ch7. Implementing Model Parallel Training and Serving Workflows

Model Parallelism and Megaton

2023.07.19 정서형

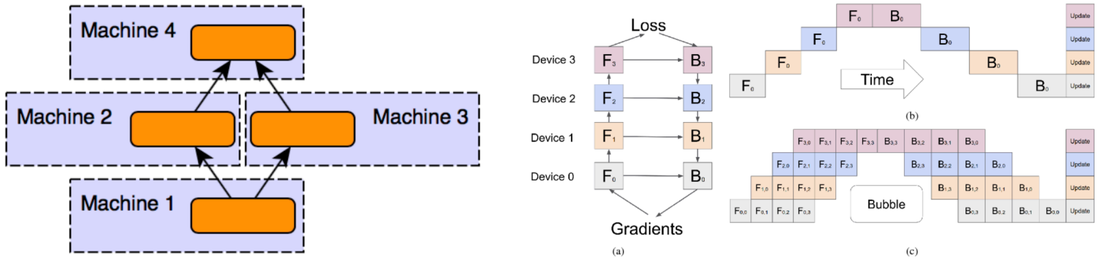
References

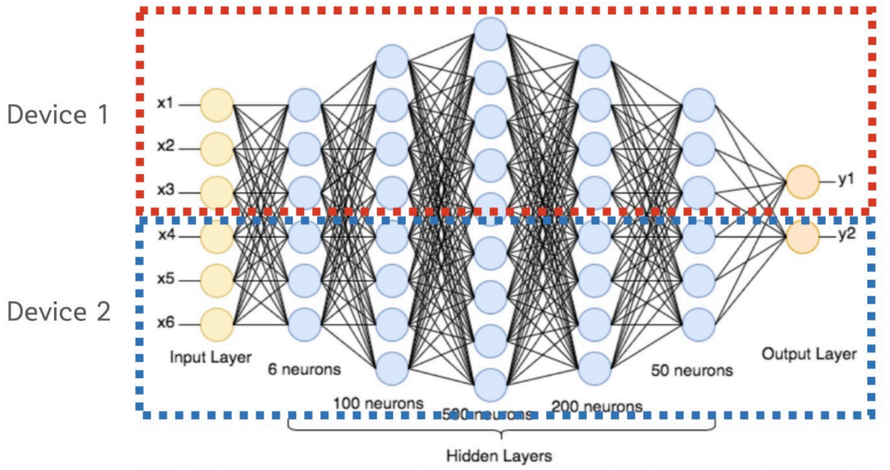
- Nvidia Megatron blog posts: https://nv-adlr.github.io/MegatronLM, https://developer.nvidia.com/blog/scaling-language-model-training-to-a-trillion-parameters-using-megatron/
- Megatron paper: https://arxiv.org/pdf/1909.08053.pdf
- Large-scale LM에 대한 얕고 넓은 지식들: https://www.youtube.com/watch?v=w4a-ARCEiqU&t=3757s
- [챗GPT 러닝데이 | 챗GPT말고 LLM] 딥러닝 병렬처리 및 Polyglot 언어모델: https://www.youtube.com/watch?v=a0TB-_WFjNk&t=5123s
- 어쩌다 보니 책 ch 9의 내용과 완벽히 겹침..

Model Parallelism

Inter-layer parallelism and Intra-layer parallelism

- Inter-layer parallelism
  - Layer-wise로 모델을 병렬화 하는 방식
  - Sequential 하게 진행되기 때문에 gpu idle time이 발생
  - Gpipe, Pipedream (pipeline parallelism)
- Intra-layer parallelism
  - Column 혹은 row 방향으로 모델을 분할하여 병렬화 하는 방식
  - Mesh-tensorflow, **Megatron-LM**





Model Parallelism

Megatron-LM

- Nvidia에서 개발한 large scale parallel transformer LM
- Tensor model parallelism으로 intra-layer parallelism을 구현
- 텐서 내적의 성질을 이용해서 병렬화 전/후의 연산 결과를 동일하게 만드는 방법 사용
  - Row parallel linear
  - Column parallel linear

