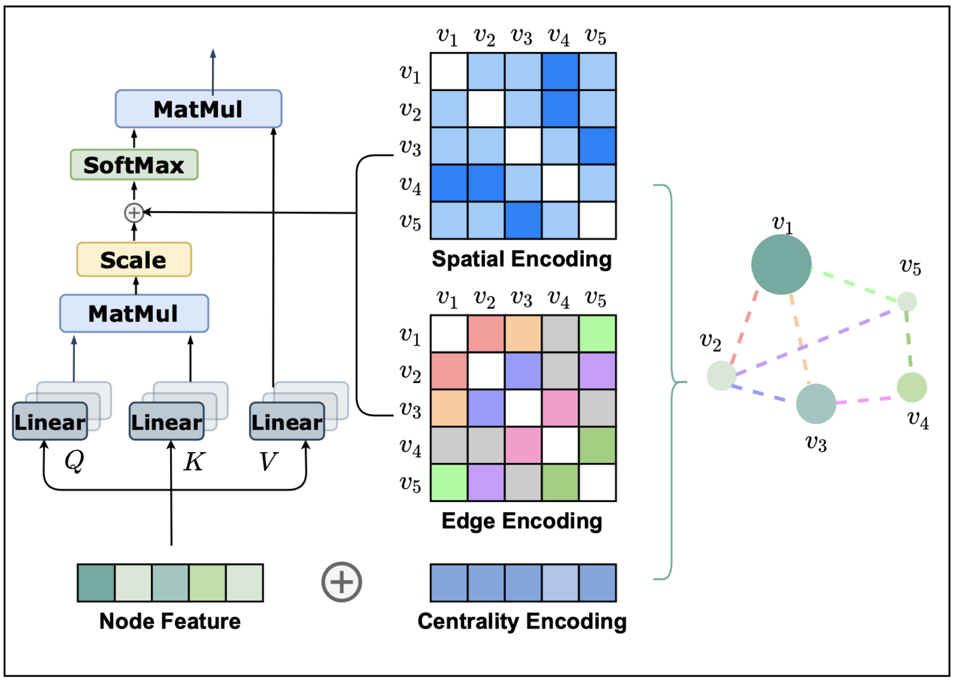
Graphormer Porting Details



This document contains the details for converting the Graphormer implementation to run on Graphcore's IPU.

@Work by MegazoneCloud & Graphcore for KPBMA.

Setting Up the Environment

1. conda create —name graphormerenv python=3.6
2. pip3 install poptorch 2.3
3. pip3 install torch==1.9.0+cu111 torchvision torchaudio -f https://download.pytorch.org/whl/torch_stable.html --upgrade-strategy only-if-needed
4. pip3 install torch-geometric ogb pytorch-lightning tqdm torch-sparse torch-scatter -f https://pytorch-geometric.com/whl/torch-1.9.0+cu111.html --upgrade-strategy only-if-needed
5. conda install -c rdkit rdkit cython

Making a Debugging Dataset

- The original PCQM4-LSC dataset is very large. While debugging, it is very time-consuming and inefficient to load the whole dataset.
- In order to shrink the dataset, we need a smaller SMILES dataset along with a 'split_dict.pt' file.

1. Make a smaller raw 'data.csv.gz' file. In this case, the dataset size will be 10,000.
    a. This must be done without deleting the last line.
        i. Doing so will corrupt the .gz file and make it unloadable.
    b. The last 1000 examples must not have a label. These are the test data.
2. Make a new 'split_dict.pt' file.
    a. This is done by using save_split in save_dataset.py in the package 'ogb.io'
    b. This produces a new split_dict.pt file that can be used to split our new smaller dataset.

```
from ogb.io import DatasetSaver
import numpy as np
import networkx as nx
import os
import torch
# constructor

dataset_name = 'ogbg-toy'
saver = DatasetSaver(dataset_name = dataset_name, is_hetero = False, version = 1)

num_data=10000
split_idx = dict()
permTrainVal = np.random.permutation(num_data*0.9) # 0 ~ 9k
permTest = np.random.permutation(np.arange(num_data*9,num_data)) # 9k ~ 10k

split_idx['train'] = permTrainVal[:int(0.8*num_data)]
split_idx['valid'] = permTrainVal[int(0.8*num_data): int(0.9*num_data)]
split_idx['test'] = permTest[int(0.9*num_data):]

torch.save(split_idx, '../split_dict.pt')
```

Code Change Summary.

