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

Vivek Bharadwaj, Austin Glover, Aydın Buluç, James Demmel

Presented by Nicole Hao

December 14, 2025

Overview

Geometric Deep Learning in Atomic Simulation

- Input: Atomic metadata and positions.
- Process: A message-passing graph neural network (MP-GNN).
- Output: 1. System energy (in milli-electronvolts, meV) 2. Atomic forces.

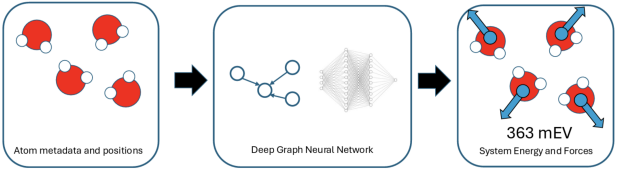


Figure: Interatomic Potential Calculation

Why do we need geometric deep learning in molecular simulations?

Geometric deep learning achieves state-of-the-art performance for fast interatomic potential calculation. A key primitive is the message-passing graph neural network.

Message-passing Graph Neural Network

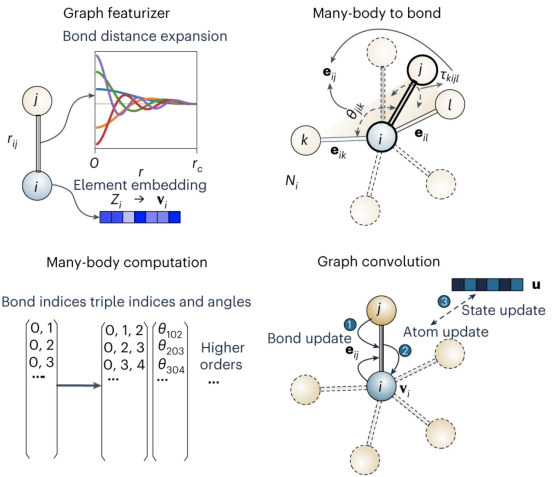


Figure: Schematic of the many-body graph potential [Chen and Ong, 2022]

O(3)-Equivariance & Respecting Physical Symmetries

- Message-passing GNNs generate messages for each node and edge.
- O(3)-Equivariance: if the input coordinate system rotates, the output energy stays the same, and the predicted forces rotate accordingly.
- Need to combine node and edge features in a highly structured, prescriptive manner.

Clebsch-Gordon (CG) Tensor Product

$$\mathbf{z} = \mathbf{W} \cdot \mathbf{P} \cdot (x \otimes y) = \mathbf{W} \sum_{i=1,j=1}^{m,n} x[i]y[j]\mathcal{P}[ij:]$$

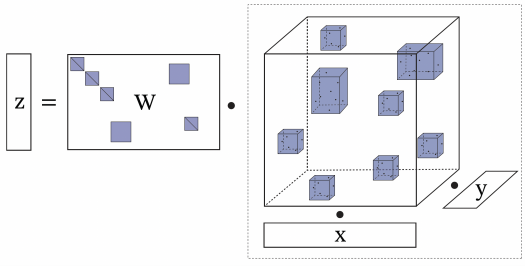


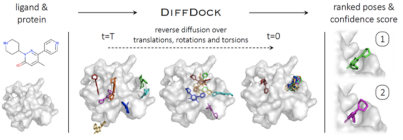
Figure: The CG tensor product, which contracts a block-sparse tensor **P** with two dense vectors to output a new vector. Each blue block is itself sparse. [Bharadwaj et al., 2025]

Bottlenecks to Scaling Equivariant Neural Networks

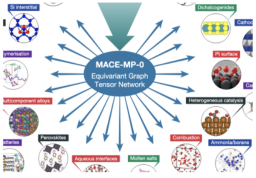
1. Computational cost: the CG tensor product and its derivatives must be evaluated millions of times on large atomic datasets.
2. Memory bottlenecks from computing large CG tensor products.
3. Existing libraries launch separate kernels for each computation.

