2.2 Writing Efficient Message Passing Code

(中文版)

DGL optimizes memory consumption and computing speed for message passing. A common practise to leverage those optimizations is to construct one's own message passing functionality as a combination of `update_all()` calls with built-in functions as parameters.

Besides that, considering that the number of edges is much larger than the number of nodes for some graphs, avoiding unnecessary memory copy from nodes to edges is beneficial. For some cases like `GATConv`, where it is necessary to save message on the edges, one needs to call `apply_edges()` with built-in functions. Sometimes the messages on the edges can be high dimensional, which is memory consuming. DGL recommends keeping the dimension of edge features as low as possible.

Here's an example on how to achieve this by splitting operations on the edges to nodes. The approach does the following: concatenate the `src` feature and `dst` feature, then apply a linear layer, i.e. $W \times (u||v)$. The `src` and `dst` feature dimension is high, while the linear layer output dimension is low. A straight forward implementation would be like:

```
import torch
import torch.nn as nn

linear = nn.Parameter(torch.FloatTensor(size=(node_feat_dim * 2, out_dim)))
def concat_message_function(edges):
     return {'cat_feat': torch.cat([edges.src['feat'], edges.dst['feat']], dim=1)}
g.apply_edges(concat_message_function)
g.edata['out'] = g.edata['cat_feat'] @ linear
```

The suggested implementation splits the linear operation into two, one applies on `src` feature, the other applies on `dst` feature. It then adds the output of the linear operations on the edges at the final stage, i.e. performing $W_l \times u + W_r \times v$. This is because $W \times (u||v) = W_l \times u + W_r \times v$, where W_l and W_r are the left and the right half of the matrix W, respectively:

```
import dgl.function as fn

linear_src = nn.Parameter(torch.FloatTensor(size=(node_feat_dim, out_dim)))
linear_dst = nn.Parameter(torch.FloatTensor(size=(node_feat_dim, out_dim)))
out_src = g.ndata['feat'] @ linear_src
out_dst = g.ndata['feat'] @ linear_dst
g.srcdata.update({'out_src': out_src})
g.dstdata.update({'out_dst': out_dst})
g.apply_edges(fn.u_add_v('out_src', 'out_dst', 'out'))
```

The above two implementations are mathematically equivalent. The latter one is more efficient because it does not need to save feat_src and feat_dst on edges, which is not memory-efficient. Plus, addition could be optimized with DGL's built-in function `u_add_v()`, which further speeds up computation and saves memory footprint.