Simplifying Equivariant GPU Kernels through Tile-based Programming

by

Mit Kotak

B.S, Engineering Physics, University of Illinois at Urbana-Champaign (2023)

Submitted to the Center for Computational Science and Engineering
in partial fulfillment of the requirements for the degree of

MASTER OF SCIENCE IN COMPUTATIONAL SCIENCE AND ENGINEERING

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

September 2025

Authored by:	Mit Kotak Center for Computational Science and Enginering August 19, 2025
Certified by:	Tess E. Smidt Associate Professor of Electrical Engineering and Computer Science Thesis Supervisor
Accepted by:	Nicolas Hadjiconstantinou Professor of Mechanical Engineering Co-Director, Center for Computational Science and Engineering

Simplifying Equivariant GPU Kernels through Tile-based Programming

Mit Kotak

ABSTRACT

E(3)-equivariant neural networks have demonstrated success across a wide range of 3D modeling tasks. Until recently, they were bottlenecked due to their high memory and wall-time requirements. In this thesis we first provide an overview of recent GPU kernel efforts by both academia and industry that address this issue. These approaches tradeoff performance for engineering complexity, while still being algorithmically bottlenecked at 10 % GPU utilization. We instead trade off engineering complexity for performance. This not only lowers the barrier to GPU programming but also builds an abstraction layer to reason about future algorithmic innovations that can improve GPU utilization. Our kernel $B3$, based on tile-based programming implements all existing optimizations in just 100 lines of PyTorch-like code. We explore the performance-simplicity tradeoff with two case studies and demonstrate the practicality of our kernel workflow through downstream integration with a production model. We hope this work serves as inspiration to broaden and deepen existing equivariant kernel efforts.

Yes, there were times, I'm sure you knew
When I bit off more than I can could chew
But through it all, when there was doubt
I ate it up and spit out
I faced it all, and I stood tall
And did it my way

—Frank Sinatra

Acknowledgments

This thesis is a product of work I have done with collaborators at MIT and Harvard.

I would like to thank my advisors Prof. Tess Smidt and Prof. Saman Amarasinghe for the constant help and support. I would like to particularly thank Jaeyeon Won and Teo Collins for setting me up in this direction.

I would like to thank the Atomic Architects and the Coffeehouse Club for the mischief. I would also like to thank my Harvard collaborators Albert Musaelian and Chuin Wei Tan for testing out my crazy ideas.

I would like to thank the National Science Foundation (NSF) Graduate Research Fellowship (GRFP) program and the NSF AI Institute for Artificial Intelligence and Fundamental Interactions (IAIFI) for their funding.

To my family, thank you for the love and support.

Contents

Abstract 2

Acknowledgments 4

List of Figures 7

1 Introduction 10

1.1 Overview of Thesis 11

2 Background and Related Work 12

2.1 Equivariant Networks 12

2.1.1 Irreducible Representations of E(3) or Irreps 12

2.1.2 Spherical Harmonics 13

2.1.3 Clebsch Gordon Tensor Product 14

2.2 GPU Fundamentals 15

2.2.1 Memory Hierarchy 15

2.2.2 Software Hierarchy 15

2.2.3 Operator Fusion 15

2.2.4 Tiling Languages 16

2.3 Low GPU Utilization in Existing Kernels 16

2.3.1 What algorithmic factors affect GPU utilization ? 16

2.3.2 Roofline Analysis for Equivariant Kernels 18

3 Tiling Tensor Products 19

3.1 Naive Tensor Product Implementation 19

3.2 *B*3 Kernel Design 19

3.3 Case Studies 21

3.3.1 NequIP 21

3.3.2 Allegro 22

4 Experimental Results 24

4.1 Experiments 24

4.1.1 Microbenchmarks 24

4.1.2 End to End Benchmark 24

4.2 Results 24

5 Conclusion **30**

A Details for Reproducibility **31**

List of Figures

2.1 Roofline plots for NequIP (blue), Allegro (red) and MACE (green) forward kernels. C is the channel and lmax is the spherical harmonic degree. MHA-128 is Multi-Headed Attention with 128 query heads and MLA-128 is Multi-Latent Attention with 128 query heads. The arithmetic intensity varies from 10-30 with 12 TFLOPS/s being the maximum GPU utilization out of a full capacity for 156 TFLOPS/s 18

3.1 The tensor product's irregular structure which is not GPU friendly mapped onto a classic scatter form which is naively parallelizable. (Left) Dataflow of the tensor product between `1x0e + 1x1o` irrep with itself using the Clebsch-Gordon coefficients (green boxes) and weights (red). (Right) Gather-scatter version of the tensor product with flattened inputs, weights and Clebsch-Gordon coefficients indexed using the metadata. 20

4.1 *B*3's kernels for NequIP (left) and Allegro (right) in < 100 lines of Helion code . . 26

4.2 NequIP kernels forward walltime speedup over e3nn. *B*3 (Helion) is 3x faster than e3nn and 2-4x slower than OpenEquivariance (CUDA) 27

4.3 Allegro kernels forward walltime speedup over e3nn. Both Helion and Triton implementations of *B*3 are comparable and are 5-10x faster over e3nn while being 2-10x slower than cuEquivariance (CUDA) 28

4.4 Allegro end-to-end inference speedups over e3nn. Upto 3x faster than e3nn and 2x slower than cuEquivariance (CUDA) 29

List of Algorithms

1 Naive Tensor Product Implementation in e3nn [13] 19
2 *B*3 Clebsch-Gordon Implementation 21
3 Naive NequIP in e3nn [13] 21
4 *B*3 NequIP Implementation 22
5 Naive Allegro in e3nn [13] 23
6 *B*3 Allegro Implementation 23

Chapter 1

Introduction

In the physical world, many systems exhibit fundamental symmetries. For example, the laws of physics remain unchanged if we rotate a molecule in 3D space, translate it to a different location, or reflect it through a mirror. Spatial symmetries are a useful inductive bias while modeling complex physical systems. Incorporating them into neural networks has been shown to significantly improve both learning efficiency and robustness [4, 11, 28, 31]. To address the specific symmetries present in 3D systems, considerable effort has been dedicated to the development of $E(3)$-equivariant neural networks [18, 19, 36, 38]. Equivariant networks have delivered strong performance across a wide range of scientific applications, including molecular force fields [3, 4, 25], catalyst discovery [22], generative models [15], charge density prediction [12], and protein structure prediction [17, 20].

Despite their robustness and practical success, these networks have been computationally bottlenecked by their high compute and memory requirements. While traditional networks are largely made up of matrix multiplication operations, these networks have much more complicated operations that don't map easily onto modern GPUs. This has naturally inspired a lot of work by both academia and industry in writing custom GPU kernels for these networks[5, 10, 21, 26]. While these kernels have helped bring down compute requirements, their engineering complexity has still limited their impact to a small class of equivariant architectures. Existing kernels are also algorithmically bottlenecked at only being able to use 10% of the available GPU compute. There is a need for both broadening as well deepening existing kernel efforts.

In this thesis we take the first step in this direction by *simplifying* the existing kernel workflow. Rather than adding more complexity to squeeze out additional performance, we focus on making the development process more accessible while still retaining the level of GPU control needed to implement *all* existing optimizations. Our approach draws inspiration from the popular FlashAttention kernel [6], whose key optimizations can be implemented within 100 lines[9].

We ask: **Can we write a 100-line simplified equivariant kernel that captures *all* known optimizations while remaining accessible to non-GPU experts ?**

We introduce $B3$[1], a kernel built on the tiling-based programming paradigm. Tiling-based languages such as Triton[37] have become the defacto choice for non-GPU experts due to their accessible GPU programming model. $B3$ preserves this accessibility by translating existing optimizations into a form that can be implemented using a tiling language. This is done by introducing additional metadata at runtime. By carefully broadcasting the inputs using this metadata, the engineering complexity reduces to 100 lines of PyTorch-like code. We then explore the performance-simplicity tradeoff for NequIP[4] and Allegro[25] kernels through microbenchmarks, and show end to end benchmarks for Allegro.

1.1 Overview of Thesis

Chapter 2 provides a background on GPUs and equivariant networks partly adapted from [39] that I co-authored and presented at ICML 2025. Chapter 2 also dives into the GPU utilization issue that motivates the direction of this thesis. Chapter 3 introduces $B3$ and dives into its implementation with case studies. Chapter 4 highlights kernel micro benchmarks and end to end integration results that I integrated into [34] which is currently under review.

[1] $B3$ is named after the 3 input broadcasts in the kernel and also happens to be my favorite ice-cream at Toscanini's

Chapter 2

Background and Related Work

2.1 Equivariant Networks

2.1.1 Irreducible Representations of E(3) or Irreps

We give a brief overview of representation theory since irreps form the fundamental datatype and dictate the rules for manipulating the tensors in these networks.

A representation ρ of a group G maps each group element g to a bijective linear transformation $\rho(g) \in GL(V)$, where V is some vector space. Representations must preserve the group multiplication property

$$\rho(g \cdot h) = \rho(g) \circ \rho(h) \quad \forall g, h \in G \tag{2.1}$$

Thus, the representation ρ defines a group action on a vector space V. The dimension of the representation ρ is simply defined as the dimension of the vector space V.

There may be subspaces $W \subset V$ which are left invariant under actions of $\rho(g)$ for all $g \in G$. If this is the case, then restricting to W also gives a representation $\rho|_W(g) \in \mathrm{GL}(W)$. If there is no nontrivial W, then we say the representation ρ is an irreducible representation (irrep).

Because E(3) is not a compact group, the usual approach is to consider irreps of the group SO(3) of 3D rotations, and compose them with the representation in which translations act as the identity.

$$\rho(R, T) = \rho'(R) \tag{2.2}$$

This is why translations are often handled in $E(3)$-equivariant neural networks by centering the system or only using relative vectors.

The 'scalar' representation ρ_{scalar} representation of $SO(3)$ is defined as:

$$\rho_{\text{scalar}}(R) = \text{id} \quad \forall R \in SO(3) \tag{2.3}$$

and is of dimension 1 over $V = \mathbf{R}$. Elements of $\mathbf{R}$ are unchanged by any rotation R. We call such elements 'scalars' to indicate that they transform under the 'scalar' representation of $SO(3)$. An example of a 'scalar' element could be energy of an atom, which does not change under rotation of coordinate frames.

