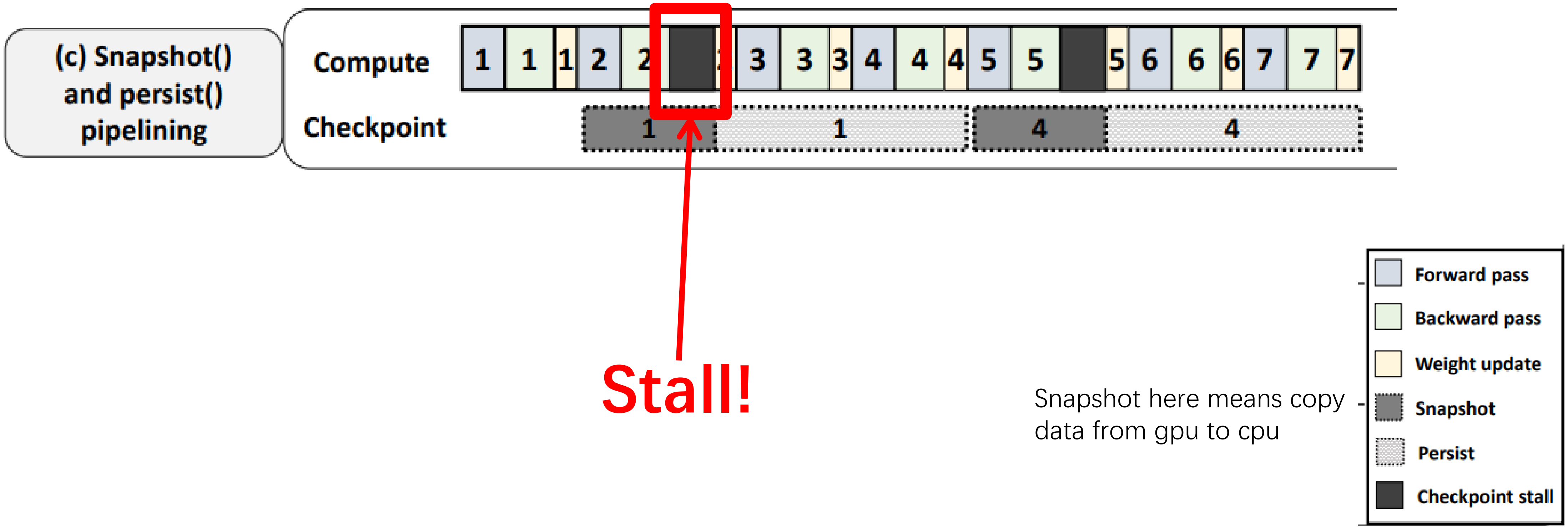
Problems in CheckFreq

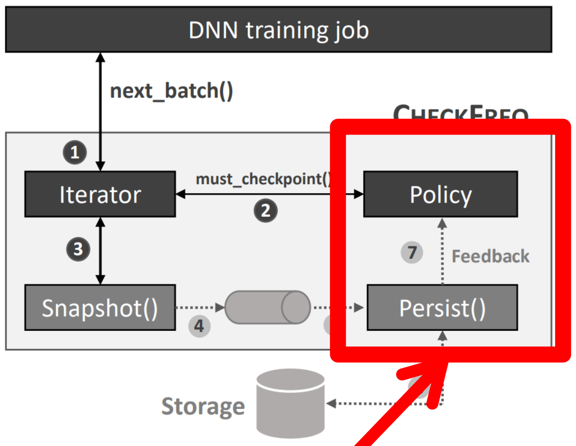
XXC

Still has Checkpoint Stall



Snapshot here means copy data from gpu to cpu

Not a Real-Time Frequency Determination



Algorithm 1 : Checkpointing frequency determination

Input: $Ti, Tw, Tc, Tg, Ts, m, M, M_{max}, p$

$T_{oc} \leftarrow max(0, T_c - (T_i - T_w))$

$T_{og} \leftarrow T_g$

if $M_{max} - M > m$ **and** $T_{og} \leq T_{oc}$ **then**

$T_o \leftarrow T_{og}$

$mode \leftarrow GPU$

else

$T_o \leftarrow T_{oc}$

