Multi-Device ONNX Proposal

Micah Villmow

Motivation

- The trend in large language models is pushing beyond the boundaries of a single device.
- Mapping network ops to devices efficiently is important for high performance execution of neural networks.
- No way to represent this in ONNX.
- With the entire graph in ONNX, custom software to handle cross device communication is not required.

New Concepts

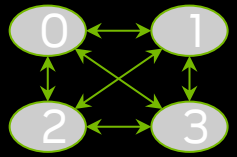
- **Instance** - A representation of an abstract computing machine that executes part of the ONNX network, i.e. a virtual device.
- **Topology** - The hierarchical ordering of instances such that the communication latency between two instances is lowest between an instance and its siblings.
- **Tiling pattern** - Vector of length rank(tensor), or scalar 1, that specifies the method of splitting the tensor into multiple tiles.
- **Tile** - A segment of a tensor that is produced by a network operation for a specific instance.
- **Tiling assignment** - Vector that maps from tiles to instances.
- **TensorDictionary** - A Tensor representation that is assigned to one or more instances.

Model extension

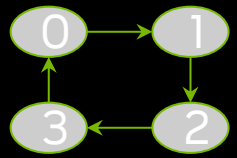
Key	Value	Description
`Model.topology`	**TopoProto**	An optional extension that defines the communication network topology information this ONNX model was defined for.
`TopoProto.instance_count`	**int64**	The maximum number of virtual ID's that can be assigned to tensors within ONNX model. This value MUST be positive. If this instance metadata is missing, then the assumed value can be one.
`TopoProto.instance_map`	**int64[][]**	A list of lists, where the I`th entry contains a list of instance IDs that the I`th entry is connected to. The I`th entry implicitly contains its own instance, since an instance can always connect with itself, so specifying that entry is not required.

Topology Examples

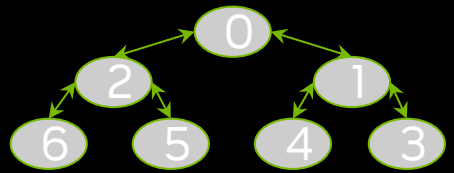
Fully Connected Mesh: {4, [[1,2,3],[2,3,0],[3,0,1],[0,1,2]]}



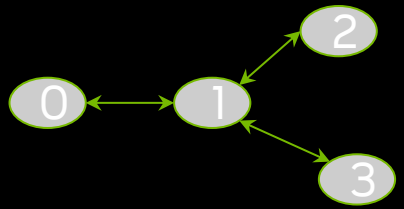
Unidirectional Ring: {4, [[1],[2],[3],[0]]}



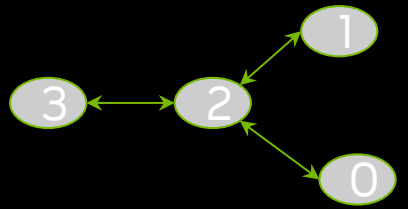
Tree: {7, [[1,2],[0,3,4],[0,5,6],[1],[1],[2],[2]]}



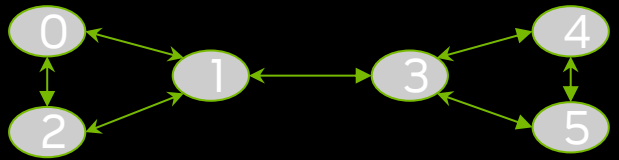
1-Centric star: {4, [[1],[0,2,3],[1],[1]]}



