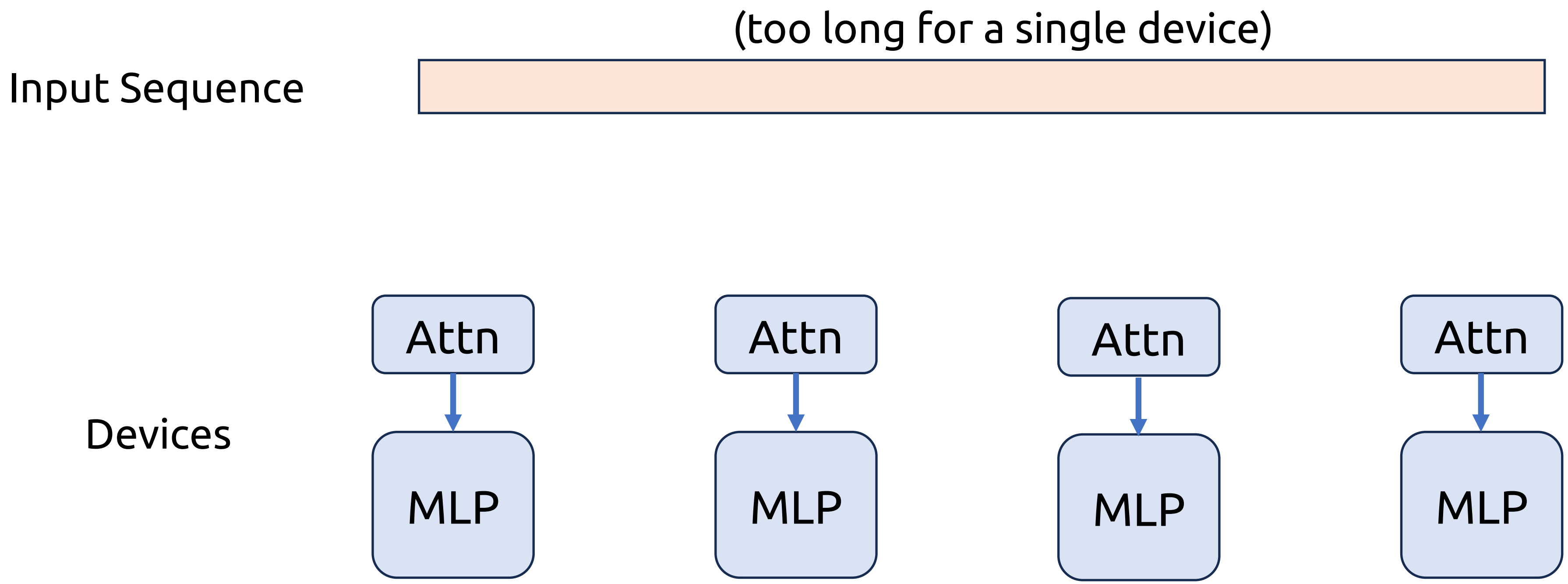
DCP: Addressing Input Dynamism In Long-Context Training via Dynamic Context Parallelism

Chenyu Jiang, Zhenkun Cai, Ye Tian, Zhen Jia, Yida Wang, Chuan Wu

The University of Hong Kong, Amazon Web Services

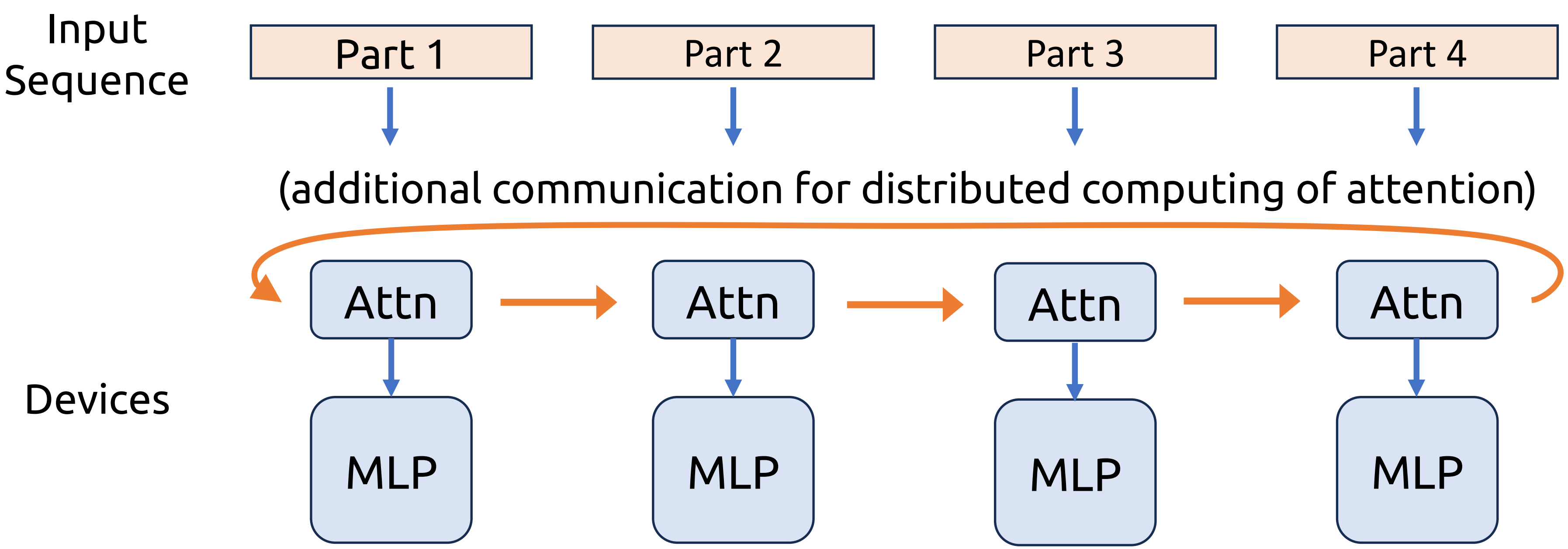
Background

Context Parallelism



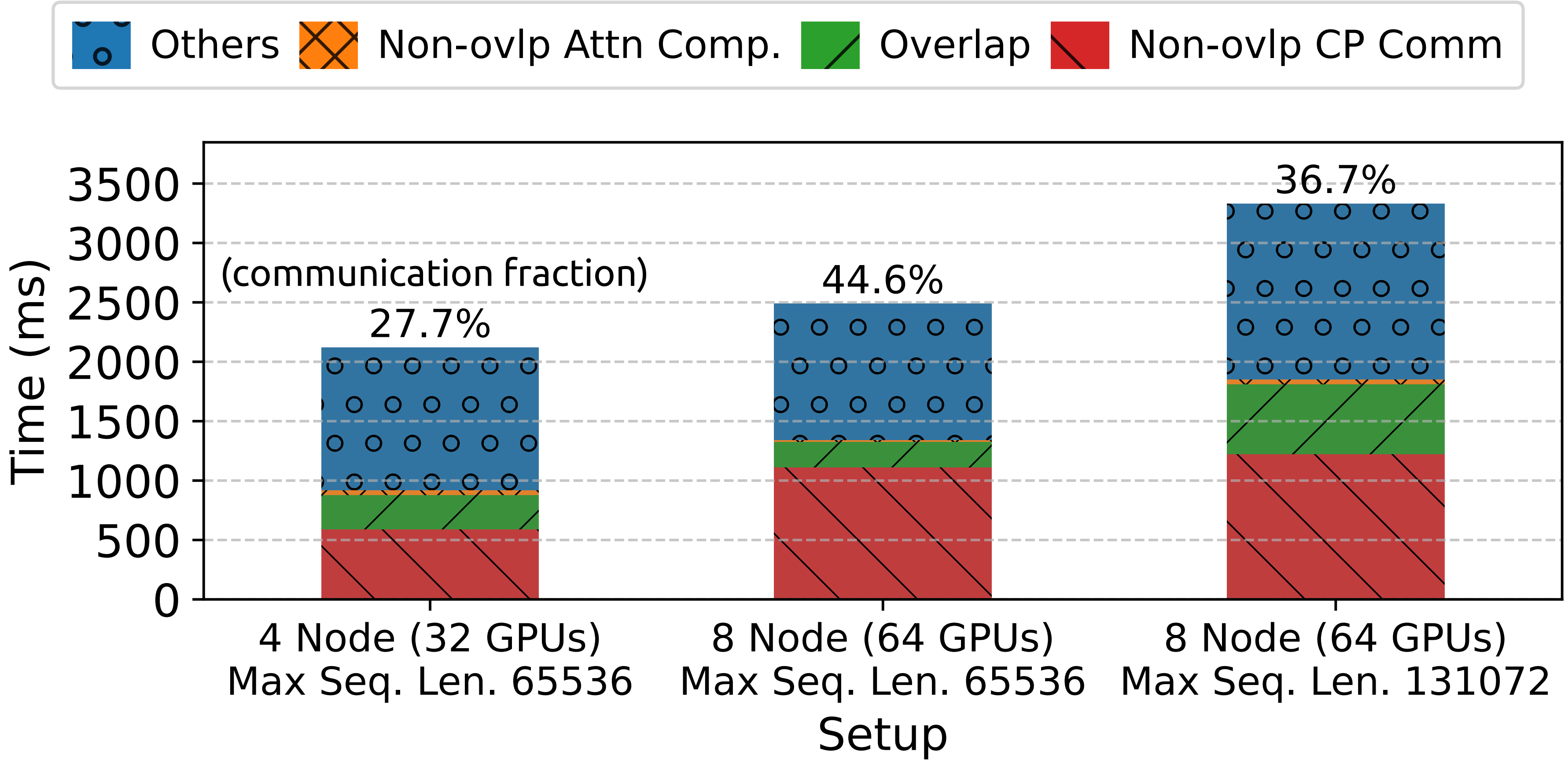
Background

Context Parallelism



Background

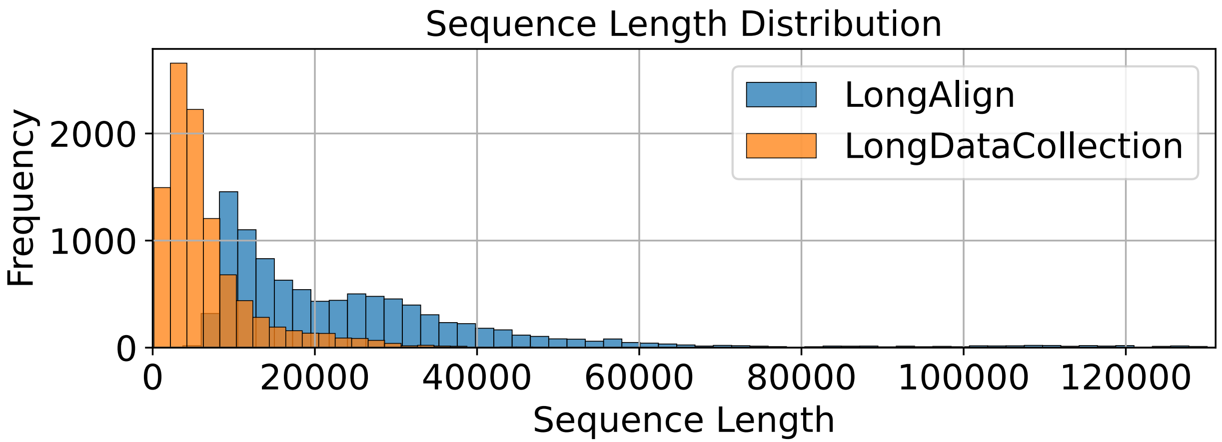
Context Parallelism



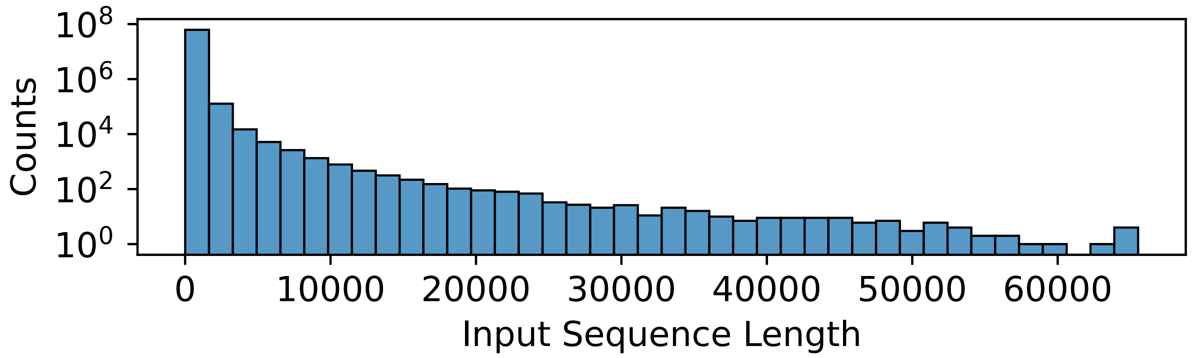
Setup: 8B model, Megatron-LM, 400Gbps interconnect between nodes, 8/16-way context parallelism cross nodes

