data. Software pipelines can consequently overlap

global-memory traffic with arithmetic, as exploited by the Mosaic GPU kernel described below.

Performance limits. For many ML kernels, particularly those with low arithmetic intensity, the rate at which data can be transferred between HBM and the GPU is the dominant performance bottleneck. In this memory-bound regime, the peak HBM bandwidth provides a useful "speed-of-light" estimate: the minimum execution time is bounded by the number of bytes that must be transferred to and from HBM divided by the maximum sustainable HBM bandwidth. Kernel efficiency can therefore be assessed by comparing its achieved HBM bandwidth with the hardware peak. Kernels with sufficiently high arithmetic intensity may instead become compute-bound, in which case peak arithmetic throughput rather than HBM bandwidth determines the relevant performance ceiling.

F.3 Overview of relevant TPU concepts

Computational units. A TPU exposes a different execution model from the SIMT hierarchy of a GPU. Rather than scheduling many independent software threads, a TPU TensorCore is programmed approximately as a sequential machine operating on wide vector tiles.

- A TPU v6e chip contains one **TensorCore**, comprising a **scalar unit**, a **vector processing unit** (VPU), and two **matrix-multiply units** (MXUs). The scalar unit handles control, indexing and scalar arithmetic, the VPU performs general vector operations, and the MXUs provide high-throughput matrix multiplication.
- The natural unit of vector computation is a two-dimensional **vector-register tile**. For 32-bit values on TPU v6e, registers are organized as 8×128 elements (sublanes $\times$ lanes). Vector instructions operate on these tiles collectively rather than on independently scheduled threads.
- Array layout is therefore closely tied to this native tile shape. Operations are most efficient when trailing dimensions map cleanly onto the 8×128 register layout; poorly aligned or very small arrays may waste vector capacity or require additional rearrangement.
- The two **MXUs** are specialized systolic units for dense matrix multiplication, while the VPU handles the more general elementwise, reduction and permutation operations used throughout Pallas kernels.

Memory hierarchy. Large tensors reside in off-chip HBM and are staged through software-managed on-chip memories before computation.

- **Vector registers (VREGs)** hold operands and intermediate array-valued results consumed by the VPU and MXUs. Their limited capacity makes register pressure and spilling important considerations.
- **Vector memory (VMEM)** is a large software-managed on-chip scratchpad for array data. TPU v6e provides approximately 128 MiB of VMEM per TensorCore, allowing substantially larger working sets to remain on-chip than in GPU shared memory.
- **Scalar memory (SMEM)** is a separate on-chip memory for scalar data such as indices and control information. TPU v6e provides approximately 1 MiB of SMEM. This should not be confused with GPU "SMEM", which denotes shared memory.
- **HBM** provides the large off-chip storage for kernel inputs and outputs. TPU v6e provides 32 GB of HBM with approximately 1.64 TB/s peak bandwidth. Data are transferred between HBM and VMEM using asynchronous DMA engines, allowing future tiles to be prefetched while the current tile is being processed.

Performance limits. As on GPUs, low-arithmetic-intensity TPU kernels may be bounded by HBM bandwidth, whereas compute-intensive kernels may instead be limited by VPU or MXU throughput. Efficient TPU kernels therefore aim to retain data in VMEM, respect the native 8×128 vector layout, and overlap HBM–VMEM transfers with computation.

F.3.1 Relevant distinction between GPU and TPU architectures

The GPU and TPU kernels are shaped by fundamentally different execution and memory models. On a GPU, computation is organized around CTAs composed of SIMT warps, with thread-local registers and a relatively small CTA-local shared-memory scratchpad. By contrast, a TPU TensorCore operates on wide 8×128 vector-register tiles and provides a much larger software-managed VMEM working memory. On H100, at most 228 KiB of shared memory is available per SM, whereas TPU v6e provides approximately 128 MiB of VMEM per TensorCore. These differences lead naturally to different kernel organizations: the GPU kernel keeps a receiver reduction local to one CTA and streams edge data through SMEM, whereas the TPU kernel operates on larger VMEM-resident windows using tile-wide vector operations.

G Reconstructions

G.1 Reconstruction of harmonic polynomials.

