

An Efficient Sparse Kernel Generator for $O(3)$ -Equivariant Deep Networks

Vivek Bharadwaj^{*†‡}Austin Glover^{*†}Aydın Buluç^{‡†}James Demmel[†]

May 12, 2025

Abstract

Rotation equivariant graph neural networks, i.e. networks designed to guarantee certain geometric relations between their inputs and outputs, yield state of the art performance on spatial deep learning tasks. They exhibit high data efficiency during training and significantly reduced inference time for interatomic potential calculations compared to classical approaches. Key to these models is the Clebsch-Gordon (CG) tensor product, a kernel that contracts two dense feature vectors with a highly-structured sparse tensor to produce a dense output vector. The operation, which may be repeated millions of times for typical equivariant models, is a costly and inefficient bottleneck. We introduce a GPU sparse kernel generator for the CG tensor product that provides significant speedups over the best existing open and closed-source implementations. Our implementation achieves high performance by carefully managing the limited GPU shared memory through static analysis at model compile-time, minimizing reads and writes to global memory. We break the tensor product into a series of smaller kernels with operands that fit entirely into registers, enabling us to emit long arithmetic instruction streams that maximize instruction-level parallelism. By fusing the CG tensor product with a subsequent graph convolution, we reduce both intermediate storage and global memory traffic over naïve approaches that duplicate input data. We also provide optimized kernels for the gradient of the CG tensor product and a novel identity for the higher partial derivatives required to predict interatomic forces. Our kernels offer up to 1.3x speedup for the over NVIDIA’s closed-source cuEquivariance package, as well as $> 10x$ speedup over the widely-used e3nn package. In FP64 precision, we offer up to 6.2x inference-time speedup for the MACE chemistry foundation model over the original unoptimized version.

1 Introduction

Equivariant deep neural network models have become immensely popular in computational chemistry over the past seven years [32, 34, 20]. Consider a function $\mathbf{f} : \mathbb{R}^n \rightarrow \mathbb{R}^m$. Informally, \mathbf{f} is *invariant* if a class of transformations applied to its argument results in no change to the function output. A function is *equivariant* if a transformation applied to any input argument of \mathbf{f} can be replaced by a compatible transformation on the output of \mathbf{f} . For example: a function predicting molecular energy based on atomic positions should not change its result if the atom coordinates are rotated, translated,

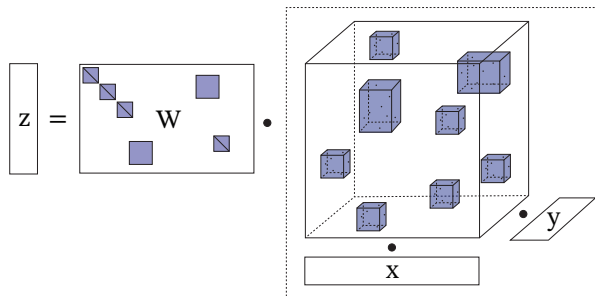


Figure 1: The CG tensor product, which contracts a block-sparse tensor P with two dense vectors to output a new vector. It is usually followed by multiplication with a structured weight matrix W , and by convention we use “CG tensor product” to refer to both operations in sequence. Each blue block is itself sparse (see Figure 2), and several blocks may share identical structure.

or reflected (invariance). Likewise, a function predicting 3D forces on point masses should rotate its predictions if the input coordinate system rotates (equivariance). The latter property is termed *rotation equivariance*, and it is the focus of our work. Rotation equivariant neural architectures appear in the AlphaFold [14] version 2 model for protein structure prediction, the DiffDock [7] generative model for molecular docking, and the Gordon Bell finalist Allegro [26] for supercomputer-scale molecular dynamics simulation, among a host of other examples [29, 1, 4, 3, 18].

A core kernel in many (though not all) rotation equivariant neural networks is the Clebsch-Gordon (CG) tensor product, which combines two feature vectors in an equivariant model to produce a new vector [32]. This multilinear operation, illustrated in Figure 1, contracts a highly-structured block sparse tensor with a pair of dense input vectors, typically followed by multiplication with a structured weight matrix. It is frequently used to combine node and edge embeddings in equivariant graph neural networks, which are used for molecular energy prediction in computational chemistry

^{*}Equal Contribution

[†]University of California, Berkeley

[‡]Lawrence Berkeley National Laboratory

[4, 2, 26] (see Figure 4). With its low arithmetic intensity and irregular computation pattern, the CG tensor product is difficult to implement efficiently in frameworks like PyTorch or JAX. Because the CG tensor product and its derivatives must be evaluated millions of times on large atomic datasets, they remain significant bottlenecks to scaling equivariant neural networks.

We introduce an open source kernel generator for the Clebsch-Gordon tensor product on both NVIDIA and AMD GPUs. Compared to the popular e3nn package [10] that is widely used in equivariant deep learning models, we offer up to one order of magnitude improvement for both forward and backward passes. Our kernels also exhibit up to 1.0-1.3x speedup over NVIDIA’s closed-source cuEquivariance v0.4.0 package [11] on configurations used in graph neural networks, although our second derivative kernel is slower by 30% on certain inputs. Our key innovations include:

Exploiting ILP and Sparsity: Each nonzero block of \mathcal{P} in Figure 1 is a structured sparse tensor (see Figure 2 for illustrations of the nonzero pattern). Popular existing codes [10] fill these blocks with explicit zeros and use optimized dense linear algebra primitives to execute the tensor product, performing unnecessary work in the process. By contrast, we use Just-in-Time (JIT) compilation to generate kernels that only perform work for nonzero tensor entries, achieving significantly higher throughput as block sparsity increases. While previous works [19, 17] use similar fine-grained approaches to optimize kernels for each block in isolation, we achieve high throughput by JIT-compiling a single kernel for the entire sparse tensor \mathcal{P} . By compiling kernels optimized for an entire sequence of nonzero blocks, we exhibit a degree of instruction level parallelism and data reuse that prior approaches do not provide.

Static Analysis and Warp Parallelism: The structure of the sparse tensor in Figure 1 is known completely at model compile-time and contains repeated blocks with identical nonzero patterns. We perform a static analysis on the block structure immediately after the equivariant model architecture is defined to generate a computation schedule that minimizes global memory traffic. We break the calculation into a series of subkernels (see Figure 3), each implemented by aggressively caching \mathbf{x} , \mathbf{y} , and \mathbf{z} in the GPU register file.

We adopt a kernel design where each GPU warp operates on distinct pieces of coalesced data and requires no block-level synchronization (e.g. `__syncthreads()`). To accomplish this, each warp manages a unique portion of the shared memory pool and uses warp-level

matrix multiplication primitives to multiply operands against nonzero blocks of the structured weight matrix.

Fused Graph Convolution: We demonstrate benefits far beyond reduced kernel launch overhead by fusing the CG tensor product with a graph convolution kernel (a very common pattern [32, 4, 2]). We embed the CG tensor product and its backward pass into two algorithms for sparse-dense matrix multiplication: a simple, flexible implementation using atomic operations and a faster deterministic version using a fixup buffer. As a consequence, our work is the first to reap significant model memory savings, a reduction in global memory writes, and data reuse at the L2 cache level.

Section 2 provides a brief introduction to equivariant neural networks, with Section 2.2 providing a precise definition and motivation for the CG tensor product. Section 3 details our strategy to generate efficient CG kernels and the design decisions that yield high performance. We validate those decisions in Section 4 on a range of benchmarks from chemical foundation models and other equivariant architectures.

2 Preliminaries and Problem Description

We denote vectors, matrices, and three-dimensional tensors in bold lowercase characters, bold uppercase characters, and script characters (e.g. \mathcal{P}), respectively. Our notation and description of equivariance follow Thomas et al. [32] and Lim and Nelson [24]. Let G be an abstract group of transformations, and let $\mathbf{D}_{\text{in}} : G \rightarrow \mathbb{R}^{n \times n}$, $\mathbf{D}_{\text{out}} : G \rightarrow \mathbb{R}^{m \times m}$ be a pair of *representations*, group homomorphisms satisfying

$$\mathbf{D}_{\text{in}}(g_1 \cdot g_2) = \mathbf{D}_{\text{in}}(g_1) \cdot \mathbf{D}_{\text{in}}(g_2) \quad \forall g_1, g_2 \in G,$$

and likewise for \mathbf{D}_{out} . A function $\mathbf{f} : \mathbb{R}^n \rightarrow \mathbb{R}^m$ is equivariant with respect to \mathbf{D}_{in} and \mathbf{D}_{out} iff

$$\mathbf{f}(\mathbf{D}_{\text{in}}(g) \cdot \mathbf{v}) = \mathbf{D}_{\text{out}}(g) \cdot \mathbf{f}(\mathbf{v}) \quad \forall \mathbf{v} \in \mathbb{R}^n, g \in G.$$

A function is invariant if the equivariance property holds with $\mathbf{D}_{\text{out}}(g) = \mathbf{I}^{m \times m}$ for all $g \in G$.

In our case, \mathbf{f} is a neural network composed of a sequence of layers, expressed as the function composition

$$\mathbf{f}(\mathbf{v}) = \phi_N \circ \dots \circ \phi_1(\mathbf{v}).$$

Here, \mathbf{D}_{in} and \mathbf{D}_{out} are derived from the dataset, and the task is to fit \mathbf{f} to a set of datapoints while maintaining equivariance to the chosen representations. Network designers accomplish this by imposing equivariance on each layer and exploiting a composition property [32]: if ϕ_i is equivariant to input / output representations

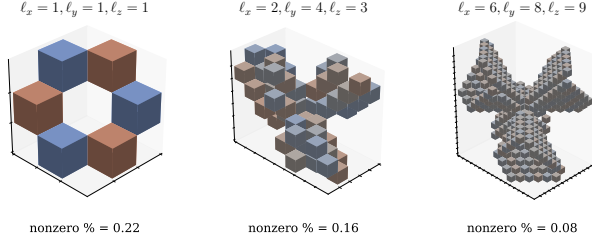


Figure 2: Three examples of coefficient tensors depicted by blue cubes in Figure 1. The fraction of zero entries increases with the tensor order, and the full CG tensor contains several copies of the blocks pictured here.