Let $T(R) \in \mathbf{R}^{3\times3}$ be the rotation matrix corresponding to a rotation $R \in SO(3)$. Then, the 'vector' representation of $SO(3)$ is defined as:

$$\rho_{\text{vector}}(R) = T(R) \quad \forall R \in SO(3) \tag{2.4}$$

and is of dimension 3 over $V = \mathbf{R}^3$. The name arises from the way vectors in $\mathbf{R}^3$ transform under a rotation of the coordinate frame. We call such elements 'vectors' to indicate that they transform under the 'vector' representation of $SO(3)$. For example, the force and position of an object in a certain coordinate frame are 'vectors'.

Weyl's theorem for the Lie group $SO(3)$ states that all finite-dimensional representations of $SO(3)$ are equivalent to direct sums of irreps. The irreps of $SO(3)$ are indexed by an integer $\ell \geq 0$, with dimension $2\ell + 1$. $\ell = 0$ corresponds to the 'scalar' representation, while $\ell = 1$ corresponds to the 'vector' representation above. We will often use m, where $-\ell \leq m \leq \ell$, to index of each of the $2\ell + 1$ components.

We say that a quantity $\mathbf{x} \in \mathbf{R}^{2\ell+1}$ is a ℓ irrep, if it transforms as the irrep of $SO(3)$ indexed by ℓ. If $\mathbf{x}_1$ is a ℓ_1 irrep and $\mathbf{x}_2$ is an ℓ_2 irrep, we say that $(\mathbf{x}_1, \mathbf{x}_2)$ is a direct sum of ℓ_1 and ℓ_2 irreps, which we call a (ℓ_1, ℓ_2) rep. Weyl's theorem states that all reps are a direct sum of ℓ_i irreps, possibly with repeats over ℓ_i: $\mathbf{x} = \oplus_{\ell_i} \mathbf{x}^{(\ell_i)}$. The multiplicity of an irrep in a rep is exactly the number of repeats.

We follow the `e3nn`[13] notation for representing irreps. For example, `64x0e + 64x1o` refers to the direct sum of a scalar `0e` and a vector `1o` irreps with multiplicity 64. If we wanted a psuedo-vector instead of vector, the representation would be `64x0e + 64x1e`.

2.1.2 Spherical Harmonics

The spherical harmonics are intimately connected to the representations of $SO(3)$ and play a key role in featurizing the geometry in $E(3)$ networks.

We define the spherical coordinates (r, θ, φ) as:

$$\begin{bmatrix} x \\ y \\ z \end{bmatrix} = \begin{bmatrix} r \sin\theta \cos\varphi \\ r \sin\theta \sin\varphi \\ r \cos\theta \end{bmatrix} \tag{2.5}$$

for $\theta \in [0, \pi), \varphi \in [0, 2\pi)$.

The spherical harmonics $Y_{\ell,m}$ are a set of functions $S^2 \rightarrow \mathbf{R}$ indexed by (ℓ, m), where again $\ell \geq 0, -\ell \leq m \leq \ell$. Here, $S^2 = \{(r, \theta, \phi) \mid r = 1\}$ denotes the unit sphere.

Indeed, as suggested by the notation, the spherical harmonics are closely related to the irreducible representations of $SO(3)$. Let Y_ℓ be the concatenation of all $Y_{\ell,m}$ over all m for a given ℓ:

$$Y_\ell(\theta, \phi) = \begin{bmatrix} Y_{\ell,-\ell}(\theta, \phi) \\ Y_{\ell,-\ell+1}(\theta, \phi) \\ \cdots \\ Y_{\ell,\ell}(\theta, \phi) \end{bmatrix} \tag{2.6}$$

When we transform the inputs to $Y_\ell(\theta, \phi)$, the output transforms as a ℓ irrep.

The spherical harmonics satisfy orthogonality conditions:

$$\int_{S^2} Y_{\ell_1,m_1} \cdot Y_{\ell_2,m_2} \, dS^2 = \delta_{\ell_1\ell_2}\delta_{m_1m_2} \tag{2.7}$$

where:

$$\int_{S^2} f \cdot g \, dS^2 = \int_{\theta=0}^{\pi} \int_{\varphi=0}^{2\pi} f(\theta,\varphi)g(\theta,\varphi) \sin\theta d\theta d\varphi \tag{2.8}$$

The orthogonality property allows us to treat the spherical harmonics as a basis for functions on S^2. We can linearly combine the spherical harmonics using irreps to approximate arbitrary functions on the sphere. Given a $(0, 1, \ldots, L)$ rep $\mathbf{x} = (\mathbf{x}^{(0)}, \mathbf{x}^{(1)}, \ldots, \mathbf{x}^{(L)})$, we can associate the function $f_{\mathbf{x}} : S^2 \longrightarrow \mathbf{R}$ as:

$$f_{\mathbf{x}}(\theta,\varphi) = \sum_{\ell=0}^{L} \sum_{m=-\ell}^{\ell} \mathbf{x}_m^{(\ell)} Y_{\ell,m}(\theta,\varphi) \tag{2.9}$$

The function $f_{\mathbf{x}}$ is uniquely determined by $\mathbf{x}$. In particular, by the orthogonality of the spherical harmonics (Equation 2.7), we can recover the $\mathbf{x}_m^{(\ell)}$ component:

$$\mathbf{x}_m^{(\ell)} = \int_{S^2} f_{\mathbf{x}} \cdot Y_{\ell,m} \, dS^2 \tag{2.10}$$

Thus, we can define the operations ToSphere and FromSphere:

$$\mathbf{x} \xrightarrow{\text{ToSphere}} f_{\mathbf{x}} \xrightarrow{\text{FromSphere}} \mathbf{x} \tag{2.11}$$

2.1.3 Clebsch Gordon Tensor Product

The most natural map is $V \times W \longrightarrow V \otimes W$ constructed by taking an outer product of the inputs. If the inputs are explicitly written as a direct sum of irreps, we can write the tensor product as

$$\mathbf{x} \otimes \mathbf{y} = \bigoplus_{\substack{\mathbf{x}^{(\ell_1)} \in \mathbf{x} \\ \mathbf{y}^{(\ell_2)} \in \mathbf{y}}} (\mathbf{x}^{(\ell_1)} \otimes \mathbf{y}^{(\ell_2)}) \tag{2.12}$$

a new basis which is the sum of tensor product reps.

The key idea of a Clebsch-Gordan tensor product is we can explicitly reduce the tensor product reps back into a direct sum of irreps with a change of basis. This change of basis is the definition of the Clebsch-Gordan coefficients, giving us

$$\mathbf{x}^{(\ell_1)} \otimes \mathbf{y}^{(\ell_2)} = \bigoplus_{\ell_3} (\mathbf{x}^{(\ell_1)} \otimes_{\text{CG}} \mathbf{y}^{(\ell_2)})^{(\ell_3)} \tag{2.13}$$

where

$$\begin{aligned} &(\mathbf{x}^{(\ell_1)} \otimes_{\text{CG}} \mathbf{y}^{(\ell_2)})_{m_3}^{(\ell_3)} \\ &= \sum_{m_1=-\ell_1}^{\ell_1} \sum_{m_2=-\ell_2}^{\ell_2} C_{\ell_1,m_1,\ell_2,m_2}^{\ell_3,m_3} \mathbf{x}_{m_1}^{(\ell_1)} \mathbf{y}_{m_2}^{(\ell_2)}. \end{aligned} \tag{2.14}$$

with the weighted version being

$$\mathbf{x}^{(\ell_1)} \otimes \mathbf{y}^{(\ell_2)} = \bigoplus_{\ell_3} w_{\ell 1,\ell 2} (\mathbf{x}^{(\ell_1)} \otimes_{\text{CG}} \mathbf{y}^{(\ell_2)})^{(\ell_3)} \tag{2.15}$$

Borrowing terminology from `e3nn`[13], we define 'paths' as valid bilinear, equivariant maps between $\ell 1, \ell 2, \ell 3$ that satisy the selection rules

$$|\ell 1 - \ell 2| \leq \ell 3 \leq \ell 1 + \ell 2 \tag{2.16}$$

2.2 GPU Fundamentals

We provide a brief overview of the modern GPU architecture and refer the curious reader to [2, 24, 33].

2.2.1 Memory Hierarchy

GPU memory can be broken down into off-chip global memory and on-chip cache memory. There's a fundamental asymmetry between the size and latencies of these two memories. For example, an NVIDIA A100 has 40-80 GB of global memory with bandwidth 1.5-2.0 TB/s and 192 KB of on-chip memory with bandwidth 19 TB/s. The on-chip memory is an order of magnitude faster than global memory and a couple orders of magnitude smaller than global memory. Carefully managing this scarce but fast on-chip memory is crucial for getting good kernel performance.

2.2.2 Software Hierarchy

Programs on a GPU are executed as kernels. A kernel loads data from global memory onto on-chip memory, performs work on it, and writes output back to global memory. The kernel is further decomposed into thread blocks which execute on streaming multiprocessors (SMs) that are roughly analogous to the cores on a CPU. Each SM in turn has 4 warp schedulers that are responsible for executing a warp. A warp is a collection of 32 threads executing the same instruction per cycle. The popular CUDA programming model (and its equivalents) offers explicit control at the block-level, warp-level and thread-level. To give a sense of the scale of parallelism that's available per cycle, an H100 has 132 SMs giving us 132 * 4 * 32 > 16,000 parallel threads.

2.2.3 Operator Fusion

Given that memory resources are slower and more scare than compute resources, we want to perform as much computation as possible on the data we bring onto the chip. This is done through an optimization called kernel fusion, which combines two or more kernels into a single kernel. For ML workloads, there are two main types of kernel fusion. Pointwise fusions that combine element-wise operations such as activations or norms into a matrix operation. These types of fusions are commonly automated through compiler passes [1, 32] or C++ templates [35]. The second type involves restructuring the algorithm to eliminate the need for writing back to global memory. For example, FlashAttention fuses the QK^T computation, softmax calculation and PV computation into a single kernel. Automating these fusions is an active area of research.

2.2.4 Tiling Languages

One common way to implement kernel fusion is through tiling which slices up the tensors to fit them into faster memory caches. Different tiling languages offer different levels of control over how these tiles get mapped onto the underlying hardware.