$mode \leftarrow CPU$

end if

$k \leftarrow \frac{T_c + T_s - T_o}{T_i}$

$k_{min} \leftarrow \left\lceil \frac{T_o}{p * T_i} \right\rceil$

$k \leftarrow max(k, k_{min})$

Output: $\boxed{k}, mode$

frequency

Information between checkpoint interval?

Environment may have already changed!

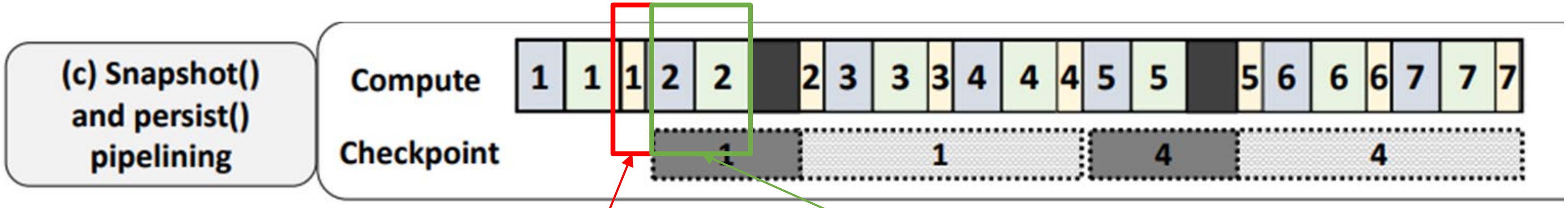
Collected Information $\xrightarrow{\text{Calculate}}$ Determine Frequency

Cause of these Problems

XXC

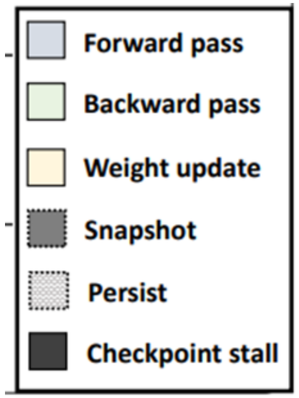
Stall: No Parallelism in Weight Update

Why can't CheckFreq Explore it?