(D_i, D_{i+1}) and ϕ_{i+1} is equivariant to (D_{i+1}, D_{i+2}) , then $\phi_{i+1} \circ \phi_i$ is equivariant to (D_i, D_{i+2}) . These intermediate representations are selected by the network designer to maximize predictive capability.

2.1 Representations of $O(3)$ In this paper, we let $G = O(3)$, the group of three-dimensional rotations including reflection. A key property of real representations of $O(3)$ is our ability to block-diagonalize them into a canonical form [25]. Formally, for any representation $D : O(3) \rightarrow \mathbb{R}^{n \times n}$ and all $g \in G$, there exists a similarity matrix P and indices i_1, \dots, i_D satisfying

$$D(g) = P^{-1} \begin{bmatrix} D^{(i_1)}(g) & & 0 \\ & \ddots & \\ 0 & & D^{(i_D)}(g) \end{bmatrix} P$$

where $D^{(0)}(g), D^{(1)}(g), \dots$ are a family of elementary, *irreducible* representations known as the Wigner D-matrices. For all $i \geq 0$, we have $D^{(i)}(g) \in \mathbb{R}^{(2i+1) \times (2i+1)}$. In the models we consider, all representations will be exactly block diagonal (i.e. P is the identity matrix), described by strings of the form

$$D(g) \cong "3x1e + 1x2o".$$

This notation indicates that D has three copies of $D^{(1)}$ along the diagonal followed by one copy of $D^{(2)}$. We refer to the term "3x1e" as an irrep (irreducible representation) with $\ell = 1$ and multiplicity 3. The suffix letters, "e" or "o", denote a parity used to enforce reflection equivariance, which is not relevant for us (we refer the reader to Thomas et al. [32] for a more complete explanation).

2.2 Core Computational Challenge Let $x \in \mathbb{R}^n, y \in \mathbb{R}^m$ be two vectors from some intermediate layer ϕ of an equivariant deep neural network. For example,

x could be the embedding associated with a node of a graph and y a feature vector for an edge (see Figure 4, bottom). We can view both vectors as functions $x(v), y(v)$ of the network input v , which are equivariant to (D_{in}, D_x) and (D_{in}, D_y) respectively. An equivariant graph convolution layer ϕ interacts x and y to produce a new vector z . To ensure layer-equivariance of ϕ , $z(v)$ must be equivariant to (D_{in}, D_z) , where D_z is a new representation selected by the network designer.

The Kronecker product provides an expressive, general method to interact x and y : if $x(v)$ and $y(v)$ are equivariant to the representations listed above, then $z(v) = x(v) \otimes y(v)$ is equivariant to $(D_{\text{in}}, D_x \otimes D_y)$. Unfortunately, $x \otimes y \in \mathbb{R}^{nm}$ may have intractable length, and we cannot drop arbitrary elements of the Kronecker product without compromising the equivariance property.

Let $P \in \mathbb{R}^{nm \times nm}$ be the similarity transform diagonalizing $D_x \otimes D_y$. To reduce the dimension of the Kronecker product, we first form $P(x(v) \otimes y(v))$, an equivariant function with block-diagonal output representation $P(D_x \otimes D_y)P^{-1}$. We can now safely remove segments of $P(x \otimes y)$ corresponding to unneeded higher-order Wigner blocks and recombine its components through a trainable, structured weight matrix. The result, $z(v) = WP(x(v) \otimes y(v))$, has a new output representation D_z and can be much shorter than $x \otimes y$.

When both D_x and D_y are representations in block-diagonal canonical form, the transform P is a highly structured block-sparse matrix containing nonzero *Clebsch-Gordon coefficients*. After potentially reducing P to k rows (removing segments corresponding to unneeded Wigner D-blocks), we can reshape it into a block-sparse tensor $\mathcal{P} \in \mathbb{R}^{m \times n \times k}$ contracted on two sides with x and y . We call this operation (along with multiplication by a structured weight matrix $W \in \mathbb{R}^{k \times k'}$) the **CG tensor product**, illustrated in Figure 1. It can be expressed by a matrix equation, a summation expression, multilinear tensor contraction (popular in the numerical linear algebra community), or Einstein notation:

$$\begin{aligned}
 (2.1) \quad z &= \boxed{\text{TP}(\mathcal{P}, x, y, W)} \\
 &:= W \cdot P \cdot (x \otimes y) \\
 &:= W \sum_{i=1, j=1}^{m, n} x[i] y[j] \mathcal{P}[ij :] \\
 &:= \mathcal{P} \times_1 x \times_2 y \times_3 W \\
 &:= \text{einsum}("ijk, i, j, kk' \rightarrow k'", \mathcal{P}, x, y, W).
 \end{aligned}$$

Our goal is to accelerate computation of $\text{TP}(\mathcal{P}, x, y, W)$ for a variety of CG coefficient tensors \mathcal{P} . Given $\partial E / \partial z$ for some scalar quantity E , we will also provide

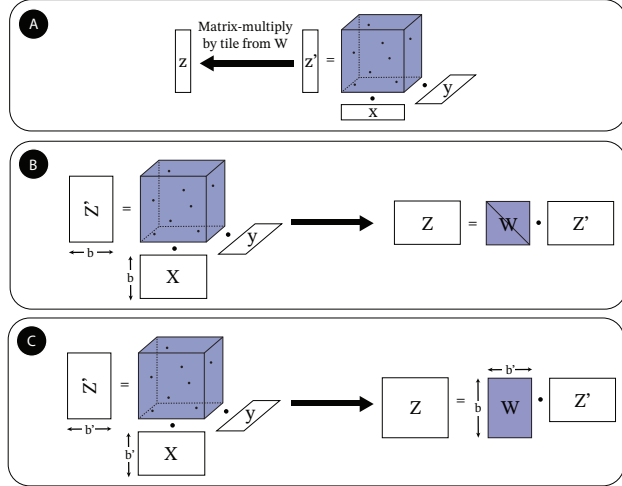


Figure 3: Fundamental subkernels that compose to implement the CG tensor product in Figure 1. In (A), \mathbf{x} , \mathbf{y} , and \mathbf{z} refer to segments of the longer vectors in Figure 1, and \mathbf{W} contains entries from the larger weight matrix rearranged appropriately. In (B) and (C), \mathbf{X} and \mathbf{Z} refer to segments from \mathbf{x} and \mathbf{z} that have been reshaped into matrices to exploit repeating sparse tensor structure. b and b' are multiples of 32 in many models.

an efficient kernel to compute $\partial E/\partial \mathbf{x}$, $\partial E/\partial \mathbf{y}$, and $\partial E/\partial \mathbf{W}$ in a single pass. These gradients are required during both training and inference for some interatomic potential models.

2.3 Structure in the Sparse Tensor and Weights

Suppose \mathbf{D}_x , \mathbf{D}_y , and \mathbf{D}_z each consist of a single Wigner block. In this case, the tensor \mathcal{P} in Figure 1 has a single nonzero block of dimensions $(2\ell_x+1) \times (2\ell_y+1) \times (2\ell_z+1)$. Figure 2 illustrates three blocks with varying parameters, which are small, (current models typically use $\ell_x, \ell_y, \ell_z \leq 4$), highly structured, and sparse.

Every nonzero block of the general tensor \mathcal{P} in Figure 1 takes the form $\mathcal{P}^{(\ell_x, \ell_y, \ell_z)}$. Therefore, we could implement the CG tensor product by repeatedly calling the kernel in Figure 3A: a small tensor contraction followed by multiplication by a tile from the weight matrix \mathbf{W} . In practice, this is an inefficient strategy because \mathcal{P} may contain hundreds of blocks with identical nonzero structures and values.

Instead, the CG tensor product splits into a sequence of subkernels [19] that match the structure in \mathcal{P} and \mathbf{W} . We target two common patterns. First, the CG tensor product may interact b unique segments of \mathbf{x} with a common segment of \mathbf{y} using the same block from \mathcal{P} , followed by multiplication by a submatrix of

weights from \mathbf{W} rearranged along a diagonal. Figure 3B illustrates Kernel B as a contraction of a sparse tensor with a matrix \mathbf{X} (containing the b rearranged segments from \mathbf{x}) and the common vector \mathbf{y} . It appears in the Nequip [4] and MACE [2] models. Kernel C is identical to kernel B, but arranges the weights in a dense matrix $\mathbf{W} \in \mathbb{R}^{b' \times b}$. Here, \mathbf{Z} and \mathbf{X} may have distinct row counts. The latter operation appears in DiffDock [7] and 3D shape classifiers by Thomas et al. [32]. Both operations can be extended to interact multiple segments from \mathbf{y} , but we could not find this pattern in existing equivariant models. While kernels besides B and C are possible, they rarely appear in practice.

2.4 A Full Problem Description Armed with the prior exposition, we now give an example of a specific CG tensor product using the notation of the e3nn software package [10]. A specification of the tensor product consists of a sequence of subkernels and the representations that partition \mathbf{x} , \mathbf{y} , and \mathbf{z} into segments for those kernels to operate on:

$$\begin{aligned}
 (2.2) \quad & \mathbf{D}_x \cong 32x2e + 32x1e \\
 & \mathbf{D}_y \cong 1x3e + 1x1e \\
 & \mathbf{D}_z \cong 32x5e + 16x2e + 32x3e \\
 & [(1, 1, 1, "B"), (1, 2, 2, "C"), (1, 2, 3, "C")].
 \end{aligned}$$

Here, \mathbf{x} and \mathbf{y} are partitioned into two segments, while \mathbf{z} is partitioned into three segments. The list of tuples on the last line specifies the subkernels to execute and the segments they operate on. The first list entry specifies kernel B with the respective first segments of \mathbf{x} , \mathbf{y} , and \mathbf{z} ($b = 32$). Likewise, the second instruction executes kernel C with the first segment of \mathbf{x} and the second segment of \mathbf{y} ($b' = 32, b = 16$) to produce the second segment of \mathbf{z} .