1. Migration from Pytorch Lightning to Pytorch + Poptorch.
2. In collator.py, changes in the batch datatype, and added fixed paddings.
3. Change of Optimizer. AdamW → Adam.

4. Unsupported operation in Graphormer's forward().
5. Moving Loss function from training loop to inside the model.
6. Pipelining the Model for Model Parallelism. Edit in forward().
7. Update optimizer after lr_scheduler is called.

Code Change Details

1. Lightning → Pytorch + Poptorch

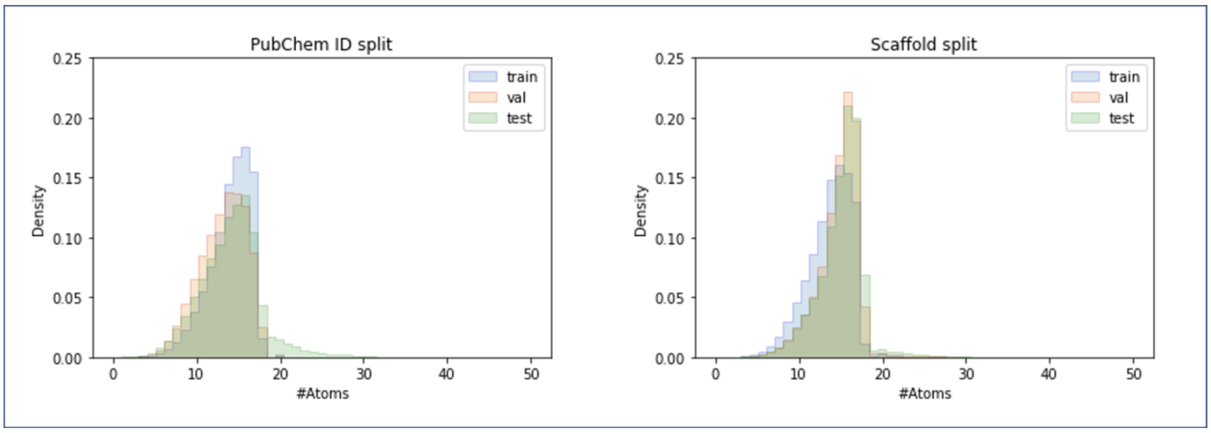
- Pytorch Lightning recently announced its support for the IPU, but there is still a lot of difficulty with debugging.
    - Lack of documentation + Numerous Layers of Wrappers.
- Therefore, the code was migrated from Pytorch Lightning to normal Pytorch.
- There should be no changes to the code itself, just differences in the expression.

---

2. Changes to Batch in collator.py

- Graphormer implements its own Batch class, which is not recognized by Poptorch.
    - When using the IPU, because of the static nature of the computational graph, it is recommended that the input variables be immutable objects such as torch.Tensor or Tuple(torch.tensor).
    - The collator was changed to return a simple Tuple containing the input tensors. **All other code that accesses batched_data, such as forward() in model.py, were edtied accordingly.**
- Additionally, input tensor sizes were fixed in collator.py by setting
    - max_node_num = 30
    - max_dist = 30
    - in collator().

- Since we arbitrarily fix the node size, there can be a sample in the batch which has a size greater than 30.
- This will result in an error.
- Therefore, we use list comprehension to remove all samples larger than 30 and replace them with a random sample from the same batch.



