名大スパコン「不老」でのLLMの利用

作成したリポジトリ
https://github.com/yu1349/llm_on_supercomp.git

25/06/19

渡邉優

本資料の主旨

●大規模計算環境でのLLMの利用

- 名大スパコン「不老」での実装
- ~~複数ノード~~複数GPUまでを順に実装

●想定聴講者

- 「不老」ユーザー
- 「大規模計算特論A or B」の履修者
- 「松原研究室」の皆さん

●渡邉が咀嚼した内容を話す

- ライブラリやアルゴリズムの詳説×

名大スパコン「不老」

●システム構成

- 機械学習向けは「Type Ⅱ」サブシステム
- GPU: Tesla V100 32GB
  - 松原研のGPU（48GB）よりVRAMが小さい

ハードウェア構成

機種名		FUJITSU Server PRIMERGY CX2570 M5
計算ノード	**CPU**	Intel Xeon Gold 6230, 20コア, 2.10 - 3.90 GHz × 2 ソケット
	GPU	NVIDIA Tesla V100 (Volta) SXM2, 2,560 FP64コア, upto 1,530 MHz × 4 ソケット
	メモリ	メインメモリ(DDR4 2933 MHz) 384 GiB (32 GiB × 6 枚 × 2 ソケット) デバイスメモリ(HBM2) 32 GiB × 4 ソケット
	理論演算性能	倍精度 33.888 TFLOPS (CPU 1.344 TFLOPS × 2 ソケット, GPU 7.8 TFLOPS × 4 ソケット)
	メモリバンド幅	メインメモリ 281.5 GB/s (23.464 GB/s × 6 枚(6チャネル) × 2 ソケット) デバイスメモリ 900 GB/s × 4 ソケット
	GPU間接続	NVLINK2 (1GPUから他の3GPUに対してそれぞれ50GB/s×双方向)
	CPU-GPU間接続	PCI-Express 3.0 (x16)