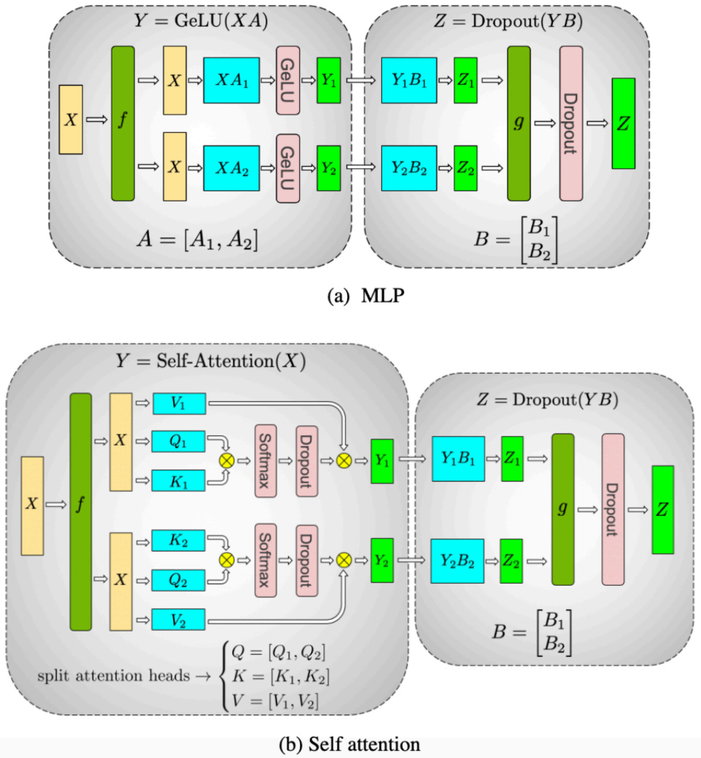
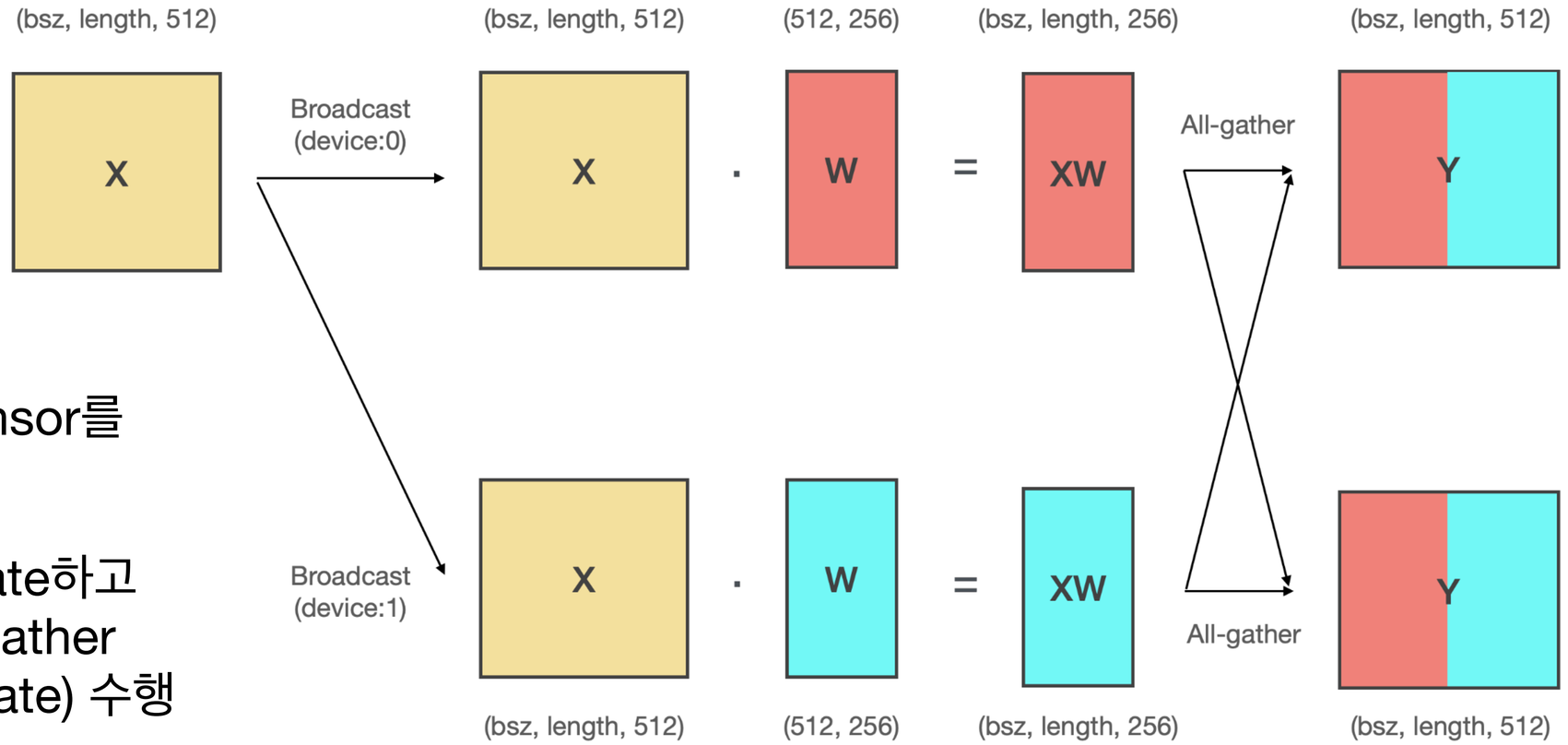


Figure 2: (a): MLP and (b): self attention blocks of transformer. ***f*** and ***g*** are conjugate, ***f*** is an **identity** operator in the forward pass and **all-reduce** in the backward pass while ***g*** is an **all-reduce** in forward and ***identity*** in backward.