2.5 Context and Related Work Variants of rotation equivariant convolutional neural networks were first proposed by Weiler et al. [34]; Kondor et al. [20]; and Thomas et al. [32]. Nequip [4], Cormorant [1], and Allegro [26] deploy equivariant graph neural networks to achieve state-of-the-art performance for molecular energy prediction. Other works have enhanced these message-passing architectures by adding higher-order equivariant features (e.g. MACE [2, 3] or ChargeE3Net [18]). Equivariance has been integrated into transformer architectures [8, 23] with similar success.

Equivariant Deep Learning Software The e3nn package [9, 10] allows users to construct CG interaction tensors, compute CG tensor products, explicitly form Wigner D-matrices, and evaluate spherical harmonic basis functions. PyTorch and JAX versions of

the package are available. The e3x package [33] offers similar functionality. Allegro [26] modifies the message passing equivariant architecture in Nequip [4] to drastically reduce the number of CG tensor products and lower inter-GPU communication.

The cuEquivariance package [11], released by NVIDIA concurrent to this project’s development, offers the fastest implementation of the CG tensor product outside of our work. However, their kernels are closed-source and do not appear to exploit sparsity *within* each nonzero block of \mathcal{P} . Our kernel performance matches or exceeds cuEquivariance, often by substantial margins. Other relevant codes include GELib [19], Sphercart [5], and Equitriton [22]. The former efficiently computes CG tensor products, while the latter two accelerate spherical harmonic polynomial evaluation.

Alternatives to CG Contraction The intense cost of the CG tensor product has fueled the search for cheap yet accurate algorithms. For example, Schütt et al. [29] propose an equivariant message passing neural network that operates in Cartesian space, eliminating the need for CG tensor products. For SO(3)-equivariant graph convolution, Passaro and Zitnick [27] align the node embeddings with spherical harmonic edge features before interacting the two. This innovation sparsifies the interaction tensor and asymptotically decreases the cost of each tensor product. Luo et al. [25] also produce asymptotic speedups by connecting the tensor product with a spherical harmonic product accelerated through the fast Fourier transform, an operation they call the *Gaunt tensor product*. Xie et al. [36] counter that the Gaunt tensor product does not produce results directly comparable to the CG tensor product, and that the former may sacrifice model expressivity.

GPU Architecture GPUs execute kernels by launching a large number of parallel threads running the same program, each accessing a small set of local registers. In a typical program, threads load data from global memory, perform computation with data in their registers, and store back results. Kernels are most efficient when groups of 32-64 threads (called “warps”) execute the same instruction or perform memory transaction on contiguous, aligned segments of data. Warps execute asynchronously and are grouped into cooperative thread arrays (CTAs) that communicate through a fast, limited pool of shared memory. Warps can synchronize at the CTA level, but the synchronization incurs overhead.

3 Engineering Efficient CG Kernels

Our task is to generate an efficient CG tensor product kernel given a problem specification outlined in Section 2.4. Algorithm 1 describes the logic of a kernel that operates on a large batch of inputs, each with a distinct set of weights (see Figure 4A). We assign each $(\mathbf{x}, \mathbf{y}, \mathbf{W})$ input triple to a single GPU warp, a choice which has two consequences. First, it enables each warp to execute contiguous global memory reads / writes for $\mathbf{x}, \mathbf{y}, \mathbf{W}$ and \mathbf{z} . Second, it allows warps to execute in a completely asynchronous fashion without any CTA-level synchronization, boosting throughput significantly. The weights \mathbf{W} are stored in a compressed form without the zero entries illustrated in Figure 1.

After the division of labor, each warp follows a standard GPU kernel workflow. The three inputs are staged in shared memory, the kernels in Equation (2.2) are executed sequentially, and each output \mathbf{z}_b is stored back. Each warp operates on a unique partition of the CTA shared memory which may not be large enough to contain the the inputs and outputs. In the latter case, chunks of $\mathbf{x}, \mathbf{y}, \mathbf{W}$, and \mathbf{z} are staged, and the computation executes in phases according to a schedule described in Section 3.1.

Algorithm 1 High-Level CGTP Algorithm

Require: Batch $\mathbf{x}_1 \dots \mathbf{x}_B, \mathbf{y}_1 \dots \mathbf{y}_B, \mathbf{W}_1 \dots \mathbf{W}_B$

```

for  $b = 1 \dots B$  do                                ▷ Parallel over warps
  for  $\text{seg}_i \in \text{schedule}$  do
    Load  $\mathbf{x}_{\text{smem}} = \mathbf{x}_b \left[ \text{seg}_i(\mathbf{x} \text{ start}) : \text{seg}_i(\mathbf{x} \text{ end}) \right]$ 
    Load  $\mathbf{y}_{\text{smem}}, \mathbf{W}_{\text{smem}}$  similarly.
    Set  $\mathbf{z}_{\text{smem}} = 0$ 
    for  $\text{kern}_j \in \text{seg}_i$  do
      Set  $\mathbf{X}_{\text{kern}}$  as a reshaped range of  $\mathbf{x}_{\text{smem}}$ 
      Set  $\mathbf{y}_{\text{kern}}, \mathbf{W}_{\text{kern}}$  similarly.
       $\mathbf{Z}_{\text{kern}} = \text{kern}_j(\mathbf{X}_{\text{kern}}, \mathbf{y}_{\text{kern}}, \mathbf{W}_{\text{kern}})$ 
      Flatten, store  $\mathbf{Z}_{\text{kern}}$  to subrange of  $\mathbf{z}_{\text{smem}}$ 
    Store  $\mathbf{z}_b \left[ \text{seg}_i(\mathbf{z} \text{ start}) : \text{seg}_i(\mathbf{z} \text{ end}) \right] += \mathbf{z}_{\text{smem}}$ 

```

We launch a number of CTAs that is a constant multiple of the GPU streaming multiprocessor count and assign 4-8 warps per CTA. The batch size for the CG tensor product can reach millions [26] for large geometric configurations, ensuring that all warps are busy. The computation required for each batch element can exceed 100K FLOPs for typical models [26, 4], ensuring that the threads within each warp are saturated with work.

3.1 Computation Scheduling A key obstacle to efficient kernel implementation is the long length of the \mathbf{x}, \mathbf{y} , and \mathbf{z} feature vectors that must be cached

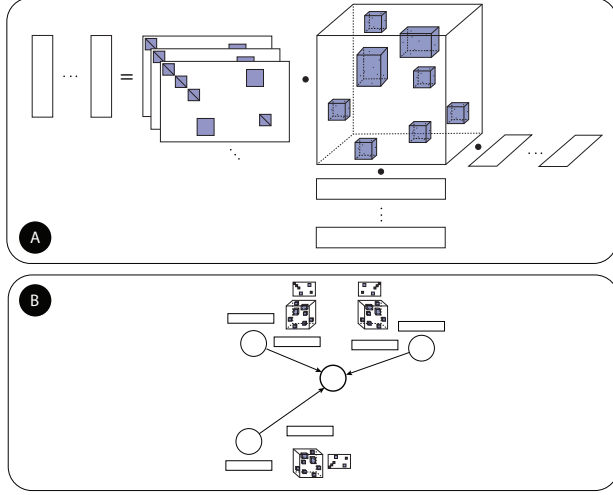


Figure 4: Applications of the CG tensor product. The simplest and most general use case (A) calls the kernel repeatedly with distinct \mathbf{x} , \mathbf{y} , and \mathbf{W} inputs. Interatomic potential models embed the operation in a graph convolution (B), where the tensor product combines node features with edge features.

in shared memory. The sum of their vector lengths for large MACE [3] and Nequip [4] configurations can exceed 10,000 data words. Given that the warps in each CTA partition the shared memory, staging all three vectors at once (along with the weights in \mathbf{W}) is infeasible.

To manage our limited shared memory, we execute the computation in phases that are scheduled at model compile-time. We break the list of instructions in Equation (2.2) into phases so that the sum of chunks from \mathbf{x} , \mathbf{y} , \mathbf{W} and \mathbf{z} required for the phase fits in each warp’s shared memory allotment. We then schedule loads and stores, hardcoding the relevant instructions into each kernel using our JIT capability. When more than a single computation phase is required, our goal is to generate a schedule that minimizes global memory reads and writes. We use a few simple heuristics:

1. If \mathbf{x} and \mathbf{y} can fit into a warp’s shared memory partition (but not \mathbf{z} and \mathbf{W}), then segments of \mathbf{z} and \mathbf{W} are streamed in through multiple phases of computation. In each phase, the kernels that touch each segment of \mathbf{z} are executed.
2. Otherwise, we use a greedy algorithm. In each phase, the shared memory pool is filled with as many segments of \mathbf{x} , \mathbf{y} , \mathbf{W} and \mathbf{z} that can fit. Segments are flushed and reloaded as needed.

Case 1 covers most problem configurations in equivariant graph neural networks and minimizes global memory writes, while Case 2 enables reasonable performance even with constrained shared memory resources. Large CG tensors (e.g. Nequip-benzene in Figure 5) may require 20-40 phases of computation per tensor product, and our scheduling substantially reduces global memory transactions.