Motivation

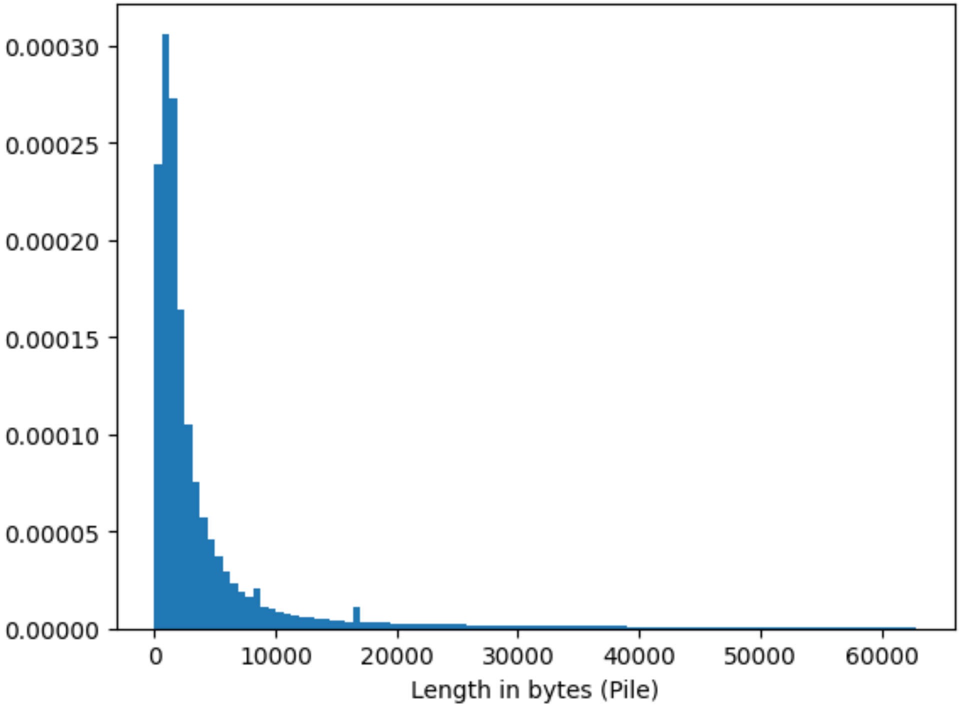
Input Dynamism *in sequence length*



Long Context Datasets



FLANv2

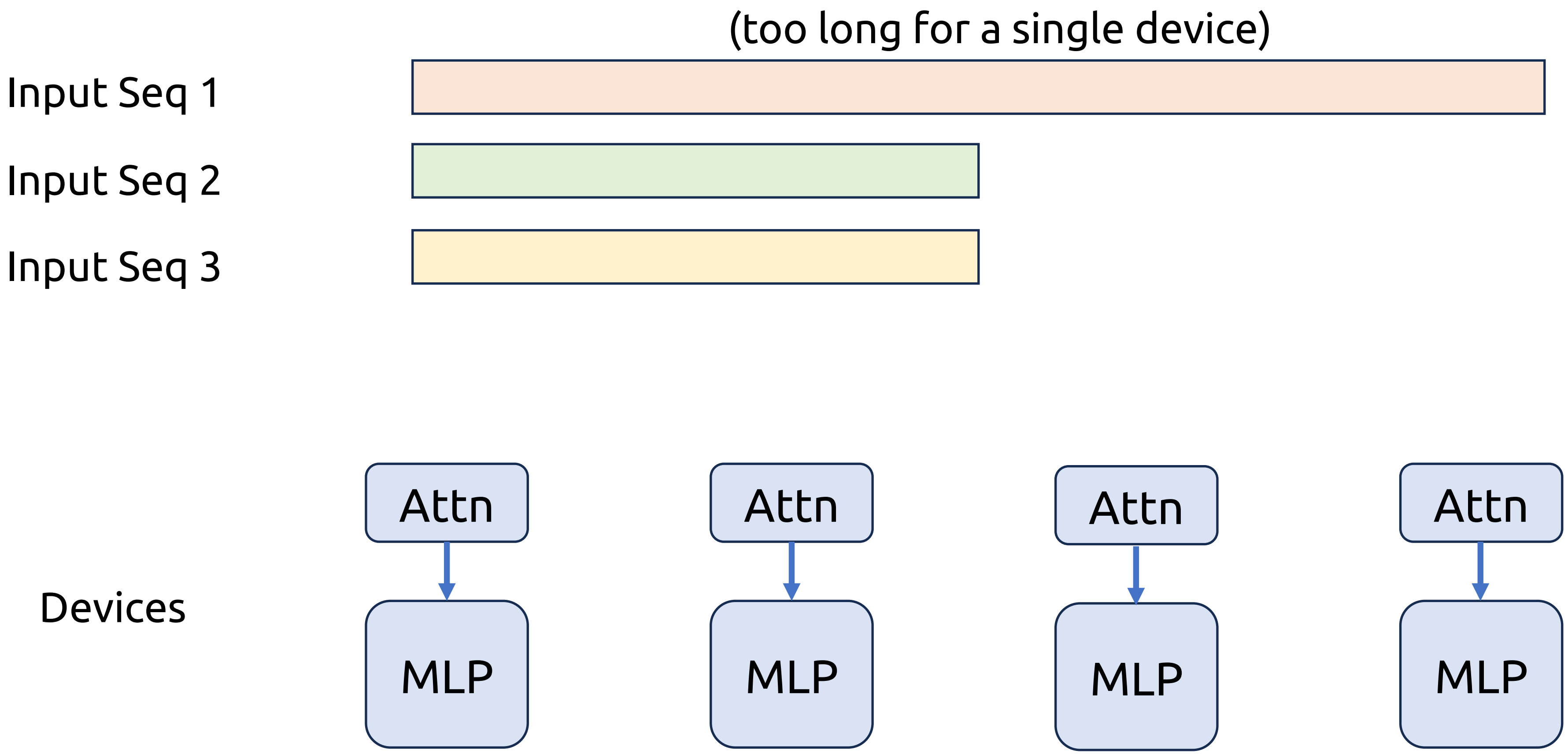


The Pile

The Pile figure from: https://arxiv.org/abs/2101.00027
FLANv2 figure from https://arxiv.org/abs/2311.10418