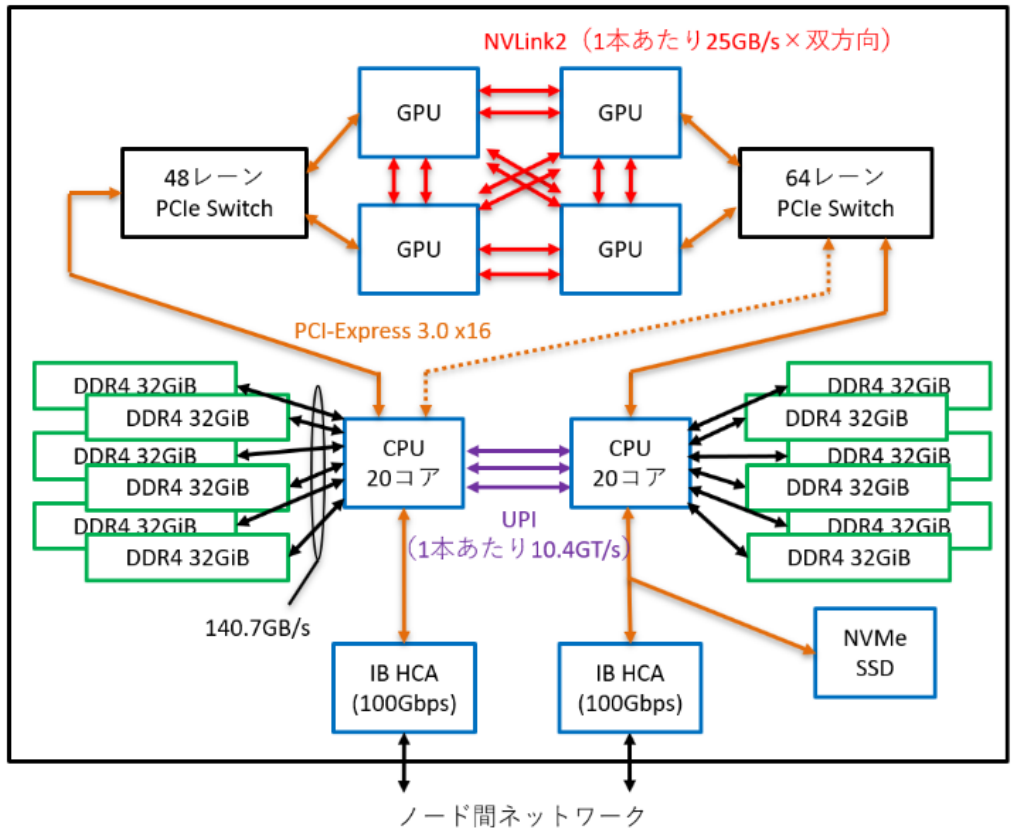
不老での計算環境（1/2）

●基本構成は1node4gpu

- GPU1枚であれば松原研GPUサーバの方が強い
- GPU4枚からスパコン利用を推奨

●モデルサイズとGPU枚数

- -10B: 32GB*1GPU
- -50B: 32GB*4GPU（1ノード）
- それ以上: 32GB*4GPU*Nノード



不老での計算環境（2/2）

●リソースグループの選択

- 用途に合わせて適切なものを選ぶ
- -shareのみGPU1枚

●小さい順に

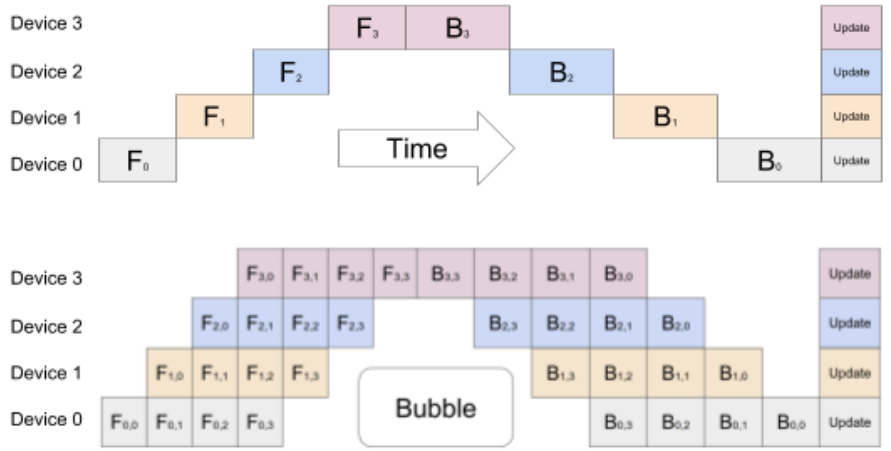
- cx-share: 1node1gpu
- cx-single: 1node4gpu
- cx-small: 8node32gpu
- 使ってもここら辺まで

リソースグループ名	最小ノード数	最大ノード数	最大CPUコア数	最長実行時間（デフォルト値）	最長実行時間（最大値）	最大メモリ容量（*）
cx-interactive	1	1	40	1時間	24時間	338 GiB
cx-debug	1	4	160	1時間	1時間	338 GiB x 4
cx-share	1/4(共有)	1/4(共有)	10	1時間	168時間	84 GiB
cx-single	1	1	40	1時間	336時間	338 GiB x 1
cx-small	1	8	320	1時間	168時間	338 GiB x 8
cx-middle	1	16	640	1時間	72時間	338 GiB x 16
cx-large	16	64	2,560	1時間	72時間	338 GiB x 64
cx-special	1	221	8,840	unlimited	unlimited	338 GiB x 221
cx-middle2	1	16	640	1時間	72時間	338 GiB x 16
cxgfs-interactive	1	1	40	1時間	168時間	338 GiB x 1
cxgfs-share	1/4（共有）	1/4（共有）	10	1時間	168時間	84 GiB x 1
cxgfs-single	1	1	40	1時間	336時間	338 GiB x 1
cxgfs-small	1	8	320	1時間	168時間	338 GiB x 8
cxgfs-middle	4	16	640	1時間	72時間	338 GiB x 16
cxgfs-special	1	50	2,000	1時間	72時間	338 GiB x 50

不老でのLLMの利用

●複数GPU環境でモデルを利用する必要

- GPU1枚当たりは32GBしかない
- **大規模な**LLMを分割して、各GPUにロード



Top: The naive model parallelism strategy leads to severe underutilization due to the sequential nature of the network. Only one accelerator is active at a time. Bottom: GPipe divides the input mini-batch into smaller micro-batches, enabling different accelerators to work on separate micro-batches at the same time.