- This is because static computational graphs require the same tensor sizes for every step.
- The following code changes were made in collator.py.

```
#ORIGINAL
def collator(items, max_node, multi_hop_max_dist, rel_pos_max):
  items = [item for item in items if item is not None and item.x.size(0) <= max_node]
  .
  .
  .
  max_node_num = max(i.size(0) for i in xs)
  max_dist = max(i.size(-2) for i in edge_inputs)
  .
  .
  .
  return Batch(
        idx=torch.LongTensor(idxs),
        attn_bias=attn_bias,
        attn_edge_type=attn_edge_type,
        rel_pos=rel_pos,
        all_rel_pos_3d_1=all_rel_pos_3d_1,
        in_degree=in_degree,
        out_degree=out_degree,
        x=x,
        edge_input=edge_input,
        y=y,
    )

#FIXED
def collator(items, max_node, multi_hop_max_dist, rel_pos_max):
  batch_size = len(items)
  items = [item for item in items if item is not None and item.x.size(0) <= max_node]

  missing_items = batch_size - len(items)
  for i in range(missing_items):
      random_idx = randint(0,batch_size-missing_items-1)
      items.append(items[random_idx])
  .
  .
  .
  max_node_num = 30
  max_dist = 30
  .
  .
  .
  returningTuple = (torch.LongTensor(idxs),attn_bias,attn_edge_type,rel_pos,all_rel_po
s_3d_1,in_degree,out_degree,x,edge_input,y)

#idx=[0], attn_bias=[1], attn_edge_type=[2], rel_pos=[3], all_rel_pos_3d_1=[4],
#in_degree=[5], out_degree=[6], x=[7], edge_input=[8], y=[9]

  return returningTuple
```

3. Optimizer changes. AdamW to Adam.

- AdamW returns the following error.

```
RuntimeError: ERROR in poptorch/python/poptorch.cpp:1141: 'std::runtime_error' excepti
on: createOutputForElementWiseOp '_adamupdater/7655/popops::map::Cast_ADDu_Cast_DIVu_C
ast_DIVu_float_1__Cast_SUBu_1zf_POWu_0z899999976f_Cast_float_4__float_d_d_float_d_floa
t_ADDu_SQUu_DIVu_float_2__Cast_SUBu_1zf_POWu_0z999000013f_Cast_float_4__float_d_d_floa
t_d_d_9z99999994em09f_d_d_float_MULu_Cast_float_5__float_float_3__d_d_float0111111/Ou
t': Shapes of input tensors do not match Context: LowerToPopart::compile  Compiler::co
mpileAndPrepareDevice popart::Session::prepareDevice: Poplar compilation
```

- Seems like a bug. The same problem occurred with another Transformer model.
- As a placeholder, the model currently uses Adam. Model convergence is not affected.

4. Handling Unsupported Operation.

- Graphormer's forward() has a certain operation,

```
rel_pos_[rel_pos_ == 0] = 1
```

- that sets all values of 0 in tensor 'rel_pos_' to 1.
- This is for setting the padding of the tensors to 1.
- Without this line of code, division by 0 will occur.
- Indexing with boolean tensors / masks is not supported with PopART, so it is replaced by the following method:

```
rel_pos_ = torch.masked_fill(rel_pos_, rel_pos_ == 0, 1)
```