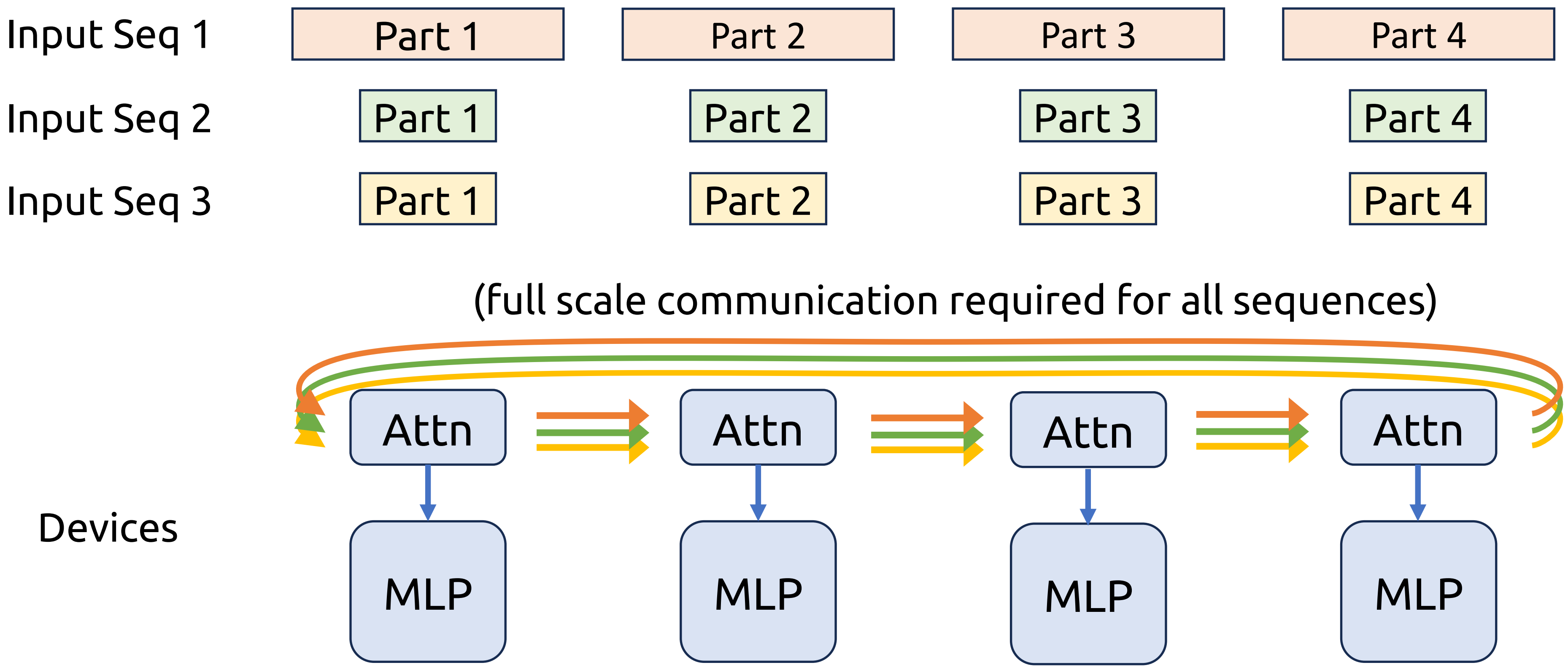
Motivation

Input Dynamism *in sequence length*



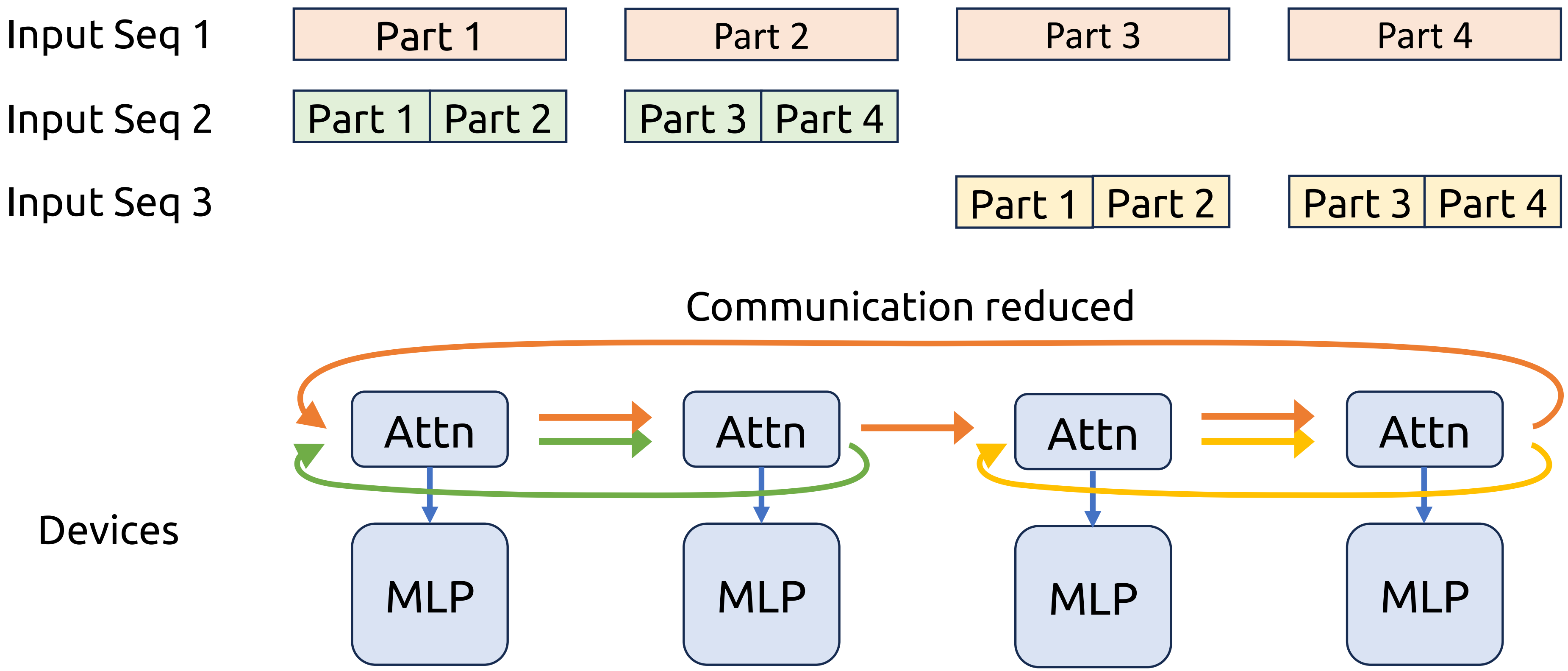
Motivation

Input Dynamism *in sequence length*



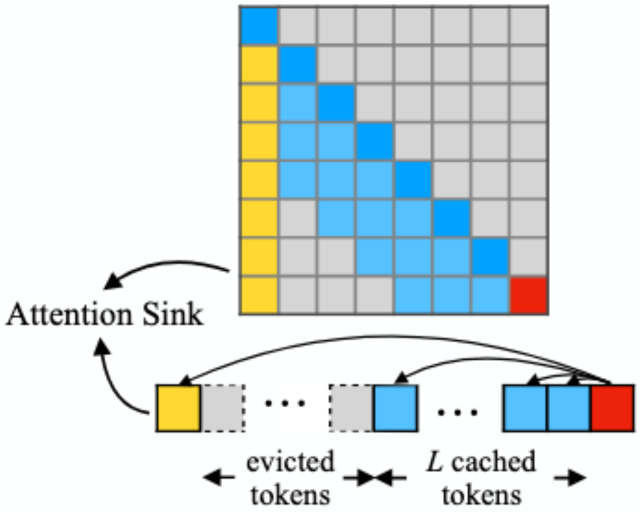
Motivation

Input Dynamism *in sequence length*



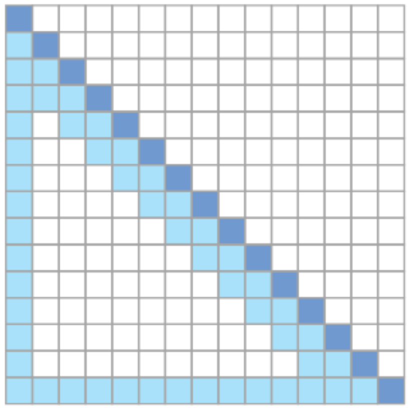
Motivation

Input Dynamism *in attention masks*



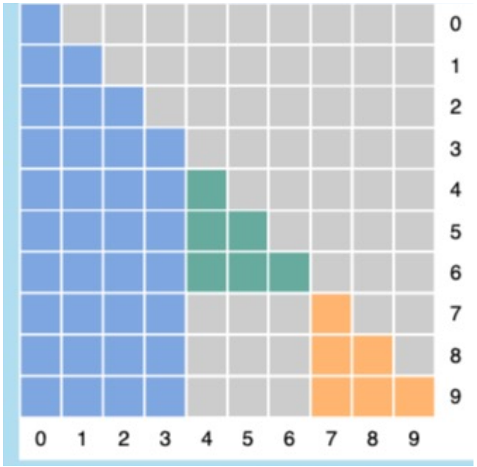
StreamingLLM

Xiao, et al., 2024



Causal Blockwise Mask

Bertsch, et al., 2025



Shared Question Mask

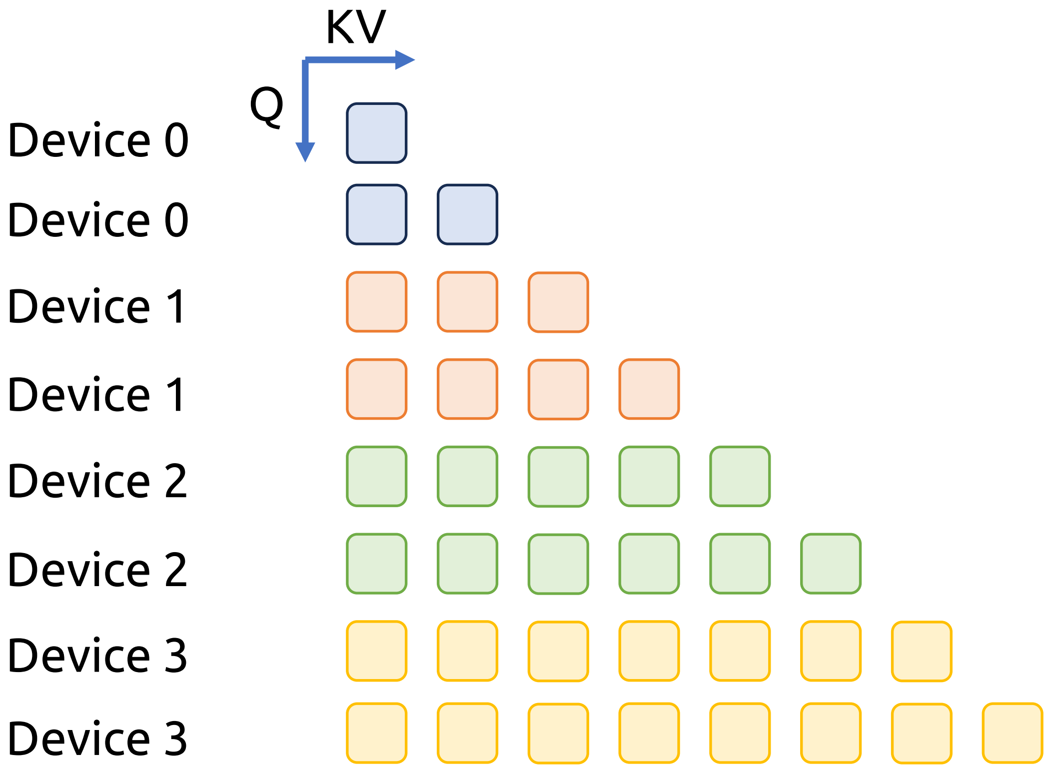
Wang, et al., 2025

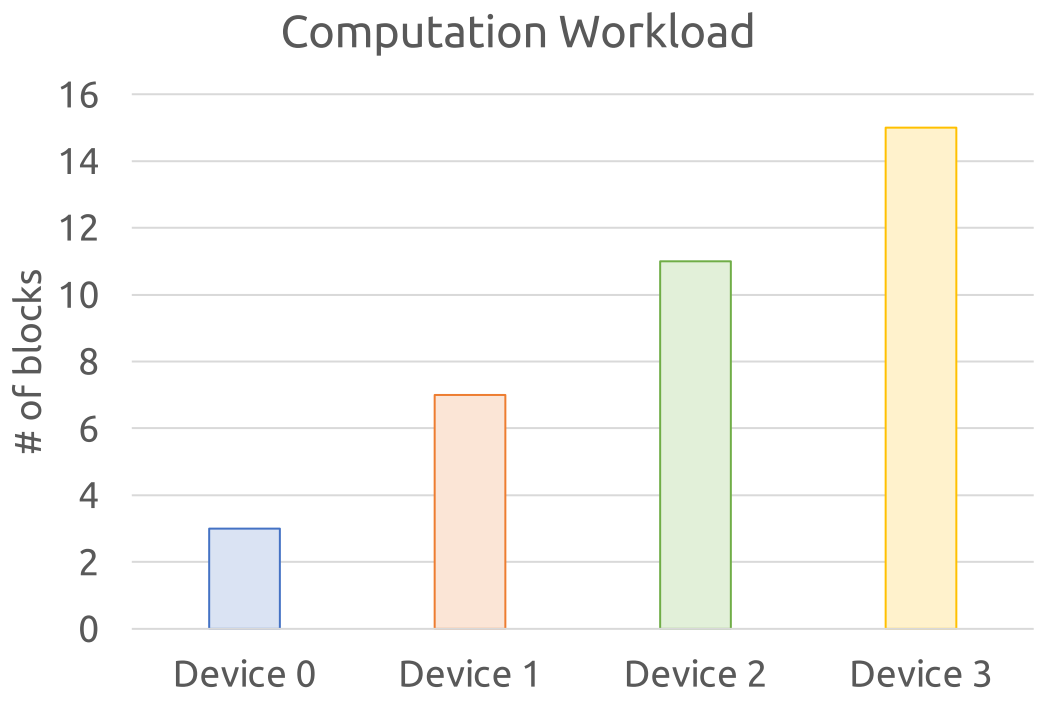
Diverse attention mask patterns on input sequences.

Xiao, et al., 2024: https://arxiv.org/pdf/2309.17453, Bertsch, et al., 2025: https://arxiv.org/pdf/2405.00200, Wang, et al.: https://arxiv.org/pdf/2410.01359