Model Parallelism

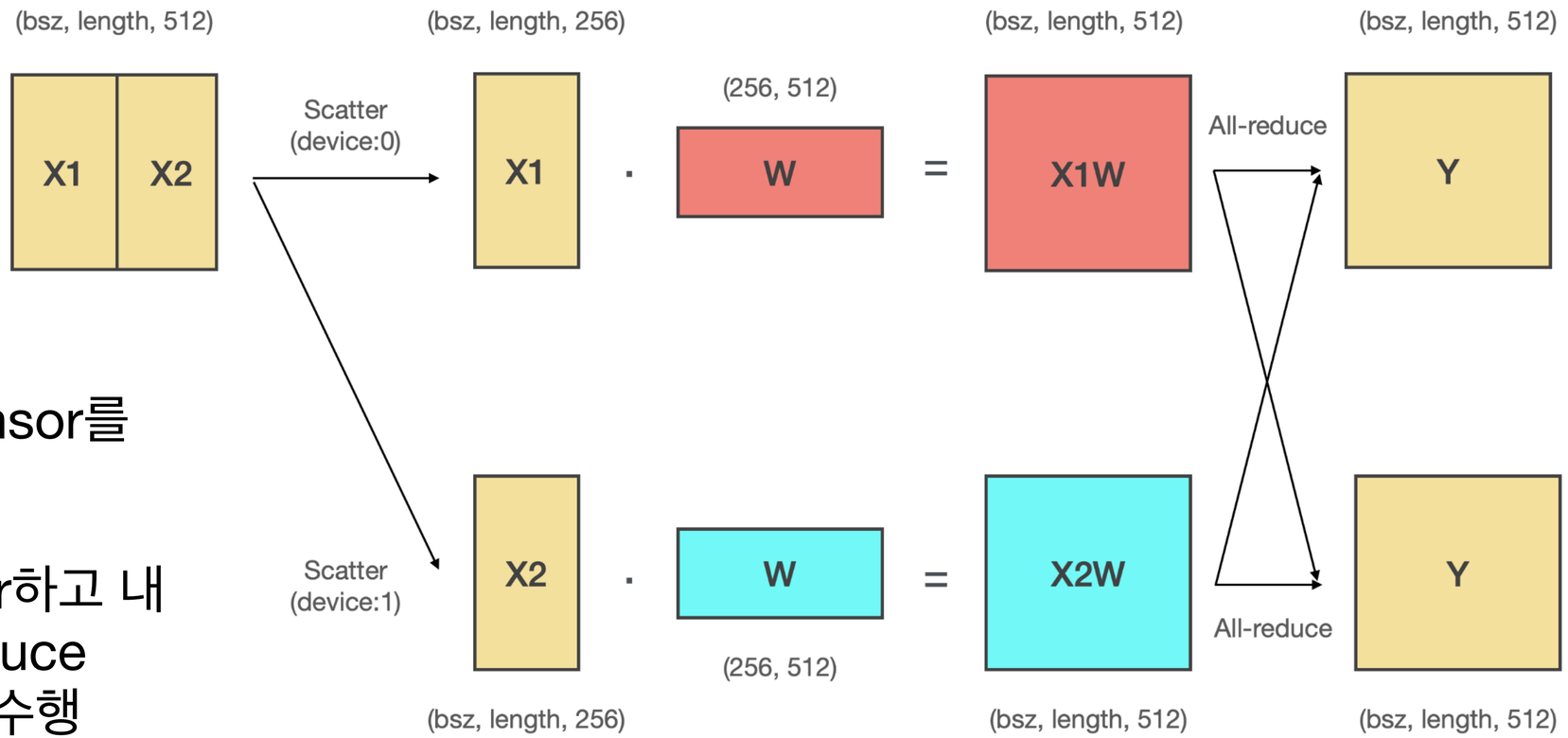
Tensor model parallelism - column parallel