3.2 JIT Subkernel Implementations In preparation to execute a subkernel, suppose we have loaded \mathbf{x} , \mathbf{y} and \mathbf{W} into shared memory and reshaped subranges of all three to form \mathbf{X}_{kern} , \mathbf{y}_{kern} , and \mathbf{W}_{kern} . For the rest of Section 3.2 and all of Section 3.3, we omit the “kern” subscripts. Algorithm 2 gives the pseudocode to execute either kernel B or C from Figure 3 using these staged operands. Each thread stages a unique row row of \mathbf{X}

Algorithm 2 Subkernel B & C Warp-Level Algorithm

Require: $\mathbf{X} \in \mathbb{R}^{b' \times (2\ell_x + 1)}$, $\mathbf{y} \in \mathbb{R}^{(2\ell_y + 1)}$, $\mathbf{W} \in \mathbb{R}^{b \times b'}$

Require: Sparse tensor $\mathcal{P}^{(\ell_x, \ell_y, \ell_z)}$ for subkernel

for $t = 1 \dots b'$ **do** ▷ Parallel over threads

Load $\mathbf{x}_{\text{reg}} = \mathbf{X}[t, :]$, $\mathbf{y}_{\text{reg}} = \mathbf{y}$

Initialize $\mathbf{z}_{\text{reg}} \in \mathbb{R}^{2\ell_z + 1}$ to 0.

for $(i, j, k, v) \in \text{nz}(\mathcal{P})$ **do** ▷ Unroll via JIT
 $\mathbf{z}_{\text{reg}}[k] += v \cdot \mathbf{x}_{\text{reg}}[i] \cdot \mathbf{y}_{\text{reg}}[j]$

if \mathbf{W} is diagonal **then** ▷ Compile-time branch

$\mathbf{Z}[t, :] += \mathbf{W}[t, t] \cdot \mathbf{z}_{\text{reg}}$

else

Store $\mathbf{Z}'[t, :] = \mathbf{z}_{\text{reg}}$, compute $\mathbf{Z} += \mathbf{W} \cdot \mathbf{Z}'$

and \mathbf{Z} , as well as the entirety of \mathbf{y} , into its local registers. Models such as Nequip [4] and MACE [3] satisfy $\ell_x, \ell_y, \ell_z \leq 4$, so the added register pressure from the operand caching is manageable. We then loop over all nonzero entries of the sparse tensor to execute the tensor contraction. Because the nonzero indices (i, j, k) and entries v of the sparse tensor \mathcal{P} are known at compile-time, we emit the sequence of instructions in the inner loop explicitly using our JIT kernel generator. Finally, the output \mathbf{Z} is accumulated to shared memory after multiplication by \mathbf{W} . When multiple subkernels execute in sequence, we allow values in \mathbf{x}_{reg} , \mathbf{y}_{reg} , and \mathbf{z}_{reg} to persist if they are reused.

The matrix multiplication by the weights at the end of Algorithm 2 depends on the structure of \mathbf{W} . When \mathbf{W} is square and diagonal (kernel B), multiplication proceeds asynchronously in parallel across all threads. When \mathbf{W} is a general dense matrix, we temporarily store \mathbf{z}_{reg} to shared memory and perform a warp-level

matrix-multiplication across all threads.

Our kernel generator maximizes instruction-level parallelism, and the output kernels contain long streams of independent arithmetic operations. By contrast, common sparse tensor storage formats (coordinate [12], compressed-sparse fiber [30], etc.) require expensive memory indirections that reduce throughput. Because we compile a single kernel to handle all nonzero blocks of \mathcal{P} , we avoid expensive runtime branches and permit data reuse at the shared memory and register level. Such optimizations would be difficult to implement in a traditional statically-compiled library.

For typical applications, b and b' are multiples of 32. When b is greater than 32, the static analysis algorithm in Section 3.1 breaks the computation into multiple subkernels with $b \leq 32$, and likewise for b' .

3.3 Backward Pass Like other kernels in physics informed deep learning models [16], the gradients of the CG tensor product are required during model *inference* as well as training for interatomic force prediction. Suppose $E(\mathbf{R}, \mathbf{W})$ is the scalar energy prediction emitted by our equivariant model for a configuration of s atoms, where \mathbf{W} contains trainable model weights and each row of $\mathbf{R} \in \mathbb{R}^{s \times 3}$ is an atom coordinate. Then $\mathbf{F}_{\text{pr}} = -\partial E / \partial \mathbf{R} \in \mathbb{R}^{s \times 3}$ is the predicted force on each atom. Conveniently, we can compute these forces by auto-differentiating $E(\mathbf{R}, \mathbf{W})$ in a framework like PyTorch or JAX, but we require a kernel to compute the gradient of the CG tensor product inputs given the gradient of its output.

To implement the backward pass, suppose $\mathbf{z} = \text{TP}(\mathcal{P}, \mathbf{x}, \mathbf{y}, \mathbf{W})$ and we have $\mathbf{g}_z = \partial E / \partial \mathbf{z}$. Because the CG tensor product is linear in its inputs, the product rule gives

$$\begin{aligned} \partial E / \partial \mathbf{x}[i] &= \sum_{(i,j,k) \in \text{nz}(\mathcal{P})} \mathcal{P}[ijk] \cdot \mathbf{y}[j] \cdot (\mathbf{W}^\top \cdot \mathbf{g}_z)[k] \\ \partial E / \partial \mathbf{y}[j] &= \sum_{(i,j,k) \in \text{nz}(\mathcal{P})} \mathcal{P}[ijk] \cdot \mathbf{x}[i] \cdot (\mathbf{W}^\top \cdot \mathbf{g}_z)[k] \\ \partial E / \partial \mathbf{W}[kk'] &= \mathbf{g}_z[k'] \cdot \sum_{(i,j,k) \in \text{nz}(\mathcal{P})} \mathcal{P}[ijk] \mathbf{x}[i] \mathbf{y}[j] \end{aligned}$$

Notice the similarity between the three equations above and Equation (2.1): all require summation over the nonzero indices of \mathcal{P} and multiplying each value with a pair of elements from distinct vectors. Accordingly, we develop Algorithm 3 with similar structure to Algorithm 2 to compute all three gradients in a single kernel. For simplicity, we list the general case where the submatrix \mathbf{W} is a general dense matrix (kernel C). There are two new key features in Algorithm 3: first, we must perform a reduction over the warp for the gradient

Algorithm 3 Subkernel C Warp-Level Backward

Require: $\mathbf{X} \in \mathbb{R}^{b' \times (2\ell_x + 1)}$, $\mathbf{y} \in \mathbb{R}^{2\ell_y + 1}$, $\mathbf{W} \in \mathbb{R}^{b \times b'}$

Require: $\mathbf{G}_Z \in \mathbb{R}^{b \times (2\ell_z + 1)}$, sparse tensor $\mathcal{P}^{(\ell_x, \ell_y, \ell_z)}$

Threads collaboratively compute $\mathbf{G}'_Z = \mathbf{W}^\top \cdot \mathbf{G}_Z$

for $t = 1 \dots b'$ **do** ▷ Parallel over threads

Load $\mathbf{x}_{\text{reg}} = \mathbf{X}[t, :]$, $\mathbf{y}_{\text{reg}} = \mathbf{y}$, $\mathbf{g}'_{z\text{reg}} = \mathbf{G}'_Z[t, :]$

Initialize $\mathbf{g}_{x\text{reg}}, \mathbf{g}_{y\text{reg}}, \mathbf{g}_{w\text{reg}}, \mathbf{z}_{\text{reg}}$ to 0.

for $(i, j, k, v) \in \text{nz}(\mathcal{P}^{(\ell_x, \ell_y, \ell_z)})$ **do** ▷ Unroll via JIT

$\mathbf{g}_{x\text{reg}}[i] += v \cdot \mathbf{y}_{\text{reg}}[j] \cdot \mathbf{g}'_{z\text{reg}}[k]$

$\mathbf{g}_{y\text{reg}}[j] += v \cdot \mathbf{x}_{\text{reg}}[i] \cdot \mathbf{g}'_{z\text{reg}}[k]$

$\mathbf{z}_{\text{reg}}[k] += v \cdot \mathbf{x}_{\text{reg}}[i] \cdot \mathbf{y}_{\text{reg}}[j]$

Store $\mathbf{g}_y = \text{warp-reduce}(\mathbf{g}_{y\text{reg}})$

Store $\mathbf{G}_x[t, :] = \mathbf{g}_{x\text{reg}}$ and $\mathbf{Z}'[t, :] = \mathbf{z}_{\text{reg}}$

Threads collaboratively compute $\mathbf{G}_W = \mathbf{G}_Z \cdot (\mathbf{Z}')^\top$

vector \mathbf{g}_y , since each thread calculates a contribution that must be summed. Second: when \mathbf{W} is not diagonal, an additional warp-level matrix multiply is required at the end of the algorithm to calculate \mathbf{G}_W . We embed Algorithm 3 into a high-level procedure akin to Algorithm 1 to complete the backward pass.

3.4 Higher Partial Derivatives For interatomic potential models, we require higher-order derivatives to optimize force predictions during training [16], as we explain below. Rather than write new kernels for these derivatives, we provide a novel (to the best of our knowledge) calculation that implements them using the existing forward and backward pass kernels.

As in Section 3.3, let $\mathbf{F}_{\text{pr}} = -\partial E / \partial \mathbf{R} \in \mathbb{R}^{s \times 3}$ be the predicted atomic forces generated by our model. During training, we must minimize a loss function of the form

$$\min_{\mathbf{W}} \mathcal{L}(\mathbf{R}, \mathbf{W}) = \min_{\mathbf{W}} \|\mathbf{F}_{\text{pr}}(\mathbf{R}, \mathbf{W}) - \mathbf{F}_{\text{gt}}(\mathbf{R})\|_F^2$$

where $\mathbf{F}_{\text{gt}}(\mathbf{R}) \in \mathbb{R}^{s \times 3}$ is a set of ground-truth forces created from a more expensive simulation. The loss function may include other terms, but only the Frobenius norm of the force difference is relevant here. We use a gradient method to perform the minimization and calculate

$$\begin{aligned} (3.3) \quad \frac{\partial \mathcal{L}}{\partial \mathbf{W}} &= 2 \cdot \text{vec}(\mathbf{F}_{\text{pr}}(\mathbf{R}, \mathbf{W}) - \mathbf{F}_{\text{gt}}(\mathbf{R}))^\top \frac{\partial \mathbf{F}_{\text{pr}}}{\partial \mathbf{W}} \\ &= -2 \cdot \text{vec}(\mathbf{F}_{\text{pr}}(\mathbf{R}, \mathbf{W}) - \mathbf{F}_{\text{gt}}(\mathbf{R}))^\top \frac{\partial^2 E}{\partial \mathbf{R} \partial \mathbf{W}} \end{aligned}$$

where “vec” flattens its matrix argument into a vector and $\partial^2 E / (\partial \mathbf{R} \partial \mathbf{W}) \in \mathbb{R}^{3s \times (\# \text{ weights})}$ is a matrix of second partial derivatives. Equation (3.3) can also be computed by auto-differentiation, but the second partial

derivative requires us to register an autograd formula for our CG tensor product *backward* kernel (i.e. we must provide a “double-backward” implementation).