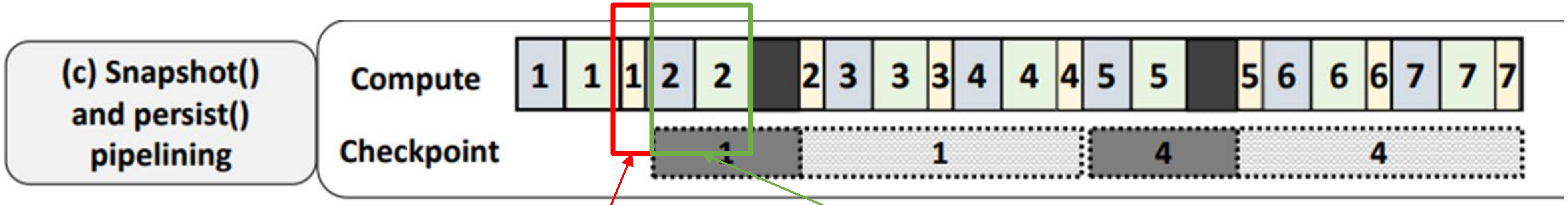


Parallelism in Weight Update Pass is not explored

Only explore parallelism in Forward & Backward Pass



Stall: Model-grained Checkpoint



Wait for the WHOLE Model to Update

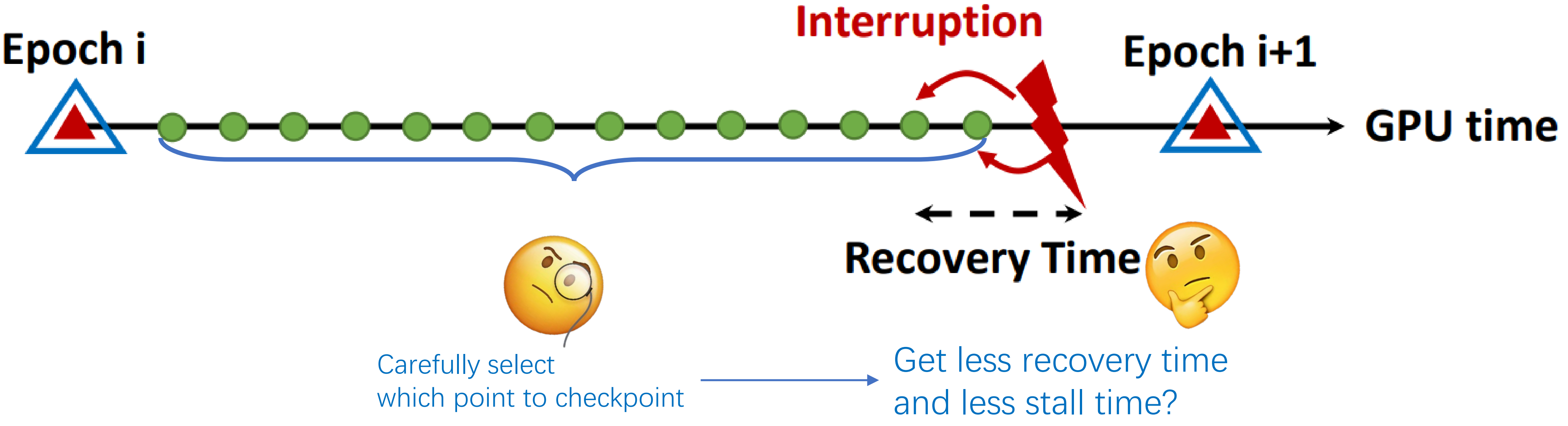
Can only start checkpoint after update the whole model

Whole Model

Store to Block Device

Serialization: We can't store what resides in memory directly into SSD/HDD, because memory is byte-addressable whereas SSD/HDD is block addressable.

Delay: Because of Checkpoint Stall

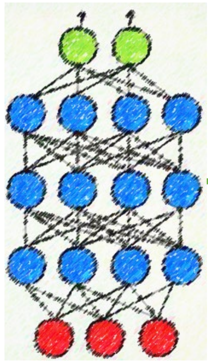


Because there may be checkpoint stall, checkpoint frequency needs to be carefully selected in order to diminish checkpoint stall while getting more checkpoints to resist failure.

One Possible Solution

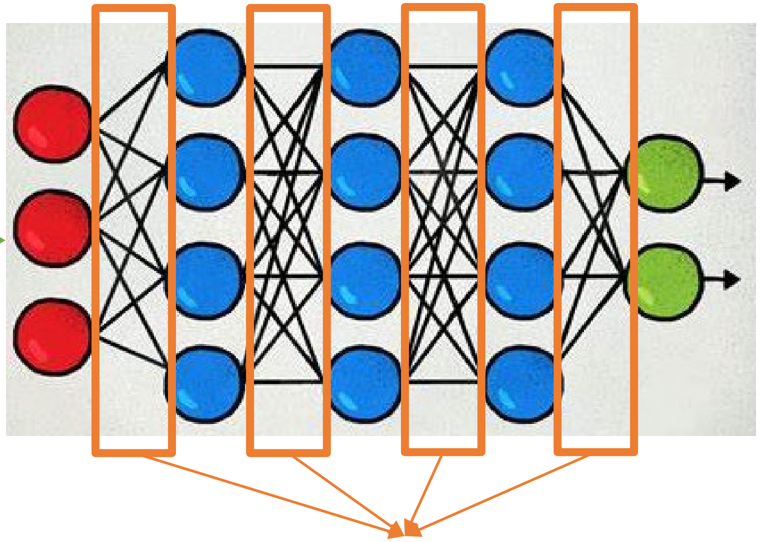
XXC

Weight-grained Serialization?