---

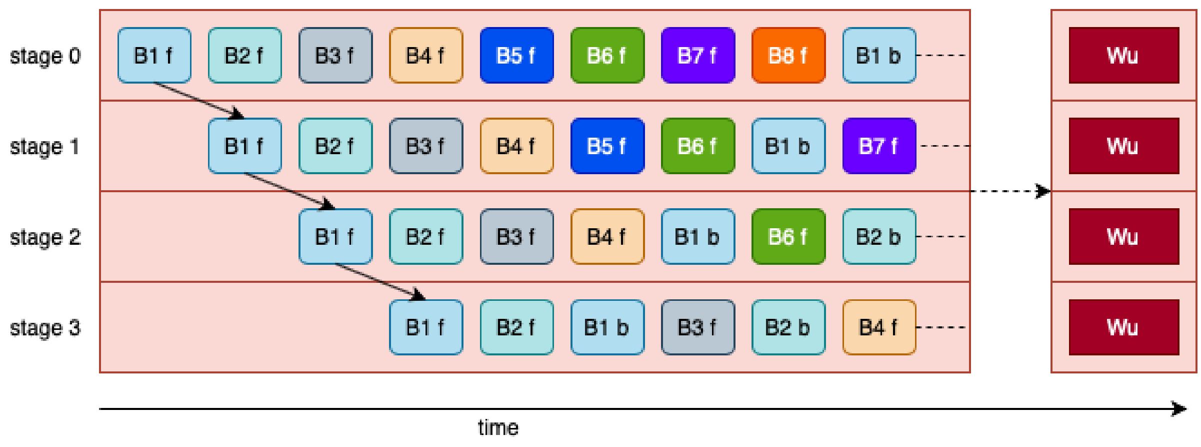
5. Moving Loss Calculation to forward().

- Normally with Pytorch, we calculate the loss, calculate the gradients, and perform optimizer.step() in the training_loop.
- Poptorch automates this process, so the model must be compiled with knowledge of how the loss is calculated.
- Therefore, we need to move the loss function into the model's forward().

```
def forward():
  .
  .
  .

  output = self.final_ln(output)
  output = self.out_proj(output[:, 0, :])

  y_hat = output.view(-1)
  y_gt = batched_data[9].view(-1)
  loss = self.loss_fn(y_hat,y_gt)

  return output, loss
```

6. Pipelining the Model for Model Parallelism.

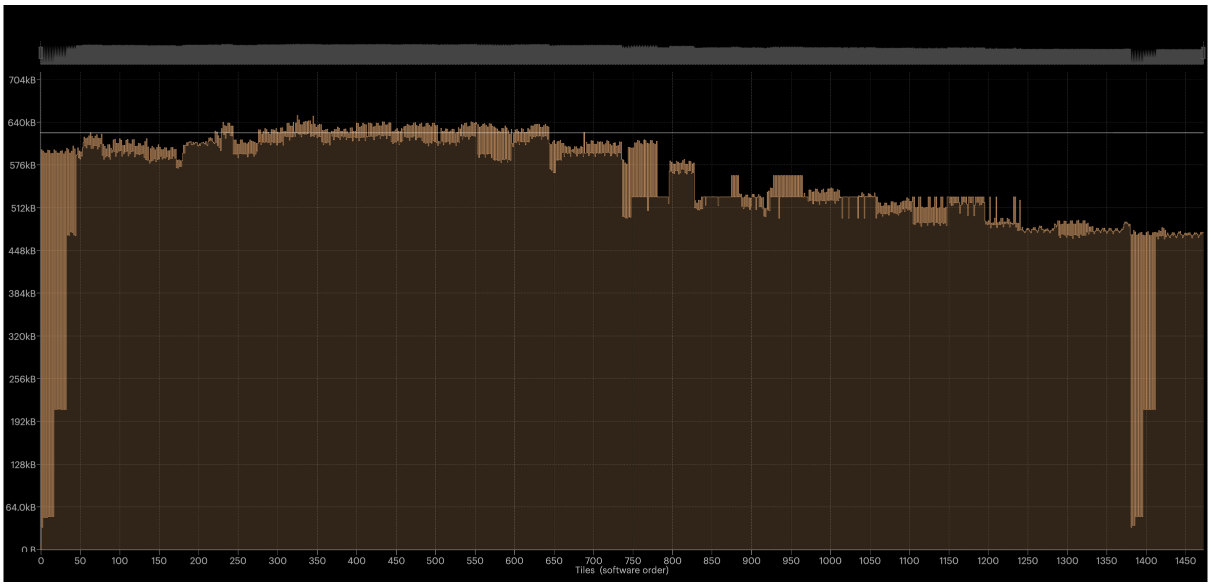
- When the model is larger than the available on-chip memory of the IPU, we split the model onto separate devices.
- This is done by defining which parts of the model go onto which IPU.
- Why is this considered model parallelism? We can utilize gradient accumulation to process multiple batches at once. This allows us to keep all of the devices busy.



