2.2 编写高效的消息传递代码

(English Version)

DGL优化了消息传递的内存消耗和计算速度。利用这些优化的一个常见实践是通过基于内置函数的 `update_all()` 来开发消息传递功能。

除此之外，考虑到某些图边的数量远远大于节点的数量，DGL建议避免不必要的从点到边的内存拷贝。对于某些情况，比如 `GATConv`，计算必须在边上保存消息，那么用户就需要调用基于内置函数的 `apply_edges()`。有时边上的消息可能是高维的，这会非常消耗内存。DGL建议用户尽量减少边的特征维数。

下面是一个如何通过对节点特征降维来减少消息维度的示例。该做法执行以下操作：拼接 源 节点和 目标 节点特征，然后应用一个线性层，即 $W \times (u||v)$。 源 节点和 目标 节点特征维数较高，而线性层输出维数较低。一个直截了当的实现方式如下：

```
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
```

建议的实现是将线性操作分成两部分，一个应用于 源 节点特征，另一个应用于 目标 节点特征。在最后一个阶段，在边上将以上两部分线性操作的结果相加，即执行 $W_l \times u + W_r \times v$，因为 $W \times (u||v) = W_l \times u + W_r \times v$，其中 W_l 和 W_r 分别是矩阵 W 的左半部分和右半部分：

```
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'))
```

以上两个实现在数学上是等价的。后一种方法效率高得多，因为不需要在边上保存feat_src和feat_dst， 从内存角度来说是高效的。另外，加法可以通过DGL的内置函数 `u_add_v` 进行优化，从而进一步加快计算速度并节省内存占用。