Whole Model Serialization

Start checkpoint
right after a weight completes
its own update

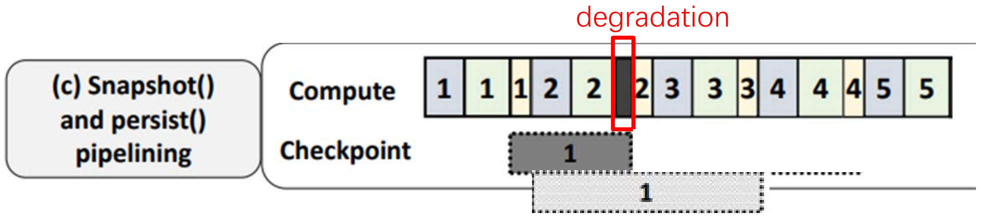


Weight-grained
serialization

Serialize each weight
instead of whole model at once

Pros:
1. This explores parallelism in weight update to some extent.
2. Also explores parallelism between snapshot and persist.

Cons:
1. When a weight cannot complete checkpoint before the next update occurs, performance degrades.
2. A checkpoint consists of many fragment files, hard to manage.



Solution for Stall

XXC

PM: A Byte-addressable Persistence Device

- PM: byte-addressability + high bandwidth + low read/write latency

Directly persist what resides in memory

More threads

  - Allow finer-grained checkpoint, more parallelism, faster speed.

- Different Strategies When encountering NEXT weight update pass before completing checkpoint:

Model-grained Serialization	Wait for the whole model gpu to cpu snapshot to complete
Weight-grained Serialization	Wait for each own weight gpu to cpu snapshot to complete
Persistent Memory (**sub-weight** grained, **without** serialization)	Chances for further less stall