To avoid spiraling engineering complexity, we will implement the double-backward pass by linearly combining the outputs from existing kernels. Let $\mathbf{z} = \text{TP}(\mathbf{x}, \mathbf{y}, \mathbf{W})$ (we omit the sparse tensor argument \mathcal{P} here) and define $\mathbf{g}_z = \partial E / \partial \mathbf{z}$ for the scalar energy prediction E . Finally, let \mathbf{a} , \mathbf{b} , and \mathbf{C} be the gradients calculated by the backward pass, given as

$$\begin{aligned} [\mathbf{a}, \mathbf{b}, \mathbf{C}] &= [\partial E / \partial \mathbf{x}, \partial E / \partial \mathbf{y}, \partial E / \partial \mathbf{W}] \\ &= \text{backward}(\mathbf{x}, \mathbf{y}, \mathbf{W}, \mathbf{g}_z). \end{aligned}$$

Now our task is to compute $(\partial \mathcal{L} / \partial \mathbf{x}, \partial \mathcal{L} / \partial \mathbf{y}, \partial \mathcal{L} / \partial \mathbf{W}, \partial \mathcal{L} / \partial \mathbf{g}_z)$ given $(\partial \mathcal{L} / \partial \mathbf{a}, \partial \mathcal{L} / \partial \mathbf{b}, \partial \mathcal{L} / \partial \mathbf{C})$. We dispatch seven calls to the forward and backward pass kernels:

$$\begin{aligned} \text{op1} &= \text{backward}(\partial \mathcal{L} / \partial \mathbf{a}, \partial \mathcal{L} / \partial \mathbf{b}, \mathbf{W}, \mathbf{g}_z) \\ \text{op2} &= \text{backward}(\mathbf{x}, \mathbf{y}, \partial \mathcal{L} / \partial \mathbf{C}, \mathbf{g}_z) \\ \text{op3} &= \text{TP}(\partial \mathcal{L} / \partial \mathbf{a}, \mathbf{y}, \mathbf{W}) \\ (3.4) \quad \text{op4} &= \text{backward}(\partial \mathcal{L} / \partial \mathbf{a}, \mathbf{y}, \mathbf{W}, \mathbf{g}_z) \\ \text{op5} &= \text{backward}(\mathbf{x}, \partial \mathcal{L} / \partial \mathbf{b}, \mathbf{W}, \mathbf{g}_z) \\ \text{op6} &= \text{TP}(\mathbf{x}, \partial \mathcal{L} / \partial \mathbf{b}, \mathbf{W}) \\ \text{op7} &= \text{TP}(\mathbf{x}, \mathbf{y}, \partial \mathcal{L} / \partial \mathbf{C}). \end{aligned}$$

By repeatedly applying the product and chain rules to the formulas for \mathbf{a} , \mathbf{b} , and \mathbf{C} in Section 3.3, we can show

$$\begin{aligned} \partial \mathcal{L} / \partial \mathbf{x} &= \text{op1}[1] + \text{op2}[1] \\ (3.5) \quad \partial \mathcal{L} / \partial \mathbf{y} &= \text{op1}[2] + \text{op2}[2] \\ \partial \mathcal{L} / \partial \mathbf{W} &= \text{op4}[3] + \text{op5}[3] \\ \partial \mathcal{L} / \partial \mathbf{g}_z &= \text{op3} + \text{op6} + \text{op7}, \end{aligned}$$

where $\text{op1}[1]$, $\text{op1}[2]$, and $\text{op1}[3]$ denote the three results calculated by the backward function, and likewise for op2 , op4 , and op5 . Equations (3.4) and (3.5) could be implemented in less than 10 lines of Python and accelerate the double-backward pass without any additional kernel engineering. In practice, we fuse the forward calls into a single kernel by calling Algorithm 2 three times with different arguments in a procedure like Algorithm 1. The backward calls fuse in a similar manner, and we adopt this approach to dramatically reduce memory traffic and kernel launch overhead.

3.5 Graph Convolution and Kernel Fusion Figure 4 illustrates two typical use cases of the CG tensor product kernel. The first case (4A) calls the kernel illustrated in Figure 1 several times with unique triples of $(\mathbf{x}, \mathbf{y}, \mathbf{W})$ inputs, and we have already addressed its implementation. The second case (4B) embeds the CG tensor product into a graph convolution

operation [32, 4, 2]. Here, the nodes of a graph typically correspond to atoms in a simulation and edges represent pairwise interactions. For a graph $G = (V, E)$, let $\mathbf{x}_1 \dots \mathbf{x}_{|V|}$, $\mathbf{y}_1 \dots \mathbf{y}_{|E|}$, and $\mathbf{W}_1 \dots \mathbf{W}_{|E|}$ be node embeddings, edge embeddings, and trainable edge weights, respectively. Then each row \mathbf{z}_j of the graph convolution output, $j \in [|V|]$, is given by

$$(3.6) \quad \mathbf{z}_j = \sum_{(j,k,e) \in \mathcal{N}(j)} \text{TP}(\mathcal{P}, \mathbf{x}_k, \mathbf{y}_e, \mathbf{W}_e),$$

where $\mathcal{N}(j)$ denotes the neighbor set of node j and $(j, k, e) \in \mathcal{N}(j)$ indicates that edge e connects nodes j and k . Current equivariant message passing networks [4, 2] implement Equation (3.6) by duplicating the node features to form $\mathbf{x}'_1, \dots, \mathbf{x}'_{|E|}$, calling the large batch kernel developed earlier, and then executing a scatter-sum (also called reduce-by-key) to perform aggregation. Unfortunately, duplicating the node features incurs significant memory and communication-bandwidth overhead when $|E| \gg |V|$ (see Table 3).

Notice that graph convolution exhibits a memory access pattern similar to sparse-dense matrix multiplication (SpMM) [37]. We provide two procedures for the fused CGTP / graph convolution based on classic SpMM methods. The first, detailed in Algorithm 4, requires row-major sorted edge indices and iterates over the phases of the computation schedule as the outer loop. The latter change enables the algorithm to keep a running buffer \mathbf{z}_{acc} that accumulates the summation in Equation (3.6) for each node. The buffer \mathbf{z}_{acc} is only flushed to global memory when a warp transitions to a new row of the graph adjacency matrix, reducing global memory writes from $O(|E|)$ to $O(|V|)$. To handle the case where two or more warps calculate contributions to the same node, we write the first row processed by each warp to a fixup buffer [37]. We developed a backward pass kernel using a similar SpMM strategy, but a permutation that transposes the graph adjacency matrix is required as part of the input.

The second algorithm, which we omit for brevity, functions almost identically to Algorithm 4, but replaces the fixup / store logic with an atomic accumulation at every inner loop iteration. This *nondeterministic* method performs $O(|E|)$ atomic storebacks, but does not require a sorted input graph or adjacency transpose permutation.

4 Experiments

Our kernel generator is available online¹ as an installable Python package. We adopted the frontend interface of e3nn [9, 10] and used QuTiP [13, 21] to generate

¹<https://github.com/PASSIONLab/OpenEquivariance>

Algorithm 4 Deterministic TP + Graph Convolution

Require: Graph $G = (V, E)$, $E[b] = (i_b, j_b)$
Require: Edges in E sorted by first coordinate.
Require: Batch $\mathbf{x}_1, \dots, \mathbf{x}_{|V|}$, $\mathbf{y}_1, \dots, \mathbf{y}_{|E|}$, $\mathbf{W}_1, \dots, \mathbf{W}_{|E|}$

```

for  $\text{seg}_i \in \text{schedule}$  do
   $(s, t) = E[k][0], E[k][1]$ 
  Set  $\mathbf{z}_{\text{acc}} = 0$ 
  for  $b = 1 \dots |E|$  do ▷ Parallel over warps
     $\mathbf{x}_{\text{smem}} = \mathbf{x}_t [\text{seg}_i(\text{x start}) : \text{seg}_i(\text{x end})]$ 
    Load  $\mathbf{y}_{\text{smem}}, \mathbf{W}_{\text{smem}}$  similarly, set  $\mathbf{z}_{\text{smem}} = 0$ .
    Run kernels as in Algorithm 1.
     $\mathbf{z}_{\text{acc}} += \mathbf{z}_{\text{smem}}$ 
    if  $b = |E|$  or  $s < E[b+1][0]$  then
      if  $s$  is first vertex processed by warp then
        Send  $\mathbf{z}_{\text{acc}}$  to fixup buffer.
      else
         $\mathbf{z}_s [\text{seg}_i(\text{z start}) : \text{seg}_i(\text{z end})] += \mathbf{z}_{\text{acc}}$ 
     $\mathbf{z}_{\text{acc}} = 0$ 
  Execute fixup kernel.

```

CG coefficients. We tested correctness against e3nn to ensure that our kernels produce identical results, up to floating point roundoff and a well-defined permutation of the weights \mathbf{W} on certain input configurations. In cases where weight reordering is required, we provide a function for easy migration. We use the NVIDIA and AMD HIP Runtime Compilers to compile our generated kernels through a C++ extension to Python.

Quantity	Value
FP32 Peak	19.5 TFLOP/s
FP64 SIMT Peak	9.7 TFLOP/s
FP64 Tensor Core Peak	19.5 TFLOP/s
HBM2 Bandwidth	2.04 TB/s

Table 1: A100-SXM4-80GB performance [6].

The majority of experiments were conducted on NVIDIA A100 GPU nodes of NERSC Perlmutter (each equipped with an AMD EPYC 7763 CPU). Table 1 lists the advertised maximum memory bandwidth and compute peaks for multiple datatypes, a yardstick for our results. Section 4.4 covers performance on other GPU models.

