Optimizing a Tensor Product using Sparsity

Jaeyeon Won Ryan Lee Nicholas Dow

***Abstract*—We present an optimized GPU implementation of the tensor product for equivariant neural networks. Tensor products in equivariant neural networks are challenging due to the block sparsity in its weights imposed by the three properties of equivariance (translation, rotations, and inversion). We show that using the right sparse tensor format is critical to performing only the effectual computations while minimizing costly synchronization between SMs. We implement a GPU implementation using Triton which outperforms the dense Jax-based baseline by 2-10× on a Nvidia RTX A4000 GPU.**

I. Introduction

Equivariant neural networks [5] are specially designed networks capable of handling input transformations such as rotations and shifts. For instance, if the input to an equivariant neural network is rotated by 90 degrees, the output feature map will transform (or permute) in the same way.

Among the libraries available for building equivariant neural networks, *e3nn-jax* [3] stands out as a state-of-the-art tool. It is designed for constructing and accelerating neural networks that are equivariant to 3D rotations, translations, and other symmetries. At its core, e3nn-jax leverages irreducible representations of the 3D rotation group (SO(3)) to process geometric data, such as atomic structures, molecules, and point clouds, within the Jax [1] framework.

A fundamental operation in e3nn-jax is the tensor product, which combines features expressed in these irreducible representations. This operation enables the network to exchange information between different rotational symmetries while preserving equivariance, ensuring that the model maintains consistency under transformations.

II. Background

A. Tensor Product

In **e3nn**, there are multiple variants of the tensor product operation. In this project, we focus on a specific form of the tensor product, which can be expressed using Einsum notation as follows:

$$Out_{b,k} = W_{i,j,k} * In1_{b,i} * In2_{b,j}$$

This operation multiplies two batched inputs, **In1** and **In2**, with a 3D weight tensor **W**. The result is stored in **Out** by reducing over dimensions i and j.

B. e3nn-jax Implementation

The current state-of-the-art implementation in e3nn-jax takes advantage of the fact that the weight tensor W exhibits a special sparsity pattern: a block-diagonal sparse structure. Instead of storing the full tensor $W_{i,j,k}$, **e3nn-jax** stores only the dense blocks within W. For each block, the output is computed as:

$$Out_{b,k} = WBlock_{i,j,k} * In1_{b,i} * In2_{b,j}$$

These computations leverage optimized vendor libraries such as cuBLAS [4] for efficiency. Additionally, all JAX code is *JIT-compiled*, which significantly accelerates performance compared to running the same code in the Python interpreter.

C. Leveraging Sparsity Limitations

Despite these optimizations, our investigation revealed that e3nn-jax does not fully exploit the sparsity in W. Specifically, we observed that each dense block within the block-diagonal sparse format has only about **10% non-zero values**, meaning a significant number of unnecessary operations (e.g., zero multiplications) are still being performed.

To address this issue, our work implements two optimized versions of the tensor product that fully leverage sparsity and GPU capabilities, reducing redundant computations and improving performance.

III. Implementation

A. Using Coordinate List (COO) Representation

The relatively high sparsity of equivariant neural networks guides us to use a *sparse representation* of the weights. Sparse representations such as Coordinate List (COO) and Compressed-Sparse Row only store non-zero values and their coordinates, which brings the benefit of skipping ineffectual computation (e.g., multiplying by zero). We use the COO format which stores the coordinates of non-zero values as a list of tuples.

```
1  # B : Batch size
2  # NNZ: Number of nonzeros
3  # W_i, W_j, W_k, W_val: weights in COO-format,
       dimensions [NNZ]
4  # Input1, Input2: [B, L_in]
5  # Output: [B, L_out]
6  def TensorProduct(W_i, W_j, W_k, W_val,
7    Input1, Input2, Output):
8      for nz in range(NNZ):
9        nz_in1 = Input1[:,W_i[nz]]
10       nz_in2 = Input2[:,W_j[nz]]
11       Output[:,W_k[nz]] += nz_in1 * nz_in2 *
             W_val[nz]
```

Listing 1. Hash table lookup code.