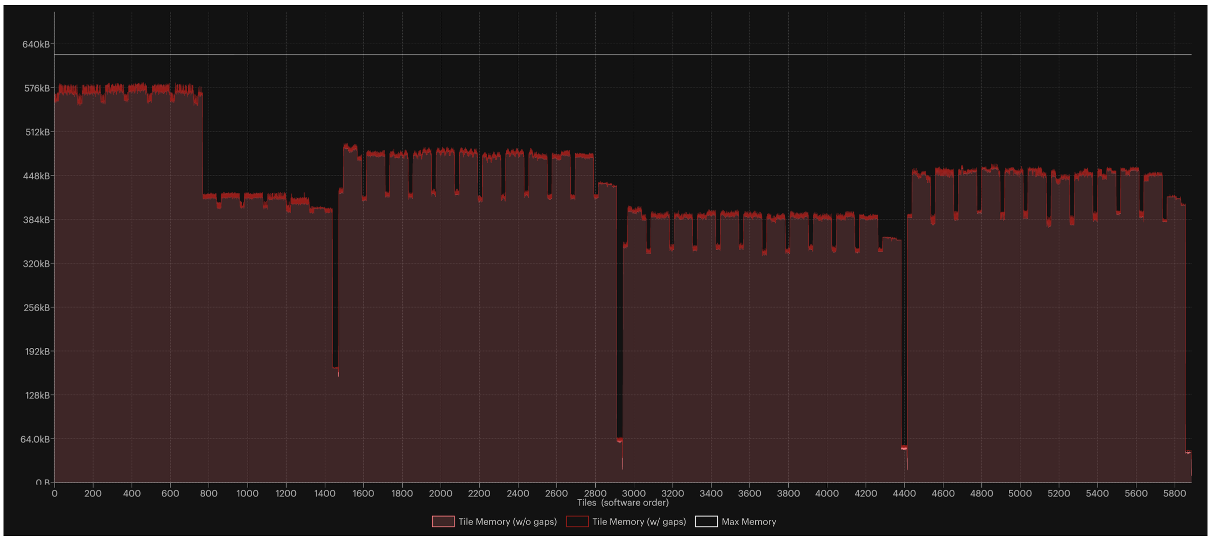
- For example, when a certain batch X's forward propagation is still being processed, a device can perform the forward/backward propagation of a different batch Y.
- This allows us to simultaneously run multiple batches on multiple devices, which we consider as parllelism.
- By utilizing ipu_utils.py, we can split the model through the command line.

```
# at train.sh
python3 src/entry.py --mini_batch_size 28 --gradient_accumulation 32 --replication_fac
tor 4 --device_iteration 1 --pipeline_splits layers/1 layers/5 layers/8 \
                    --checkpoint 0 --test 0
```

- We can visualize how our split occupies the on-chip memory using Graphcore's PopVision Graph Analyzer.



- The first memory profile shows that the tiles from 250~650 cross the white line, which represents the maximum memory capacity of each IPU tile.



- Therefore we split the computation onto 4 different IPUs, which was found empirically to give the highest throughput for this model.

7. Updating the model's optimizer when calling lr_scheduler.step().

- Since the optimizer is uploaded with the model to the IPU, we need to update the optimizer on the IPU after each lr_scheduler.step() call.

```
# in training loop in entry.py

lr_scheduler.step()
training_model.setOptimizer(optimizer)
```

- Otherwise, the new learning rate will not be applied.