Existing Implementations

(a) Nequip [Bat+22]



(b) DiffDock [Cor+23]



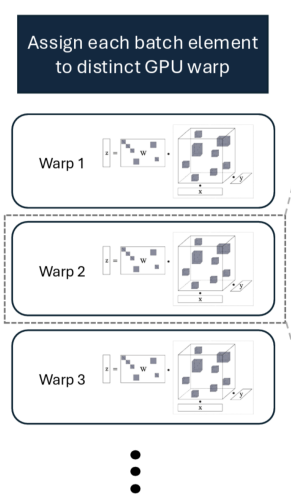
(c) MACE-MP0 [Bat+24]

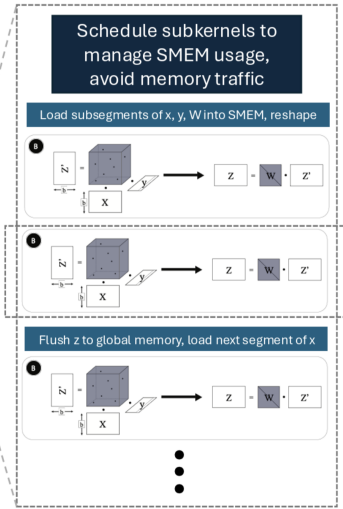
Solution: *OpenEquivariance*

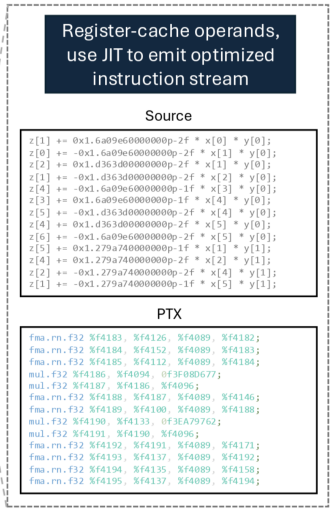
OpenEquivariance is an open source CUDA kernel generator for the CG tensor product and its gradients. It achieves 10x speedup over the e3nn package, and 2x over NVIDIA cuEquivariance.

- Exploits sparse structure in the multilinear map tensor via JIT (Just-In-Time Compilation), register caching, and loop unrolling.
- Sparsity-aware forward and backward passes.
- Fuses CG tensor product with graph convolution.

A Roadmap to Efficient CG Kernels







Warp-Level Forward Parallelism

Algorithm Subkernel C Warp-Level Algorithm

Require: $\boldsymbol{X} \in \mathbb{R}^{b' \times (2\ell_x+1)}, \boldsymbol{y} \in \mathbb{R}^{(2\ell_y+1)}, \boldsymbol{W} \in \mathbb{R}^{b \times b'}$
Require: Sparse tensor $\mathcal{P}^{(\ell_x,\ell_y,\ell_z)}$ for subkernel
  for $t = 1 \ldots b'$ **do** ▷ Parallel over threads
    Load $\boldsymbol{x}_{\text{reg}} = \boldsymbol{X}[t,:], \boldsymbol{y}_{\text{reg}} = \boldsymbol{y}$
    Initialize $\boldsymbol{z}_{\text{reg}} \in \mathbb{R}^{2\ell_z+1}$ to 0.

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

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

Gradient Calculations

- Given $\frac{\partial E}{\partial z}$, want to compute $\frac{\partial E}{\partial x}$, $\frac{\partial E}{\partial \mathbf{y}}$, $\frac{\partial E}{\partial \mathbf{W}}$.
- Recompute forward pass to avoid memory traffic.
- Weight matrix handled through warp matrix multiplication.