The tensor product kernel is implemented using Triton [6]. We design the kernel such that each Triton program works on a *block of non-zeros*. The pseudocode is given in Listing 1. Given the input `i`, `j`, and `k` coordinate values of the COO, each Triton program loads a block of these coordinates, performs an indirect load on the input tensors to grab the corresponding

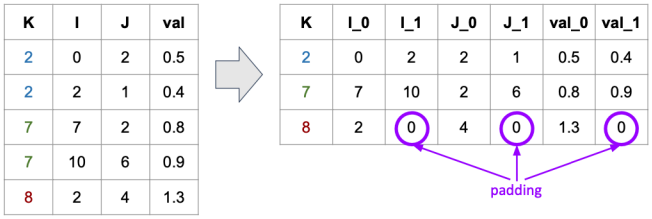

Fig. 1. Transforming COO to a grouped COO representation with a group size of 2.

values (`i` coordinates from the first tensor and `j` coordinates from the second tensor), and multiplies them with the weight. The kernel then performs scatter indirect stores to the `k` coordinates of the output tensor.

Our tensor kernel's performance is highly dependent on the number of gather loads of the input tensors and the scatter writes to the output tensor. In particular, the scatter writes must be performed *atomically* because different triton programs (i.e., thread blocks in CUDA-land) can write to the same coordinate of the output tensor. As we will see later in section IV, the atomic add operations dominate our throughput such that the naive implementation is worse than the dense baseline.

B. Reducing Atomic Operations

We reduce the number of atomic operations by *grouping multiple input coordinates that contribute to the same output coordinate*. Our key insight is that multiple input coordinates that contribute to the same output coordinate can be handled by a single thread, which removes the need for atomic operations on those FMAs.

An example of such grouping is shown in Figure 1. We define a new parameter called a *group size* which indicates how many input coordinates are grouped together for the same output coordinate. Notice that the ideal group size varies per output coordinate, as the number of input coordinate pairs that have the same output coordinate differ. To make this amenable to GPUs, we choose a single group size for all output coordinates and *pad* the input coordinates' values with zeros if they are not divisible by the group size. For example, Figure 1 shows that output coordinate 8 only has one input coordinate for it, so we pad it to match the group size of 2.

```
# B : Batch size
# G_NNZ: Number of grouped output coordinates
# G: group size
# W_i, W_j, W_val: [G_NNZ, G]
# W_k: [G_NNZ]
# Input1, Input2: [B, L_in]
# Output: [B, L_out]
def TensorProduct(W_i, W_j, W_k, W_val,
  Input1, Input2, Output):
    for nz in range(G_NNZ):
      for g in range(G)
        nz_in1 = Input1[:,W_i[nz, g]]
        nz_in2 = Input2[:,W_j[nz, g]]
        accum += nz_in1 * nz_in2 * W_val[nz, g]
      Output[:,W_k[nz]] += accum
```

Listing 2. Hash table lookup code.

The pseudocode for our improved tensor product kernel is shown in Listing 2. Each program now loads a block of groups, where each group contains the input coordinate pairs for a given output coordinate. Because each thread now works on a single group, it can accumulate the results across that group locally before storing it back to global memory.

Because grouping introduces padded values, it increases ineffectual computation: compared to the naive implementation which performed $2NNZ \times B$ floating-point multiplies, grouping now performs $2G\_NNZ \times G \times B$ multiplies where $G\_NNZ >= NNZ$. The increase in work is determined by the distribution of non-zeros in the weight matrix. As we will show in the evaluation (section IV) we sweep the group size parameter that produces the fastest runtime for our application.

IV. Evaluation

We evaluated our implementation on a Nvidia RTX A4000 GPU, with randomly initialized inputs that follow the sparsity pattern expected in a real **e3nn** applications; the sparsity is generated using **e3nn's** `e3nn.clebsch_gordon` function who's computation is opaque but generates the weight tensor for the tensor product. We benchmarked both the Naive COO and Padded COO's performances against the baseline to show the effectiveness of each of the improvments. The parameters that are relevant to check performance against are the group size parameter (denoted as Padding size), the size of the batch dimension, and the lmax parameter, which controls the number of non-zeroes and size of **In1, In2, and W**. While lmax changes a number of different qualities of the tensor product, the main one we are concerned with is sparsity.