Challenges of Using PM for Checkpoint

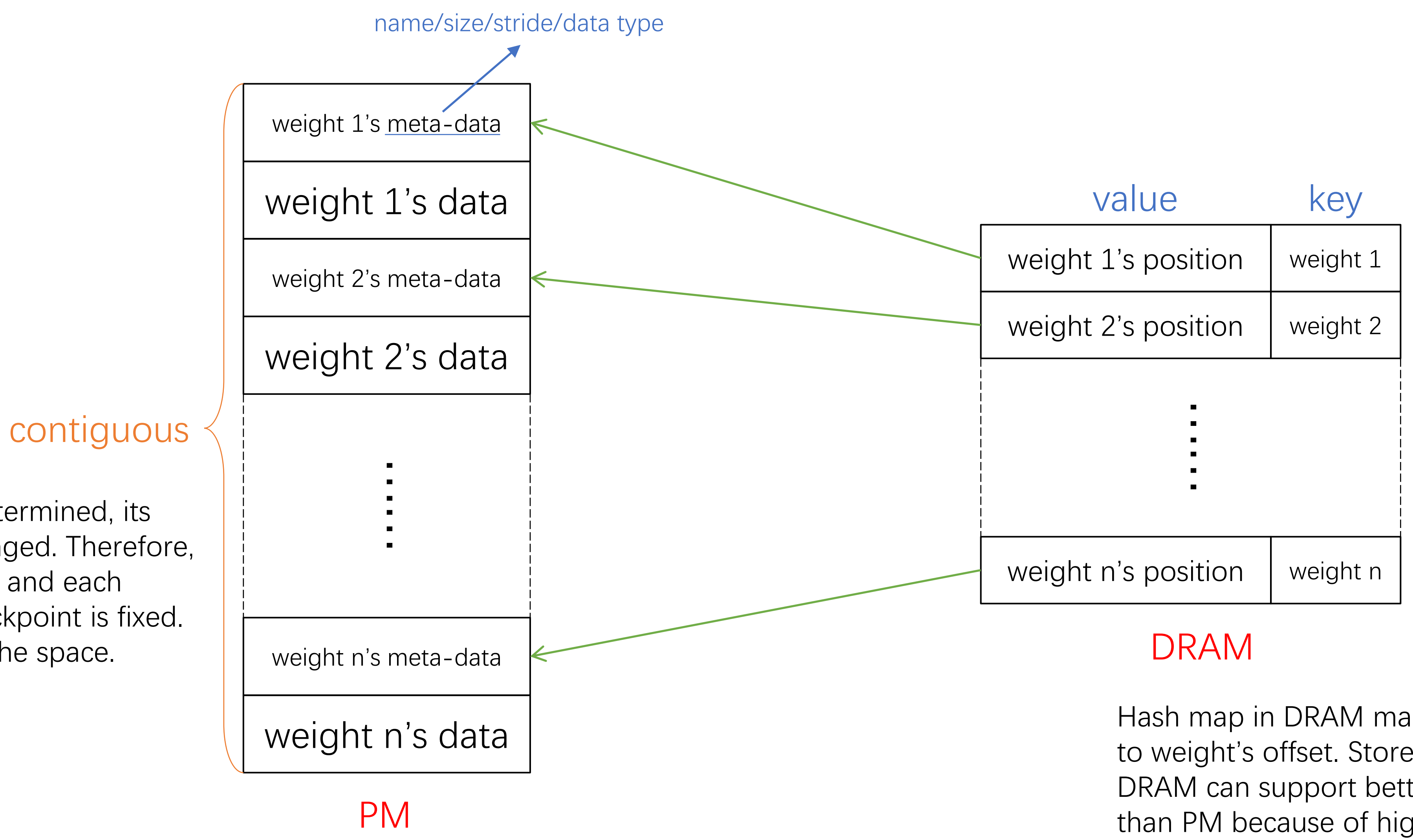
- How to structure checkpoint data and metadata in PM and DRAM
  - DRAM has many <u>runtime data</u> (such as python headers) and <u>fragment data</u> (such as different pointers).
  - It's not a good idea to store these into PM
- How to reduce overhead when encountering <u>NEXT weight update pass</u> before completing checkpoint
- How to maintain consistency within a checkpoint
- How to do checkpoints as many as possible