Python embedded languages: While tiling has historically been a crucial part of the ML-compiler stack, it was made popular within the ML community with Triton[37]. Triton strikes a balance between performance and productivity by aggressively specializing on dense block computations (e.g attention, matrix-multiplications). It exposes explicit control only at the block-level, letting its autotuner pick the best configurations at the warp and thread level. While this introduces a performance tradeoff since the most optimized implementation go down to thread-level, the programming experience is drastically simplified with the user only worrying about how to tile their computation. Another language which takes this high level tiling paradigm even further is Helion[30]. While Triton automatically makes a lot of low-level decisions compared to CUDA, the user still has to compute all of the indices and offsets for the tiles which is error-prone and has performance trade-offs that are not immediately obvious. Helion automates even this part by directly tracing PyTorch code and automatically generating Triton. It also comes with a much more powerful autotuner that can search over multiple valid Triton programs with different memory layouts and code structures. On the other end of the spectrum, we have Pallas [16] and Gluon [**?**] that go lower than Triton. These languages are motivated by the need for deeper pipelining and asynchronous programming in post-Hopper GPU architectures.

C++ embedded languages Among C++ languages we have CuTe[27] and CUTLASS[**?**]. Unlike Triton and Helion, they offer control at the warp and thread-level, making them the current "workhorse" for implementing state of the art kernels.

2.3 Low GPU Utilization in Existing Kernels

2.3.1 What algorithmic factors affect GPU utilization ?

In order to understand the GPU utilization in existing kernels let's start with a simple question: What does it take to hit close to 100 % GPU utilzation ?

Every algorithm's GPU wall-time can be roughly broken down time spent doing compute and time spent moving data across various memory-bandwidths. These bandwidths are distributed both within a GPU and across GPUs. For our use case, we only need to look at single GPU. Thus, borrowing notation from [2], we can define these times as

$$T_{math} = \frac{\text{Computation FLOPS}}{\text{GPU FLOPS/s}} \tag{2.17}$$

$$T_{mem} = \frac{\text{Memory Bytes}}{\text{GPU Bandwidth Bytes/s}} \tag{2.18}$$

Typically (but not always), for a single GPU, computation can be overlapped with data movement. This means we can lower bound the wall time by using the maximum of computation and data movement time.

$$T_{lower} = \max(T_{math}, T_{mem}) \tag{2.19}$$

If we assume we can perfectly overlap data movement and computation, when $T_{math} T_{mem}$, we see full GPU utilization. We call this being "compute-bound". $T_{mem} T_{math}$, we tend to be "memory-bound" and at least some fraction of our GPU FLOPS/s is wasted waiting for data to be passed around. One way to tell if an algorithm will be compute or memory-bound is to look at its "arithmetic intensity".

The arithmetic intensity of an algorithm is given by the ratio of the total FLOPS it performs to the total bytes it needs to move within a GPU or across GPUs.

$$\text{Arithmetic Intensity} = \frac{\text{Computation FLOPS}}{\text{Memory Bytes}} \tag{2.20}$$

Arithmetic intensity measures the "FLOPS per byte" of a given operation. When arithmetic intensity is low (e.g. ReLU), we spend most of the time moving memory to and from the compute cores, and waste available FLOPS. Once the arithmetic intensity is high (e.g. matrix multiplication), we typically spend more time in compute T_{comm} than memory transfers T_{mem}. The point where this crossover happens is the "ridge point" of of our GPU, the ratio of peak FLOPS/s to peak GB/s.

$$\begin{aligned}
T_{math} > T_{mem} &\Leftrightarrow \frac{\text{Computation FLOPs}}{\text{Accelerator FLOPs/s}} > \frac{\text{Communication Bytes}}{\text{Bandwidth Bytes/s}} \\
&\Leftrightarrow \frac{\text{Computation FLOPs}}{\text{Memory Bytes}} > \frac{\text{GPU FLOPs/s}}{\text{GPU Bandwidth Bytes/s}} \\
&\Leftrightarrow \text{Intensity(Computation)} > \text{Intensity(GPU)}
\end{aligned}$$

Intensity(GPU) is the arithmetic intensity at which our accelerator achieves its peak FLOPS/s. For an A100 this is about 153 FLOPS/byte. That means if an algorithm has lower arithmetic intensity than 153, it will be bound by byte loading and thus won't make good use of our hardware.

We can visualize this tradeoff between memory and compute using a roofline plot, which plots peak achievable FLOPS/s (throughput) of an algorithm on our hardware (y-axis) against the arithmetic intensity of that algorithm (x-axis).

2.3.2 Roofline Analysis for Equivariant Kernels

We appply the roofline analysis to equivariant tensor product kernels for NequIP[4], Allegro[25] and MACE[3]. We profile cuEquivariance[26] kernels for MACE, Allegro and OpenEquivariance[5] kernels for NequIP. While there are other implementations[10, 21], since all of them implement the same set of optimizations, we went for ones that were readily available from the code repositories. Moreover, here we were more interested in the peak GPU utilization set by the arithmetic intensity of different tensor products rather then any specific kernel GPU utilization.

In Figure 2.1 we see that existing kernels are very close to the FP32 roofline with a maximum arithmetic intensity of 30. FP32 FLOPS unfortunately takes up < 10% of the available compute on modern GPUs. Thus, there is a need for increasing the FLOPS allocated to matrix multiplication-style operations as well as the increasing the arithmetic intensity, both by an order of magnitude to hit GPU utilization levels similar to attention. We highlight the algorithmic progess in improving attention's GPU utilization by plotting the increase in GPU utilization from the original 2022 Multiheaded Attention (MHA) kernel[6] to the latest DeepSeek's Multi-latent attention (MLA)[7].

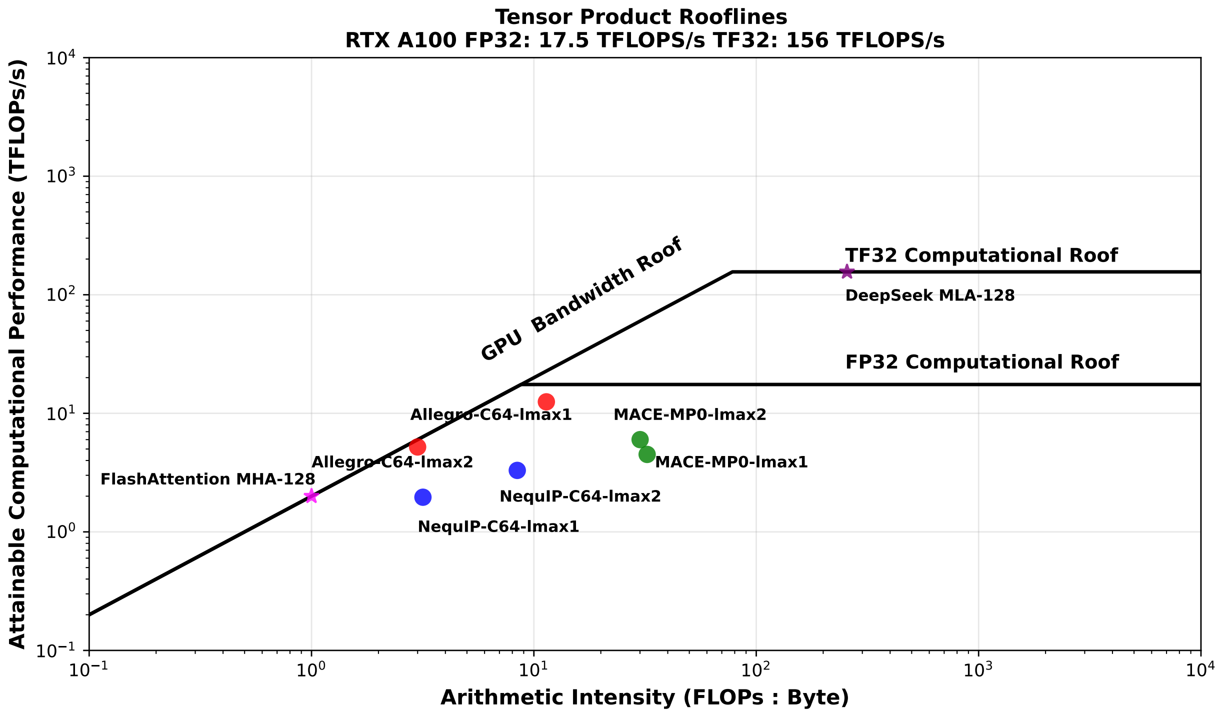

Figure 2.1: Roofline plots for NequIP (blue), Allegro (red) and MACE (green) forward kernels. C is the channel and lmax is the spherical harmonic degree. MHA-128 is Multi-Headed Attention with 128 query heads and MLA-128 is Multi-Latent Attention with 128 query heads. The arithmetic intensity varies from 10-30 with 12 TFLOPS/s being the maximum GPU utilization out of a full capacity for 156 TFLOPS/s

Chapter 3

Tiling Tensor Products

3.1 Naive Tensor Product Implementation

We first look at a naive tensor product algorithm as implemented in e3nn [13]. `e3nn`[13] provides functionality to specify what paths the user wants between their input and output irreps, and how to parameterize them. During runtime, the tensor product algorithm then loops over all of the paths specified by the user.

In the naive implementation, we pass in edge embedding X_{E,C,L_x}, spherical harmonic embedding Y_{E,L_y}, edge radial weight $W_{E,C,P}$ after running through an equivariant MLP, Clebsch-Gordon tensors $CG_{l_x,l_y,l_o,p}$ as inputs, and get edge embedding O_{E,C,L_o} as the output.

E is the number of edges, C is the number of channels, L_x, L_y, L_o are irrep dimensions, and P is the number of paths in the tensor product.