出典: GPipe: Easy Scaling with Micro-Batch Pipeline Parallelism

環境設定

●スパコン

- cuda/12.4.1
- openmpi_cuda/4.0.5
- nccl/2.19.3

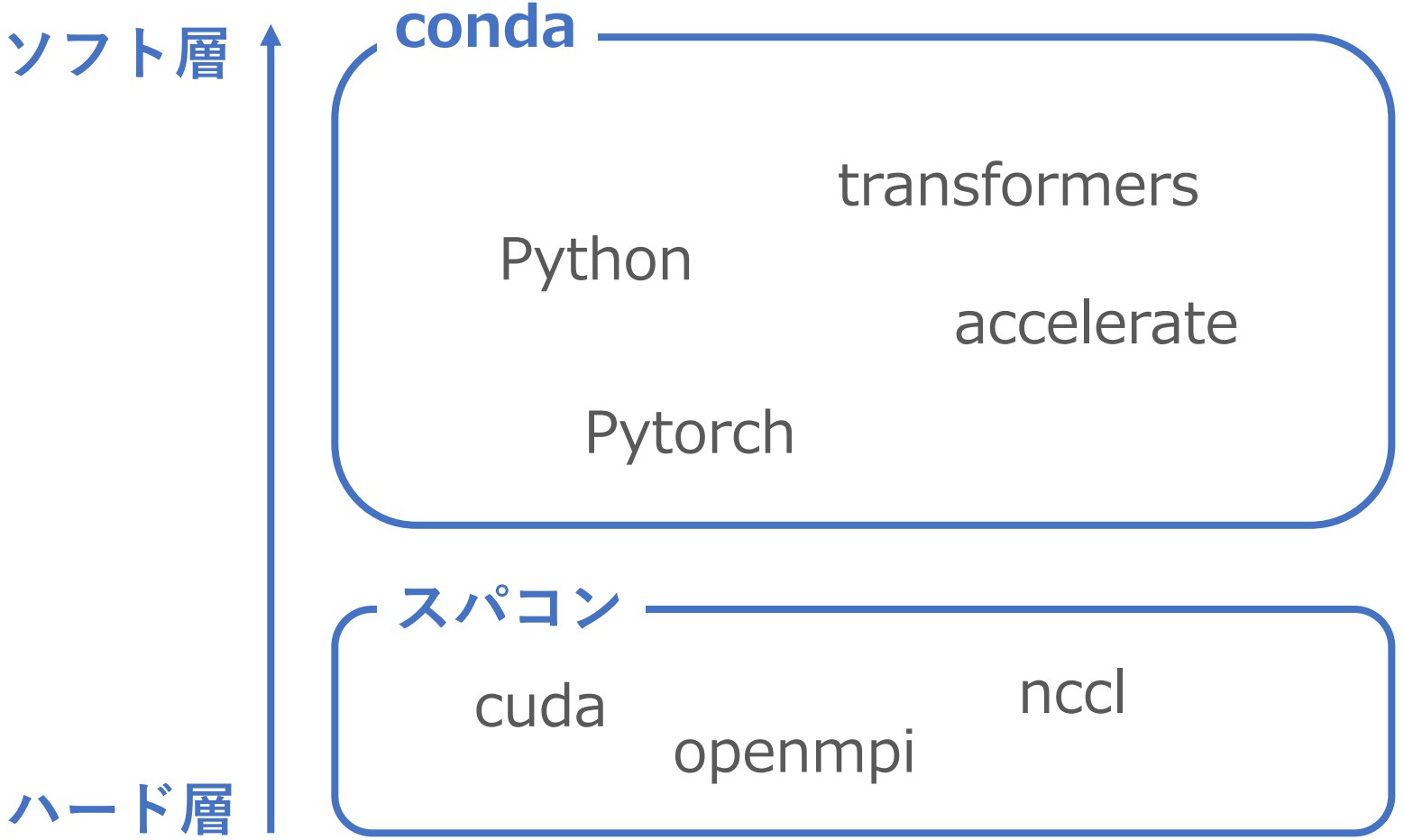
●~~Singularity (Docker)~~

- Pytorch/2.6.0↓condaへ

●Miniconda

- Python/3.11
- Transformers/4.51.3
- など

●環境のイメージ

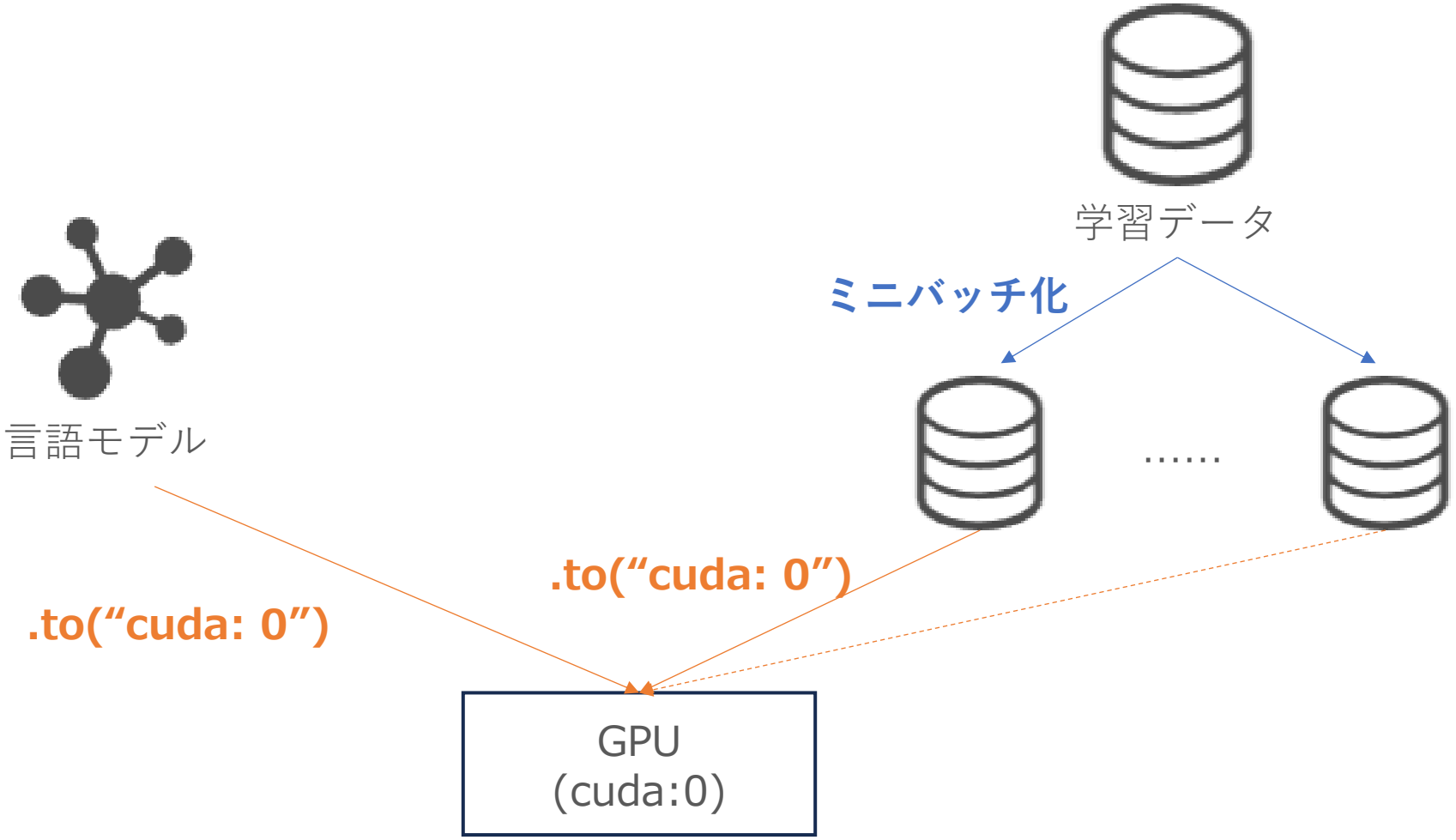


実装

- 1GPU
- 1node4gpu

1GPUでの推論

●概念



1GPUでの推論

●inference_on_1gpu.py

- main関数

```python
# 入力データの読み込み
jsonl_data = read_jsonl(args.input_data_path)
print("input_data_size:::", len(jsonl_data))
print("input_sample:::", jsonl_data[0])

# モデルとトークナイザーの初期化
## 本実装では読み込みの際はCPUにロードする -> pipe作成時に
tokenizer, model = intialize_tokenizer_and_model(args.m      _name_or_path)

# プロンプトの作成
input_texts = create_inst(jsonl_data, tokenizer)
input_dataset = Dataset.from_generator(lambda: metainfo_generator_fn(input_texts))
print("input_prompt:::", input_dataset.__getitem__(0))

# パイプラインの作成
pipe = pipeline(
    task="text-generation",                    # LLMはtext-generationで基本的にOK
    model=model,
    tokenizer=tokenizer,
    max_new_tokens=512,
    do_sample=False,
    device=0,                                  # GPU1枚なのでcuda:0を指定
    )

# 推論
outs = pipe(input_dataset['text'], truncation=True, padding=True, max_length=512)
```