Algorithm Subkernel C Warp-Level Backward

Require: $\boldsymbol{X} \in \mathbb{R}^{b' \times (2\ell_x+1)}, \boldsymbol{y} \in \mathbb{R}^{2\ell_y+1}, \boldsymbol{W} \in \mathbb{R}^{b \times b'}$
Require: $\boldsymbol{G}_Z \in \mathbb{R}^{b \times (2\ell_z+1)}$, sparse tensor $\mathcal{P}^{(\ell_x,\ell_y,\ell_z)}$
Threads collaboratively compute $\boldsymbol{G}'_Z = \boldsymbol{W}^\top \cdot \boldsymbol{G}_Z$

for $t = 1 \ldots b'$ **do** ▷ Parallel over threads
Load $\boldsymbol{x}_{\text{reg}} = \boldsymbol{X}[t,:]$, $\boldsymbol{y}_{\text{reg}} = \boldsymbol{y}$, $\boldsymbol{g}'_{z\text{reg}} = \boldsymbol{G}'_Z[t:]$
Initialize $\boldsymbol{g}_{\text{x reg}}, \boldsymbol{g}_{\text{y reg}}, \boldsymbol{g}_{\text{w reg}}, \boldsymbol{z}_{\text{reg}}$ to 0.

for $(i,j,k,v) \in \text{nz}(\mathcal{P}^{(\ell_x,\ell_y,\ell_z)}$ **do** ▷ Unroll via JIT
$\boldsymbol{g}_{\text{x reg}}[i] \mathrel{+}= v \cdot \boldsymbol{y}_{\text{reg}}[j] \cdot \boldsymbol{g}'_{z\text{ reg}}[k]$
$\boldsymbol{g}_{\text{y reg}}[j] \mathrel{+}= v \cdot \boldsymbol{x}_{\text{reg}}[i] \cdot \boldsymbol{g}'_{z\text{ reg}}[k]$
$\boldsymbol{z}_{\text{reg}}[k] \mathrel{+}= v \cdot \boldsymbol{x}_{\text{reg}}[i] \cdot \boldsymbol{y}_{\text{reg}}[j]$

Store $\boldsymbol{g}_y = \text{warp-reduce}(\boldsymbol{g}_{\text{y reg}})$
Store $\boldsymbol{G}_x[t,:] = \boldsymbol{g}_{\text{x reg}}$ and $\boldsymbol{Z}'[t,:] = \boldsymbol{z}_{\text{reg}}$
Threads collaboratively compute $G_W = \boldsymbol{G}_Z \cdot (\boldsymbol{Z}')^\top$

Kernel Fusion for Graph Neural Networks (GNNs)

- Graph convolution has an SpMM memory access pattern.
- Fuse both kernels to save compute *and* memory.

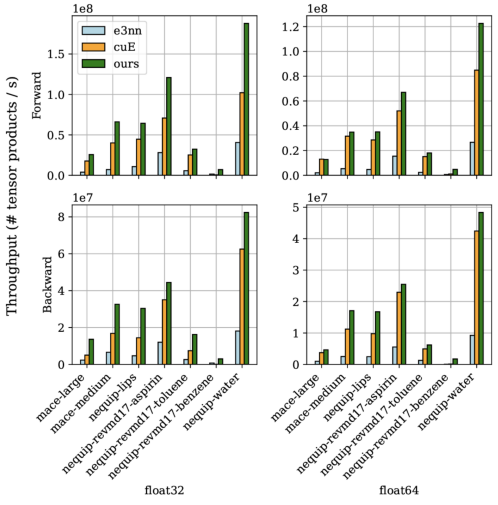
Algorithm Deterministic TP + Graph Convolution

Require: Graph $G = (V, E)$, $E[b] = (i_b, j_b)$
Require: Batch $\boldsymbol{x}_1, ..., \boldsymbol{x}_{|V|}, \boldsymbol{y}_1, ...\boldsymbol{y}_{|E|}, \boldsymbol{W}_1, ..., \boldsymbol{W}_{|E|}$
  for $\text{segment}_i \in \text{schedule}$ **do**
    $(s, t) = E[k][0], E[k][1]$
    Set $\boldsymbol{z}_{\text{acc}} = 0$ ▷ Parallel over Warps
    for $b = 1...|E|$ **do**
      Execute segment subkernel sequence
      $\boldsymbol{z}_{\text{acc}} \mathrel{+}= \boldsymbol{z}_{\text{smem}}$
      if $b = |E|$ or $s < E[b+1][0]$ **then**
        if s is first vertex processed by warp **then**
          Send $\boldsymbol{z}_{\text{acc}}$ to fixup buffer.
        else
          Store z_{acc} to global memory.
        $\boldsymbol{z}_{\text{acc}} = 0$
  Execute fixup kernel.