Motivation

Input Dynamism *in attention masks*

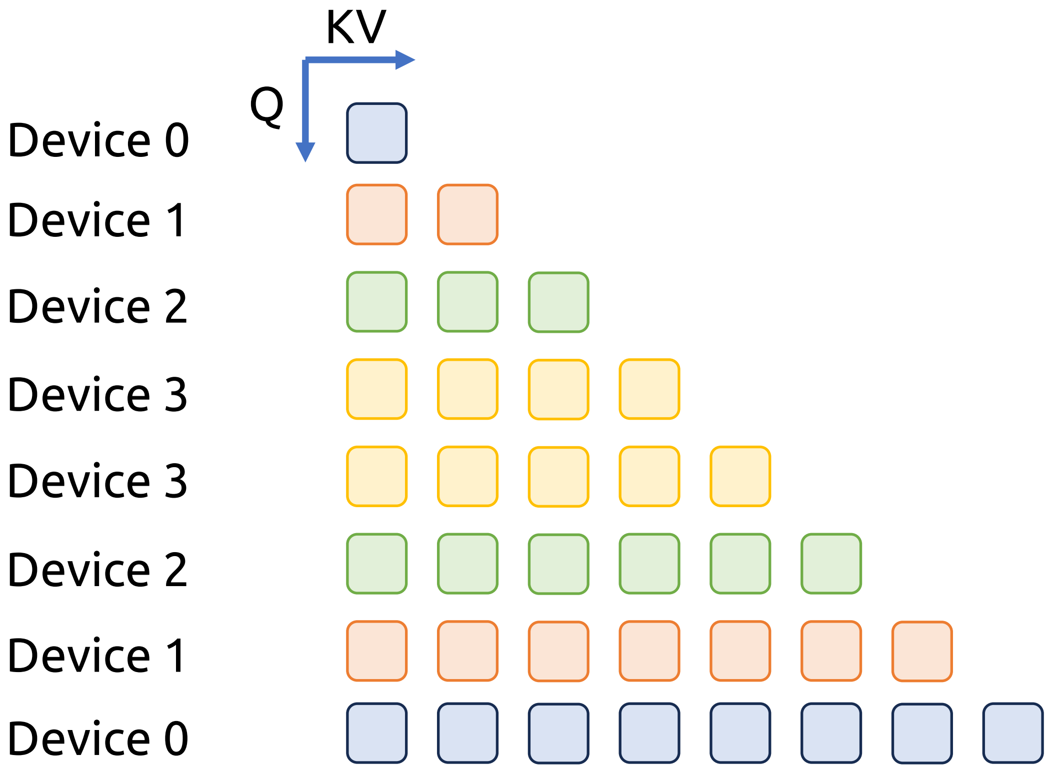


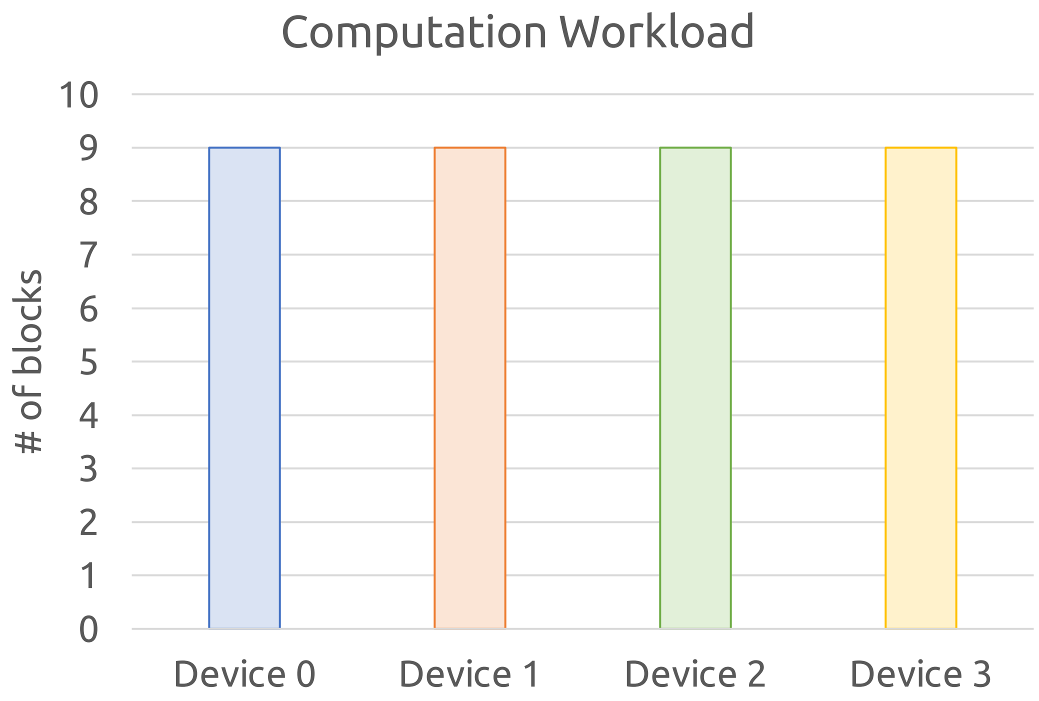


Special Data Placement for Causal Mask

Motivation

Input Dynamism *in attention masks*



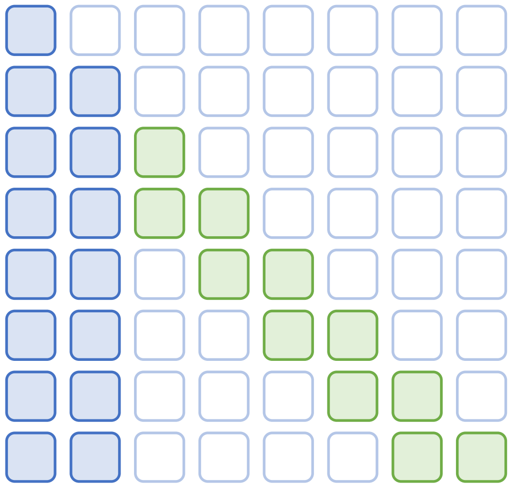


Special Data Placement for Causal Mask

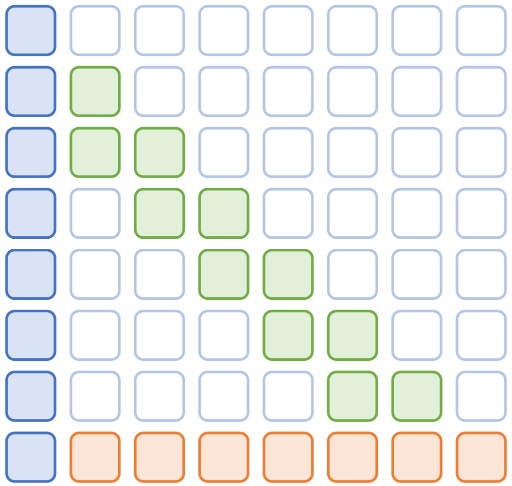
Motivation

Input Dynamism *in attention masks*

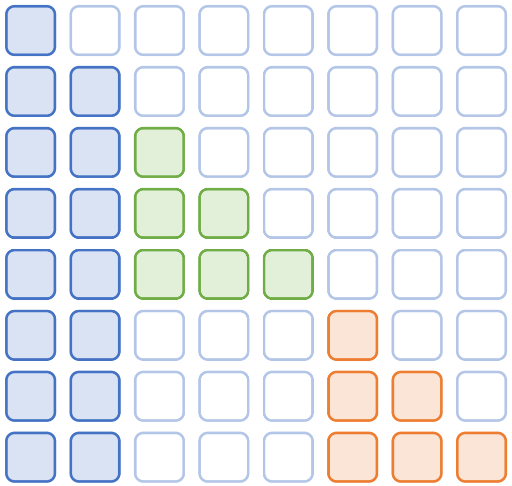
Other masks…



StreamingLLM



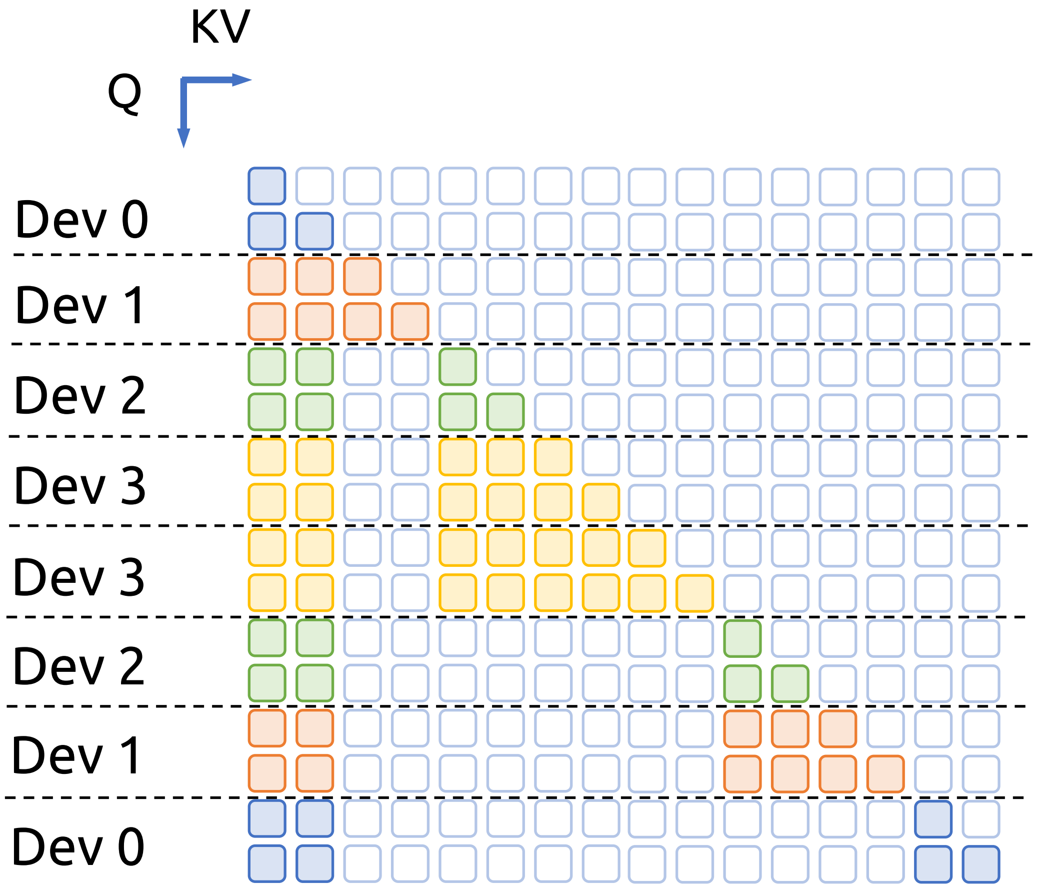
Causal Blockwise Mask

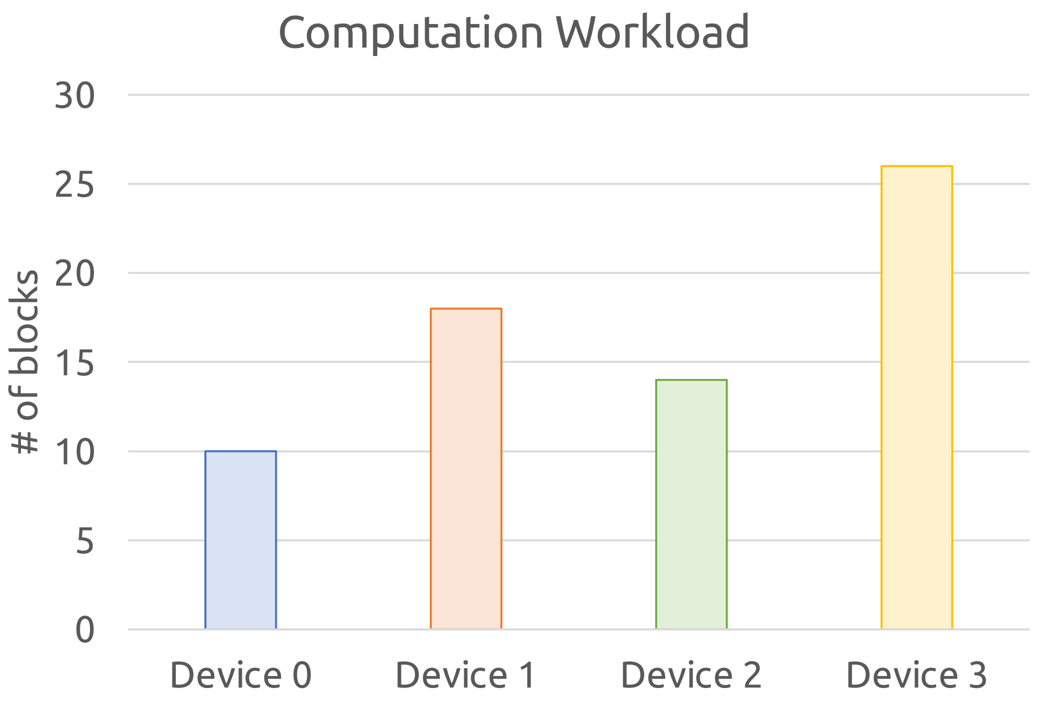


Shared Question Mask

Motivation

Input Dynamism *in attention masks*

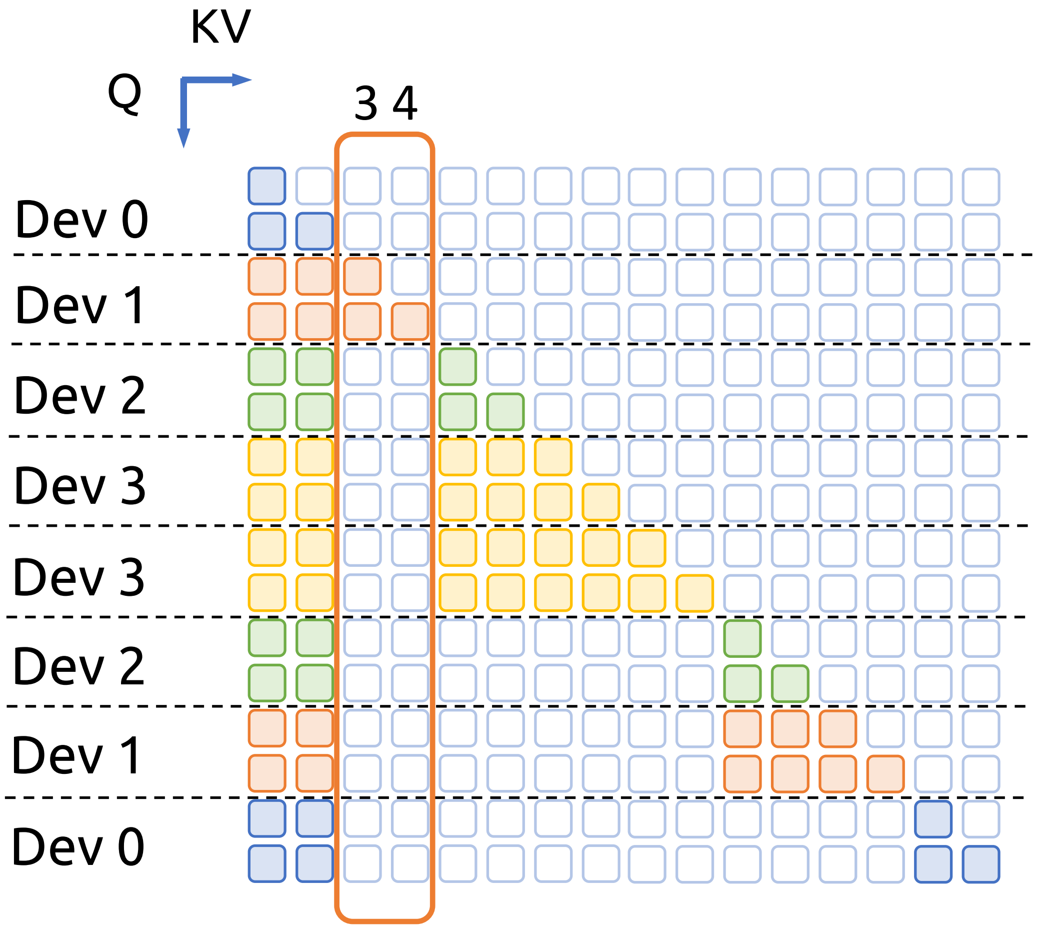


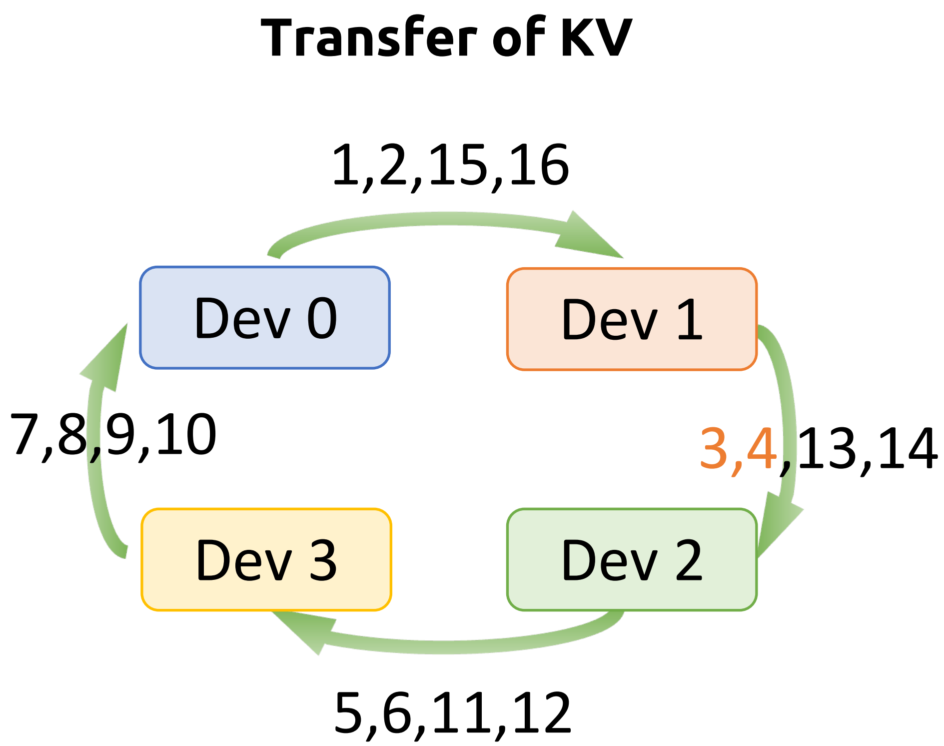


Computation Imbalance

Motivation

Input Dynamism *in attention masks*





Redundant Communication

Motivation

Input Dynamism *in distributed attention*

Observation:

Different batches require **different parallelization strategy** for optimal performance.

Question:

How to automatically optimize parallelization strategy for each input batch?

How to build a system that flexibly adapts to such dynamic parallelism?

Design

Path to automatic parallelization strategy optimization

1. Optimize the placement of data and computation (parallelization)

2. Determine the schedule of communication and computation

Design

Path to automatic parallelization strategy optimization

1. Optimize the placement of data and computation (parallelization)

2. Determine the schedule of communication and computation

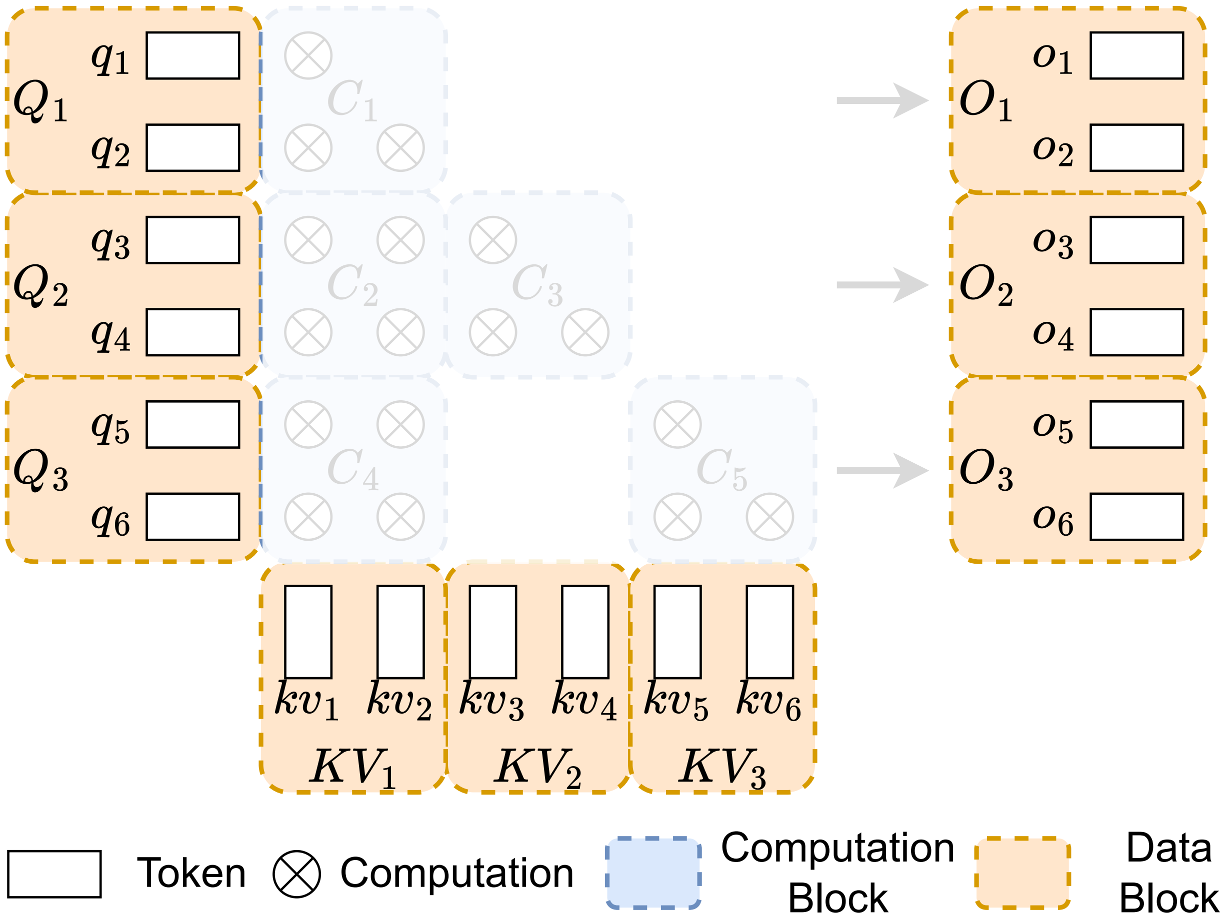
Design

Optimize parallelization with Hypergraph Partition

Attention described with two type of blocks:

Data blocks

Computation blocks



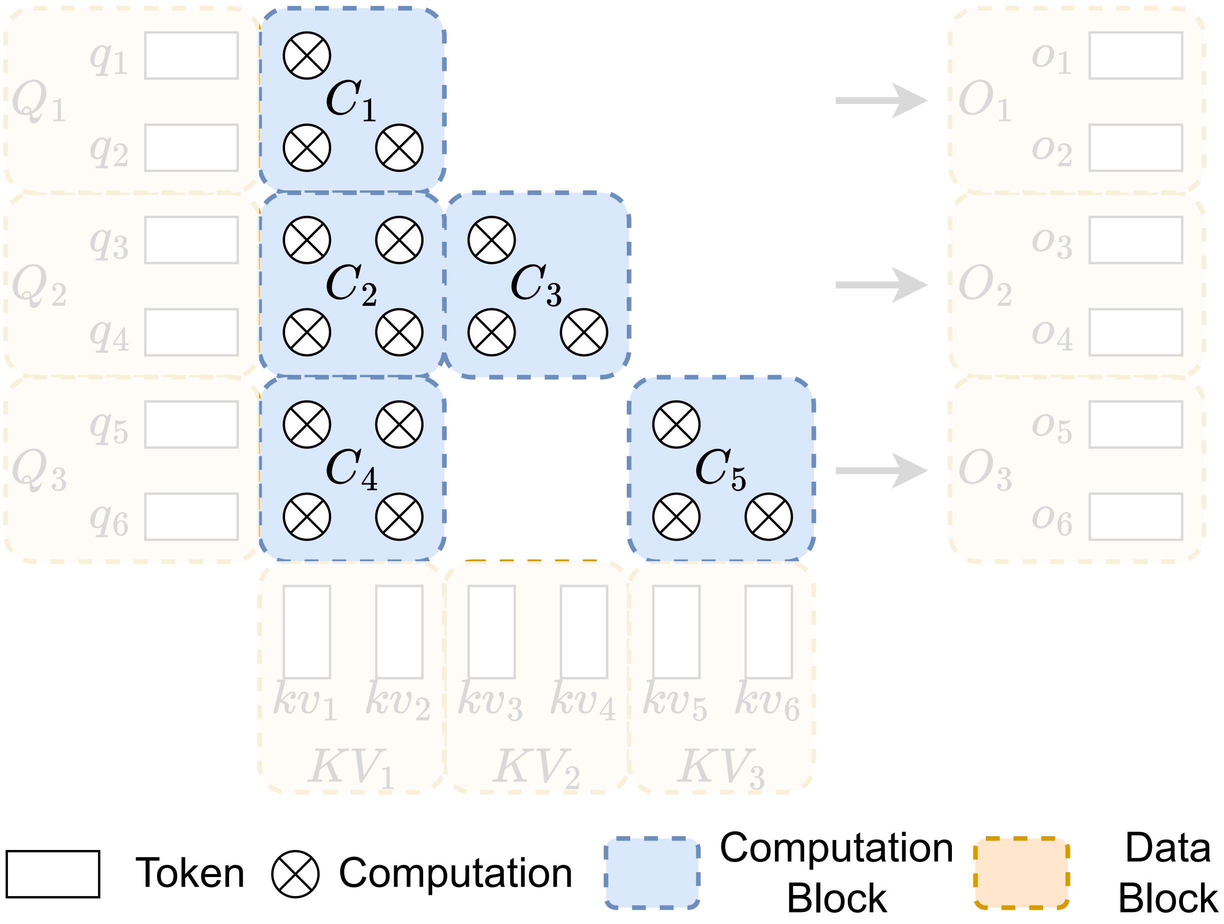
Design

Optimize parallelization with Hypergraph Partition

Attention described with two type of blocks:

Data blocks

Computation blocks



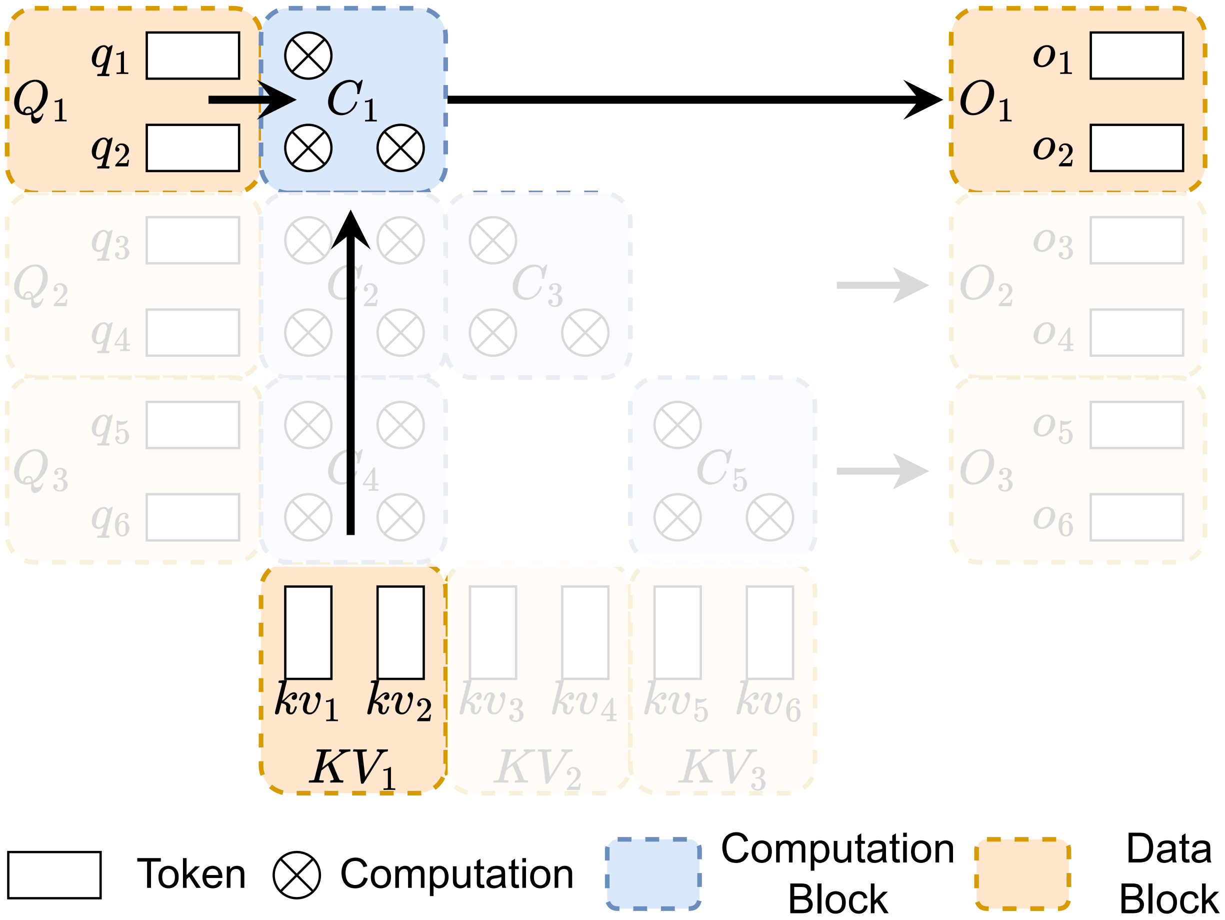
Design

Optimize parallelization with Hypergraph Partition

Attention described with two type of blocks:

Data blocks

Computation blocks



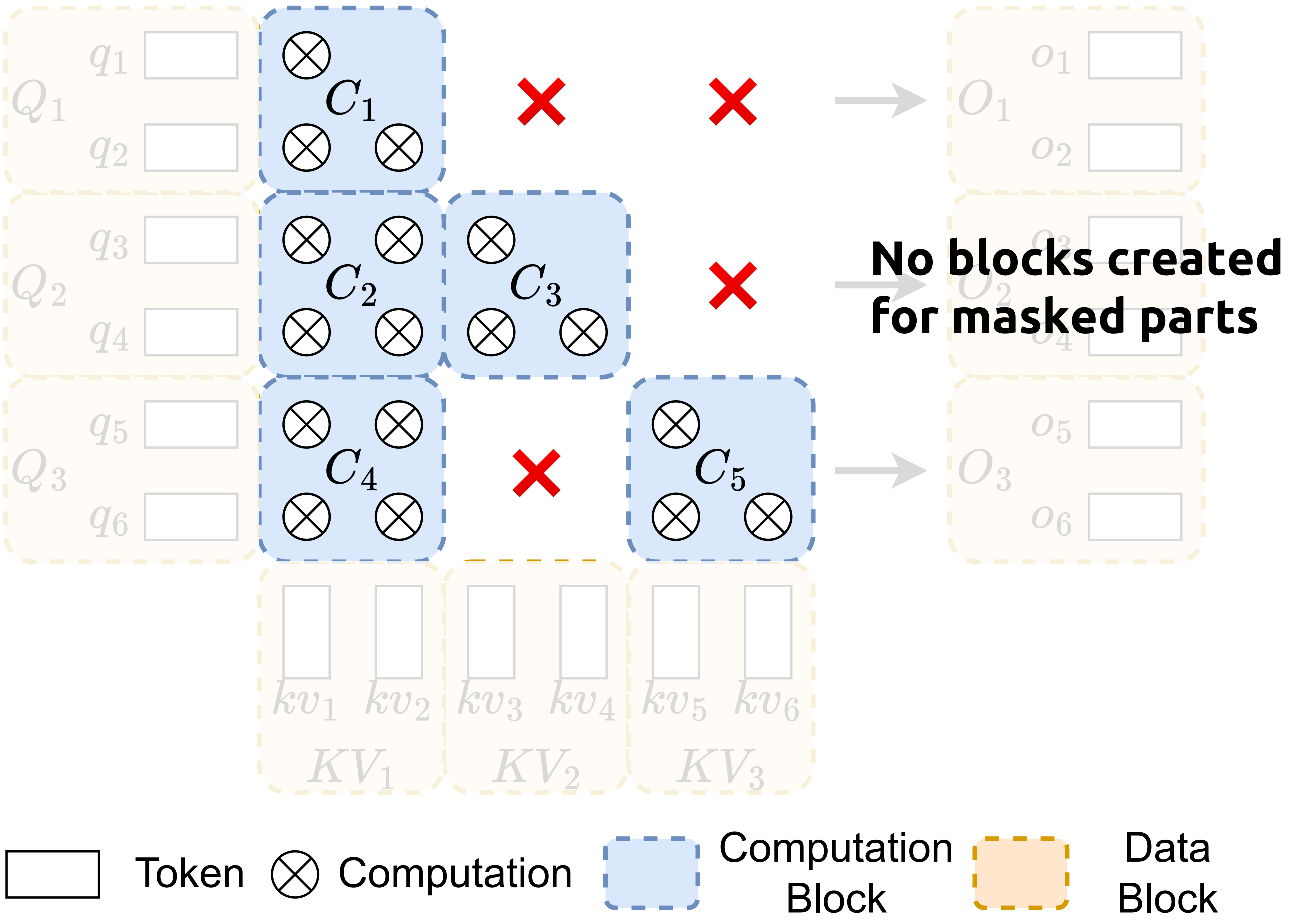
Design

Optimize parallelization with Hypergraph Partition

Attention described with two type of blocks:

Data blocks

Computation blocks



Design

Optimize parallelization with Hypergraph Partition

Goal: find a balanced partition of data and computation, while minimizing communication

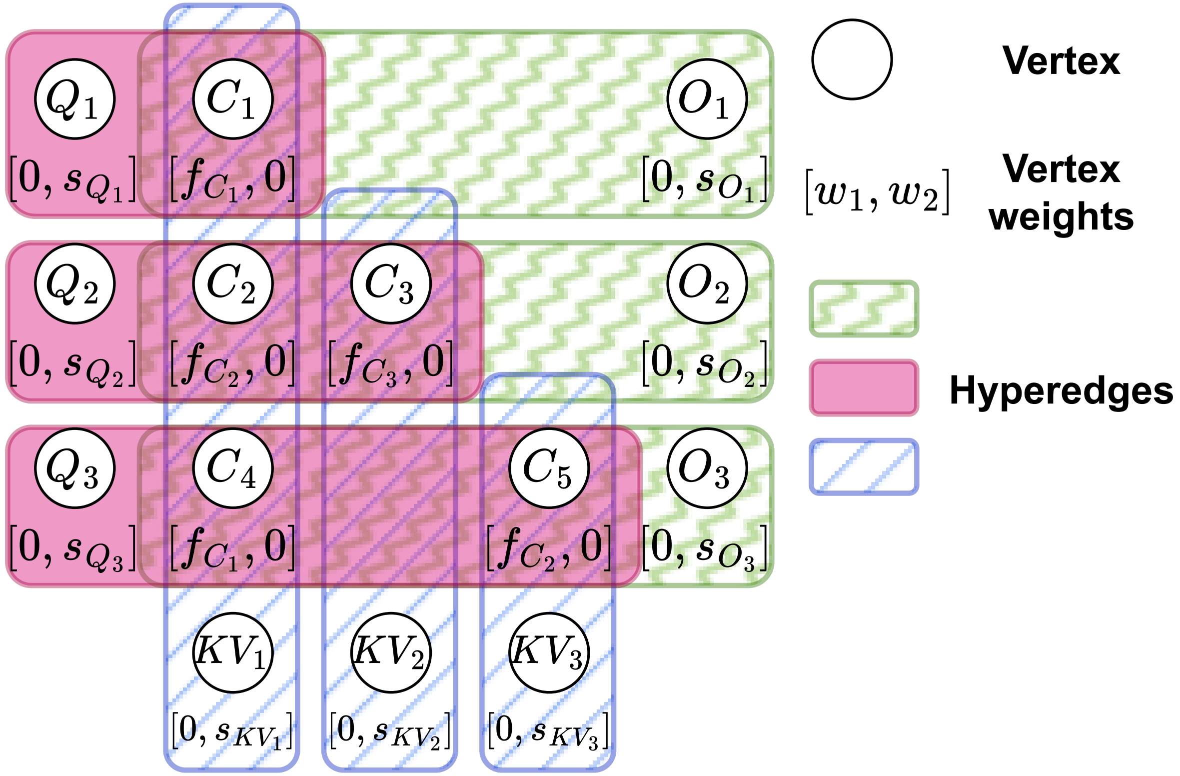
Hypergraph: each edge connects >=2 vertices

Vertices: data and computation blocks

Hyperedges: dependency between vertices (one for each data block)

Communication: for each hyper-edge, required communication is:

$$(\#\ cut - 1) \times Size(data\ block)$$



Design

Optimize parallelization with Hypergraph Partition

Goal: find a balanced partition of data and computation, while minimizing communication

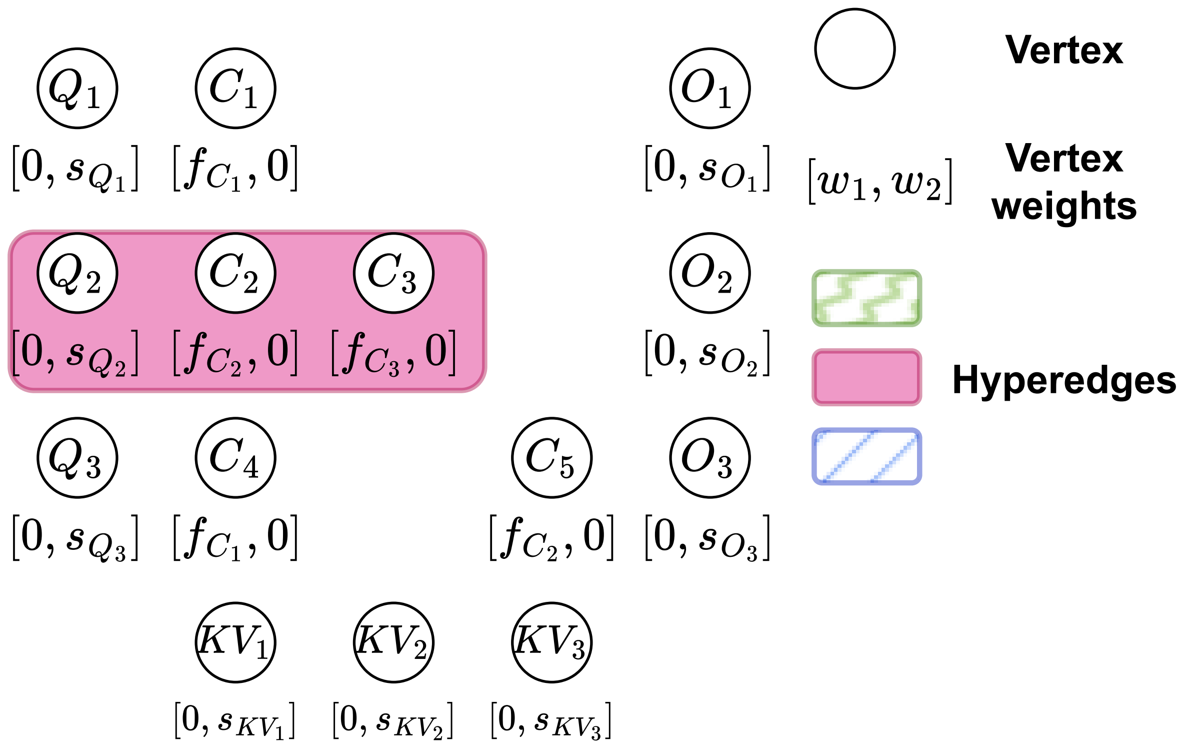
Hypergraph: each edge connects >=2 vertices

Vertices: data and computation blocks

Hyperedges: dependency between vertices (one for each data block)

Communication: for each hyper-edge, required communication is:

$$(\#\ cut - 1) \times Size(data\ block)$$



Design

Optimize parallelization with Hypergraph Partition

Goal: find a balanced partition of data and computation, while minimizing communication