Call this **conflict** in the following slides

Design of PM-DNN checkpoint

XXC

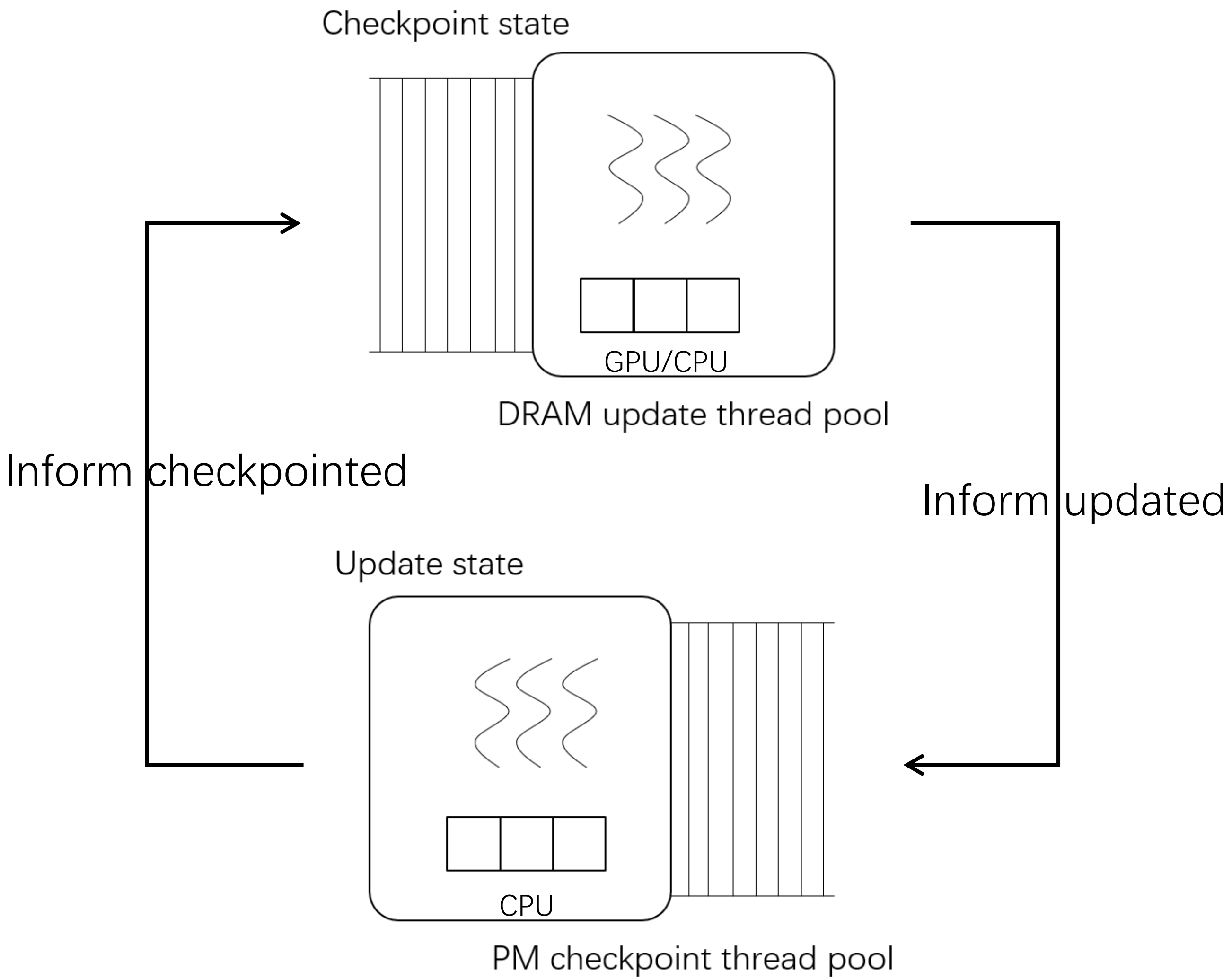
Checkpoint Structure



Once the model is determined, its structure will not changed. Therefore, the size of checkpoint and each weight's offset in checkpoint is fixed. We can pre-allocate the space.

Hash map in DRAM map weight's name to weight's offset. Store Hash map in DRAM can support better concurrency than PM because of higher bandwidth