Generatorを設定して
pipelineが使用できるように

GPU1枚なのでdevice=0

Pipelineの実行

●inference_on_1gpu.sh

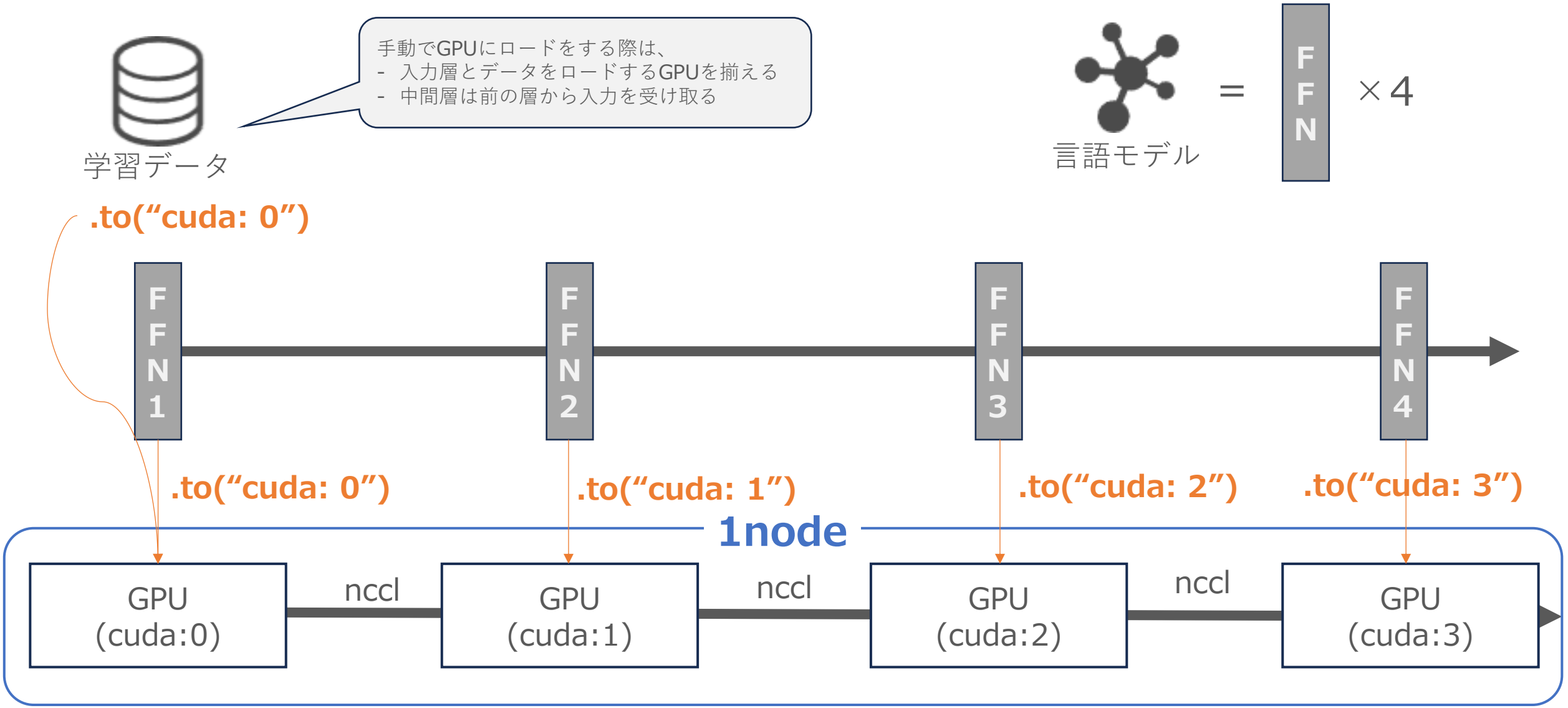
```bash
#!/bin/bash -x
#PJM -L rscgrp=cx-share
#PJM -L gpu=1
#PJM -L elapse=1:00:00  # REWRITE ME!!!!!
#PJM -j
#PJM -S

module load gcc/11.3.0 cuda/12.4.1 openmpi_cuda/4.0.5 nccl/2.19.3

eval "$(~/miniconda3/bin/conda shell.bash hook)"
conda activate llm_on_supercomp
cd `dirname $0`

# 使用モデルの設定
export MODEL_NAME_OR_PATH="meta-llama/Llama-3.1-8B-Instruct"

# データパス
export INPUT_DATA_PATH="../data/metainfo_top30.jsonl"
export MODEL_OUTPUT_DIR="./output"

python inference_on_1gpu.py \
--model_name_or_path $MODEL_NAME_OR_PATH \
--input_data_path $INPUT_DATA_PATH \
--model_output_dir $MODEL_OUTPUT_DIR
```

GPU1枚は-shareのみ

モジュールのロード

Anacondaの有効化

4GPUでの推論(Naive Model Parallel)



1node4gpuでの推論

●inference_on_1node4gpu.py

- 下記の関数

```python
def intialize_tokenizer_and_model(model_name_or_path:str):
    try:
        # トークナイザーの初期化
        ## DecoderはPADがないため、EOSで代用
        tokenizer = AutoTokenizer.from_pretrained(model_name_or_path)
        tokenizer.pad_token = tokenizer.eos_token
        tokenizer.pad_token_id = tokenizer.eos_token_id
        tokenizer.padding_side='right'
        # モデルの初期化
        model = AutoModelForCausalLM.from_pretrained(
            model_name_or_path,
            device_map="auto",
            torch_dtype=torch.bfloat16,
            trust_remote_code=True
            )
    except:
        print("Tokenizer & Model Initialize Error")
        return
    # 学習をしない
    model.eval()
    return tokenizer, model
```