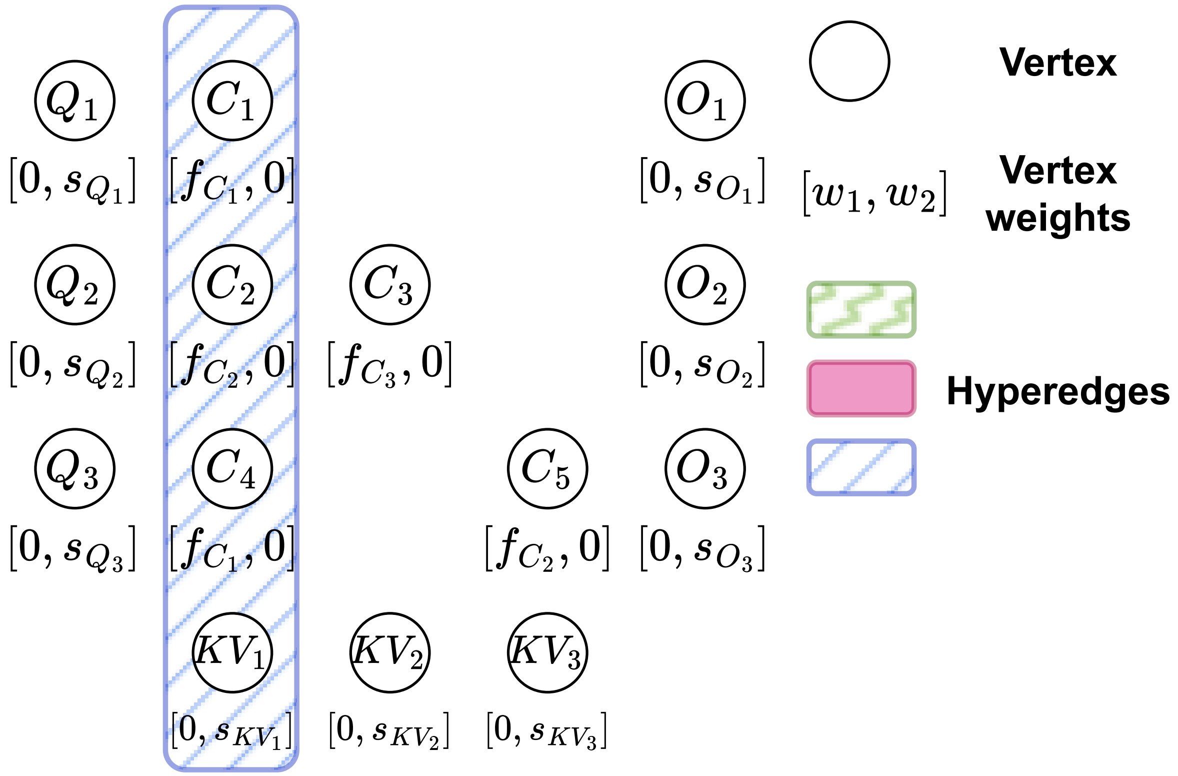
Hypergraph: each edge connects >=2 vertices

Vertices: data and computation blocks

Hyperedges: dependency between vertices (one for each data block)

Communication: for each hyper-edge, required communication is:

$$(\#\ cut - 1) \times Size(data\ block)$$



Design

Optimize parallelization with Hypergraph Partition

Goal: find a balanced partition of data and computation, while minimizing communication

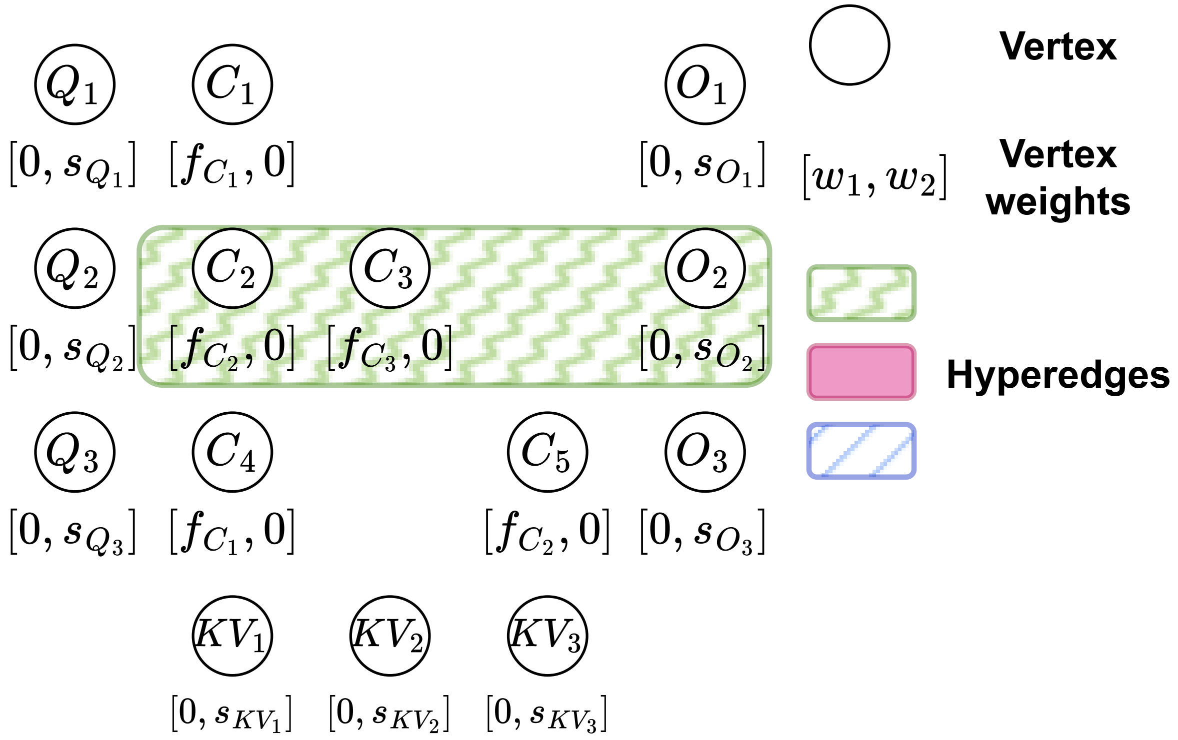
Hypergraph: each edge connects >=2 vertices

Vertices: data and computation blocks

Hyperedges: dependency between vertices (one for each data block)

Communication: for each hyper-edge, required communication is:

$$(\#\ cut - 1) \times Size(data\ block)$$



Design

Optimize parallelization with Hypergraph Partition

Goal: find a **balanced** partition of data and computation, while **minimizing** communication

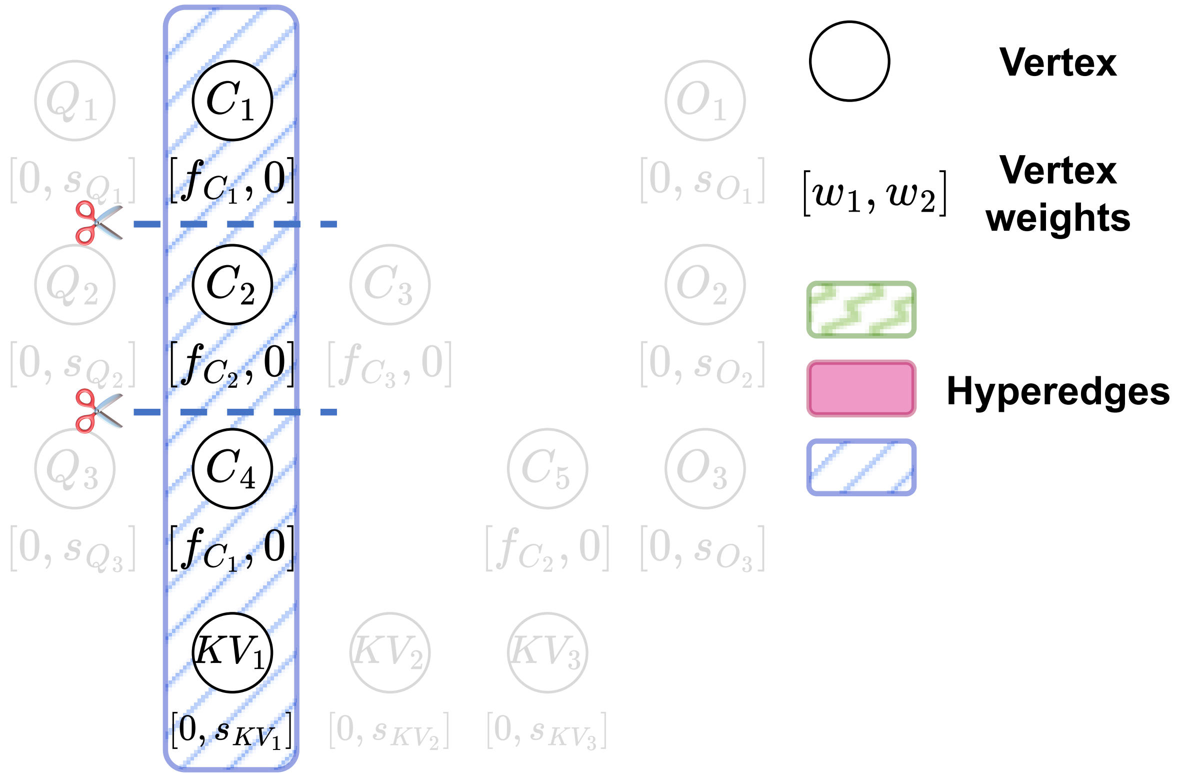
Hypergraph: each edge connects >=2 vertices

Vertices: data and computation blocks

Hyperedges: dependency between vertices (one for each data block)

Communication: for each hyper-edge, required communication is:

$$(\# \, cut - 1) \times Size(data \; block)$$

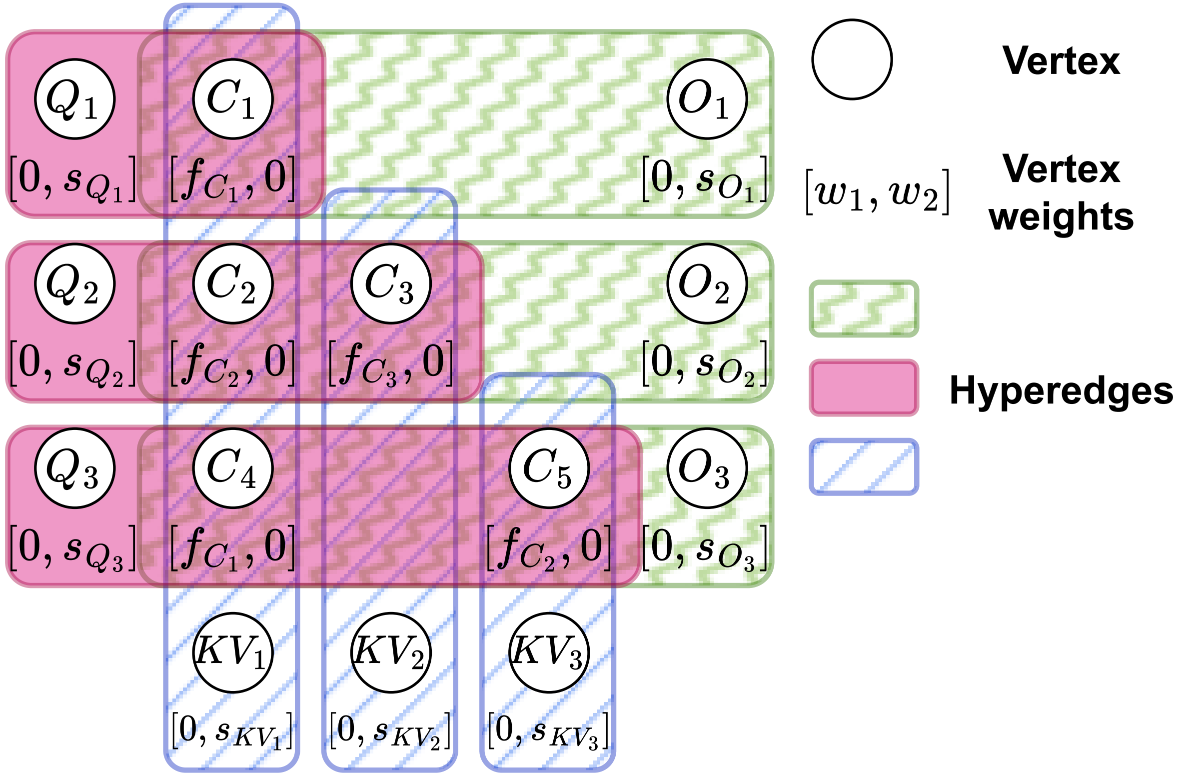


Design

Optimize parallelization with Hypergraph Partition

Goal: find a balanced partition of data and computation, while minimizing communication

Solving the balanced hyper-graph partitioning problem yields the optimal data and computation placement.



