Optimizing 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 XXX× on a Nvidia RTX A4000 GPU.**

I. Introduction

II. Background

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.

```
# B : Batch size
# NNZ: Number of nonzeros
# W_i, W_j, W_k, W_val: weights in COO-format,
    dimensions [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(NNZ):
      nz_in1 = Input1[:,W_i[nz]]
      nz_in2 = Input2[:,W_j[nz]]
      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 [1]. 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 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.

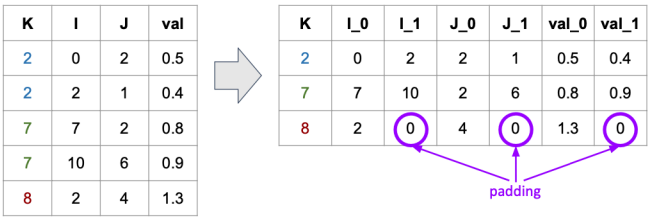

Fig. 1. Transforming COO to a grouped COO representation with a group size of 2.

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

References

[1] Tillet, Philippe, Hsiang-Tsung Kung, and David Cox. "Triton: an intermediate language and compiler for tiled neural network computations." Proceedings of the 3rd ACM SIGPLAN International Workshop on Machine Learning and Programming Languages. 2019.