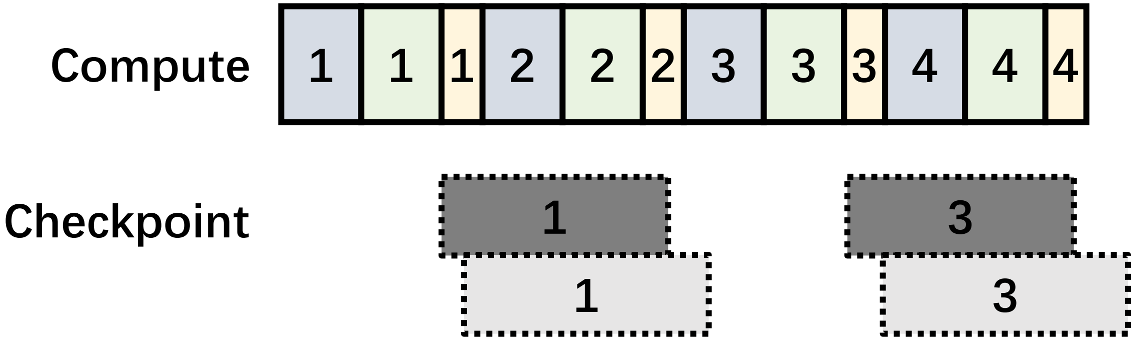
Checkpoint Pipeline



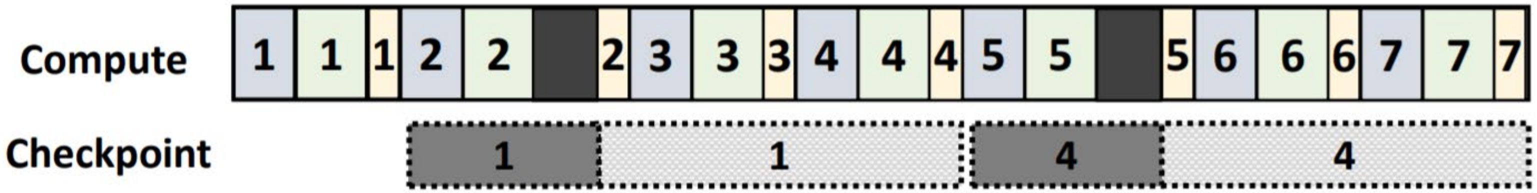

```python
def update_weight(w):
    ckpt_state = check_ckpt_state(w)
    if ckpt_state is checkpoint:
        pass
    if ckpt_state is not checkpoint:
        HandleConflict(w)
    update(w)
    change_updt_state(w, update)
```

```python
def checkpoint_weight(w):
    while(check_updt_state(w) is not update):
        continue
    checkpoint(w)
    change_ckpt_state(w, checkpoint)
```