- Weight tensor를 수직분할
- X를 replicate하고 내적 후에 gather (concatenate) 수행

Model Parallelism

Tensor model parallelism - row parallel

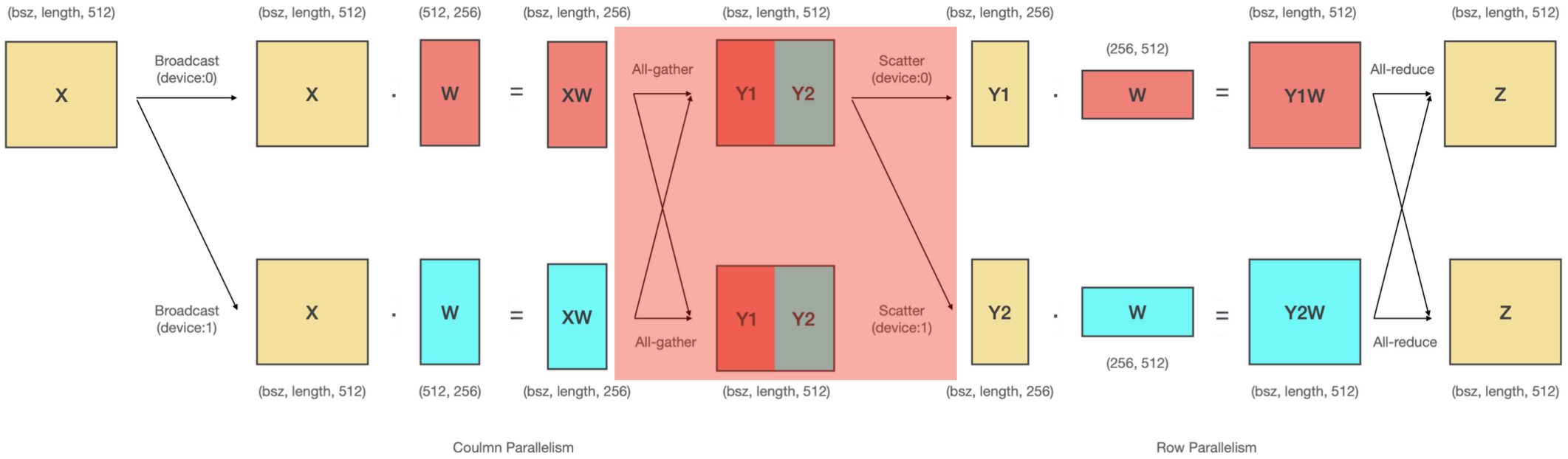


- Weight tensor를 수평분할
- X를 scatter하고 내적 후에 reduce (addition) 수행

Model Parallelism

Tensor model parallelism - mlp layer

- Column-row 순서로 linear layer를 연결하게 되면 all-gather와 scatter 연산 생략 가능



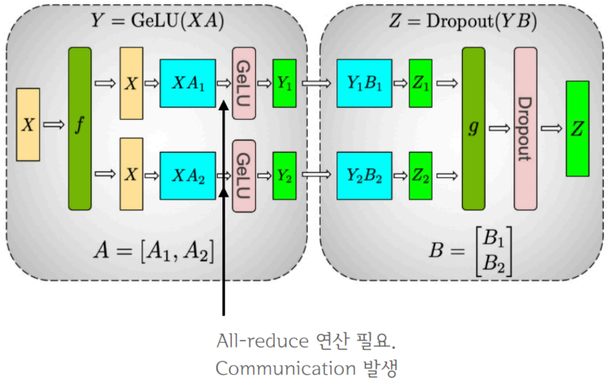
Model Parallelism

Tensor model parallelism - mlp layer (h -> 4h)

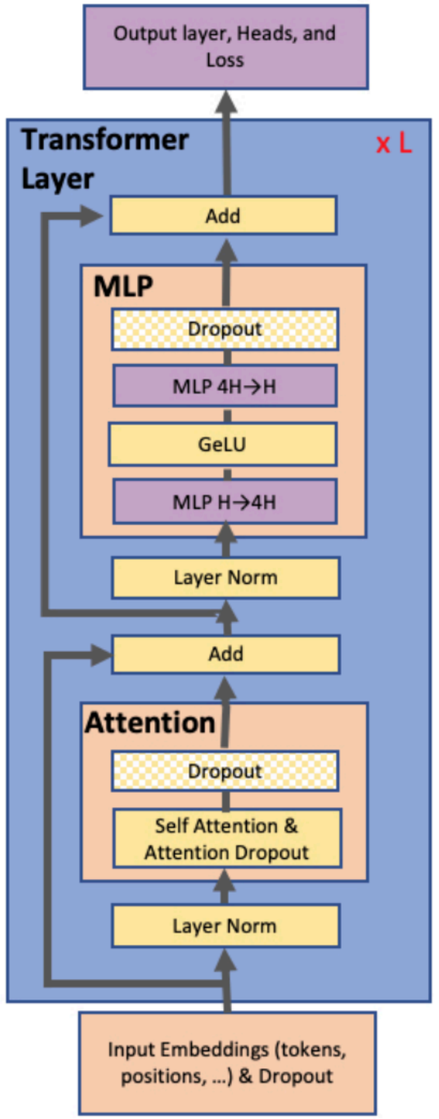
- row-wise로 병렬화 할 경우, GeLU에서 linearity가 보장되지 않음

$$X = [X_1, X_2],\ A = \begin{bmatrix} A_1 \\ A_2 \end{bmatrix}. \tag{2}$$

This partitioning will result in $Y = \text{GeLU}(X_1A_1 + X_2A_2)$. Since GeLU is a nonlinear function, $\text{GeLU}(X_1A_1 + X_2A_2) \neq \text{GeLU}(X_1A_1) + \text{GeLU}(X_2A_2)$ and this approach will require a synchronization point before the GeLU function.



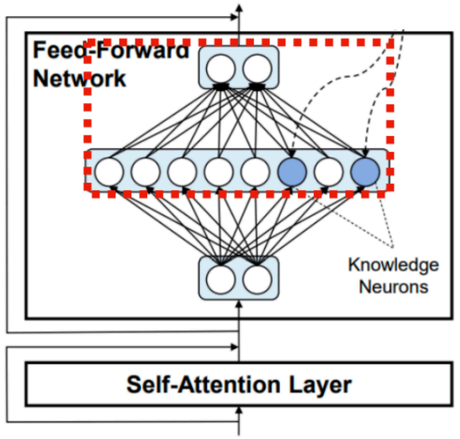
- colum-wise로 병렬화 진행 (GeLU 병렬화, 동기화 결과가 동일)
  - 병렬화: $GeLU(XA_1)\ cat\ GeLU(XA_2)$
  - 동기화: $GeLU(XA_1\ cat\ XA_2)$