As baselines, we used the PyTorch versions of e3nn (v0.5.6) [10] and NVIDIA cuEquivariance (v0.4.0) [11]. The e3nn implementation was accelerated with `torch.compile` except where prohibited by memory constraints. For Figures 5, 6, 7, and 9, we benchmarked all functions through a uniform PyTorch interface and

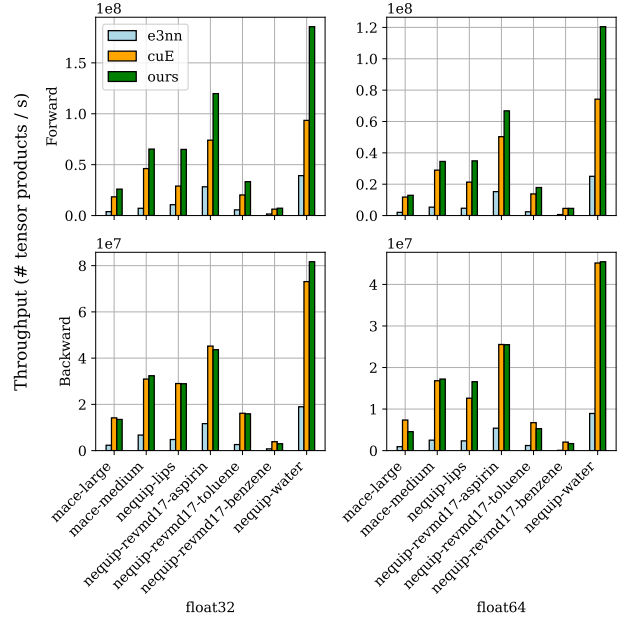


Figure 5: Throughput of CG tensor products (batch size 50K), kernel B configurations without SpMM kernel fusion. On difficult configurations like Nequip-benzene with massive output vector lengths, we exhibit more than 10x improvement over e3nn.

included any overhead in the measured runtime. Figures 8 and 10 (right) rely on kernel runtime measurements without PyTorch overhead.

cuEquivariance experienced a significant efficiency increase since v0.2.0, the latest version available when our first preprint was released (see Figures 8, 10). Since that early release, the authors also added JIT capability and fused convolution, although the closed source kernel backend renders the details opaque. Unless otherwise noted, we report all benchmarks against cuE v0.4.0.

4.1 Forward / Backward Throughput We first profiled our kernels on a large collection of model configurations used by Nequip [4] and MACE [2]. For each model, we selected an expensive representative tensor product to benchmark. Figure 5 shows the results of our profiling on configurations that use only Kernel B (see Figure 3). Our median FP32 speedup over e3nn was 5.9x, resp. 4.8x, for the forward and backward passes, with a maximum of 9.2x for the forward pass. We observed a median speedup of 1.6x over cuE for the FP32 forward pass, which drops to 1.3x in FP64 precision. Our median performance approaches parity with cuE on the backward pass, with a minimum and

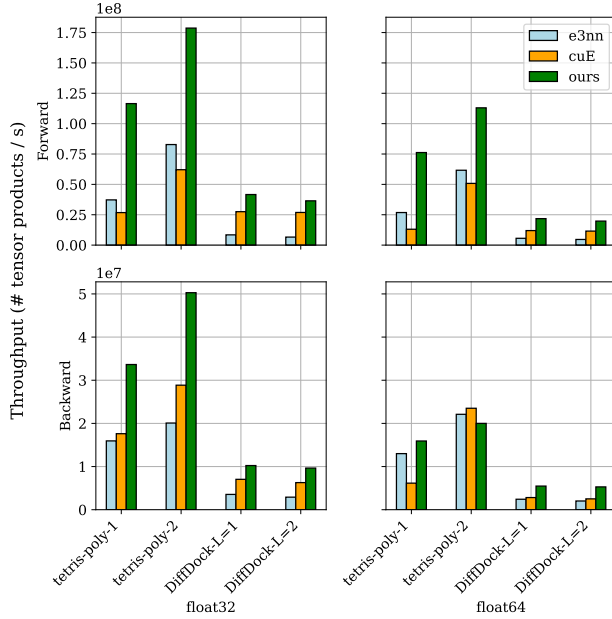


Figure 6: Throughput of CG tensor products (batch size 50K), kernel C configurations. We exhibit up to 2x speedup over cuE on the DiffDock tensor products.

maximum speedup of 0.72x and 1.32x in FP64 precision.

To benchmark kernel C, we used the Tetris polynomial from e3nn’s documentation [10] and two configurations based on DiffDock [7]. We exhibit between 1.4x and 2.0x speedup over cuE for both forward and backward passes on DiffDock. Our speedups over e3nn are less dramatic for kernel C, which has a workload dominated by the small dense matrix multiplication in Algorithms 2 and 3.

4.2 Second Derivative Performance We analyzed the double backward pass for the tensor products from the prior section (excluding the Tetris polynomial, which does not require it). Figure 7 shows our results. Across datatypes and model configurations, our speedup ranges from 5.5x to 35x over e3nn and 0.69x-1.69x over cuE. Although our median speedup over cuE is 0.73x in FP32 precision and 0.93x for FP64 precision, we exhibit lower runtime on all DiffDock tensor products and several Nequip configurations.

For many Nequip / MACE configurations, the performance gap between our implementation and cuE could likely shrink with some judicious kernel tuning. In particular, we could improve our heuristic selection of the warp count per block, the number of blocks, and the shared memory allotted to each block. We leave

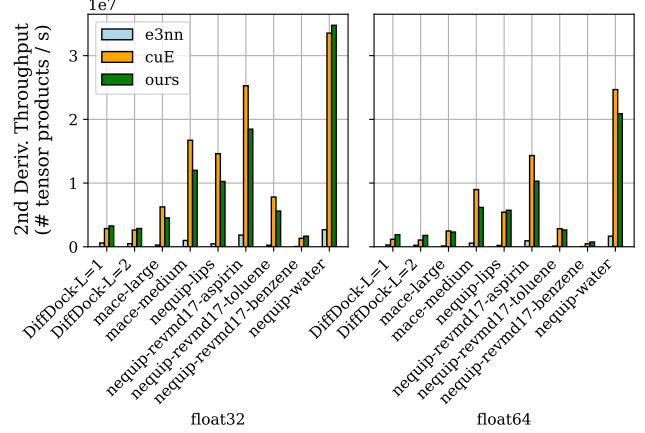


Figure 7: Throughput of second derivative kernels (batch size 20K) for chemistry / protein models.

tuning these hyperparameters as future work.

4.3 Roofline Analysis We conducted a roofline analysis [35] by profiling our forward / backward pass implementations on varied input configurations. We profiled tensor products with a single “B” subkernel (see Figure 3) with FP32 precision, core building blocks for models like Nequip and MACE. The arithmetic intensity of the CG tensor product depends on the structure of the sparse tensor, and we profiled configurations with progressively increasing arithmetic intensity.

Figure 8 shows our results, which indicate consistently high compute and bandwidth utilization for both our kernels and the latest version of cuE. An earlier version of cuE (v0.2.0, indicated on the plot as cuE-old) exhibited significantly lower efficiency, which has since been corrected by the package authors. The performance of all kernels saturates at 58% of the FP32 peak, likely because Algorithms 2 and 3 contain a significant fraction of non fused-multiply-add (FMA) instructions.

4.4 Additional GPU Models We also tested our kernels on the NVIDIA A5000 and a single GPU die of the AMD MI250x. Table 2 compares the MACE tensor product runtime across architectures and kernel providers; our codebase contains a more complete set of benchmarks. The A5000 performance matches our expectations given its lower memory bandwidth compared to the A100. While we also saw significant speedup on the MI250x, we detected somewhat lower memory bandwidth utilization than predicted.

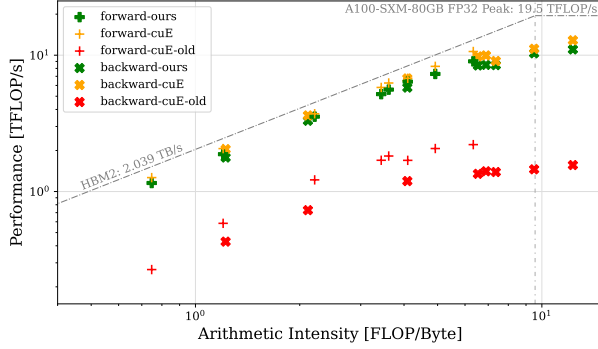


Figure 8: Roofline analysis for input configurations of varying arithmetic intensity, batch size 200K. Our kernels and the latest version of cuE closely track the slope of the global memory roofline, indicating high efficiency. An older version of cuE (v0.2.0) is also included to highlight the performance improvement in their package (see top of Section 4).

GPU	forward			backward		
	e3nn	cuE	ours	e3nn	cuE	ours
A100	13	2.8	2.0	21	3.5	3.7
A5000	29	4.2	3.8	42	9.3	11
MI250x	41	-	3.0	128	-	15

Table 2: MACE-large isolated tensor product runtime (ms), batch size 50K, FP32 unfused.

4.5 Kernel Fusion Benchmarks We conducted our kernel fusion experiments on three molecular structure graphs listed in Table 3. We downloaded the atomic structures of human dihydrofolate deductase (DHFR) and the SARS-COV-2 glycoprotein spike from the Protein Data Bank and constructed a radius-neighbors graph for each using Scikit-Learn [28]. The carbon lattice was provided to us as a representative workload for MACE [3].

Graph	Nodes	Adj. Matrix NNZ
DHFR 1DRF	1.8K	56K
COVID spike 6VXX	23K	136K
Carbon lattice	1K	158K

Table 3: Molecular graphs for kernel fusion experiments.

Figure 9 shows the speedup of fused implementations benchmarked on the most expensive tensor product in the MACE-large model. The baseline, “cuE-

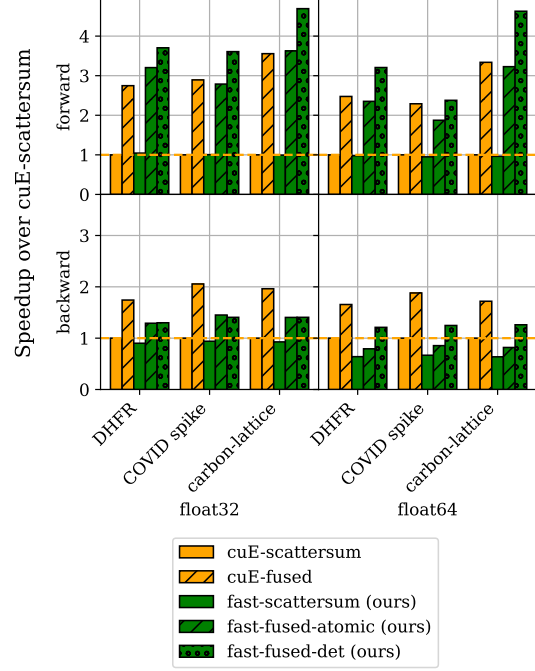


Figure 9: Speedup of convolution kernels over cuE-scattersum, which calls cuEquivariance and follows it by a scatter-sum operation. cuE-fused refers to a new kernel introduced in cuE v0.4.0.

scattersum”, implements the unfused strategy in Section 3.5 by duplicating node embeddings, executing a large batch of tensor products with cuEquivariance, and finally performing row-based reduction by keys. Our deterministic fused algorithm offers the greatest speedup in FP64 precision on the carbon lattice forward pass over cuE (roughly 1.3x speedup). On the other hand, cuE-fused offers 1.3-1.4x speedup on the backward pass, and we aim to close this performance gap.