Algorithm 1 Naive Tensor Product Implementation in e3nn [13]

```
Input: $\mathbf{X} \in \mathbb{R}^{E\times C\times L_x}, \mathbf{Y} \in \mathbb{R}^{E\times L_y}, \mathbf{W} \in \mathbb{R}^{E\times C\times P}, \mathbf{CG} \in \mathbb{R}^{L_x\times L_y\times L_o\times P}$
Output: Output $\mathbf{O} \in \mathbb{R}^{E\times C\times L_o}$
  # $p$ is path indexing variable
  $p = 0$
  for $l_x, l_y, l_o$ in valid tensor product paths do
      Load $X_{E,C,l_x}, Y_{E,l_y}, CG_{l_x,l_y,l_o,p}, W_{E,C,p}$ from GMEM
      Compute $O_{E,C,l_o} = \sum_{l_x} \sum_{l_y} X_{E,C,l_x} \times Y_{E,l_y} \times CG_{l_x,l_y,l_o,p} \times W_{E,C,p}$
      Write $O_{E,C,l_o}$ to GMEM
      $p+ = 1$
  Return $\mathbf{O}$
```

3.2 $B3$ Kernel Design

Similar to FlashAttention, the goal is to reduce I/O traffic between on-chip and off-chip memories by fusing all the operations into a single kernel.

Parallelization Axes: We observe that the naive tensor product can be parallelized at two levels:

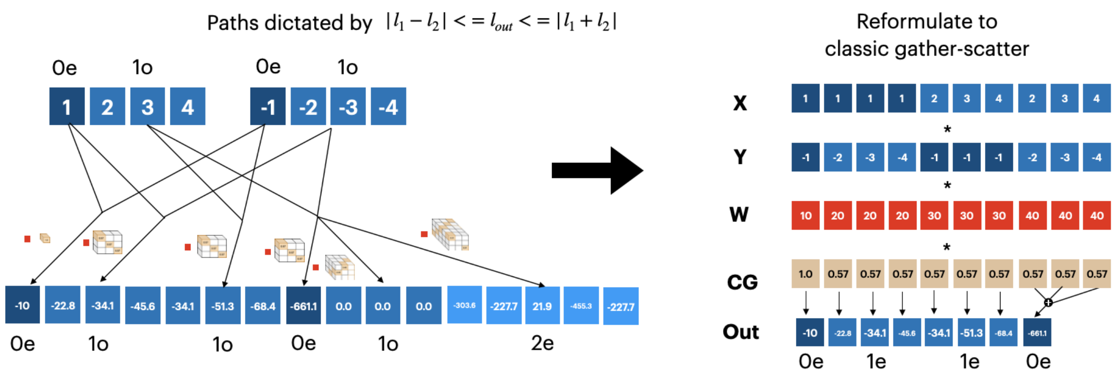
Paths dictated by $|l_1 - l_2| <= l_{out} <= |l_1 + l_2|$
Reformulate to classic gather-scatter
0e
1o
0e
1o
X
Y
W
CG
Out
0e
1o
1o
0e
1o
2e
0e
1e
1e
0e

Figure 3.1: The tensor product's irregular structure which is not GPU friendly mapped onto a classic scatter form which is naively parallelizable. (Left) Dataflow of the tensor product between `1x0e + 1x1o` irrep with itself using the Clebsch-Gordon coefficients (green boxes) and weights (red). (Right) Gather-scatter version of the tensor product with flattened inputs, weights and Clebsch-Gordon coefficients indexed using the metadata.

1. *Path level*: Every path can be independently scheduled onto different thread blocks.
2. *Coefficient level*: Every path has a CG_{ijk} tensor that is reduced over i, j through a dense BLAS operation. The sparsity of CG_{ijk} (due to selection rules) can be leveraged by converting to a sparse COO tensor $CGvals \in \mathbb{R}^{NNZ}$ and then accumulating i, j over the non-zero Clebsch-Gordon coefficients at the thread-level.

Tiling Strategy: $B3$'s tiling strategy is built on these insights. Tiling over the edge EE E and channel C dimensions is straightforward. The challenge lies in tiling over different paths which read and write back to variable slices of $\mathbf{X}, \mathbf{Y}, \mathbf{W}, \mathbf{CG}, \mathbf{O}$. This complicates the engineering since the slices need to fit onto limited on-chip shared memory. Existing works schedule the paths onto hardware resources either explicitly in their kernel or by generating the kernel at compile time. $B3$ instead moves the complexity from compile time to runtime by passing in additional metadata, letting the tiling language's heuristics handle the low-level hardware scheduling. This allows $B3$ to exploit path-level parallelization, and coefficient-level parallelization inside the tile-block.

Gather-Scatter Reformulation: $B3$'s tiling strategy reformulates the tensor product into a classic gather-scatter form 3.1. The metadata maps L_x, L_y, P dimensions onto NNZ and then back to L_o. More concretely, $M^{L_x \rightarrow NNZ}$, $M^{L_y \rightarrow NNZ}$, $M^{P \rightarrow NNZ}$ specify how to map the inputs to NNZ dimension (gather), and then $indptr^{NNZ \rightarrow L_o}$ specifies the NNZ slices that are accumulated over to map back to the output (scatter).

Algorithm 2 *B*3 Clebsch-Gordon Implementation

```
Input: $\mathbf{X} \in \mathbb{R}^{E\times C\times L_x}$, $\mathbf{Y} \in \mathbb{R}^{E\times L_y}$, $\mathbf{W} \in \mathbb{R}^{E\times C\times P}$, $\mathbf{CGvals} \in \mathbb{R}^{NNZ}$,
Input: Metadata $M^{L_x\to NNZ}$, $M^{L_y\to NNZ}$, $M^{P\to NNZ}$, $indptr^{(NNZ+1)\to L_o}$
Input: Twiddle factors $E$ tile size $E_{tile}$, $C$ tile size $C_{tile}$, $L_o$ tile size $L_{tile}$.
Output: Output $\mathbf{O} \in \mathbb{R}^{E\times C\times L_o}$
  for SMs in parallel across $E/E_{tile} \times C/C_{tile} \times L_o/L_{tile}$ do
     Load $indptr^{NNZ\to L_o}_{L_{tile}}$
     for $nnz_{tile}$ indices in $indptr^{NNZ\to L_o}_{L_{tile}}$ do
        Load mappers $M^{L_x\to NNZ}_{nnz_{tile}}$, $M^{L_y\to NNZ}_{nnz_{tile}}$, $M^{P\to NNZ}_{nnz_{tile}}$ from GMEM to SMEM.
        # Gather: $L_{tile} \longrightarrow nnz_{tile}$
        Load $X_{E_{tile}C_{tile}L_{tile}\to nnz_{tile}}$, $Y_{E_{tile}L_{tile}\to nnz_{tile}}$, $CG_{nnz_{tile}}$, $W_{E_{tile}C_{tile}L_{tile}\to nnz_{tile}}$
        from GMEM to SMEM.
        # Already in $nnz_{tile}$
        Load $CGvals_{nnz_{tile}}$.
        # Scatter: $nnz_{tile} \longrightarrow L_{tile}$
        On-chip, compute $O_{E_{tile}C_{tile}L_{tile}} += X_{E_{tile}C_{tile}nnz_{tile}} \times Y_{E_{tile}nnz_{tile}} \times CG_{nnz_{tile}} \times W_{E_{tile}C_{tile}nnz_{tile}}$.
     Write $O_{E_{tile}C_{tile}L_{tile}}$ to GMEM
  Return $\mathbf{O}$
```

3.3 Case Studies

3.3.1 NequIP

For NequIP, the core algorithm remains the same though now with a different list of paths. An additional optimization involves fusing with the message passing step that does a gather-scatter between nodes and edge for the input X_{N,C,L_x} and output O_{N,C,L_y} features.

Algorithm 3 Naive NequIP in e3nn [13]

```
Input: $\mathbf{X} \in \mathbb{R}^{N\times C\times L_x}$, $\mathbf{Y} \in \mathbb{R}^{E\times L_y}$, $\mathbf{W} \in \mathbb{R}^{E\times C\times P}$, $\mathbf{CG} \in \mathbb{R}^{L_x\times L_y\times L_o\times P}$, $\mathbf{src} \in \mathbb{R}^{E}$, $\mathbf{dst} \in \mathbb{R}^{E}$
Output: Output $\mathbf{O} \in \mathbb{R}^{N\times C\times L_o}$
  for $n$ indices in src do
     # $p$ is path indexing variable
     $p = 0$
     for $l_x, l_y, l_o$ in valid tensor product paths do
        # Gather $n \longrightarrow e$, $L_{tile} \longrightarrow nnz_{tile}$
        Load $X_{N\to E,C,l_x}$, $Y_{E,l_y}$, $CG_{l_x,l_y,l_o,p}$, $W_{E,C,p}$ from GMEM
        Compute $O_{E,C,l_o} = \sum_{l_x}\sum_{l_y} X_{E,C,l_x} \times Y_{E,l_y} \times CG_{l_x,l_y,l_o,p} \times W_{E,C,p}$
        # Scatter $e \longrightarrow n$, $nnz_{tile} \longrightarrow L_{tile}$
        Write $O_{E->N,C,l_o}$ to GMEM at $dst$ indices
        $p += 1$
  Return $\mathbf{O}$
```

We highlight the modifications from $B3$ tensor product algorithm to $B3$'s NequIP version in green. $B3$'s NequIP version naively parallelize over the edge E and channel C dimemsions as before and uses the mappers M's to handle the tiling across the irreps dimensions. The message passing from nodes N to edges E is handled inside the tensor product through edge lists src and dst each containing node indices. src is used to move the X node embedding to edge embedding before doing the tensor product, and dst to write the edge output back to the nodes. This introduces a double gather-scatter in the algorithm since the tensor product is also gather-scatter based.