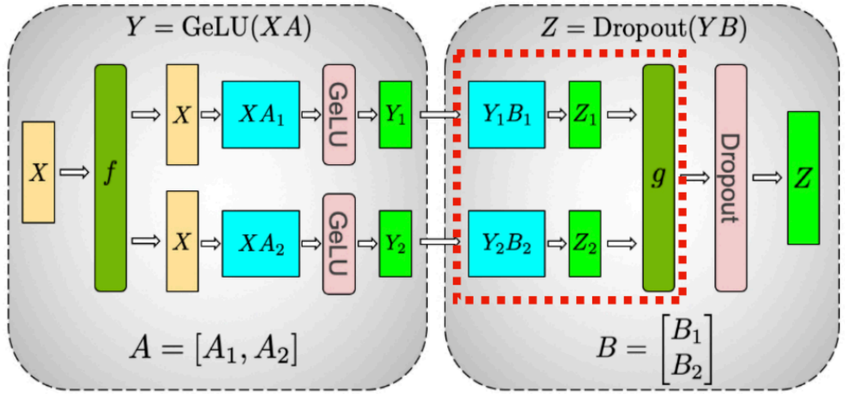


Model Parallelism

Tensor model parallelism - mlp layer (4h -> h)

- row-wise로 병렬화
- 이전 layer output을 Concat하는 비용을 줄일 수 있음



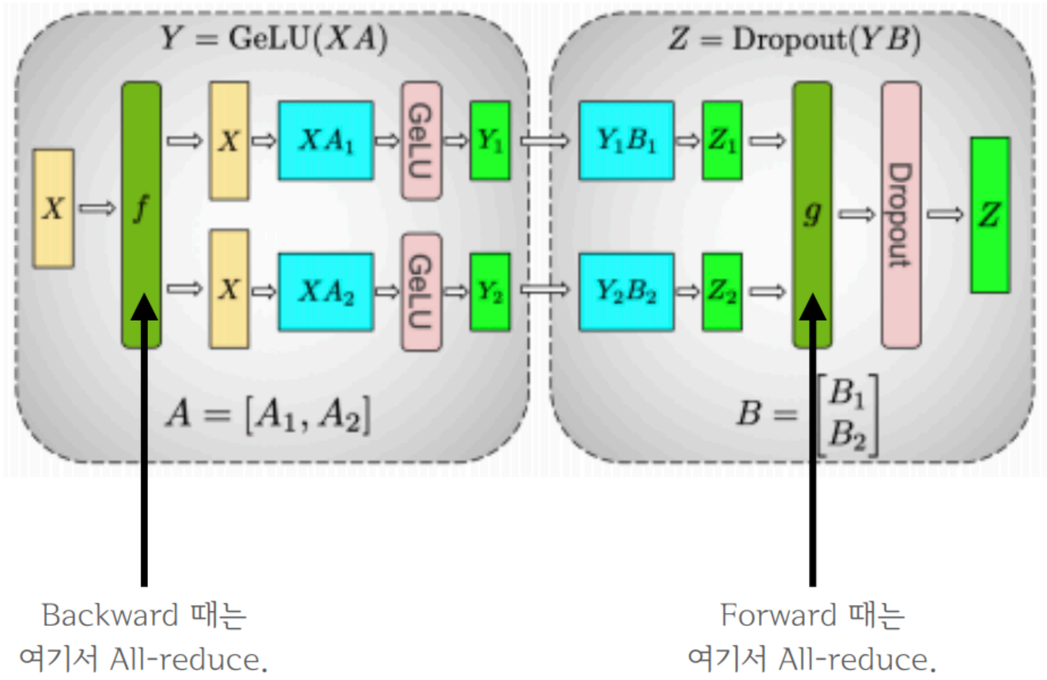



```python
class ParallelMLP(MegatronModule):
    """MLP.

    MLP will take the input with h hidden state, project it to 4*h
    hidden dimension, perform nonlinear transformation, and project the
    state back into h hidden dimension.
    """

    def __init__(self, init_method, output_layer_init_method):
        super(ParallelMLP, self).__init__()
        args = get_args()


        # Project to 4h.
        self.dense_h_to_4h = tensor_parallel.ColumnParallelLinear(
            args.hidden_size,
            args.ffn_hidden_size,
            gather_output=False,
            init_method=init_method,
            skip_bias_add=True,
            async_tensor_model_parallel_allreduce=args.async_tensor_model_parallel_allreduce,
            **_args_to_kwargs())

        self.bias_gelu_fusion = args.bias_gelu_fusion
        self.activation_func = F.gelu
        if args.openai_gelu:
            self.activation_func = openai_gelu
        elif args.onnx_safe:
            self.activation_func = erf_gelu

        # Project back to h.
        self.dense_4h_to_h = tensor_parallel.RowParallelLinear(
            args.ffn_hidden_size,
            args.hidden_size,
            input_is_parallel=True,
            init_method=output_layer_init_method,
            skip_bias_add=True,
            **_args_to_kwargs())
```

Model Parallelism

Tensor model parallelism - mlp layer (h -> 4h -> h)