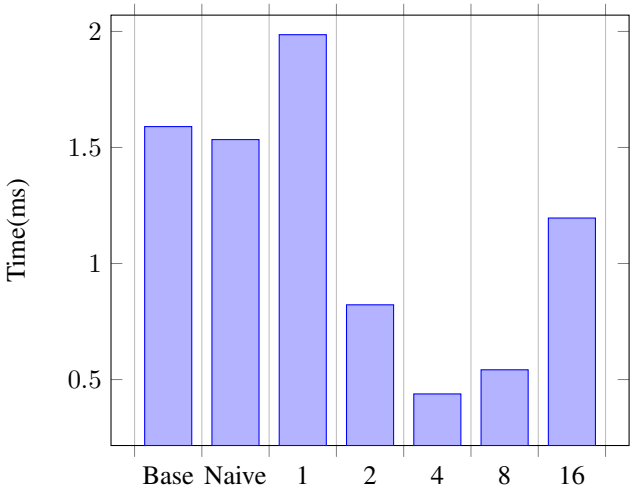

Fig. 2. Evaluation over padding sizes, Batch size = 1e4, Lmax = 5

A. Finetuning group size

As seen in Figure 2 the group size parameter has a huge impact on the performance of the padded COO implementation. We've found that performance across batch sizes for a certain group size was independent, but as lmax changed, the optimal group size would change as well. This observation

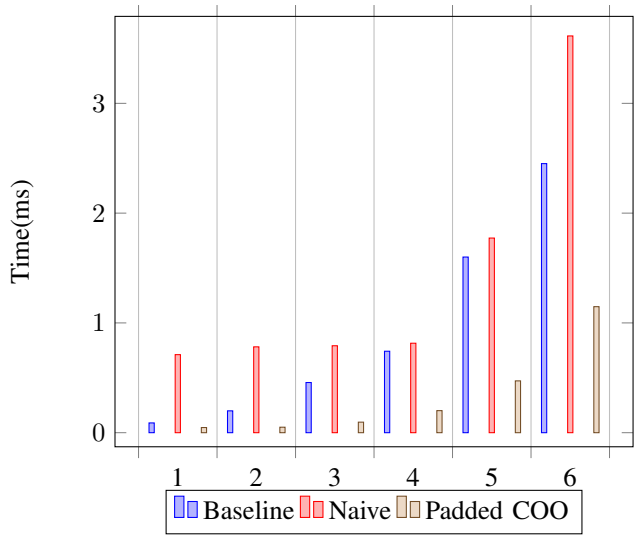

Fig. 3. Evaluation over Lmax, (Batch size = 1e4, Padding = 4)

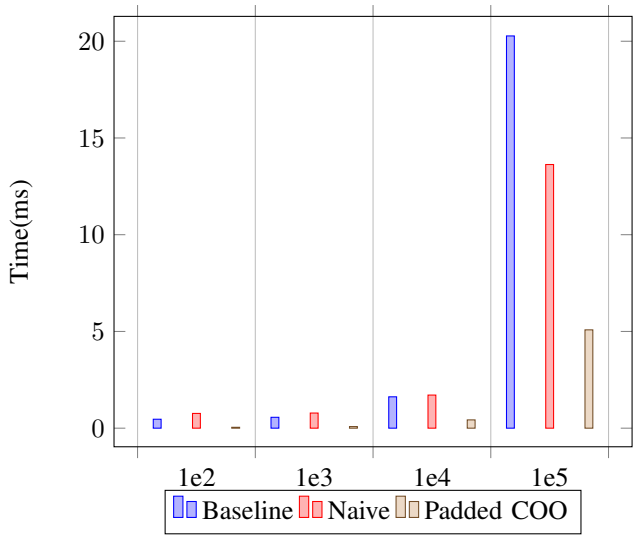

Fig. 4. Evaluation over Batch size, (Padding = 4, Lmax = 5)

makes sense intuitively as changing lmax changes the number and distribution of non-zero entries in **W**, so a group size setting for a high lmax might add too many padding zeroes to compute compared to the contention it reduces in the tensor product. However, for all relevant lmax settings(1-5) had group size $\in [3, 5]$.