Algorithm 4 $B3$ NequIP Implementation

```
Input: $\mathbf{X} \in \mathbb{R}^{E\times C\times L_x}$, $\mathbf{Y} \in \mathbb{R}^{E\times L_y}$, $\mathbf{W} \in \mathbb{R}^{E\times C\times P}$, $\mathbf{CGvals} \in \mathbb{R}^{NNZ}$, $\mathbf{src} \in \mathbb{R}^{E}$, $\mathbf{dst} \in \mathbb{R}^{E}$
Input: Metadata $M^{L_x\to NNZ}$, $M^{L_y\to NNZ}$, $M^{P\to NNZ}$, $indptr^{(NNZ+1)\to L_o}$
Input: Twiddle factors $E$ tile size $E_{tile}$, $C$ tile size $C_{tile}$, $L_o$ tile size $L_{tile}$.
Output: Output $\mathbf{O} \in \mathbb{R}^{N\times C\times L_o}$
  for SMs in parallel across $E/E_{tile} \times C/C_{tile} \times L_o/L_{tile}$ do
      Load $src_{E_{tile}}$, $dst_{E_{tile}}$
      Load $indptr^{NNZ\to L_o}_{L_{tile}}$
      for $nnz_{tile}$ indices in $indptr^{NNZ\to L_o}_{L_{tile}}$ do
          Load mappers $M^{L_x\to NNZ}_{nnz_{tile}}$, $M^{L_y\to NNZ}_{nnz_{tile}}$, $M^{P\to NNZ}_{nnz_{tile}}$ from GMEM to SMEM.
          # Gather: $N_{tile} \to E_{tile}$, $L_{tile} \to nnz_{tile}$
          Load $X_{N_{tile}\to E_{tile}C_{tile}L_{tile}\to nnz_{tile}}$, $Y_{E_{tile}L_{tile}\to nnz_{tile}}$, $CG_{nnz_{tile}}$, $W_{E_{tile}C_{tile}L_{tile}\to nnz_{tile}}$
          from GMEM to SMEM
          # Already in $nnz_{tile}$
          Load $CGvals_{nnz_{tile}}$.
          # Scatter: $nnz_{tile} \to L_{tile}$
          On-chip, compute $O_{E_{tile}C_{tile}L_{tile}} += X_{E_{tile}C_{tile}nnz_{tile}} \times Y_{E_{tile}nnz_{tile}} \times CG_{nnz_{tile}} \times W_{E_{tile}C_{tile}nnz_{tile}}$.
      # Scatter: $E_{tile} \to N_{tile}$
      Write $O_{E_{tile}C_{tile}L_{tile}}$ to GMEM at $dst_{E_{tile}}$
  Return $\mathbf{O}$
```

3.3.2 Allegro

We first highlight the input modifications for Allegro. The weight $W_{E,C,P}$ is not parameterized on the edge dimension giving us $W_{C,P}$. The spherical harmonic embedding gains an additional channel dimension. So Y_{E,L_y} becomes Y_{E,C,L_y}.

On the algorithm side, since Allegro is an edge based architecture compared to the node-based for NequIP and MACE, the message passing is done on the spherical harmonic embedding Y_{N,C,L_y} instead of the output embedding. Since the scatter-gather operation here happens outside $B3$'s inner loop, we are only able to partially fuse the gather from nodes to edges in the kernel.

Algorithm 5 Naive Allegro in e3nn [13]

```
Input: $\mathbf{X} \in \mathbb{R}^{N \times C \times L_x}$, $\mathbf{Y} \in \mathbb{R}^{E \times C \times L_y}$, $\mathbf{W} \in \mathbb{R}^{C \times P}$, $\mathbf{CG} \in \mathbb{R}^{L_x \times L_y \times L_o \times P}$, $\mathbf{src} \in \mathbb{R}^{E}$
Output: Output $\mathbf{O} \in \mathbb{R}^{E \times C \times L_o}$
  for $n$ indices in src do
      # $p$ is path indexing variable
      $p = 0$
      for $l_x, l_y, l_o$ in valid tensor product paths do
          # Gather $n \to e$, $L_{tile} \to nnz_{tile}$
          Load $X_{N \to E,C,l_x}$, $Y_{E,C,l_y}$, $CG_{l_x,l_y,l_o,p}$, $W_{C,p}$ from GMEM
          Compute $O_{E,C,l_o} = \sum_{l_x} \sum_{l_y} X_{E,C,l_x} \times Y_{E,C,l_y} \times CG_{l_x,l_y,l_o,p} \times W_{C,p}$
          Write $O_{E,C,l_o}$ back to GMEM
          $p+ = 1$
  Return $\mathbf{O}$
```

As before, the modifications with respect to the naive implementation are highlighted in green. The partial graph convolution is in moving the spherical harmonic embedding Y_{N,C,L_y} from nodes to edges Y_{E,C,L_y} by using *dst* node index list.

Algorithm 6 *B*3 Allegro Implementation

```
Input: $\mathbf{X} \in \mathbb{R}^{N \times C \times L_x}$, $\mathbf{Y} \in \mathbb{R}^{E \times C \times L_y}$, $\mathbf{W} \in \mathbb{R}^{C \times P}$, $\mathbf{CGvals} \in \mathbb{R}^{NNZ}$, $\mathbf{src} \in \mathbb{R}^{E}$
Input: Metadata $M^{L_x \to NNZ}$, $M^{L_y \to NNZ}$, $M^{P \to NNZ}$, $indptr^{NNZ \to L_o}$
Input: Twiddle factors $E$ tile size $E_{tile}$, $C$ tile size $C_{tile}$, $L_o$ tile size $L_{tile}$.
Output: Output $\mathbf{O} \in \mathbb{R}^{E \times C \times L_o}$
  for SMs in parallel across $E/E_{tile} \times C/C_{tile} \times L_o/L_{tile}$ do
      Load $src_{E_{tile}}$
      Load $indptr^{NNZ \to L_o}_{L_{tile}}$
      for $nnz_{tile}$ indices in $indptr^{NNZ \to L_o}_{L_{tile}}$ do
          Load mappers $M^{L_x \to NNZ}_{nnz_{tile}}$, $M^{L_y \to NNZ}_{nnz_{tile}}$, $M^{P \to NNZ}_{nnz_{tile}}$ from GMEM to SMEM.
          # Gather: $N_{tile} \to E_{tile}$, $L_{tile} \to nnz_{tile}$
          Load $X_{N_{tile} \to E_{tile} C_{tile} L_{tile} \to nnz_{tile}}$, $Y_{E_{tile} L_{tile} \to nnz_{tile}}$, $CG_{nnz_{tile}}$, $W_{C_{tile} L_{tile} \to nnz_{tile}}$
          from GMEM to SMEM
          # Already in $nnz_{tile}$
          Load $CGvals_{nnz_{tile}}$.
          # Scatter: $nnz_{tile} \to L_{tile}$
          On-chip, compute $O_{E_{tile} C_{tile} L_{tile}} + = X_{E_{tile} C_{tile} nnz_{tile}} \times Y_{E_{tile} C_{tile} nnz_{tile}} \times CG_{nnz_{tile}} \times W_{C_{tile} nnz_{tile}}$.
      Write $O_{E_{tile} C_{tile} L_{tile}}$ to GMEM
  Return $\mathbf{O}$
```

Chapter 4

Experimental Results

4.1 Experiments

We evaluate our kernels through microbenchmarks on NequIP and Allegro kernels, and an end to end benchmark on the Allegro model. We use an NVIDIA A100 80 GB for all of our experiments and only benchmark the forward pass of the models. All `e3nn` baselines are compiled with the PyTorch2 [1] compiler.

4.1.1 Microbenchmarks

NequIP: We compare the runtime and memory of a Helion implementation of $B3$ with respect to OpenEquivariance. Our irrep configurations Table 4.1 are based on the SevenNetl3i5[29] and GNoME[23] set of NequIP models with multiplicity fixed to 64 since our kernel cannot handle variable multiplicity.
Allegro: Here we compare both a Helion implementation and a Triton implementation that we integrated into Allegro as part of their infrastructure upgrade [34]. We could not use Helion for the integration since it doesn't yet support PyTorch's Ahead-of-Time-Inductor functionality[1].

The hyperparams are based on the latest Allegro pre-print [34].

4.1.2 End to End Benchmark

We use the same setup as the microbenchmarks except calling the whole model instead of just the kernel. Model details can be found in Appendix A

4.2 Results

B3 simplifies the code: $B3$'s Helion and Triton implementations span hundreds of lines compared to thousands in CUDA-based implementations. The Helion implementation Figure 4.1 in particular takes < 100 lines of code.

[1]https://github.com/pytorch/helion/issues/143

Model	Kernel Irreps
NequIP	`64x0e+64x1o`
	`64x0e+64x1o+64x2e`
	`64x0e+64x1o+64x2e+64x3o`
Allegro	`64x0e+64x0o+64x1e+64x1o`
	`64x0e+64x0o+64x1e+64x1o+64x2e+64x2o`
	`64x0e+64x0o+64x1e+64x1o+64x2e+64x2o+64x3e+64x3o`