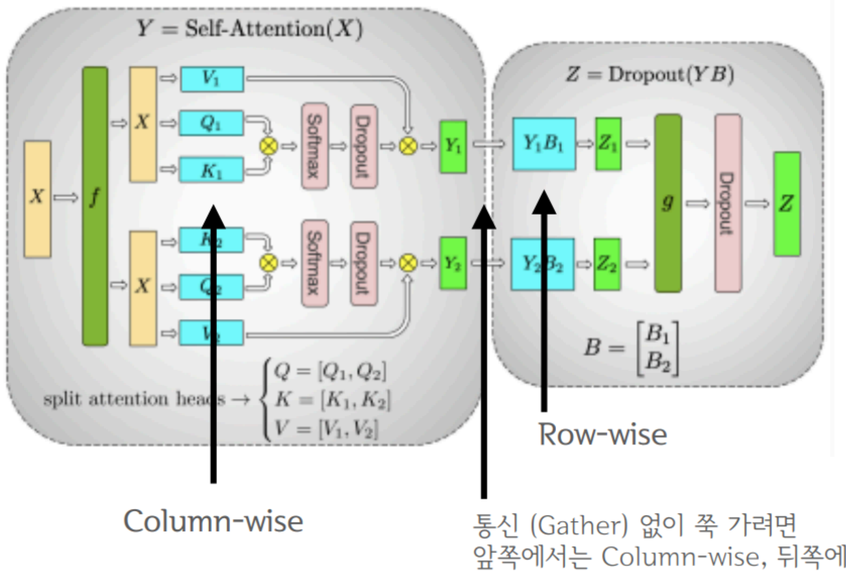
- 2번의 all reduce 연산으로 구현 가능 (forward에서 1번, backward에서 1번)



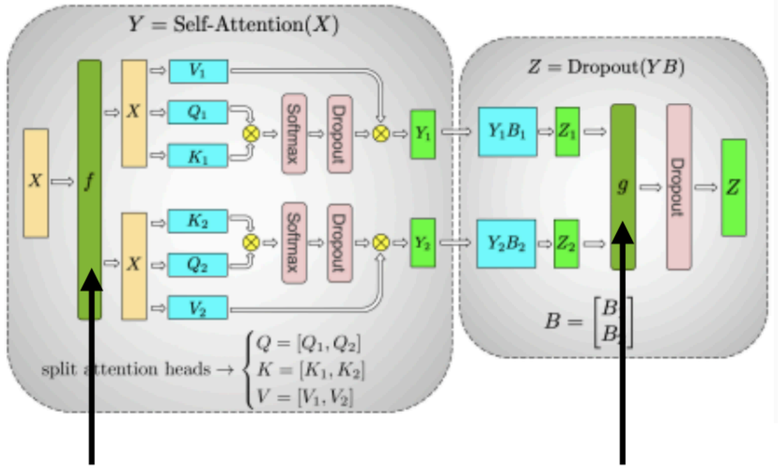
Model Parallelism

Tensor model parallelism - attention layer

- Q, K, V 연산은 column parallel로, output project은 row parallel로 병렬화
- Column-row 순서로 병렬화하여 all-gather, scatter 연산 생략 가능



통신 (Gather) 없이 쭉 가려면
앞쪽에서는 Column-wise, 뒤쪽에서는 Row-wise를 써줘야 함.



Backward 때는
여기서 All-reduce.

Forward 때는
여기서 All-reduce.

Model Parallelism

Tensor model parallelism - attention layer

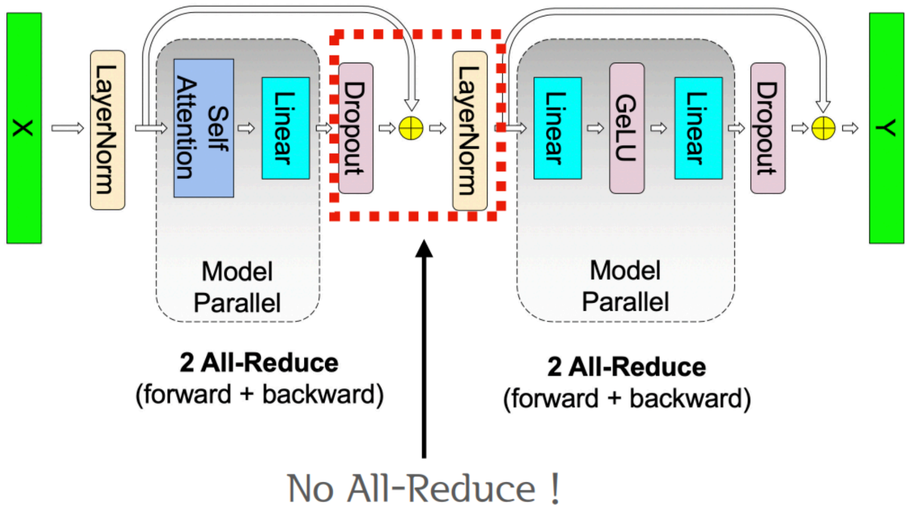
```
self.query_key_value = tensor_parallel.ColumnParallelLinear(
    args.hidden_size,
    3 * projection_size,
    gather_output=False,
    init_method=init_method,
    async_tensor_model_parallel_allreduce=args.async_tensor_model_parallel_allreduce,
    **_args_to_kwargs())
```

```
# Output.
self.dense = tensor_parallel.RowParallelLinear(
    projection_size,
    args.hidden_size,
    input_is_parallel=True,
    init_method=output_layer_init_method,
    skip_bias_add=True,
    **_args_to_kwargs())
```