Design

Path to automatic parallelism optimization

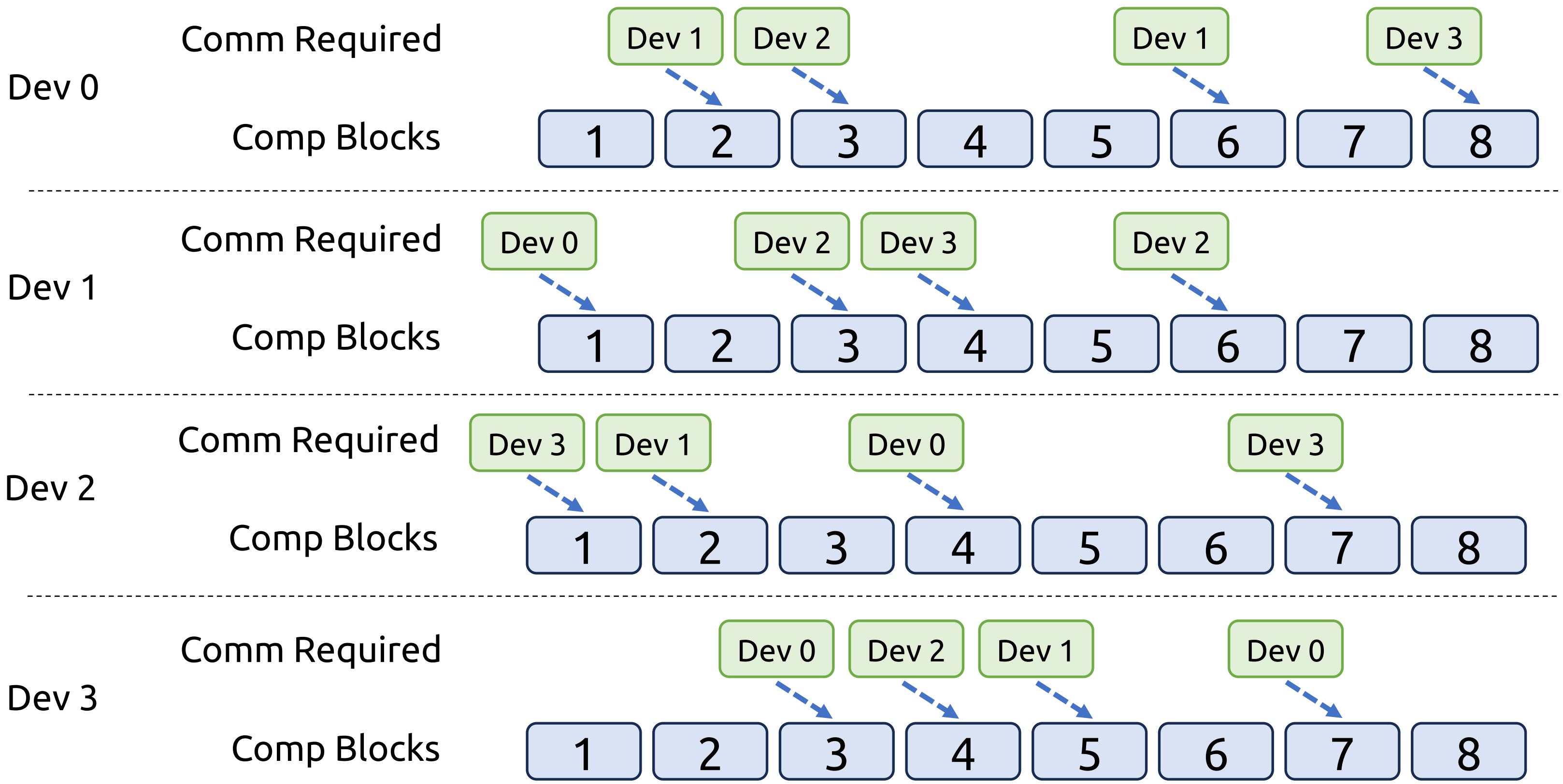
1. Optimize the placement of data and computation (parallelization)

2. Determine the schedule of communication and computation

Design

Block scheduling for overlapping computation and communication

Goal: maximize communication-computation overlap while avoiding congestion



Design

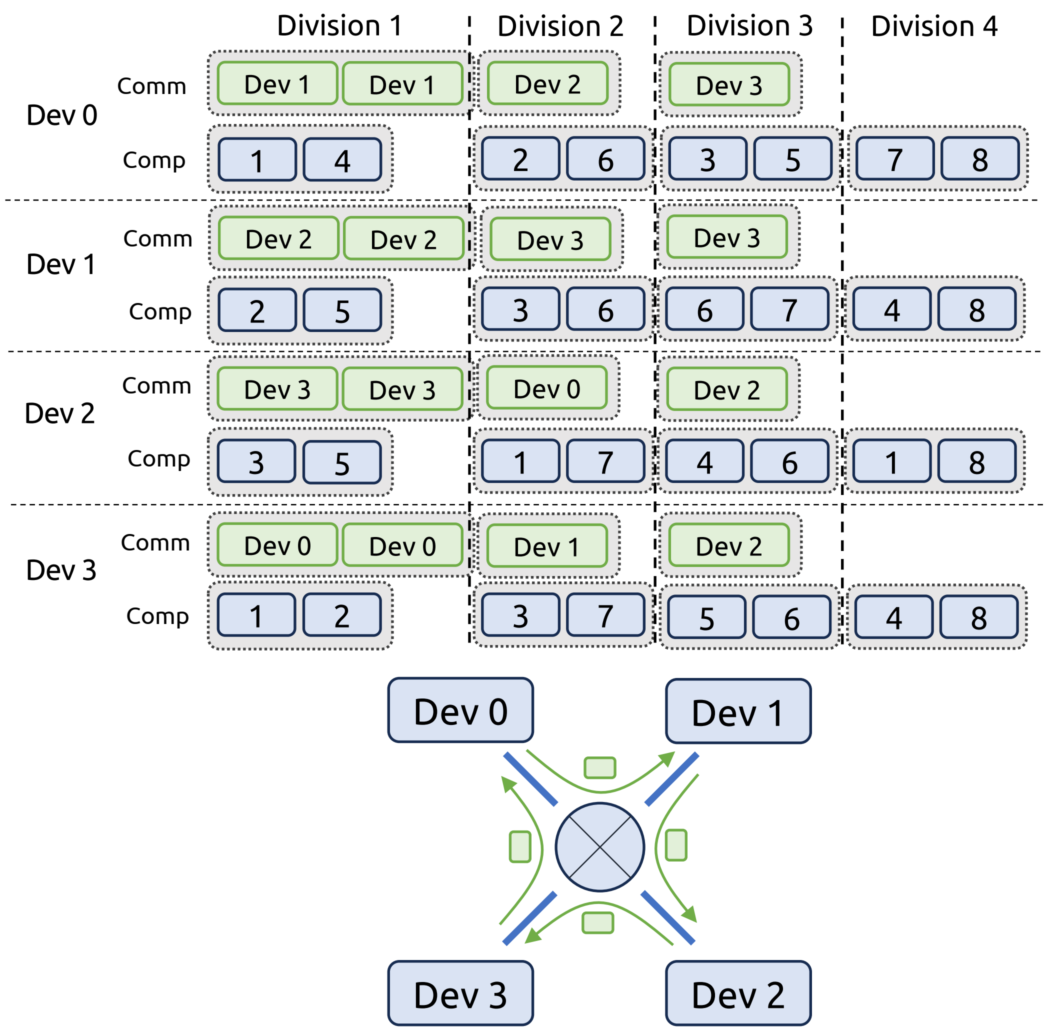
Block scheduling for overlapping computation and communication

Partition comm. and comp. on each device into **divisions.**

Within each division, desire **balanced computation and communication.**

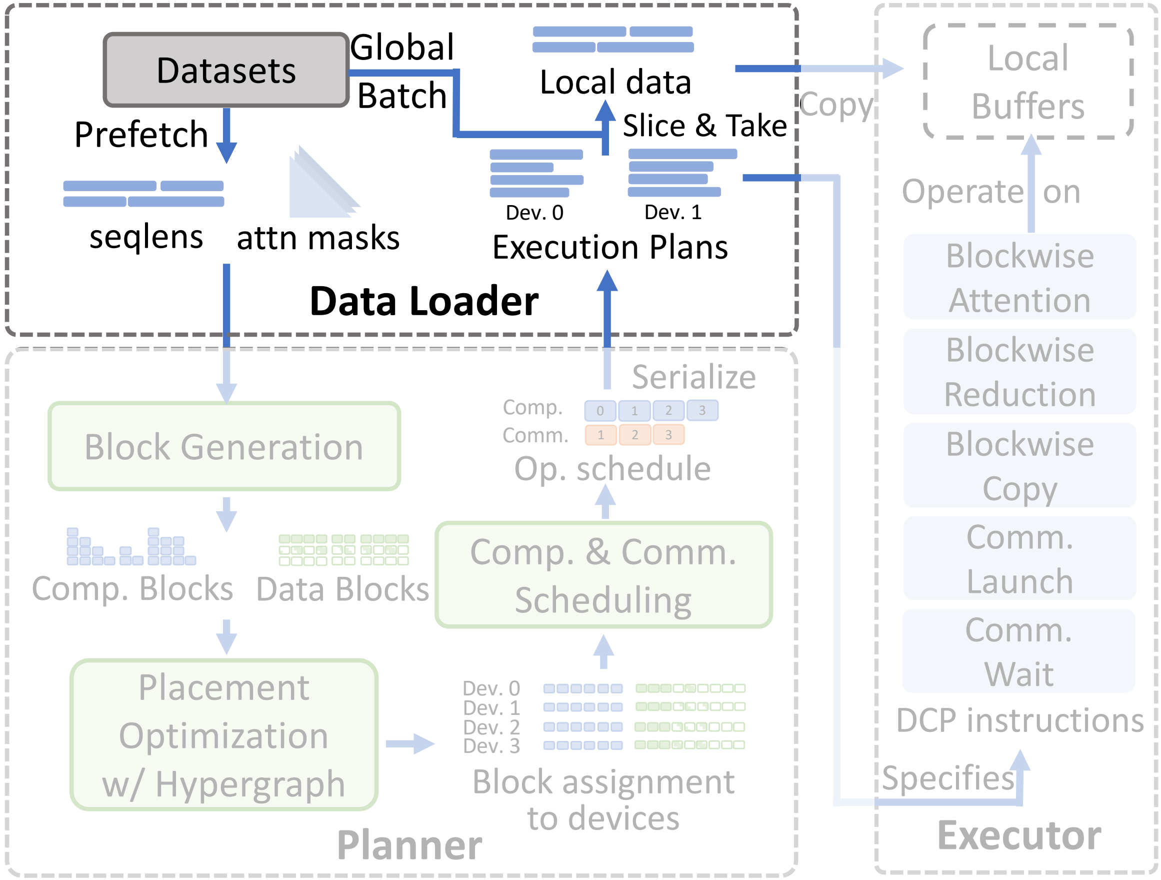
Communication required by the next division can **overlap** with computation in the current division.

Problem is **NP-hard**, using a greedy heuristic.



Design

System Overview

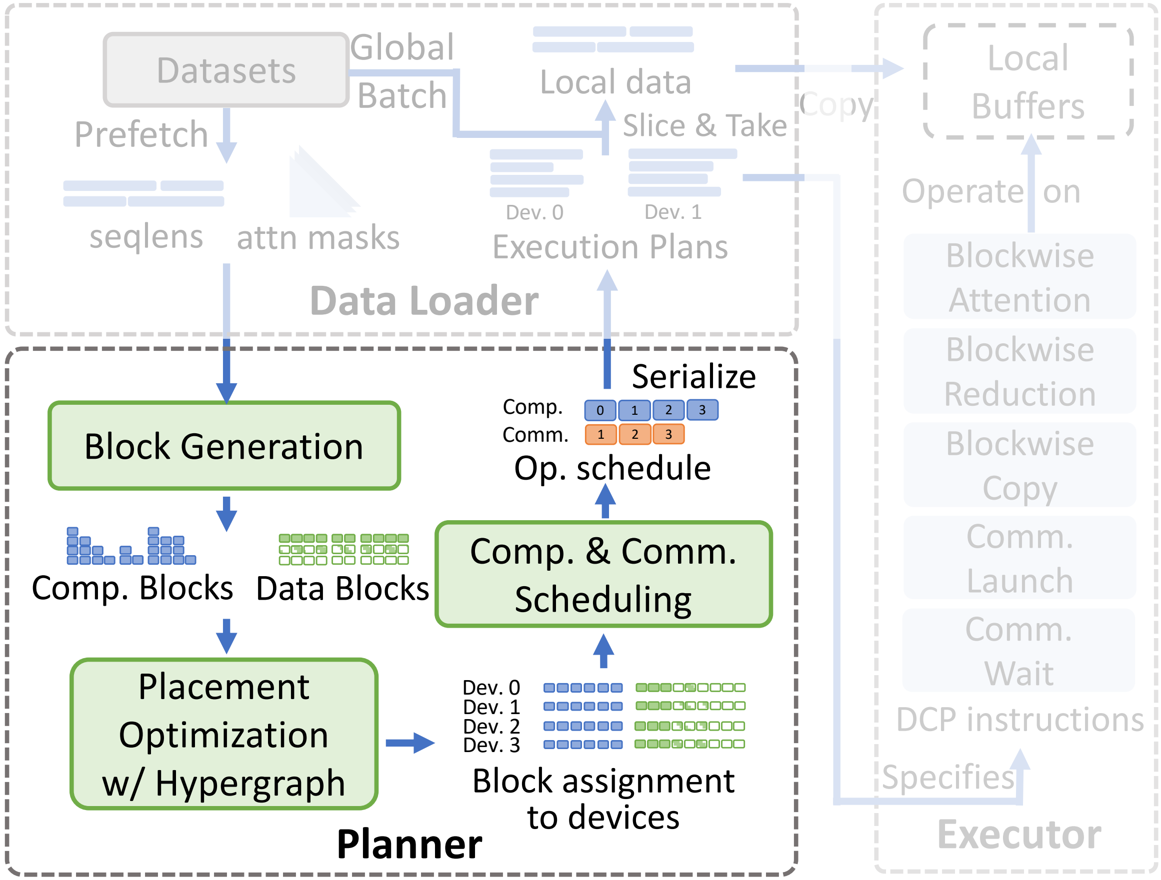



```
# when defining models
class TransformerLayer(...):
  def forward(..., dcp_executor):
    ...
    # replace attention implementation with DCPAttn
    core_attn_out = DCPAttn.apply(dcp_executor, q, kv)
    ...
```

```
# in training script
dcp_dataloader = DCPDataloader(dataset, mask_fn)
# dcp_group is a communicator that connects all devices
# (e.g., torch.distributed.ProcessGroup)
dcp_executor = DCPExecutor(group=dcp_group)
# training iterations
for (local_data, execution_plan) in dcp_dataloader:
  # set execution plan and create buffers
  dcp_executor.prepare(execution_plan)
  # execute model
  loss = model(local_data, dcp_executor)
  ...
```

Design

System Overview

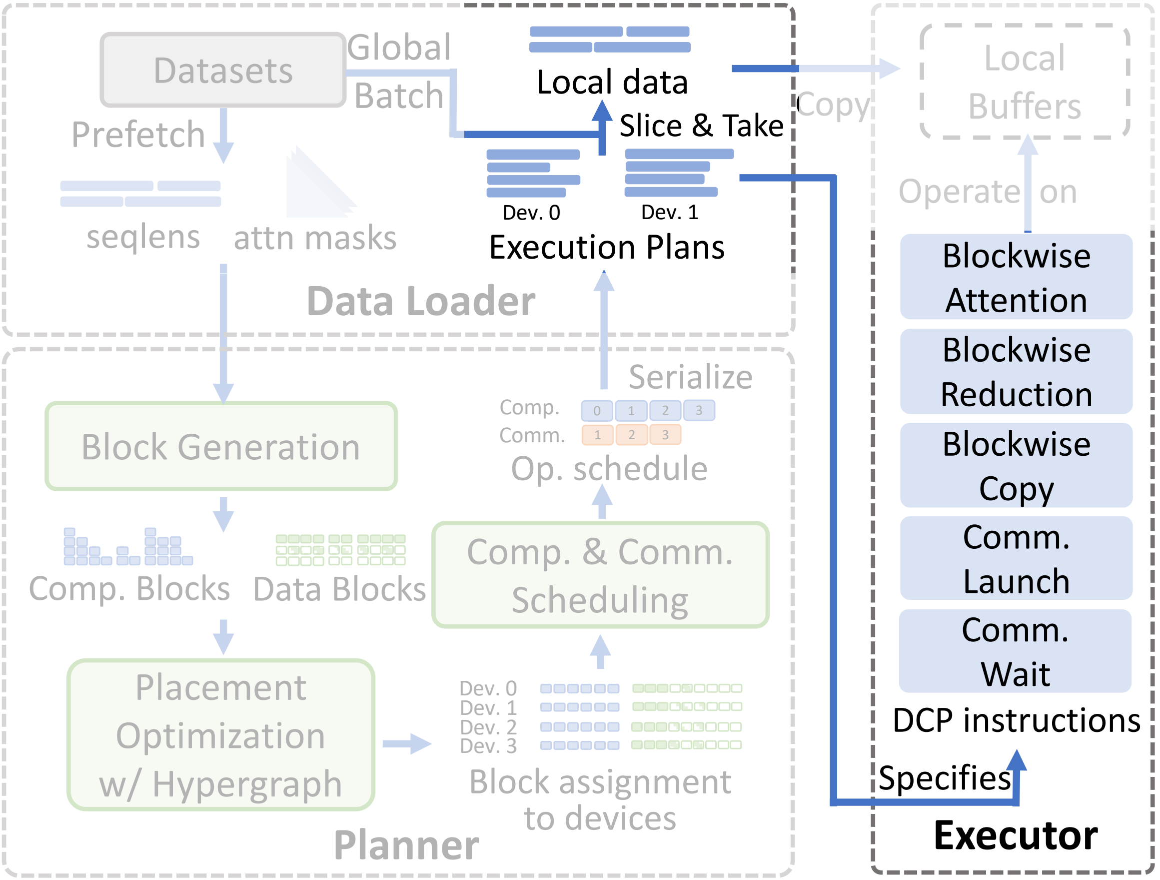



```python
# when defining models
class TransformerLayer(...):
  def forward(..., dcp_executor):
    ...
    # replace attention implementation with DCPAttn
    core_attn_out = DCPAttn.apply(dcp_executor, q, kv)
    ...
```

```python
# in training script
dcp_dataloader = DCPDataloader(dataset, mask_fn)
# dcp_group is a communicator that connects all devices
# (e.g., torch.distributed.ProcessGroup)
dcp_executor = DCPExecutor(group=dcp_group)
# training iterations
for (local_data, execution_plan) in dcp_dataloader:
  # set execution plan and create buffers
  dcp_executor.prepare(execution_plan)
  # execute model
  loss = model(local_data, dcp_executor)
  ...
```