Table 4.1: Irreps configs for Allegro and NequIP used in the benchmarks

B3 is faster than e3nn: Due to its tiling strategy, *B*3 is 3x faster for NequIP **??** and up to 10x faster for Allegro over compiled `e3nn`.
B3 is as memory-efficient as CUDA implementations: By tying the memory efficiency gains to the kernel's overall design rather than its specific hardware mapping, *B*3 is as efficient as CUDA-based implementations. *B*3 implements the graph convolution fusion for NequIP [5] and outer product fusion for Allegro[34].
Additional metadata upper bounds the speedup: While the metadata simplifies the tiling on the irreps dimension, it also introduces additional overhead compared to CUDA implementations which we see here in the 2-4x gap in NequIP and 10x gap for Allegro. We leave it to future work to further sweep the performance-simplicity front.
Kernel speedups ≠ End-to-end speedups: We are reminded of Amdahl's law[8] in 4.4 where 100x gains on the kernel side only translate to 6x end-to-end speedup. This underscores the need for equivariant algorithms that can not only be made more efficient with kernels but also take up a sizeable chunk of the overall runtime.

```python
@helion.kernel(dot_precision="ieee")
def helion_nequip(
    X: torch.Tensor,
    Y: torch.Tensor,
    out: torch.Tensor,
    M_x_to_nnz: torch.Tensor,
    M_y_to_nnz: torch.Tensor,
    M_p_to_nnz: torch.Tensor,
    indptr: torch.Tensor,
    CGvals: torch.Tensor,
    src: torch.Tensor,
    dst: torch.Tensor,
    weight: torch.Tensor,
) -> torch.Tensor:
    """Helion kernel for NequIP convolution."""
    *,C, * = X.shape
    E = src.shape[0]
    *, *,L_out = out.shape
    # Tile across the E, C, and L_out dimensions
    for tile_e, tile_c, tile_o in hl.tile([E, C, L_out]):
        # Load src_tile_e, dst_tile_e
        src_idx_chunk = src[tile_e]
        dst_idx_chunk = dst[tile_e]

        # Get inptr_NNZ_to_L_out_tile_o
        start_ptr_chunk = indptr[tile_o]
        end_ptr_chunk = indptr[tile_o.index+1]
        nnz = end_ptr_chunk - start_ptr_chunk
        max_nnz = nnz.amax()

        acc = hl.zeros((tile_e, tile_c, tile_o), dtype=X.dtype
              )

        for nnz_tile in hl.tile(max_nnz, block_size=1):

            # Gather: tile_o -> tile_nnz
            offsets = start_ptr_chunk + nnz_tile.index
            mask = nnz_tile.index < nnz

            x_idx_chunk = hl.load(
                M_x_to_nnz,
                [offsets],
                extra_mask=mask,
            )
            y_idx_chunk = hl.load(
                M_y_to_nnz,
                [offsets],
                extra_mask=mask,
            )
            w_idx_chunk = hl.load(
                M_p_to_nnz,
                [offsets],
                extra_mask=mask,
            )
            CGval_chunk = hl.load(
                CGvals,
                [offsets],
                extra_mask=mask,
            )

            # Gather: tile_n -> tile_e
            # Scatter: tile_nnz -> tile_o
            acc += (X[src_idx_chunk, tile_c, x_idx_chunk] *
                    Y[tile_e, l2_idx_chunk][:, None, :] *
                    CGval_chunk *
                    weight[tile_e, tile_c, w_idx_chunk])

        # Scatter:  tile_e -> tile_n
        hl.atomic_add(out, [dst_idx_chunk, tile_c, tile_o],
              acc)
    return out
```

```python
@helion.kernel(dot_precision="ieee")
def helion_kernel_allegro(
    X: torch.Tensor,
    Y: torch.Tensor,
    out: torch.Tensor,
    M_x_to_nnz: torch.Tensor,
    M_y_to_nnz: torch.Tensor,
    M_p_to_nnz: torch.Tensor,
    indptr: torch.Tensor,
    CGvals: torch.Tensor,
    src: torch.Tensor
    weight: torch.Tensor
) -> torch.Tensor:
    E, C, L_out = output.shape
    # Tile across the E, C, and L_out dimensions
    for tile_e, tile_c, tile_o in hl.tile([E, C, L_out]):
        start_ptr_chunk = indptr[tile_o]
        end_ptr_chunk = indptr[tile_o.index+1]
        nnz = end_ptr_chunk - start_ptr_chunk
        max_nnz = nnz.amax()

        src_chunk = src[tile_e]
        acc = hl.zeros((tile_e, tile_c, tile_o), dtype=X.dtype
              )
        for nnz_tile in hl.tile(max_nnz, block_size=1):

            # Gather: tile_o -> tile_nnz
            offsets = start_ptr_chunk + nnz_tile.index
            mask = nnz_tile.index < nnz

            x_idx_chunk = hl.load(
                M_x_to_nnz,
                [offsets],
                extra_mask=mask,
            )
            y_idx_chunk = hl.load(
                M_y_to_nnz,
                [offsets],
                extra_mask=mask,
            )
            w_idx_chunk = hl.load(
                M_p_to_nnz,
                [offsets],
                extra_mask=mask,
            )
            CGval_chunk = hl.load(
                CGvals,
                [offsets],
                extra_mask=mask,
            )

            # Gather: tile_n -> tile_e
            # Scatter: tile_nnz -> tile_o
            acc += X[tile_e, tile_c, x_idx_chunk] *
                   Y[src_chunk, tile_c, y_idx_chunk] *
                   CGval_chunk *
                   weight[tile_c, w_idx_chunk][None, :, :]
        output[tile_e, tile_c, tile_o] = acc
    return output
```

Figure 4.1: *B*3’s kernels for NequIP (left) and Allegro (right) in < 100 lines of Helion code

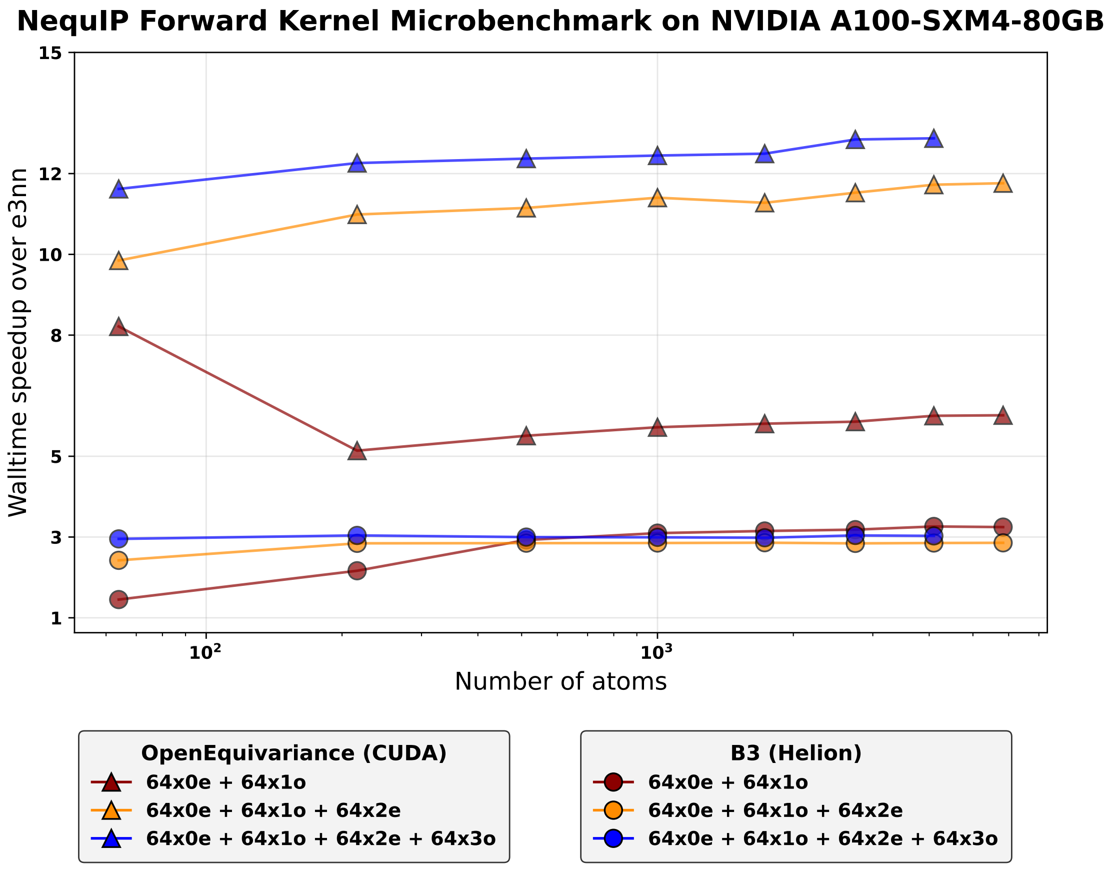

Figure 4.2: NequIP kernels forward walltime speedup over e3nn. *B*3 (Helion) is 3x faster than e3nn and 2-4x slower than OpenEquivariance (CUDA)

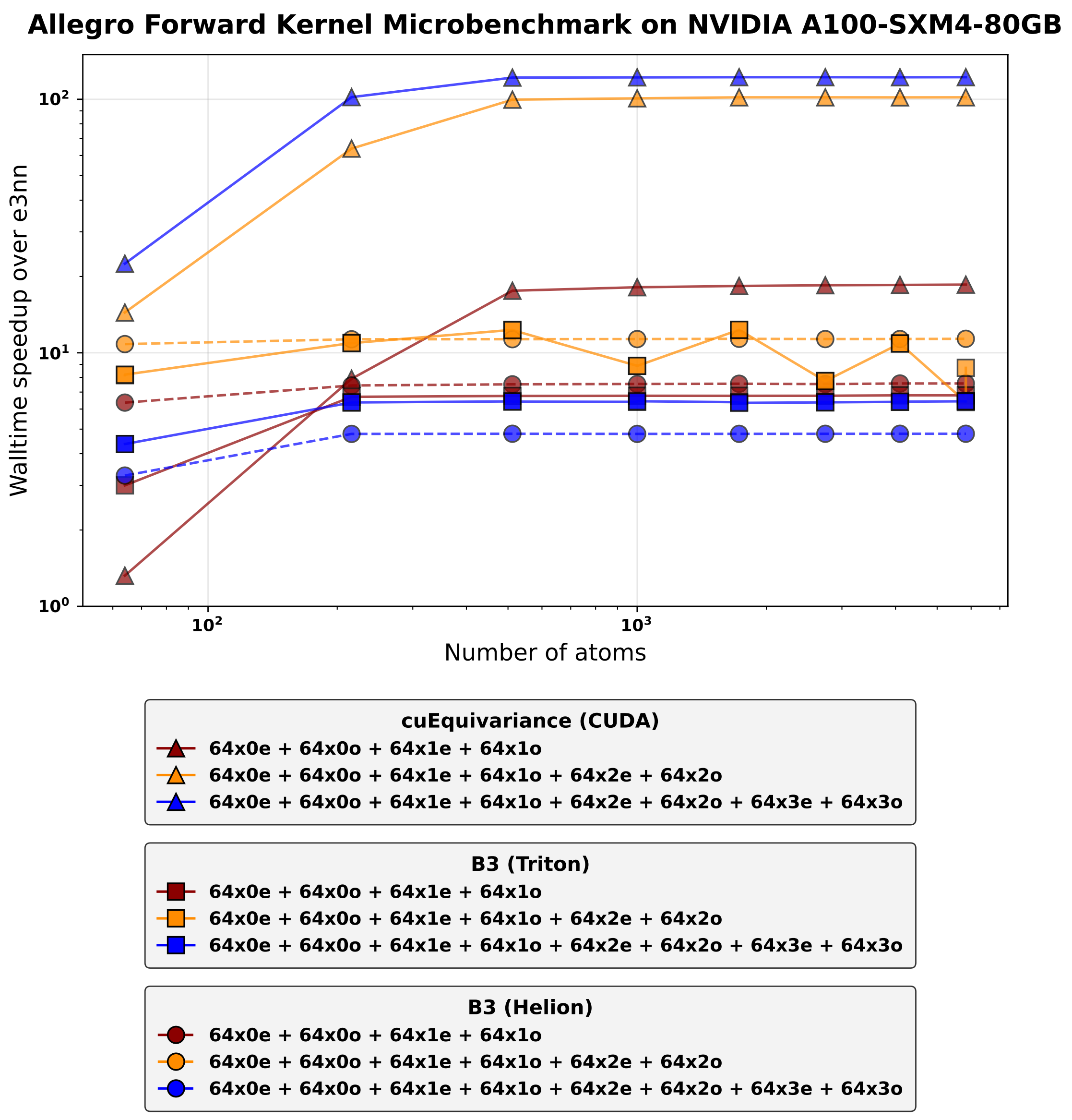

Figure 4.3: Allegro kernels forward walltime speedup over e3nn. Both Helion and Triton implementations of *B*3 are comparable and are 5-10x faster over e3nn while being 2-10x slower than cuEquivariance (CUDA)