Model Parallelism

Tensor model parallelism - layer norm, dropout

- Layer norm과 dropout은 병렬화 하지 않음

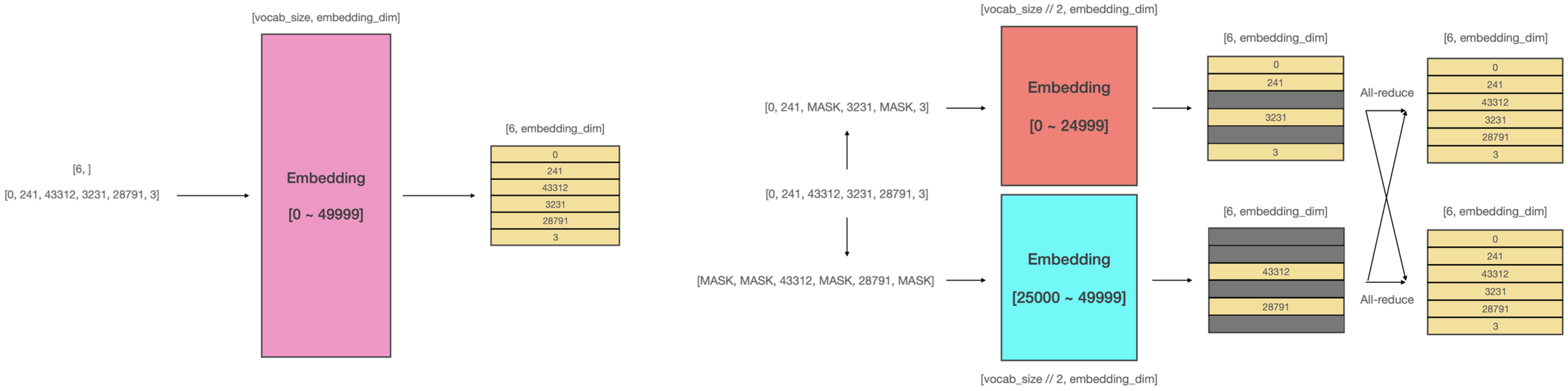


we maintain duplicate copies of layer normalization parameters on each GPU, and take the output of the model parallel region and run dropout and residual connection on these tensors before feeding them as input to the next model parallel regions. To optimize the model we allow each model parallel worker to optimize its own set of parameters. Since all values are either local to or duplicated on a GPU, there is no need for communicating updated parameter values in this formulation.

Model Parallelism

Tensor model parallelism - embedding layer

- Vocab size dimension을 기준으로 병렬화 진행, n개의 device에 각각 [bsz, seq_len, vocab_size/n]의 logit이 존재
- Partitioning 된 vocab 범위에서 벗어나면 masking 처리
- logit을 all-gather 할 필요 없이 loss까지 계산 후 (cross entropy layer와 fuse) scalar 값들을 all-reduce하여 backdrop



Model Parallelism

Tensor model parallelism - embedding layer

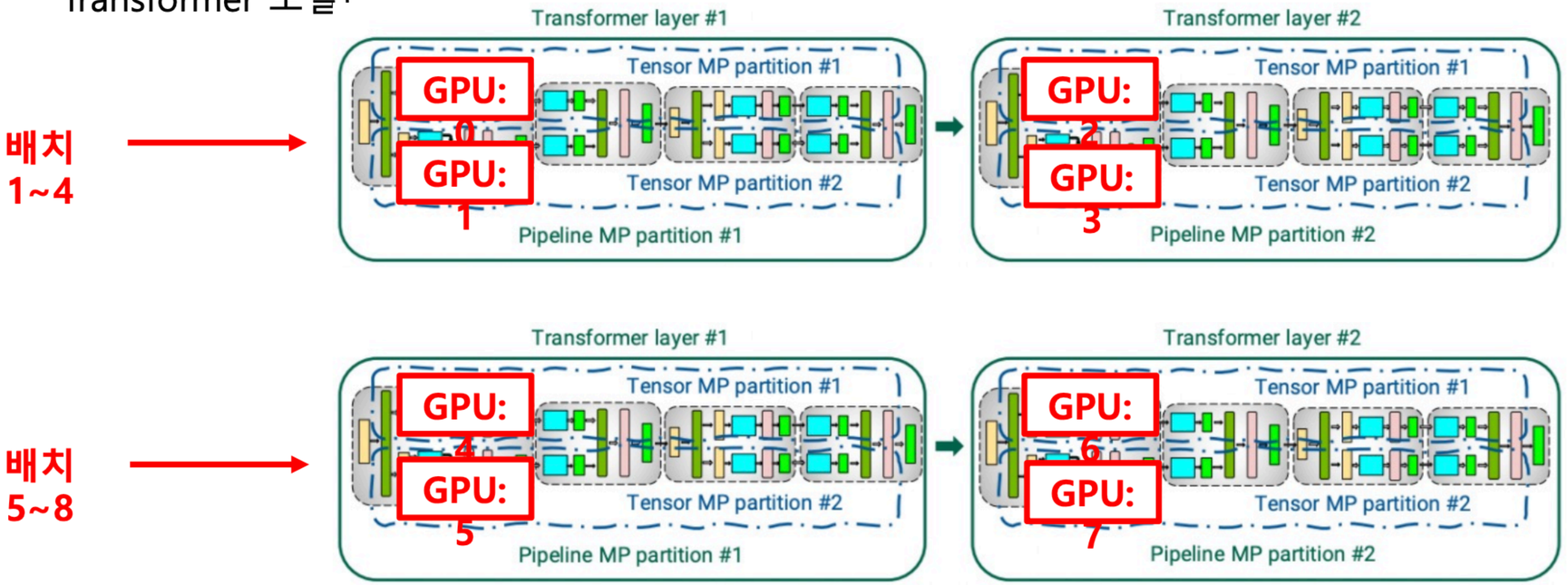
```
# Get the partition's vocab indecies
get_vocab_range = VocabUtility.vocab_range_from_per_partition_vocab_size
partition_vocab_size = vocab_parallel_logits.size()[-1]
rank = get_tensor_model_parallel_rank()
world_size = get_tensor_model_parallel_world_size()
vocab_start_index, vocab_end_index = get_vocab_range(
    partition_vocab_size, rank, world_size)

# Create a mask of valid vocab ids (1 means it needs to be masked).
target_mask = (target < vocab_start_index) | (target >= vocab_end_index)
masked_target = target.clone() - vocab_start_index
masked_target[target_mask] = 0
```

```
# Get predicted-logits = logits[target].
# For Simplicity, we convert logits to a 2-D tensor with size
# [*, partition-vocab-size] and target to a 1-D tensor of size [*].
logits_2d = vocab_parallel_logits.view(-1, partition_vocab_size)
masked_target_1d = masked_target.view(-1)
arange_1d = torch.arange(start=0, end=logits_2d.size()[0],
                         device=logits_2d.device)
predicted_logits_1d = logits_2d[arange_1d, masked_target_1d]
predicted_logits_1d = predicted_logits_1d.clone().contiguous()
predicted_logits = predicted_logits_1d.view_as(target)
predicted_logits[target_mask] = 0.0
# All reduce is needed to get the chunks from other GPUs.
torch.distributed.all_reduce(predicted_logits,
                             op=torch.distributed.ReduceOp.SUM,
                             group=get_tensor_model_parallel_group())

# Sum of exponential of logits along vocab dimension across all GPUs.
exp_logits = vocab_parallel_logits
torch.exp(vocab_parallel_logits, out=exp_logits)
sum_exp_logits = exp_logits.sum(dim=-1)
torch.distributed.all_reduce(sum_exp_logits,
                             op=torch.distributed.ReduceOp.SUM,
                             group=get_tensor_model_parallel_group())

# Loss = log(sum(exp(logits))) - predicted-logit.
loss = torch.log(sum_exp_logits) - predicted_logits
```