---

8. Results.

Time to Train

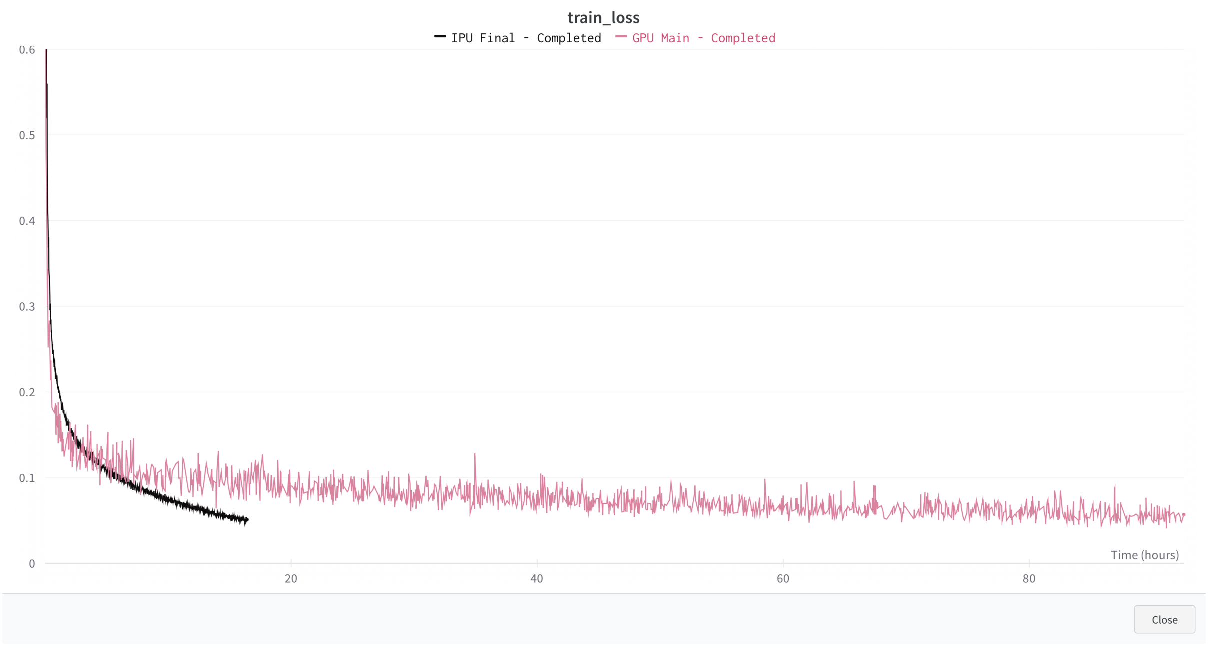
IPU * 16

- Time to Train -> 16.406 hours. Final train_loss == 0.05113

A100 * 1

- Time to Train -> 92.563 hours. Final train_loss == 0.05323

→ ~5.64x faster time to train on the IPU.



** Black: IPU, Pink: GPU.

** GPU on Pytorch Lightning. IPU on Pytorch + Poptorch. Overhead from logging could be different.

** Did not run validation loop during training on IPU.

** IPU did not use automatic mixed precision.

Throughput

IPU * 16

- Training Throughput -> ~15500 samples/s
- Inference Throughput → ~41300 samples/s

A100 * 1

- Training Throughput -> ~1500 samples/s
- Inference Throughput → ~9000 samples/s

→ ~10x higher training throughput.

** Throughput : samples processed by the device per second.

** A "process" refers to one iteration starting from optimizer.zero_grad() to optimizer.step().

** Throughput does not include lr_scheduler.step() on both sides.

** GPU Total Batch Size == 256, IPU Total Batch Size == 3584.

Final Validation MAE

IPU * 16

- Valid MAE → 0.05331

A100 * 1

- Valid MAE → 0.04727