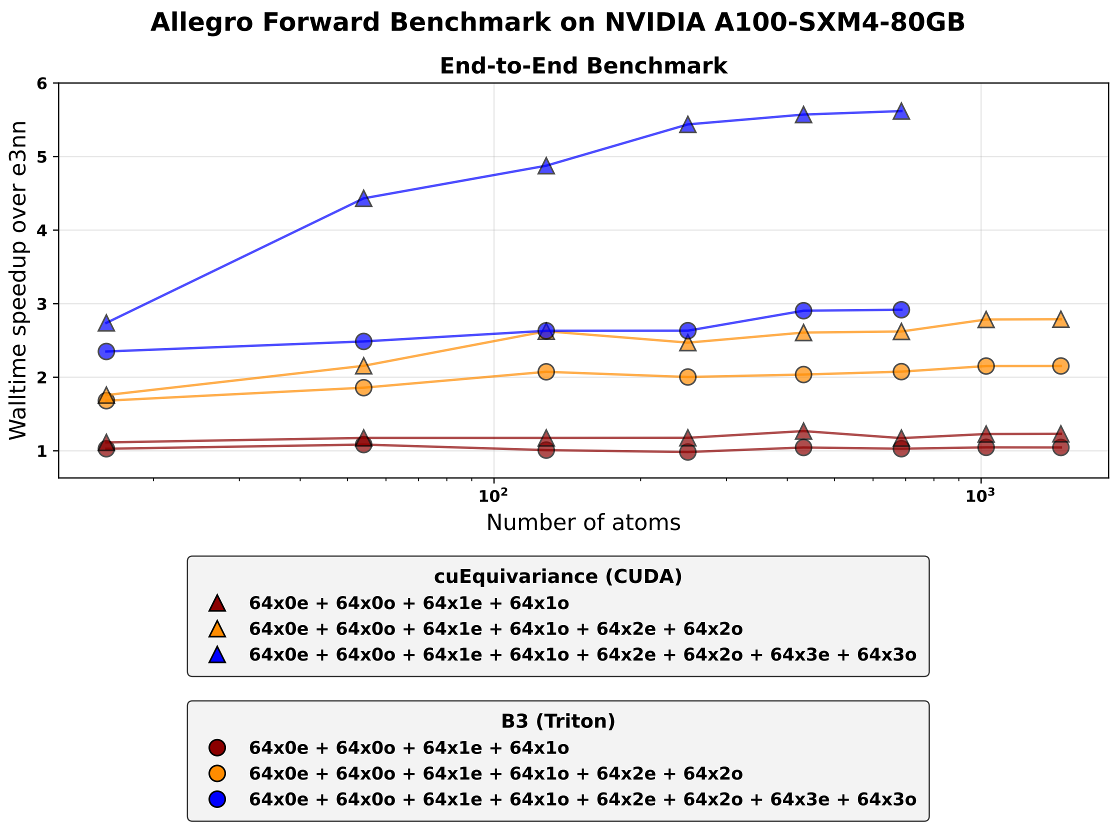

Figure 4.4: Allegro end-to-end inference speedups over e3nn. Upto 3x faster than e3nn and 2x slower than cuEquivariance (CUDA)

Chapter 5

Conclusion

In this thesis we presented *B*3, a simplified 100 line equivariant kernel that covers all known optimizations, while providing walltime and memory efficiency gains over PyTorch baseline. While existing works trade off performance for simplicity, we instead explore trading off simplicity for performance to improve kernel development velocity.
We highlight some promising future extensions of this work:

- **Backward kernels**: Since the focus our work was in identifying a simplified abstraction layer, we did not invest that much effort in having a complete production set of kernels that include the backward and double backward kernels, typically needed for training computational chemistry models with forces or hessians.
- **Reducing metadata overhead**: The metadata strategy that we took is definitely not the only way to deal with the tensor product sparsity and parallel path execution. We invite future work to find more cleaner and performant alternatives.
- **Higher arithmetic intensity**: Our main goal was to provide an accessible interface for prototyping different path pruning strategies that bring up the arithmetic intensity along with expanding to megafusions that implement a single kernel for the entire layer or network [14].

Appendix A

Details for Reproducibility

Code: All of the kernels and benchmarking code can be found at
https://github.com/mitkotak/tiling_tensorproduct

Input Generation: The graphs are generated using MACE's ASE benchmarking script[1] that scales up a diamond crystal lattice. Radial cutoff is set 6 A.

End to End Model: We list the hyperparams for the Allegro model we used in our benchmarking in Table A.1

Timing: All benchmarking was done using Triton's `do_bench` function. We do 25 warmups and 100 runs, and report the median value.

[1]https://github.com/ACEsuit/mace/blob/main/tests/test_benchmark.py

Table A.1: Allegro Model Hyperparameters

Parameter	**Value**
Common Configuration	
cutoff	6.0
chemical_symbols	["C"]
seed	123
model_dtype	"float32"
type_names	["C"]
r_max	6.0
per_type_energy_shifts	{"C": 0.0}
per_type_energy_scales	{"C": 1.0}
Allegro Model Configuration	
target	allegro.model.AllegroModel
l_max	3
parity	True
num_layers	2
num_scalar_features	64
num_tensor_features	64
tp_path_channel_coupling	True
radial_chemical_embed	allegro.nn.TwoBodyBesselScalarEmbed
radial_chemical_embed_dim	128
scalar_embed_mlp_hidden_layers_depth	2
scalar_embed_mlp_hidden_layers_width	64
scalar_embed_mlp_nonlinearity	"silu"
allegro_mlp_hidden_layers_depth	2
allegro_mlp_hidden_layers_width	256
allegro_mlp_nonlinearity	"silu"
readout_mlp_hidden_layers_depth	1
readout_mlp_hidden_layers_width	128
readout_mlp_nonlinearity	None
avg_num_neighbors	25.0

Bibliography

[1] Jason Ansel, Edward Yang, Horace He, Natalia Gimelshein, Animesh Jain, Michael Voznesensky, Bin Bao, Peter Bell, David Berard, Evgeni Burovski, Geeta Chauhan, Anjali Chourdia, Will Constable, Alban Desmaison, Zachary DeVito, Elias Ellison, Will Feng, Jiong Gong, Michael Gschwind, Brian Hirsh, Sherlock Huang, Kshiteej Kalambarkar, Laurent Kirsch, Michael Lazos, Mario Lezcano, Yanbo Liang, Jason Liang, Yinghai Lu, C. K. Luk, Bert Maher, Yunjie Pan, Christian Puhrsch, Matthias Reso, Mark Saroufim, Marcos Yukio Siraichi, Helen Suk, Shunting Zhang, Michael Suo, Phil Tillet, Xu Zhao, Eikan Wang, Keren Zhou, Richard Zou, Xiaodong Wang, Ajit Mathews, William Wen, Gregory Chanan, Peng Wu, and Soumith Chintala. Pytorch 2: Faster machine learning through dynamic python bytecode transformation and graph compilation. In *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*, ASPLOS '24, page 929–947, New York, NY, USA, 2024. Association for Computing Machinery. ISBN 9798400703850. doi:10.1145/3620665.3640366. URL https://doi.org/10.1145/3620665.3640366.

[2] Jacob Austin, Sholto Douglas, Roy Frostig, Anselm Levskaya, Charlie Chen, Sharad Vikram, Federico Lebron, Peter Choy, Vinay Ramasesh, Albert Webson, and Reiner Pope. How to scale your model. 2025. Retrieved from https://jax-ml.github.io/scaling-book/gpus.

[3] Ilyes Batatia, David Peter Kovacs, Gregor N. C. Simm, Christoph Ortner, and Gabor Csanyi. MACE: Higher Order Equivariant Message Passing Neural Networks for Fast and Accurate Force Fields. In Alice H. Oh, Alekh Agarwal, Danielle Belgrave, and Kyunghyun Cho, editors, *Advances in Neural Information Processing Systems*, 2022. URL https://openreview.net/forum?id=YPpSngE-ZU.

[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. 13, May 2022. URL https://doi.org/10.1038/s41467-022-29939-5.

[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.

[6] Tri Dao, Daniel Y. Fu, Stefano Ermon, Atri Rudra, and Christopher Ré. Flashattention: Fast and memory-efficient exact attention with io-awareness, 2022. URL https://arxiv.org/abs/2205.14135.