4.6 Acceleration of Nequip and MACE The Nequip [4, 31] and MACE [2, 3] interatomic potential models implement the equivariant graph neural network architecture in Figure 4B. Both have similar message passing structures, while MACE incorporates higher order interactions for its node features. Our first benchmark uses the Nequip-ASE calculator interface to evaluate forces on a large box of water molecules. Due to recent updates to Nequip’s software [31] and time constraints, we only used the nondeterministic fused convolution for our measurements, which appear in Table 4. For simplicity, we report speedup with respect to the Nequip Python interface without JITScript or `torch.compile`, although our package is fully compatible with these subsystems.

GPU	Speedup over Unmodified Nequip	
	ours-unfused	ours-fused
A100	6.3x	7.8x
MI250x	3.9x	4.4x

Table 4: Force evaluation speedup for our kernels on a 4-layer FP32 Nequip model, 5184-atom water box system.

For MACE, we patched the code to sort nonzeros of the atomic adjacency matrix according to compressed sparse row (CSR) order. We then substituted our deterministic fused convolution into the model and conducted our benchmark on the carbon lattice in Table 3. MACE uses a distinct set of weight matrices for each atomic species, and an inefficiency in the baseline code causes its runtime to increase disproportionately to the useful computation involved. Our model has a species dictionary of eight elements (to trigger the problem in the baseline code, even though the carbon lattice only requires one), and our package includes a module to optimize away the inefficiency.

Figure 10 (left) compares the rate of molecular dynamics simulation among the different kernel providers. We benchmarked cuE with the optimal data layout for its irreps and included optimizations for symmetric tensor contraction, linear combination layers, and graph convolution. In FP32 precision, we provide a 5.1x speedup over e3nn and 1.5x over the older v0.2.0 cuE package, noting that the latter does not provide kernel fusion. A similar speedup exists for the FP64 models. Our implementation, however, achieves 0.73x speedup compared to the latest version of cuE that introduces kernel fusion.

To further investigate these benchmarks, Figure 10 (right) breaks down device runtime spent in various kernels. Our time spent on the tensor product (CTP kernels) falls within 2-3 milliseconds of cuE, a highly competitive result. Our performance suffers due to the remaining model components, which contribute to less than 15% of the unoptimized model runtime. To address this, we created a hybrid model (ours-cuE-hybrid) that combines our fused convolution with the linear / symmetric contraction layers offered by cuE. While the hybrid model closes the gap further, the runtime of the remaining kernels is still higher than cuE. This is because the hybrid model preserves the original data layout of MACE layers, whereas cuE transposes several key weight matrices to achieve higher performance. As a consequence, cuE requires a data reordering function for models trained without the package, whereas our kernels have no such restriction.

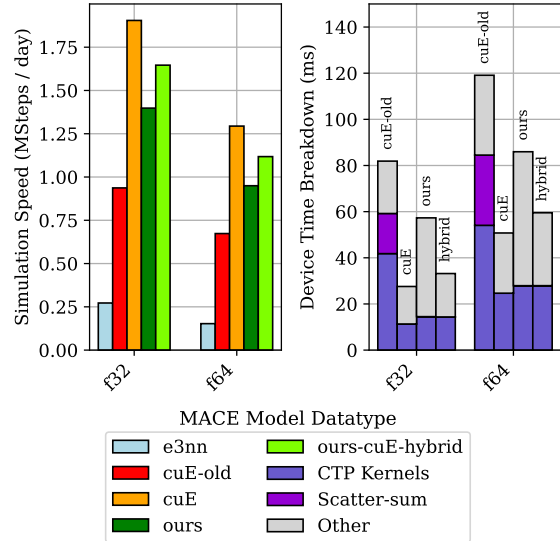


Figure 10: Simulation speed of MACE for varying kernel provider (left) and device time breakdown (right). cuE-old refers to version 0.2.0 of their package, while the hybrid implementation combines our fused convolution with other model primitives optimized by cuE.

5 Conclusions and Further Work

We have established that our sparse kernels achieve consistently high performance on several common primitives used in $O(3)$ -equivariant deep neural networks. We see several avenues for future progress:

- Low Precision Data and MMA Support:** Our kernels rely on the single-instruction multiple thread (SIMT) cores for FP32 and FP64 floating point arithmetic. Modern GPUs offer specialized hardware for lower-precision calculation, both using SIMT cores and within matrix-multiply-accumulate (MMA) units. We hope to harness these capabilities in the future.
- Stable Summation During Convolution:** Our kernel generator allows us to easily extend our methods to use stable (Kahan) summation [15] within fused graph convolution. Kahan summation reduces numerical roundoff error during feature vector aggregation across node neighborhoods, promoting energy conservation in simulations.
- Integration into new models:** Our open-source software remains accessible to newcomers while delivering the high performance required for massive workloads. In conjunction with domain experts, we hope to apply our library to train larger, more expressive equivariant deep neural networks.

Acknowledgements and Disclaimers

We thank the anonymous referees for feedback that strengthened our draft. This research was supported by the U.S. Department of Energy, Office of Science, Office of Advanced Scientific Computing Research, Department of Energy Computational Science Graduate Fellowship under Award Number DE-SC0022158. This research is also supported by the Office of Science of the DOE under Award Number DE-AC02-05CH11231. We used resources of the National Energy Research Scientific Computing Center (NERSC), a Department of Energy Office of Science User Facility using NERSC award ASCR-ERCAP-33069. We also used resources of the Oak Ridge Leadership Computing Facility at the Oak Ridge National Laboratory, which is supported by the Office of Science of the U.S. Department of Energy under Contract No. DE-AC05-00OR22725.

This report was prepared as an account of work sponsored by an agency of the United States Government. Neither the United States Government nor any agency thereof, nor any of their employees, makes any warranty, express or implied, or assumes any legal liability or responsibility for the accuracy, completeness, or usefulness of any information, apparatus, product, or process disclosed, or represents that its use would not infringe privately owned rights. Reference herein to any specific commercial product, process, or service by trade name, trademark, manufacturer, or otherwise does not necessarily constitute or imply its endorsement, recommendation, or favoring by the United States Government or any agency thereof. The views and opinions of authors expressed herein do not necessarily state or reflect those of the United States Government or any agency thereof.

References

- [1] Brandon Anderson, Truong-Son Hy, and Risi Kondor. Cormorant: covariant molecular neural networks. In *Proceedings of the 33rd International Conference on Neural Information Processing Systems*, Red Hook, NY, USA, 2019. Curran Associates Inc.
- [2] Ilyes Batatia, David Peter Kovacs, Gregor N. C. Simm, Christoph Ortner, and Gabor Csanyi. MACE: Higher order equivariant message passing neural networks for fast and accurate force fields. In *Advances in Neural Information Processing Systems*, 2022.
- [3] Ilyes Batatia, Philipp Benner, Yuan Chiang, Alin M. Elena, Dávid P. Kovács, Janosh Riebesell, Xavier R. Advincula, Mark Asta, Matthew Avaylon, William J. Baldwin, Fabian Berger, Noam Bernstein, Arghya Bhowmik, Samuel M. Blau, Vlad Cărare, James P. Darby, Sandip De, Flaviano Della Pia, Volker L. Deringer, Rokas Elijošius, Zakariya El-Machachi, Fabio Falcioni, Edvin Fako, Andrea C. Ferrari, Annalena Genreith-Schriever, Janine George, Rhys E. A. Goodall, Clare P. Grey, Petr Grigorev, Shuang Han, Will Handley, Hendrik H. Heenen, Kersti Hermansson, Christian Holm, Jad Jaafar, Stephan Hofmann, Konstantin S. Jakob, Hyunwook Jung, Venkat Kapil, Aaron D. Kaplan, Nima Karimitari, James R. Kermode, Namu Kroupa, Jolla Kullgren, Matthew C. Kuner, Domantas Kuryla, Guoda Liepuoniute, Johannes T. Margraf, Ioan-Bogdan Magdău, Angelos Michaelides, J. Harry Moore, Aakash A. Naik, Samuel P. Niblett, Sam Walton Norwood, Niamh O'Neill, Christoph Ortner, Kristin A. Persson, Karsten Reuter, Andrew S. Rosen, Lars L. Schaaf, Christoph Schran, Benjamin X. Shi, Eric Sivonxay, Tamás K. Stenczel, Viktor Svahn, Christopher Sutton, Thomas D. Swinburne, Jules Tilly, Cas van der Oord, Eszter Varga-Umbrich, Tejs Vegge, Martin Vondrák, Yangshuai Wang, William C. Witt, Fabian Zills, and Gábor Csányi. A foundation model for atomistic materials chemistry, 2024. URL <https://arxiv.org/abs/2401.00096>.
- [4] Simon Batzner, Albert Musaelian, Lixin Sun, Mario Geiger, Jonathan P. Mailoa, Mordechai Kornbluth, Nicola Molinari, Tess E. Smidt, and Boris Kozinsky. E(3)-equivariant graph neural networks for data-efficient and accurate interatomic potentials. *Nature Communications*, 13(1):2453, May 2022. ISSN 2041-1723. doi: 10.1038/s41467-022-29939-5.
- [5] Filippo Bigi, Guillaume Fraux, Nicholas J. Brownling, and Michele Ceriotti. Fast evaluation of spherical harmonics with sphericart. *J. Chem. Phys.*, 159:064802, 2023.
- [6] NVIDIA Corporation. NVIDIA A100 tensor core GPU architecture. Technical report, NVIDIA, 2020. URL <https://images.nvidia.com/aem-dam/en-zz/Solutions/data-center/nvidia-ampere-architecture-whitepaper.pdf>.
- [7] Gabriele Corso, Hannes Stärk, Bowen Jing, Regina Barzilay, and Tommi S. Jaakkola. Diffdock: Diffusion steps, twists, and turns for molecular docking. In *The Eleventh International Conference on Learning Representations*, 2023.
- [8] Fabian Fuchs, Daniel Worrall, Volker Fischer,