Design

System Overview

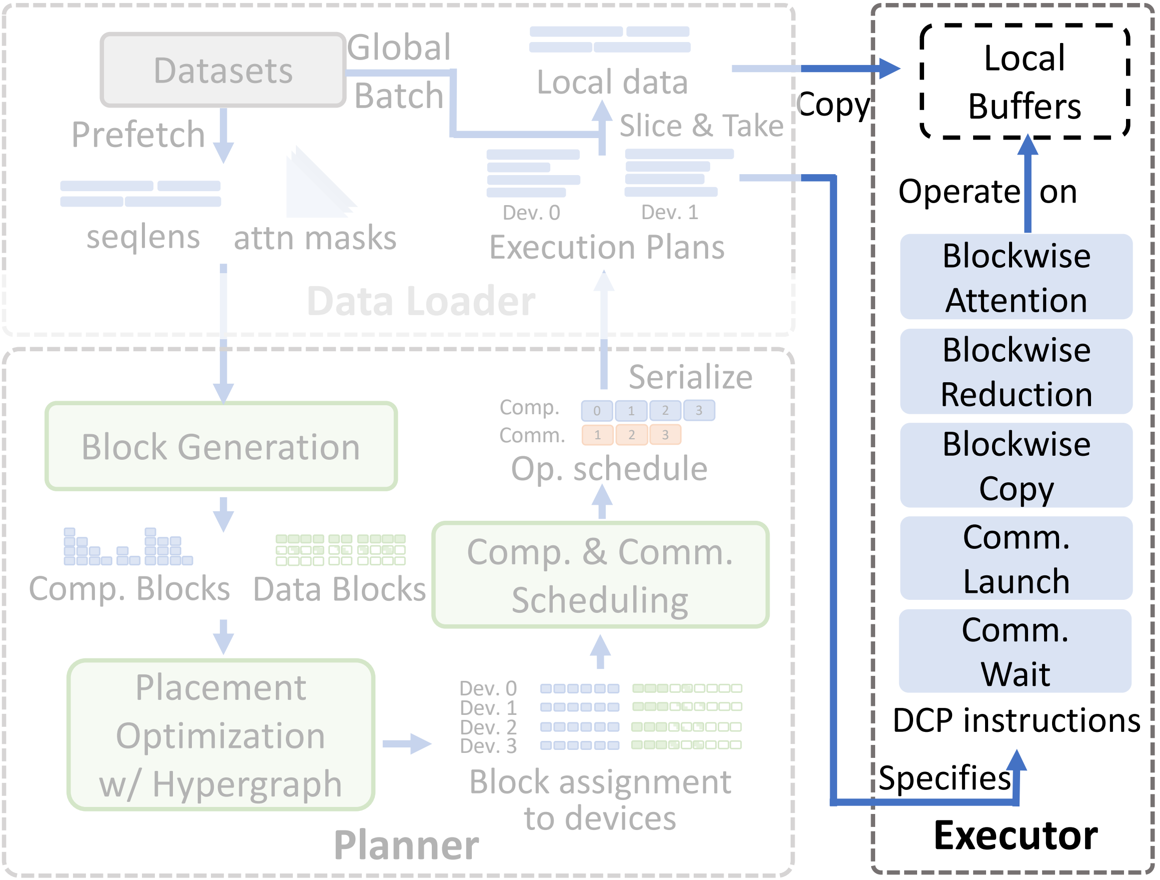



```
# when defining models
class TransformerLayer(...):
  def forward(..., dcp_executor):
    ...
    # replace attention implementation with DCPAttn
    core_attn_out = DCPAttn.apply(dcp_executor, q, kv)
    ...
```

```
# in training script
dcp_dataloader = DCPDataloader(dataset, mask_fn)
# dcp_group is a communicator that connects all devices
# (e.g., torch.distributed.ProcessGroup)
dcp_executor = DCPExecutor(group=dcp_group)
# training iterations
for (local_data, execution_plan) in dcp_dataloader:
  # set execution plan and create buffers
  dcp_executor.prepare(execution_plan)
  # execute model
  loss = model(local_data, dcp_executor)
  ...
```

Design

System Overview

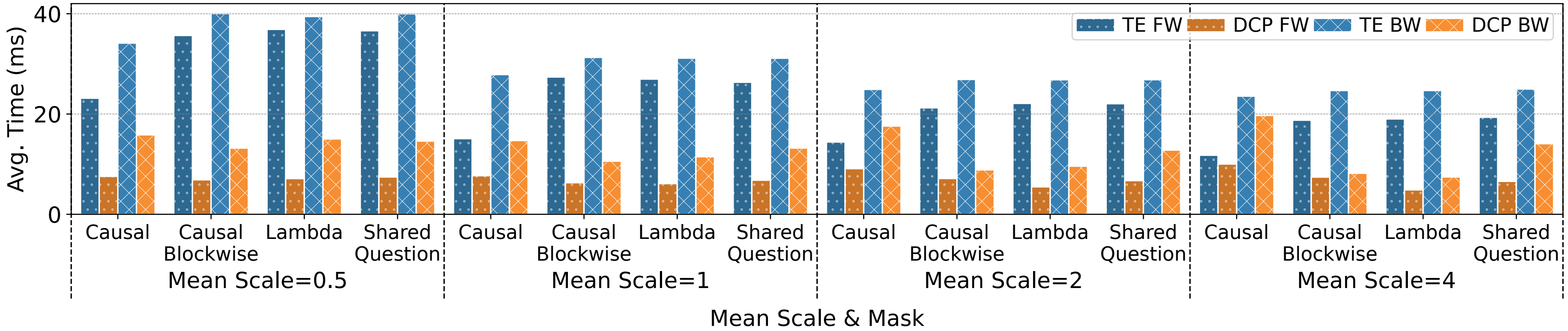



```python
# when defining models
class TransformerLayer(...):
  def forward(..., dcp_executor):
    ...
    # replace attention implementation with DCPAttn
    core_attn_out = DCPAttn.apply(dcp_executor, q, kv)
    ...
```

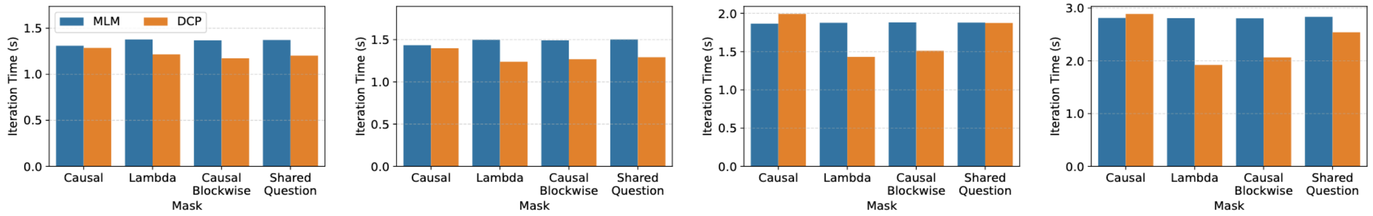
```python
# in training script
dcp_dataloader = DCPDataloader(dataset, mask_fn)
# dcp_group is a communicator that connects all devices
# (e.g., torch.distributed.ProcessGroup)
dcp_executor = DCPExecutor(group=dcp_group)
# training iterations
for (local_data, execution_plan) in dcp_dataloader:
  # set execution plan and create buffers
  dcp_executor.prepare(execution_plan)
  # execute model
  loss = model(local_data, dcp_executor)
  ...
```

Key Results

Configuration: 8 nodes, each with 8 A100-80GB GPUs, 400 Gbps interconnect between nodes
Model: follow llama3-8B setup

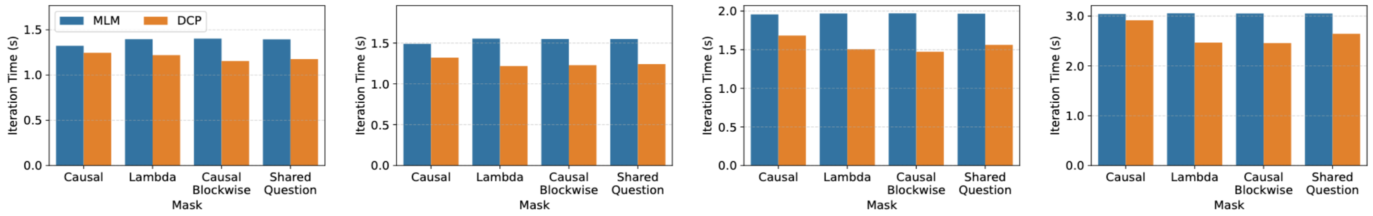


Attention microbenchmark: speed up 1.19x~2.45x under causal masks, 2.15x~3.77x under sparse masks



(a) Max Seq. Len. 16384. (b) Max Seq. Len. 32768. (c) Max Seq. Len. 65536. (d) Max Seq. Len. 131072.

End-to-end training performance on the LongAlign dataset.

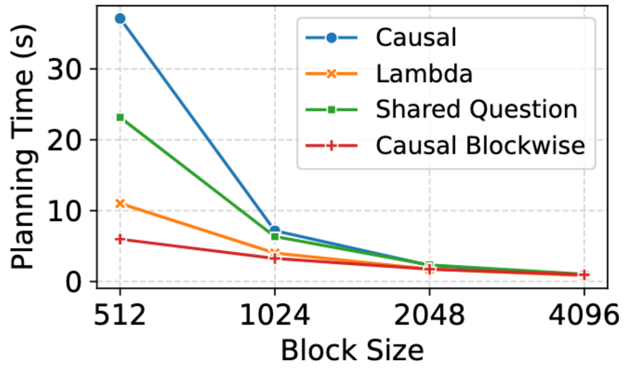


(a) Max Seq. Len. 16384. (b) Max Seq. Len. 32768. (c) Max Seq. Len. 65536. (d) Max Seq. Len. 131072.

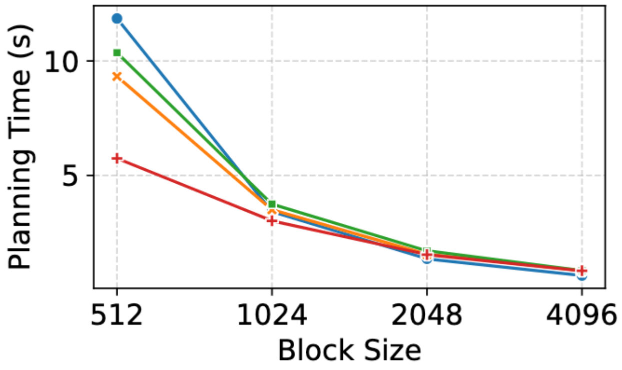
End-to-end training performance on the LongDataCollections dataset.

End-to-end benchmarks: speed up 0.94x~1.16x under causal masks, 1.00x~1.46x under sparse masks

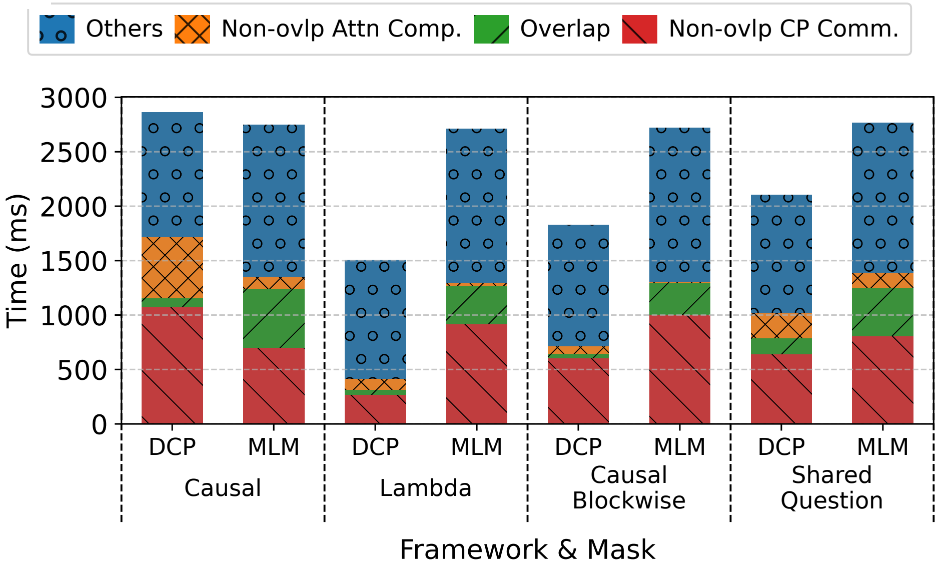
Key Results



LongAlign.



LongDataCollections.



Caption

Planning time: < 10s per iteration under reasonable block size, full overlap with model execution when parallelized onto more than 10 CPU cores.

Timeline decomposition: communication time greatly reduced. Potential performance improvement with better communication scheduling.

Takeaway

Dynamism in model input — sequence length and attention mask — can be exploited to accelerate context parallelism training.

Context parallelization strategies can be described and optimized in two layers:

1. data and computation placement (modelled and optimized with hyper-graph representation)
2. Computation/communication scheduling.

Thank you

GitHub Repo

Contact: jchenyu@connect.hku.hk