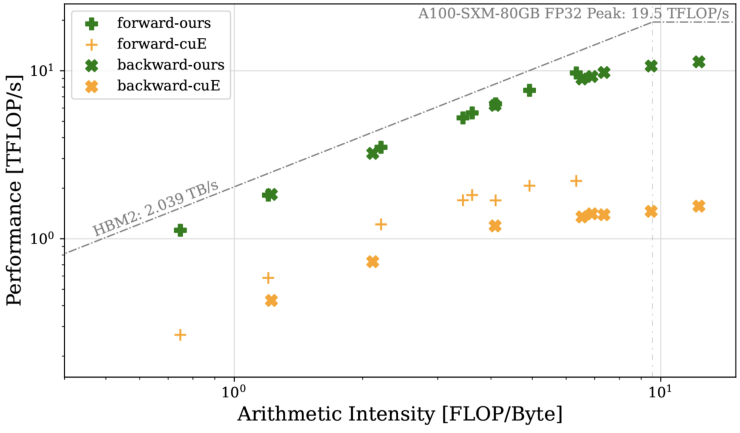
Experimental Results & Performance Gains

1. Throughput without Kernel Fusion.
2. Roofline Analysis.
3. Speedup with Kernel Fusion.
4. Acceleration of MACE Foundation Model.

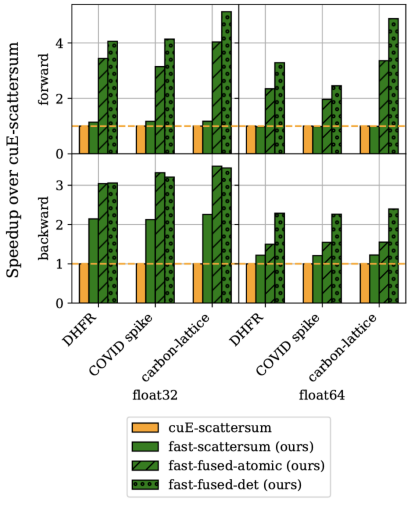
Throughput without Kernel Fusion



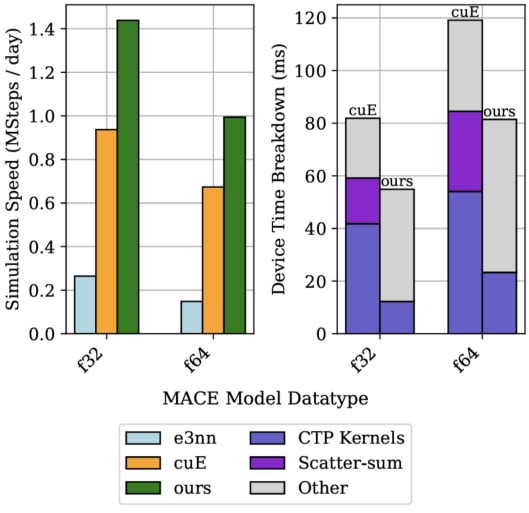
Roofline Analysis (F32)



Speedup with Kernel Fusion



Acceleration of MACE Foundation Model



Key Takeaways

- Sparse kernels achieve consistently high performance on the most common primitives used in equivariant deep neural networks.
- A sparsity-aware, JIT-compiled kernel is crucial.
- Future directions:
  - Reliance on the single-instruction multiple thread (SIMT) cores for FP32 and FP64 floating point arithmetic.
  - Expansion to specialized hardware for lower-precision computation.
  - Compile kernels to HIP for AMD GPUs.
  - Offer bindings for JAX and the Julia scientific computing language.

References


Bharadwaj, V., Glover, A., Buluc, A., and Demmel, J. (2025).
An efficient sparse kernel generator for o(3)-equivariant deep networks.


Chen, C. and Ong, S. (2022).
A universal graph deep learning interatomic potential for the periodic table.
Nature Computational Science, 2:718–728.