2-Centric star: {4, [[2],[2],[0,1,3],[2]]}

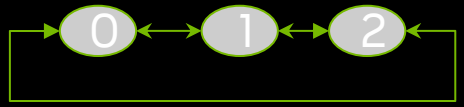


Disjoint rings w/ single crosslink:
{6, [[1,2],[0,2,3],[1,2],[2,4,5],[3,5][3,4]}

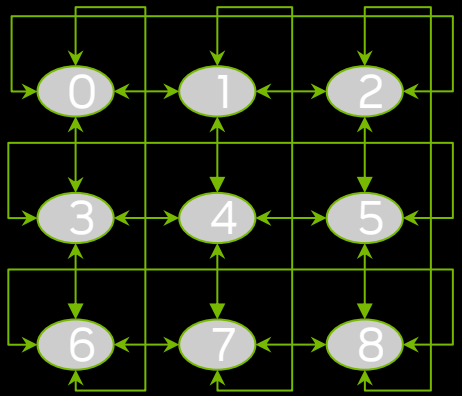


Torus topology

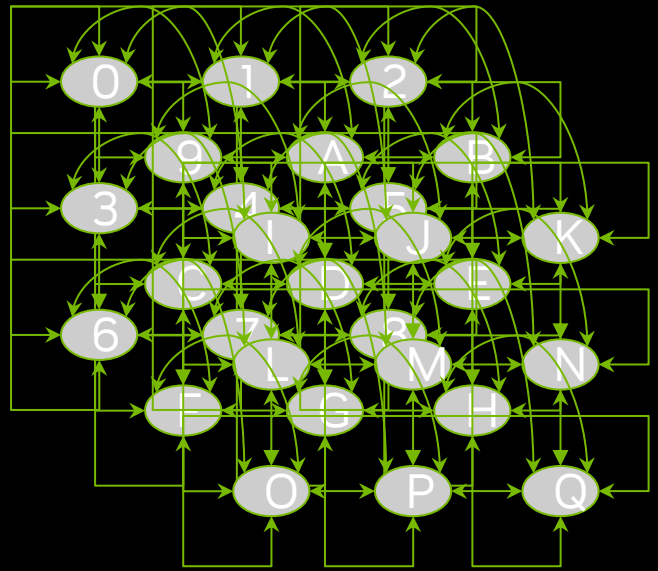
1-D: {3, [[1,2],[0,2],[1,2]]}



2-D: {9, [[1,2,3,6],[0,2,4,7],[1,2,5,8],
[4,5,0,6],[3,5,1,7],[3,4,2,8],
[0,3,7,8],[1,4,6,8],[2,5,6,7]]



3-D: {27, [[1,2,3,6,9,I,],[0,2,4,7,A,J],[1,2,5,8,B,K],
[4,5,0,6,C,L,],[3,5,1,7,D,M],[3,4,2,8,E,N],
[0,3,7,8,F,O],[1,4,6,8,G,P],[2,5,6,7,H,Q],...]



Tensor/ValueInfo extensions

Key	Value	Description
`Tensor.tile_pattern` `ValueInfo.tile_pattern`		Optional field that contains the pattern that splits the tensor into multiple tiles that are mappable onto instances. The tile pattern has the same rank as the Tensor itself, or a scalar with value 1. If the tensor pattern is a scalar 1, then the entire tensor is the same size across all instances, since slicing the tensor by 1 is an identity operation. If the pattern does not exist, then the tile pattern is left up to the importer to determine. For simplification, if the leading dimensions are all 1, then they can be omitted.
`Tensor.tile_assignment` `ValueInfo.tile_assignment`		Optional field that specifies the instances that the tiles are assigned to. If the tile_pattern is a scalar of value 1, then there can be multiple instances and the tensor is duplicated across all instances. Otherwise, the size of the tile_assignment list must be equal to the volume of the tile_pattern. If there are no tile_assignments, and a tile_pattern exists, then the assignment is [0, volume(tile_pattern)-1]. The I`th entry in the tile_assignment list is assigned the to the I`th tile after applying the tile_pattern size formula to the tensor shape. If the tile assignment entry is -1, then it is suppressed.

Tile computations

```
entries = volume(pattern);
if (entries == 1) entries = rank(assignment);
```

Tile data start computation

```
for (j: entries)
    for (i: rank(S))
        V_start[j][i] = floor(((volume(pattern) != 1 ? j : 0) % pattern[i % rank(pattern)]) * V_orig[i] /
pattern[i % rank(pattern)])
```

Tile data stop computation

```
for (j: entries)
    for (i: rank(S))
        V_stop[j][i] = min(V_orig[i], floor((((volume(pattern) != 1 ? j : 0) % pattern[i % rank(pattern)]) + 1) *
V_orig[i] / pattern[i % rank(pattern)]))
```

Tile data size computation

```
for (j: rank(assignment))
    for (i: rank(S))
        V_size [assignment[j]][i] = V_stop[j][i] - V_start[j][i]
```