B. Impact of Lmax on performance

As seen in subsection IV-A we can see that Padded COO performs a factor of 4 times better than the baseline and Naive COO for all relevant lmax settings (1-5), but baseline starts to catch up as lmax grows, with only a 2.5x speedup at lmax = 6. While the sparsity of the tensor product grows as lmax grows and therefore the theorectical advantage of an implementation leveraged for that sparsity, it seems there are other factors at play limiting that advantage for this tensor product.

Combining that observation with the earlier observation that the group size does not change much with different lmax settings, we theorized that while the sparsity increases with lmax, the distribution of non-zeroes migth be becoming adversarial to our padding approach. With some rudimentary analysis, we saw that the distribution of non-zeroes per k coordinate was skewed to the left but still maintained outliers at the max value, which increases with lmax. Perhaps this skew is re-introducing the atomic add contention for some values of k that were previously reduced by grouping, while also increasing the amount of padding we need to add.

C. Impact of batch size on performance

As seen in Figure 4, Padded COO performs at least a factor of 4 times better than the baseline and Naive COO for all batch sizes we could test, but had a bigger advantage at smaller batch sizes before speedup plateaued.

V. Discussion

With our project, we successfully leveraged the structured sparsity within the **e3nn** tensor product to increase the performance of the tensor product by 2-10x(4-10x for relevant parameters). We also showed that the performance of the padded COO implementation is independent of the batch size and group size settings. We also identified a potential scaling problem with our approach that would limit speedup on higher values of lmax.

A. Related Work

For the tensor product itself, to the best of our knowledge the **e3nn-jax** tensor product is the state-of-the-art implementation, and there are no other public efforts to boost it's performance using it's intrinsic sparsity. While we ended up using Triton as our implementation of choice, we unsuccessfully attempted a lower level implementation using NVIDIA's cuSPARSE library [2]; the many dimensions of the einsum and converting some of the necessary functions to prepare the benchmark from **e3nn** was too difficult compared to focusing our efforts on Triton.

References

[1] J. Bradbury, R. Frostig, P. Hawkins, M. J. Johnson, C. Leary, D. Maclaurin, G. Necula, A. Paszke, J. VanderPlas, S. Wanderman-Milne, and Q. Zhang, "JAX: composable transformations of Python+NumPy programs," 2018. [Online]. Available: http://github.com/jax-ml/jax

[2] N. Corporation, *NVIDIA cuSPARSE Library*, 2024, version 12.6, available at https://developer.nvidia.com/cusparse.

[3] M. Geiger, T. Smidt, A. M., B. K. Miller, W. Boomsma, B. Dice, K. Lapchevskyi, M. Weiler, M. Tyszkiewicz, S. Batzner, D. Madisetti, M. Uhrin, J. Frellsen, N. Jung, S. Sanborn, M. Wen, J. Rackers, M. Rød, and M. Bailey, "Euclidean neural networks: e3nn," Apr. 2022. [Online]. Available: https://doi.org/10.5281/zenodo.6459381

[4] NVIDIA Corporation, *NVIDIA cuBLAS Library*, 2021, version 11.4. [Online]. Available: https://developer.nvidia.com/cublas

[5] N. Thomas, T. Smidt, S. Kearnes, L. Yang, L. Li, K. Kohlhoff, and P. Riley, "Tensor field networks: Rotation-and translation-equivariant neural networks for 3d point clouds," *arXiv preprint arXiv:1802.08219*, 2018.

[6] P. Tillet, H.-T. Kung, and D. 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*, 2019.