device_map=“auto”設定

●inference_on_1gpu.sh

```bash
#!/bin/bash -x
#PJM -L rscgrp=cx-single
#PJM -L node=1
#PJM -L elapse=1:00:00  # REWRITE ME!!!!!
#PJM -j
#PJM -S

module load gcc/11.3.0 cuda/12.4.1 openmpi_cuda/4.0.5 nccl/2.19.3

eval "$(~/miniconda3/bin/conda shell.bash hook)"
conda activate llm_on_supercomp
cd `dirname $0`

# 使用モデルの設定
export MODEL_NAME_OR_PATH="Qwen/Qwen3-32B"

# データパス
export INPUT_DATA_PATH="../data/metainfo_top30.jsonl"
export MODEL_OUTPUT_DIR="./output"

python inference_on_1node4gpu.py \
--model_name_or_path $MODEL_NAME_OR_PATH \
--input_data_path $INPUT_DATA_PATH \
--model_output_dir $MODEL_OUTPUT_DIR
```

1nodeは-singleのみ

device_map

- **4種類**
  - auto: balancedと一緒。今後変更されるらしい
  - balanced: 全てのGPUに均等にモデルを分割
  - balanced_low_0: GPU0を除いたGPUに均等にモデルを分割
    - GPU0を入出力の処理として容量を開けておく
  - sequential: 1つのGPUが容量が最大になってから、次のGPUにモデル分割
    - 割り当てが貪欲法

- **infer_auto_device_mapで挙動を確認**
  - infer_device_map_on_1node4gpu.shを実行
  - OrderedDict[…, ('model.layers.31', 0), ('model.layers.33', 1), …]

まとめ

- **「不老」でのLLMの利用**
  - 不老の概要と軽い実装の解説
  - コードを公開したので確認してほしい

- **スパコンのススメ**
  - （研究用途で）大規模計算環境に触れるチャンス
  - 大規模なローカルLLMを利用可

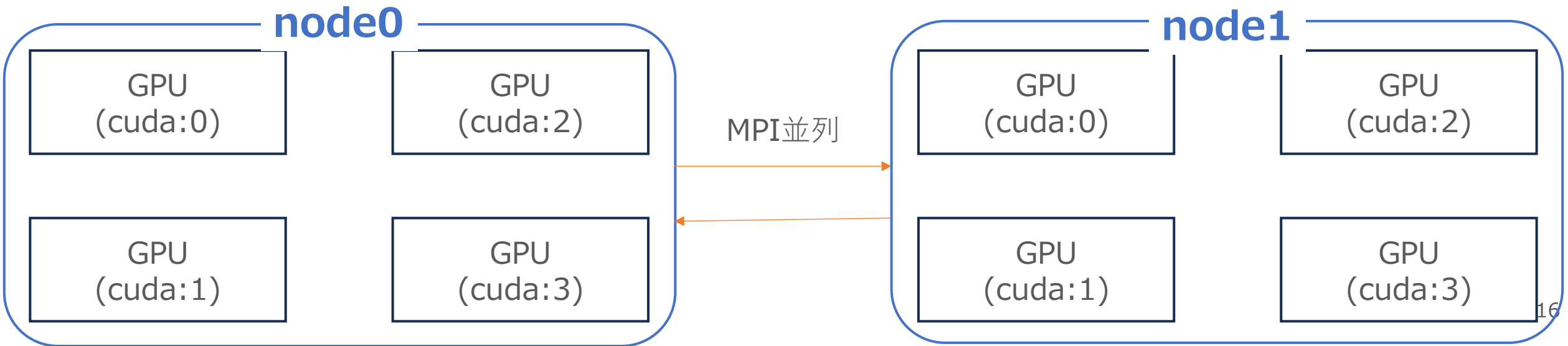
- **蛇足**
  - 資料は上手くいった部分だけを共有している
  - ので、実際はこの10倍くらいはバグっている（特に複数ノード）

今後

- **今回公開したリポジトリは随時更新予定**
  - 直近はなんとかマルチノードを動かしたい

2node8gpuでの推論

- **ノード間を超えるときは、MPI並列が必要**
  - ノード内は自由
    - Accelerateで1node4gpuやってもいい
    - torch.distributedでノード内で4並列を立ち上げてもいい