[7] DeepSeek-AI, Aixin Liu, Bei Feng, Bin Wang, Bingxuan Wang, Bo Liu, Chenggang Zhao, Chengqi Dengr, Chong Ruan, Damai Dai, Daya Guo, Dejian Yang, Deli Chen, Dongjie Ji,

Erhang Li, Fangyun Lin, Fuli Luo, Guangbo Hao, Guanting Chen, Guowei Li, H. Zhang, Hanwei Xu, Hao Yang, Haowei Zhang, Honghui Ding, Huajian Xin, Huazuo Gao, Hui Li, Hui Qu, J. L. Cai, Jian Liang, Jianzhong Guo, Jiaqi Ni, Jiashi Li, Jin Chen, Jingyang Yuan, Junjie Qiu, Junxiao Song, Kai Dong, Kaige Gao, Kang Guan, Lean Wang, Lecong Zhang, Lei Xu, Leyi Xia, Liang Zhao, Liyue Zhang, Meng Li, Miaojun Wang, Mingchuan Zhang, Minghua Zhang, Minghui Tang, Mingming Li, Ning Tian, Panpan Huang, Peiyi Wang, Peng Zhang, Qihao Zhu, Qinyu Chen, Qiushi Du, R. J. Chen, R. L. Jin, Ruiqi Ge, Ruizhe Pan, Runxin Xu, Ruyi Chen, S. S. Li, Shanghao Lu, Shangyan Zhou, Shanhuang Chen, Shaoqing Wu, Shengfeng Ye, Shirong Ma, Shiyu Wang, Shuang Zhou, Shuiping Yu, Shunfeng Zhou, Size Zheng, T. Wang, Tian Pei, Tian Yuan, Tianyu Sun, W. L. Xiao, Wangding Zeng, Wei An, Wen Liu, Wenfeng Liang, Wenjun Gao, Wentao Zhang, X. Q. Li, Xiangyue Jin, Xianzu Wang, Xiao Bi, Xiaodong Liu, Xiaohan Wang, Xiaojin Shen, Xiaokang Chen, Xiaosha Chen, Xiaotao Nie, Xiaowen Sun, Xiaoxiang Wang, Xin Liu, Xin Xie, Xingkai Yu, Xinnan Song, Xinyi Zhou, Xinyu Yang, Xuan Lu, Xuecheng Su, Y. Wu, Y. K. Li, Y. X. Wei, Y. X. Zhu, Yanhong Xu, Yanping Huang, Yao Li, Yao Zhao, Yaofeng Sun, Yaohui Li, Yaohui Wang, Yi Zheng, Yichao Zhang, Yiliang Xiong, Yilong Zhao, Ying He, Ying Tang, Yishi Piao, Yixin Dong, Yixuan Tan, Yiyuan Liu, Yongji Wang, Yongqiang Guo, Yuchen Zhu, Yuduan Wang, Yuheng Zou, Yukun Zha, Yunxian Ma, Yuting Yan, Yuxiang You, Yuxuan Liu, Z. Z. Ren, Zehui Ren, Zhangli Sha, Zhe Fu, Zhen Huang, Zhen Zhang, Zhenda Xie, Zhewen Hao, Zhihong Shao, Zhiniu Wen, Zhipeng Xu, Zhongyu Zhang, Zhuoshu Li, Zihan Wang, Zihui Gu, Zilin Li, and Ziwei Xie. Deepseek-v2: A strong, economical, and efficient mixture-of-experts language model, 2024. URL https://arxiv.org/abs/2405.04434.

[8] Christina Delimitrou and Christos Kozyrakis. Amdahl's law for tail latency. *Commun. ACM*, 61(8):65–72, July 2018. ISSN 0001-0782. doi:10.1145/3232559. URL https://doi.org/10.1145/3232559.

[9] Alex Dremov. Understanding Flash Attention: Writing the Algorithm from Scratch in Triton. https://alexdremov.me/understanding-flash-attention-writing-the-algorithm-from-scratch-in-triton/, 2025. Accessed: September 2, 2025.

[10] Jesun Firoz, Franco Pellegrini, Mario Geiger, Darren Hsu, Jenna A. Bilbrey, Han-Yi Chou, Maximilian Stadler, Markus Hoehnerbach, Tingyu Wang, Dejun Lin, Emine Kucukbenli, Henry W. Sprueill, Ilyes Batatia, Sotiris S. Xantheas, MalSoon Lee, Chris Mundy, Gabor Csanyi, Justin S. Smith, Ponnuswamy Sadayappan, and Sutanay Choudhury. Optimizing data distribution and kernel performance for efficient training of chemistry foundation models: A case study with mace, 2025. URL https://arxiv.org/abs/2504.10700.

[11] Nathan C Frey, Ryan Soklaski, Simon Axelrod, Siddharth Samsi, Rafael Gómez-Bombarelli, Connor W Coley, and Vijay Gadepally. Neural scaling of deep chemical models. *Nature Machine Intelligence*, 5(11):1297–1305, October 2023.

[12] Xiang Fu, Andrew Rosen, Kyle Bystrom, Rui Wang, Albert Musaelian, Boris Kozinsky, Tess Smidt, and Tommi Jaakkola. A recipe for charge density prediction, 2024.

[13] Mario Geiger and Tess Smidt. e3nn: Euclidean Neural Networks, 2022.

[14] Hazy Research. ThunderMLA: FlashMLA, Faster and Fused-er! https://hazyresearch.stanford.edu/blog/2025-03-04-thundermla, March 2025.

[15] Emiel Hoogeboom, Victor Garcia Satorras, Clément Vignac, and Max Welling. Equivariant diffusion for molecule generation in 3d, 2022.

[16] JAX Team. Pallas: a JAX kernel language, 2024. URL https://github.com/jax-ml/jax. Part of JAX experimental features.

[17] John Jumper, Richard Evans, Alexander Pritzel, Tim Green, Michael Figurnov, Olaf Ronneberger, Kathryn Tunyasuvunakool, Russ Bates, Augustin Žídek, 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, Michal 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, 2021.

[18] Risi Kondor. N-body networks: a covariant hierarchical neural network architecture for learning atomic potentials, 2018. URL https://arxiv.org/abs/1803.01588.

[19] Risi Kondor, Zhen Lin, and Shubhendu Trivedi. Clebsch-gordan nets: a fully fourier space spherical convolutional neural network. *Advances in Neural Information Processing Systems*, 31, 2018.

[20] Jae Hyeon Lee, Payman Yadollahpour, Andrew Watkins, Nathan C. Frey, Andrew Leaver-Fay, Stephen Ra, Kyunghyun Cho, Vladimir Gligorijevic, Aviv Regev, and Richard Bonneau. Equifold: Protein structure prediction with a novel coarse-grained structure representation. *bioRxiv*, 2022. doi:10.1101/2022.10.07.511322. URL https://www.biorxiv.org/content/early/2022/10/08/2022.10.07.511322.

[21] Seung Lee, Hojoon Kim, Yutack Park, Dawoon Jeong, Seungwu Han, Yeonhong Park, and Jae W. Lee. FlashTP: Fused, Sparsity-Aware Tensor Product for Machine Learning Interatomic Potentials. In *Proceedings of the 42nd International Conference on Machine Learning*, ICML 2025, 2025. URL https://github.com/SNU-ARC/flashTP.

[22] 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.

[23] Amil Merchant, Simon Batzner, Samuel S. Schoenholz, Muratahan Aykol, Gowoon Cheon, and Ekin Dogus Cubuk. Scaling deep learning for materials discovery. *Nature*, 2023. doi:10.1038/s41586-023-06735-9.

[24] Modal. Gpu glossary, 2025. URL https://modal.com/gpu-glossary. Glossary of GPU and CUDA terminology.

[25] 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, 2023.

[26] NVIDIA. cuequivariance, 2024. URL https://docs.nvidia.com/cuda/cuequivariance/index.html.

[27] NVIDIA CUTLASS Team. CuTe: C++ CUDA template abstractions for hierarchically multidimensional layouts, 2023. URL https://github.com/NVIDIA/cutlass. Part of CUTLASS 3.0+.

[28] Cameron J Owen, Steven B Torrisi, Yu Xie, Simon Batzner, Kyle Bystrom, Jennifer Coulter, Albert Musaelian, Lixin Sun, and Boris Kozinsky. Complexity of many-body interactions in transition metals via machine-learned force fields from the TM23 data set. *Npj Comput. Mater.*, 10(1), May 2024.

[29] Yutack Park, Jaesun Kim, Seungwoo Hwang, and Seungwu Han. Scalable parallel algorithm for graph neural network interatomic potentials in molecular dynamics simulations. *J. Chem. Theory Comput.*, 20(11):4857–4868, 2024. doi:10.1021/acs.jctc.4c00190.

[30] PyTorch Team. Helion: A python-embedded dsl that makes it easy to write fast, scalable ml kernels with minimal boilerplate. https://github.com/pytorch/helion, 2025. BSD-style licensed.

[31] Joshua A Rackers, Lucas Tecot, Mario Geiger, and Tess E Smidt. A recipe for cracking the quantum scaling limit with machine learned electron densities. *Mach. Learn.: Sci. Technol.*, 4(1):015027, February 2023.

[32] Amit Sabne. Xla : Compiling machine learning for peak performance, 2020.

[33] Benjamin F. Spector, Simran Arora, Aaryan Singhal, Daniel Y. Fu, and Christopher Ré. Thunderkittens: Simple, fast, and adorable ai kernels, 2024. URL https://arxiv.org/abs/2410.20399.

[34] 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.

[35] Vijay Thakkar, Pradeep Ramani, Cris Cecka, Aniket Shivam, Honghao Lu, Ethan Yan, Jack Kosaian, Mark Hoemmen, Haicheng Wu, Andrew Kerr, Dustyn Blasig, Fei Wang, Paul Springer, Xiaoyu Zhang, Neal Vehr, Rakesh Shivaram, Manish Gupta, Jin Wang, Dhananjay Khanna, Youngbin Kim, Julien Demouth, Stephen Jones, Buck Shlegeris, Rami Hasaj, Dave Schmerfeld, Richard Suda, Keren Zhou, Alicia Klinvex, and Jeremy Appleyard. CUTLASS: CUDA templates for linear algebra subroutines, 2025. URL https://github.com/NVIDIA/cutlass.

[36] 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.

[37] Philippe Tillet, H. T. Kung, and David Cox. Triton: an intermediate language and compiler for tiled neural network computations. In *Proceedings of the 3rd ACM SIGPLAN International Workshop on Machine Learning and Programming Languages*, MAPL 2019, page 10–19, New York, NY, USA, 2019. Association for Computing Machinery. ISBN 9781450367196. doi:10.1145/3315508.3329973. URL https://doi.org/10.1145/3315508.3329973.

[38] Maurice Weiler, Mario Geiger, Max Welling, Wouter Boomsma, and Taco Cohen. 3d steerable cnns: Learning rotationally equivariant features in volumetric data, 2018. URL https://arxiv.org/abs/1807.02547.

[39] YuQing Xie, Ameya Daigavane, Mit Kotak, and Tess Smidt. The price of freedom: Exploring expressivity and runtime tradeoffs in equivariant tensor products, 2025. URL https://arxiv.org/abs/2506.13523.