The basis $(Y_l^m)_{-l\leq m\leq l} \in \mathbb{Y}_l$ of spherical harmonic polynomials of order l depends on the basis (x, y, z) of $\mathbb{R}^3$ by the spin equations demanding that Y_l^m is an eigenvector of z-axis (infinitesimal) rotations,

$$\sigma_z \cdot Y_l^m = imY_l^m \tag{19}$$

Noting that σ_z can be written as the infinitesimal rotation $r\partial_\phi$ of the longitude angle ϕ, equation (19) implies that Y_l^m varies as $\mathrm{e}^{im\phi}$ along the longitude angle ϕ. Up to a choice of normalization factor c_l, for $m = \pm l$, this implies that $Y_l^{\pm l}$ is the fastest oscillating degree l-monomial with respect to ϕ:

$$Y_l^{\pm l}(x, y, z) = c_l\,(x + iy)^l = c_l\, r^l \cos(\theta)^l\, \mathrm{e}^{\pm il\phi}. \tag{20}$$

In general, polynomials of lower absolute spin Y_l^m with $|m| < l$ similarly take the form

$$Y_l^m(x, y, z) = r^l P_{lm}(\cos\theta, \sin\theta)\, \mathrm{e}^{im\phi}, \tag{21}$$

and is an eigenvector of σ_z with eigenvalue im (in particular, Y_l^0 is always invariant with respect to z-axis rotations). They can be obtained from $Y_l^{\pm l}$ by iterating the ladder operators $\sigma_\pm = \sigma_x \pm i\sigma_y$, which increase / decrease the magnetic quantum number m by 1.

Note that our choice of generators is in agreement with physicists and chemists' quantum state notation $|lm\rangle$, while real harmonics Y_{lm} are not eigenvalues of σ_z : only $|m|$ is determined.

In practice, the computation of polynomials Y_l^m is cached and staged out of XLA compilation. It returns an integer-valued array of exponents `Y.exp` :: $[M, 3]$, and a sparse coefficient matrix `Y.coef` :: $[N, M]$ where N is the number of polynomials in the current basis, and M the number of distinct monomials required for their evaluation.

G.2 Reconstruction of Clebsch Gordan coefficients

The irreducible CG array $C^{LM}_{lm,l'm'}$ can be reconstructed from $C^{LL}_{ll,l'l'}$, expressing the top-spin eigenvectors with respect to the pure generators. The total z-spin $M = m + m'$ can indeed be lowered by the ladder operator σ_-, acting as $\sigma_- \otimes 1 + 1 \otimes \sigma_-$ on the tensor product space by the Leibniz rule, and any eigenvector $|LM\rangle$ for $M < L$ can be obtained up to a normalization factor by iterating σ_- from $|LL\rangle$.

Therefore one only need to solve the eigenvalue equations $\vec{\sigma}^2|LL\rangle = L(L+1)|LL\rangle$ and $\sigma_z|LL\rangle = L|LL\rangle$ to obtain the top-spin generators of each irreducible subspace of $\mathbb{Y}_l \otimes \mathbb{Y}_{l'}$, for each value of $L \in \{|l - l'|, \ldots, l + l'\}$. The eigenvalue equations can be expressed and easily solved as a triangular system, with the algorithm documented below.

By the second order Leibniz rule, the operator J_+J_- acts on a dyadic tensor product as:

$$J_+J_- = (J_+J_- \otimes 1) + (1 \otimes J_+J_-) + (J_+ \otimes J_-) + (J_- \otimes J_+) \tag{22}$$

Since $J^2 = J_+J_- + J_z + J_z^2$, the action of squared angular momentum on a pure state $|lm, l'm'\rangle$ is given by

$$J^2|lm,l'm'\rangle = \{l(l+1)+l'(l'+1)+2mm'\}\cdot|lm,l'm'\rangle + c_-(m)c'_+(m')\cdot|l(m-1),l'(m'+1)\rangle + c_+(m)c'_-(m')\cdot|l(m+1),l'(m'-1)\rangle, \tag{23}$$

where the term in curly brackets contains the independent action of J^2 on each operand, and twice the action of $J_z \otimes J_z$.

Assuming $V = \sum V_{m,m'}|lm,l'm'\rangle$ is an eigenvector $|LL\rangle$ for some L, by additivity of the z-spin $M = L$ we can write more succinctly:

$$V = \sum_m V_m \cdot |lm,l'(L-m)\rangle = \sum_m V_m \cdot |m,(L-m)\rangle \tag{24}$$

so that the eigenvalue equation reads as follows:

$$L(L+1)\cdot V_m = \{l(l+1)+l'(l'+1)+2mm'\}\cdot V_m + c-(m+1)c'+(m'-1)\cdot V_{m+1} + c+(m-1)c'-(m'+1)\cdot V_{m-1}. \tag{25}$$

Given m is constrained to $[-L,L]$, coordinates V_m of the max-spin eigenvector $|LL\rangle$ can easily be constructed by solving the above triangular system in dense or sparse format.