Model Parallelism

Tensor model parallelism

- 3D Parallelism
  - Data + Pipeline + Tensor Parallelism
  - 아래 예시는 8장의 GPU로 DP=2, PP=2, TP=2로 분할한 경우. (2개의 레이어를 갖는 Transformer 모델)



Model Parallelism

Tensor vs pipeline parallelism

Pipeline model parallelism, on the other hand, uses much cheaper point-to-point communication that can be performed across nodes without bottlenecking the entire computation. However, pipeline parallelism can spend significant time in the pipeline bubble. The total number of pipeline stages should thus be limited so that the number of microbatches in the pipeline is a reasonable multiple of the number of pipeline stages. Consequently, we saw peak performance when the tensor-parallel size was equal to the number of GPUs in a single node (8 on Selene with DGX A100 nodes).

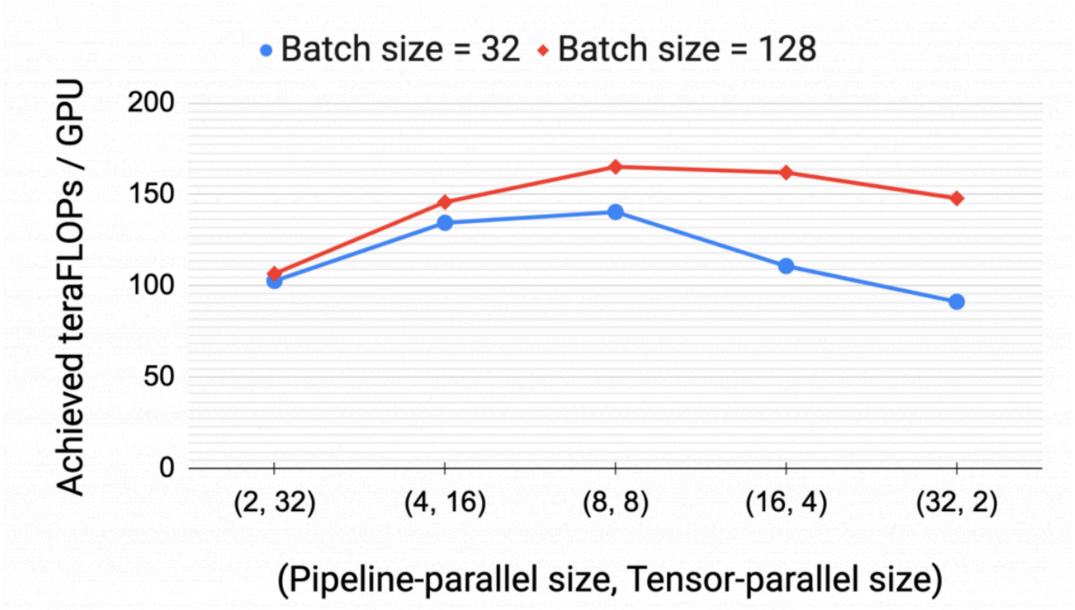


Figure 9. Throughput per GPU of various parallel configurations that combine pipeline and tensor model parallelism using a GPT model with 162.2 billion parameters, two different batch sizes, and 64 A100 GPUs.