Examples

Tensor	Pattern	Assignment	Tile Start	Tile Stop	Tile Sizes
{1, 4}	{1, 4} count = 4	{0, 1, 2, 3} Instances = 4	0: {0, 0} 1: {0, 1} 2: {0, 2} 3: {0, 3}	0: {1, 1} 1: {1, 2} 2: {1, 3} 3: {1, 4}	0: {1, 1} 1: {1, 1} 2: {1, 1} 3: {1, 1}
{1, 4}	{1, 4} count = 4	Default instance count	0: {0, 0} 1: {0, 1} 2: {0, 2} 3: {0, 3}	0: {1, 1} 1: {1, 2} 2: {1, 3} 3: {1, 4}	0: {1, 1} 1: {1, 1} 2: {1, 1} 3: {1, 1}
{7, 4}	{5, 1} count = 5	{3, 2, 4, 1, 0} instances = 5	0: {5, 0} 1: {4, 0} 2: {1, 0} 3: {0, 0} 4: {2, 0}	0: {7, 4} 1: {5, 4} 2: {2, 4} 3: {1, 4} 4: {4, 4}	0: {2, 4} 1: {1, 4} 2: {1, 4} 3: {1, 4} 4: {2, 4}
{4, 4, 2, 2}	{1, 3, 1, 1} count = 3	{2, 0, 3} instances = 3	0: {0, 1, 0, 0} 2: {0, 0, 0, 0} 3: {0, 2, 0, 0}	0: {4, 2, 2, 2} 2: {4, 1, 2, 2} 3: {4, 4, 2, 2}	0: {4, 1, 2, 2} 2: {4, 1, 2, 2} 3: {4, 2, 2, 2}
{2, 4, 8}	{1} count = 1	{3, 2} instances = 2	2: {0, 0, 0} 3: {0, 0, 0}	2: {2, 4, 8} 3: {2, 4, 8}	2: {2, 4, 8} 3: {2, 4, 8}

TensorDictionaries(TDs)

Tensor for a single device

In
{1:0}

1-D Tensor duplicated across two devices

In
{1:0,1}

2-D Tensor split across two devices

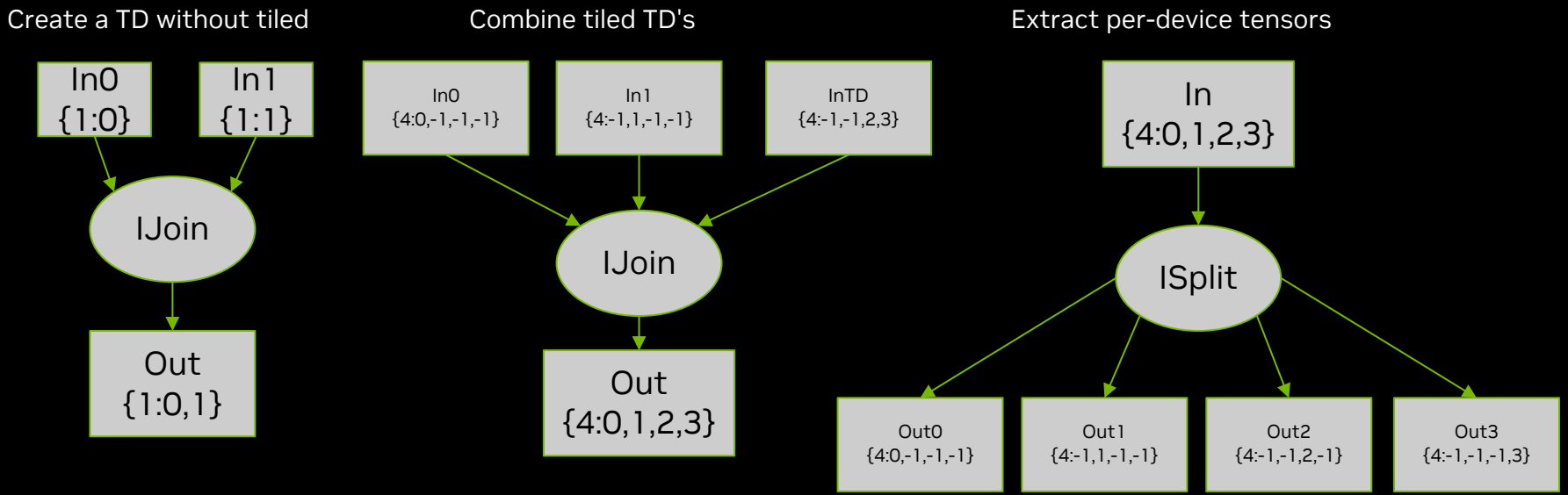
In
{2,1:0,1}

3-D Tensor split across 6 devices

In
{3,2,1:5,4,3,2,1,0}

How to create TDs?
How to index into TDs?
How to extract Tensors from TDs?

Tensor Dictionaries



Runtime evaluation

- Implementation performs input to output shape inference.
- Implementation performs output to input communication inference.
  - This is a naive implementation for exemplar purposes and can be further optimized.
  - If there is a mismatch on communication inference between input and output tile patterns
    - If input tile pattern is a scalar and output tile pattern is a vector, insert a broadcast, then tile the output per instance based on the earlier formulas.
    - If input tile pattern is a vector and output tile pattern is a scalar, insert an all-gather.
    - If input tile pattern is a vector and output tile pattern is a vector, insert an all-gather, then tile the output per instance based on the earlier formulas.

Questions?