- and Max Welling. SE(3)-transformers: 3d rotation equivariant attention networks. In H. Larochelle, M. Ranzato, R. Hadsell, M.F. Balcan, and H. Lin, editors, *Advances in Neural Information Processing Systems*, volume 33, pages 1970–1981. Curran Associates, Inc., 2020.
- [9] Mario Geiger and Tess Smidt. e3nn: Euclidean neural networks, 2022. URL <https://arxiv.org/abs/2207.09453>.
- [10] Mario Geiger, Tess Smidt, Alby M., Benjamin Kurt Miller, Wouter Boomsma, Bradley Dice, Kostiantyn Lapchevskyi, Maurice Weiler, Michał Tyszkiewicz, Simon Batzner, Dylan Madiseti, 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://github.com/e3nn/e3nn/releases/tag/0.5.0>.
- [11] Mario Geiger, Emine Kucukbenli, Becca Zandstein, and Kyle Tretina. Accelerate drug and material discovery with new math library NVIDIA cuEquivariance, 11 2024. URL <https://developer.nvidia.com/blog/accelerate-drug-and-material-discovery-with-new-math-library-nvidia-cuequivariance/>.
- [12] Ahmed E. Helal, Jan Laukemann, Fabio Checconi, Jesmin Jahan Tithi, Teresa Ranadive, Fabrizio Petrini, and Jeewhan Choi. ALTO: adaptive linearized storage of sparse tensors. In *Proceedings of the 35th ACM International Conference on Supercomputing, ICS '21*, page 404–416, New York, NY, USA, 2021. Association for Computing Machinery. ISBN 9781450383356. doi: 10.1145/3447818.3461703.
- [13] J.R. Johansson, P.D. Nation, and F. Nori. QuTiP 2: A Python framework for the dynamics of open quantum systems. *Computer Physics Communications*, 184(4):1234–1240, apr 2013. doi: 10.1016/j.cpc.2012.11.019.
- [14] John Jumper, Richard Evans, Alexander Pritzel, Tim Green, Michael Figurnov, Olaf Ronneberger, Kathryn Tunyasuvunakool, Russ Bates, Augustin Židek, Anna Potapenko, Alex Bridgland, Clemens Meyer, Simon A. A. Kohl, Andrew J. Ballard, Andrew Cowie, Bernardino Romera-Paredes, Stanislav Nikolov, Rishub Jain, Jonas Adler, Trevor Back, Stig Petersen, David Reiman, Ellen Clancy, Michał Zielinski, Martin Steinegger, Michalina Pacholska, Tamas Berghammer, Sebastian Bodenstein, David Silver, Oriol Vinyals, Andrew W. Senior, Koray Kavukcuoglu, Pushmeet Kohli, and Demis Hassabis. Highly accurate protein structure prediction with AlphaFold. *Nature*, 596(7873):583–589, Aug 2021. ISSN 1476-4687. doi: 10.1038/s41586-021-03819-2.
- [15] W. Kahan. Pracniques: further remarks on reducing truncation errors. *Commun. ACM*, 8(1):40, January 1965. ISSN 0001-0782. doi: 10.1145/363707.363723.
- [16] George Em Karniadakis, Ioannis G. Kevrekidis, Lu Lu, Paris Perdikaris, Sifan Wang, and Liu Yang. Physics-informed machine learning. *Nature Reviews Physics*, 3(6):422–440, Jun 2021. ISSN 2522-5820. doi: 10.1038/s42254-021-00314-5.
- [17] Teddy Koker. e3nn.c, November 2024. URL <https://github.com/teddykoker/e3nn.c>.
- [18] Teddy Koker, Keegan Quigley, Eric Taw, Kevin Tibbetts, and Lin Li. Higher-order equivariant neural networks for charge density prediction in materials. *npj Computational Materials*, 10(1):161, Jul 2024. ISSN 2057-3960. doi: 10.1038/s41524-024-01343-1.
- [19] Risi Kondor and Erik Henning Thiede. Gelib, July 2024. URL <https://github.com/risi-kondor/Gelib>.
- [20] Risi Kondor, Zhen Lin, and Shubhendu Trivedi. Clebsch–Gordan nets: a fully Fourier space spherical convolutional neural network. In *Proceedings of the 32nd International Conference on Neural Information Processing Systems, NIPS'18*, page 10138–10147, Red Hook, NY, USA, 2018. Curran Associates Inc.
- [21] Neill Lambert, Eric Giguère, Paul Menczel, Boxi Li, Patrick Hopf, Gerardo Suárez, Marc Gali, Jake Lishman, Rushiraj Gadhvi, Rochisha Agarwal, Asier Galicia, Nathan Shammah, Paul D. Nation, J. R. Johansson, Shah Nawaz Ahmed, Simon Cross, Alexander Pitchford, and Franco Nori. QuTiP 5: The quantum toolbox in Python, 2024. URL <https://arxiv.org/abs/2412.04705>.
- [22] Kin Long Kelvin Lee, Mikhail Galkin, and Santiago Miret. Scaling computational performance of spherical harmonics kernels with Triton. In *AI for Accelerated Materials Design - Vienna 2024*, 2024.

- [23] Yi-Lun Liao and Tess Smidt. Equiformer: Equivariant graph attention transformer for 3D atomistic graphs. In *International Conference on Learning Representations*, 2023.
- [24] Lek-Heng Lim and Bradley J Nelson. What is... an equivariant neural network? *Notices of the American Mathematical Society*, 70(4):619–624, 4 2023. ISSN 0002-9920, 1088-9477. doi: 10.1090/noti2666.
- [25] Shengjie Luo, Tianlang Chen, and Aditi S. Krishnapriyan. Enabling efficient equivariant operations in the Fourier basis via gaunt tensor products. In *The Twelfth International Conference on Learning Representations*, 2024.
- [26] Albert Musaelian, Simon Batzner, Anders Johansson, Lixin Sun, Cameron J. Owen, Mordechai Kornbluth, and Boris Kozinsky. Learning local equivariant representations for large-scale atomistic dynamics. *Nature Communications*, 14(1):579, Feb 2023. ISSN 2041-1723. doi: 10.1038/s41467-023-36329-y.
- [27] Saro Passaro and C. Lawrence Zitnick. Reducing $SO(3)$ convolutions to $SO(2)$ for efficient equivariant GNNs. In Andreas Krause, Emma Brunskill, Kyunghyun Cho, Barbara Engelhardt, Sivan Sabato, and Jonathan Scarlett, editors, *Proceedings of the 40th International Conference on Machine Learning*, volume 202 of *Proceedings of Machine Learning Research*, pages 27420–27438. PMLR, 23–29 Jul 2023.
- [28] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay. Scikit-learn: Machine learning in Python. *Journal of Machine Learning Research*, 12:2825–2830, 2011.
- [29] Kristof Schütt, Oliver Unke, and Michael Gastegger. Equivariant message passing for the prediction of tensorial properties and molecular spectra. In Marina Meila and Tong Zhang, editors, *Proceedings of the 38th International Conference on Machine Learning*, volume 139 of *Proceedings of Machine Learning Research*, pages 9377–9388. PMLR, 18–24 Jul 2021.
- [30] Shaden Smith and George Karypis. Tensor-matrix products with a compressed sparse tensor. In *Proceedings of the 5th Workshop on Irregular Applications: Architectures and Algorithms*, New York, NY, USA, 2015. Association for Computing Machinery. ISBN 9781450340014. doi: 10.1145/2833179.2833183.
- [31] Chuin Wei Tan, Marc L. Descoteaux, Mit Kotak, Gabriel de Miranda Nascimento, Seán R. Kavanagh, Laura Zichi, Menghang Wang, Aadit Saluja, Yizhong R. Hu, Tess Smidt, Anders Johansson, William C. Witt, Boris Kozinsky, and Albert Musaelian. High-performance training and inference for deep equivariant interatomic potentials, 2025. URL <https://arxiv.org/abs/2504.16068>.
- [32] 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>.
- [33] Oliver T. Unke and Hartmut Maennel. E3x: E(3)-equivariant deep learning made easy, 2024. URL <https://arxiv.org/abs/2401.07595>.
- [34] Maurice Weiler, Mario Geiger, Max Welling, Wouter Boomsma, and Taco Cohen. 3D steerable cnns: learning rotationally equivariant features in volumetric data. In *Proceedings of the 32nd International Conference on Neural Information Processing Systems, NIPS’18*, page 10402–10413, Red Hook, NY, USA, 2018. Curran Associates Inc.
- [35] Samuel Williams, Andrew Waterman, and David Patterson. Roofline: an insightful visual performance model for multicore architectures. *Commun. ACM*, 52(4):65–76, April 2009. ISSN 0001-0782. doi: 10.1145/1498765.1498785.
- [36] YuQing Xie, Ameya Daigavane, Mit Kotak, and Tess Smidt. The price of freedom: Exploring trade-offs between expressivity and computational efficiency in equivariant tensor products. In *ICML 2024 Workshop on Geometry-grounded Representation Learning and Generative Modeling*, 6 2024.
- [37] Carl Yang, Aydın Buluç, and John D. Owens. Design principles for sparse matrix multiplication on the gpu. In *Euro-Par 2018: Parallel Processing: 24th International Conference on Parallel and Distributed Computing, Turin, Italy, August 27 - 31, 2018, Proceedings*, page 672–687, Berlin, Heidelberg, 2018. Springer-Verlag. ISBN 978-3-319-96982-4. doi: 10.1007/978-3-319-96983-1_48.