6.7 Using GPU for Neighborhood Sampling

DGL since 0.7 has been supporting GPU-based neighborhood sampling, which has a significant speed advantage over CPU-based neighborhood sampling. If you estimate that your graph can fit onto GPU and your model does not take a lot of GPU memory, then it is best to put the graph onto GPU memory and use GPU-based neighbor sampling.

For example, OGB Products has 2.4M nodes and 61M edges. The graph takes less than 1GB since the memory consumption of a graph depends on the number of edges. Therefore it is entirely possible to fit the whole graph onto GPU.

Using GPU-based neighborhood sampling in DGL data loaders

One can use GPU-based neighborhood sampling with DGL data loaders via:

- Put the graph onto GPU.
- Put the `train_nid` onto GPU.
- Set `device` argument to a GPU device.
- Set `num_workers` argument to 0, because CUDA does not allow multiple processes accessing the same context.

All the other arguments for the `DataLoader` can be the same as the other user guides and tutorials.

```
g = g.to('cuda:0')
train_nid = train_nid.to('cuda:0')
dataloader = dgl.dataloading.DataLoader(
    g,                                  # The graph must be on GPU.
    train_nid,                          # train_nid must be on GPU.
    sampler,
    device=torch.device('cuda:0'),      # The device argument must be GPU.
    num_workers=0,                      # Number of workers must be 0.
    batch_size=1000,
    drop_last=False,
    shuffle=True)
```

Note

GPU-based neighbor sampling also works for custom neighborhood samplers as long as (1) your sampler is subclassed from `BlockSampler`, and (2) your sampler entirely works on GPU.

Using CUDA UVA-based neighborhood sampling in DGL data loaders

Note

New feature introduced in DGL 0.8.

For the case where the graph is too large to fit onto the GPU memory, we introduce the CUDA UVA (Unified Virtual Addressing)-based sampling, in which GPUs perform the sampling on the graph pinned in CPU memory via zero-copy access. You can enable UVA-based neighborhood sampling in DGL data loaders via:

- Put the `train_nid` onto GPU.
- Set `device` argument to a GPU device.
- Set `num_workers` argument to 0, because CUDA does not allow multiple processes accessing the same context.
- Set `use_uva=True`.

All the other arguments for the `DataLoader` can be the same as the other user guides and tutorials.

```
train_nid = train_nid.to('cuda:0')
dataloader = dgl.dataloading.DataLoader(
    g,
    train_nid,                          # train_nid must be on GPU.
    sampler,
    device=torch.device('cuda:0'),      # The device argument must be GPU.
    num_workers=0,                      # Number of workers must be 0.
    batch_size=1000,
    drop_last=False,
    shuffle=True,
    use_uva=True)                       # Set use_uva=True
```

UVA-based sampling is the recommended solution for mini-batch training on large graphs, especially for multi-GPU training.

Note

To use UVA-based sampling in multi-GPU training, you should first materialize all the necessary sparse formats of the graph before spawning training processes. Refer to our GraphSAGE example for more details.

UVA and GPU support for PinSAGESampler/RandomWalkNeighborSampler

PinSAGESampler and RandomWalkNeighborSampler support UVA and GPU sampling. You can enable them via:

- Pin the graph (for UVA sampling) or put the graph onto GPU (for GPU sampling).
- Put the `train_nid` onto GPU.

```
g = dgl.heterograph({
    ('item', 'bought-by', 'user'): ([0, 0, 1, 1, 2, 2, 3, 3], [0, 1, 0, 1, 2, 3, 2, 3]),
    ('user', 'bought', 'item'): ([0, 1, 0, 1, 2, 3, 2, 3], [0, 0, 1, 1, 2, 2, 3, 3])})

# UVA setup
# g.create_formats_()
# g.pin_memory_()

# GPU setup
device = torch.device('cuda:0')
g = g.to(device)

sampler1 = dgl.sampling.PinSAGESampler(g, 'item', 'user', 4, 0.5, 3, 2)
sampler2 = dgl.sampling.RandomWalkNeighborSampler(g, 4, 0.5, 3, 2, ['bought-by', 'bought'])

train_nid = torch.tensor([0, 2], dtype=g.idtype, device=device)
sampler1(train_nid)
sampler2(train_nid)
```

Using GPU-based neighbor sampling with DGL functions

You can build your own GPU sampling pipelines with the following functions that support operating on GPU:

- `dgl.sampling.sample_neighbors()`
- `dgl.sampling.random_walk()`

Subgraph extraction ops:

- `dgl.node_subgraph()`
- `dgl.edge_subgraph()`
- `dgl.in_subgraph()`
- `dgl.out_subgraph()`

Graph transform ops for subgraph construction:

- `dgl.to_block()`

- `dgl.compact_graph()`