2node8gpuでの推論

●shファイルから

```
#!/bin/bash -x
#PJM -L rscgrp=cxgfs-small
#PJM -L node=2
#PJM --mpi proc=8
#PJM -L elapse=1:00:00  # REWRITE ME!!!!!
#PJM -j
#PJM -S

module load gcc/11.3.0 cuda/12.4.1 openmpi_cuda/4.0.5 nccl/2.19.3

# マルチノード実行用にIPアドレスを記録
export MASTER_ADDR=$(head -1 ${PJM_O_NODEINF})
# ポートもJOB_IDを参照して動的に割り当て
export MASTER_PORT=$(( 50000 + (${PJM_JOBID:-0} % 10000) ))

eval "$(~/miniconda3/bin/conda shell.bash hook)"
conda activate llm_on_supercomp
cd `dirname $0`

# 使用モデルの設定
export MODEL_NAME_OR_PATH="meta-llama/Llama-3.1-8B-Instruct"
# export MODEL_NAME_OR_PATH="meta-llama/Llama-3.1-70B-Instruct"

# データパス
export INPUT_DATA_PATH="../data/metainfo_top30.jsonl"
export MODEL_OUTPUT_DIR="./output"
```

```
# 合計GPU数
export NUM_PROCESSES=$(( ${PJM_NODE} * 4 ))

# 通信手段の明示
export OMPI_MCA_plm_rsh_agent="/bin/pjrsh"
export OMPI_MCA_btl_tcp_if_include=ib0
export NCCL_SOCKET_IFNAME=ib0
export GLOO_SOCKET_IFNAME=ib0

# デバッグ
# PyTorch 分散の詳細ログ
export TORCH_DISTRIBUTED_DEBUG=DETAIL
# NCCL のログレベルを INFO に（通信トポロジーや障害を追跡しやすい）
export NCCL_DEBUG=INFO

mpirun -np $PJM_MPI_PROC \
-machinefile $PJM_O_NODEINF \
-x MASTER_ADDR=$MASTER_ADDR \
-x MASTER_PORT=$MASTER_PORT \
-mca pml ob1 -mca btl ^openib \
python3 inference_on_2node8gpu.py \
--backend nccl \
--model_name_or_path $MODEL_NAME_OR_PATH \
--input_data_path $INPUT_DATA_PATH \
--model_output_dir $MODEL_OUTPUT_DIR
```

torchでの実装

- torch.distributed.pipelineを使用
  - プロセスの同期

```
# OpenMPIでdevice情報を探る
LOCAL_RANK = int(os.environ['OMPI_COMM_WORLD_LOCAL_RANK'])
WORLD_SIZE = int(os.environ['OMPI_COMM_WORLD_SIZE'])
WORLD_RANK = int(os.environ['OMPI_COMM_WORLD_RANK'])

# 各プロセスの同期
def init_processes(backend):
    dist.init_process_group(backend, rank=WORLD_RANK, world_size=WORLD_SIZE)
```

torchでの実装

- torch.distributed.pipelineを使用
  - モデル分割

```
# Pipeline split spec
decoders_per_rank = (model.config.n_layer + WORLD_SIZE - 1) // WORLD_SIZE
print(f"decoders_per_rank = {decoders_per_rank}")
split_spec = {
    f'transformer.h.{i * decoders_per_rank}': SplitPoint.BEGINNING
    for i in range(1, args.world_size)
}

# Create pipeline representation
pipe = pipeline(
    model,
    mb_args=(),
    mb_kwargs=mb_inputs,
    split_spec=split_spec,
)

device = torch.device(f"cuda:{LOCAL_RANK % 4}")
print("device:", device)

# Create schedule runtime
stage = pipe.build_stage(
    WORLD_RANK,
    device=device,
)
```

torchでの実装

- torch.distributed.pipelineを使用
  - 最終的な推論

```
# Run
if WORLD_RANK == 0:
    schedule.step(**inputs)
else:
    out = schedule.step()
```