Parallelization of Tensor Transpose Operation on GPUs

Problem Statement:

Parallelizing tensor transpose operation on GPUs. A tensor can be thought of as a high-dimensional generalization of vectors and matrices. Tensor transpose operation is a permutation of the indices of a tensor. That is, the transpose operation with an input tensor ‘A’ and an output tensor ‘B’ can be seen as $B_{\rho(i0,i1,i2,\cdots,id-1)} \leftarrow A_{i0,i1,i2,\cdots,id-1}$; where ρ denotes the index permutation for the transposition. A 2D tensor transpose is a matrix transpose: $B(i_1, i_0) \leftarrow A(i_0, i_1)$.

Motivation:

Tensor transpose is an important layout transformation primitive for many domains like machine learning, deep learning, computational chemistry, and so on that use tensor as a core data structure.

Algorithm:

A 6D tensor (rank=6) transpose operation with permutation ‘5 4 0 2 1 3 ’.

Algorithm: Direct loop based tensor transposition

```
// $out[i_6, i_5, i_1, i_3, i_2, i_4] \leftarrow in[i_1, i_2, i_3, i_4, i_5, i_6]$, range of each index = $N$
Procedure Transpose($in$, $out$)
    for $i_1 = 1$ to $N$ do
        for $i_2 = 1$ to $N$ do
            for $i_3 = 1$ to $N$ do
                for $i_4 = 1$ to $N$ do
                    for $i_5 = 1$ to $N$ do
                        for $i_6 = 1$ to $N$ do
                            $out[i_6, i_5, i_1, i_3, i_2, i_4] \leftarrow in[i_1, i_2, i_3, i_4, i_5, i_6]$
                        end
                    end
                end
            end
        end
    end
```

Note: The above algorithm shows the index range as N for all the indices. You implementation should be for different ranges for different indices.

Challenges:

Achieving coalesced memory accesses for both input and output tensors.

```
Algorithm: Direct loop based tensor transposition
// $out[i_6, i_5, i_1, i_3, i_2, i_4] \leftarrow in[i_1, i_2, i_3, i_4, i_5, i_6]$, range of each index $= N$
Procedure Transpose(in, out)
    for $i_1 = 1$ to $N$ do
        for $i_2 = 1$ to $N$ do
            for $i_3 = 1$ to $N$ do
                for $i_4 = 1$ to $N$ do                                    $\mathcal{O}(N^6)$ moves
                    for $i_5 = 1$ to $N$ do
                        for $i_6 = 1$ to $N$ do
                            $\underbrace{out[i_6, i_5, i_1, i_3, i_2, i_4] \leftarrow in[i_1, i_2, i_3, i_4, i_5, i_6]}_{\text{High access stride for the output tensor}}$
                        end
                    end
                end
            end
        end
    end
```

Main focus:

Create two implementations: (1) targeting dense tensors (2) targeting sparse tensors. Main focus should be given for improving the performance of sparse tensor transpositions.

Aim for an implementation that gives good performance for varying ranks, varying permutations and varying index ranges.

Correctness:

Test the output of the parallel implementation against the sequential implementation.

Input+Output:

Input: Rank of a tensor, Index Ranges, Permutation. Input can be taken as command-line arguments.

For example: Command-line arguments “6   16 18 20 32 16 17   1 3 0 2 5 4” represents the input with a 6D tensor, with index ranges `16 18 20 32 16 17’ and the tensor transpose operation should be performed for the permutation `1 3 0 2 5 4’.

Initialize the input tensor with dummy double/float values. For example:

```
 void transpose_init(type *A, int size)
 {   int i = 0;
        for(i = 0; i < size; i++){
               A[i] = i;
        }
}
```

Output: Transposed tensor.

Benchmarks:

To create a diverse dataset, consider several transpose test cases that cover different ranks, volumes, extents (sizes of dimensions/index ranges) and orderings among the extents.

(1) Tensor ranks ranging from 3 to 6 and include all possible permutations.

(2) Volumes of the tensors ranging from 16MB to 2GB.

(3) Different orderings among extents include (example shown in brackets for a 3D tensor with indices i_0, i_2, $i_{3,}$ s_j is the extent of i_j): (i) all same ($s_0 = s_1 = s_2$) (ii) monotonically increasing ($s_0 < s_1 < s_2$) (iii) monotonically decreasing ($s_0 > s_1 > s_2$) (iv) increasing till the center dimension and then decreasing ($s_0 < s_1 > s_2$) (v) decreasing till the center dimension and then increasing ($s_0 > s_1 < s_2$).