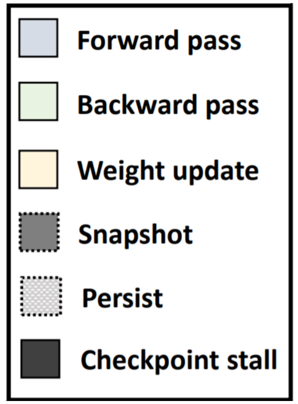
Checkpoint Pipeline



PM Checkpoint



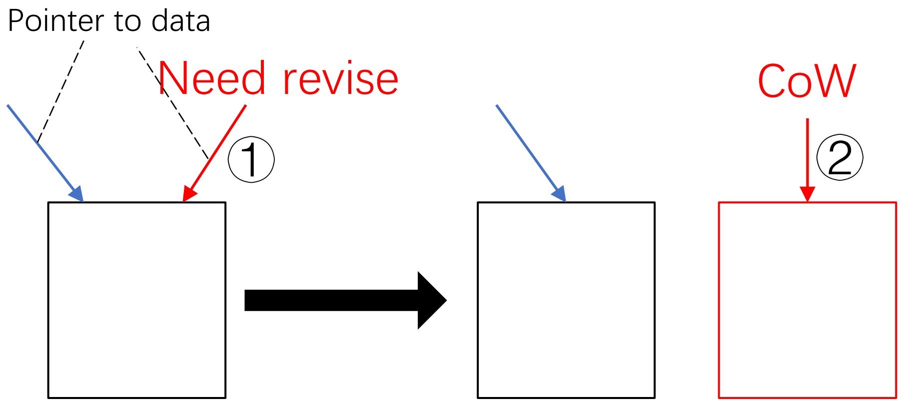
CheckFreq



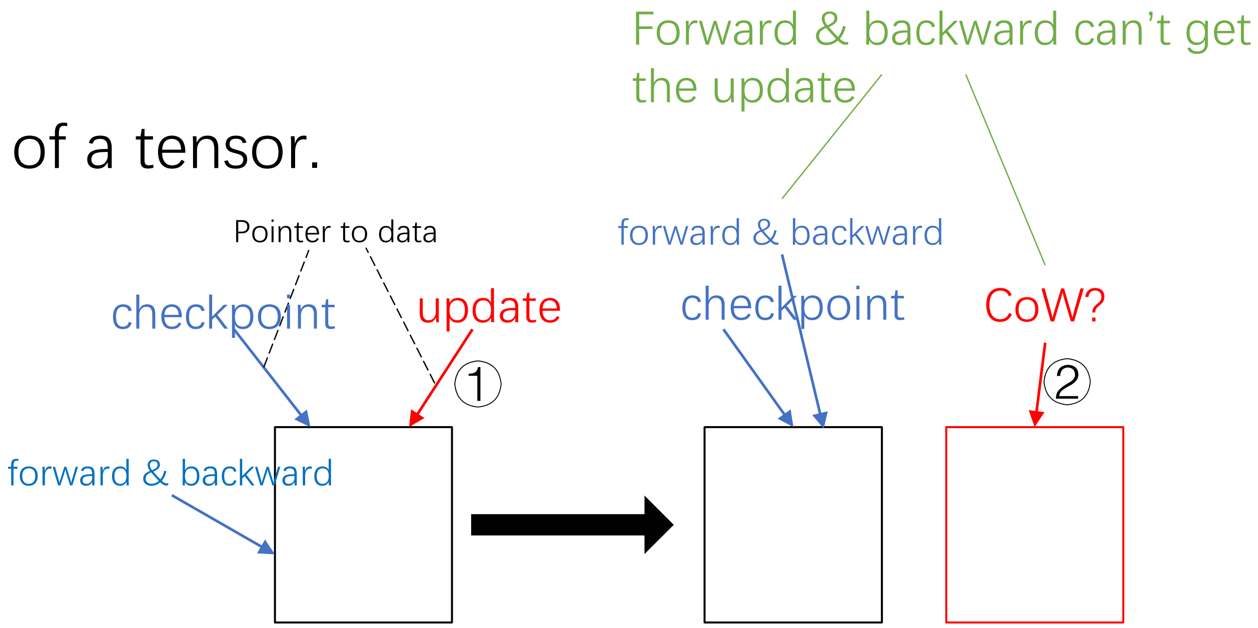
*: snapshot here means from GPU Memory to CPU Memory

Handle Conflict

- Ideal Solution: CoW
  - No conflict: read → compute → write in place
  - Conflict(CoW): read → compute → write to another place
- Not realistic in PyTorch:
  - We cannot change data pointer of a tensor.



General CoW



Compare with Checkpoint

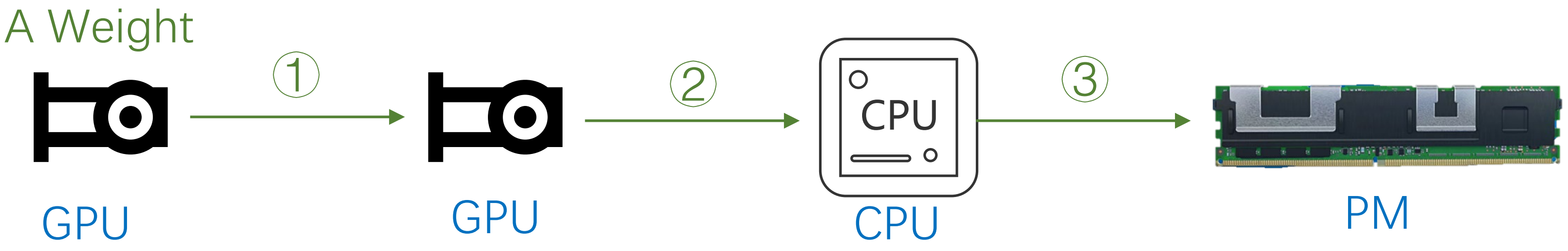
Handle Conflict

- Strategy 1:

①. In-Memory Snapshot a weight;
②. Snapshot the weight from GPU to CPU (if in GPU);
③. Persist the weight.

- Comments:
  - Pros: Stall will not occur.
  - Cons: Large GPU Memory consumption.

Update: read → compute → write in place
In-Memory Snapshot: read → write to another place

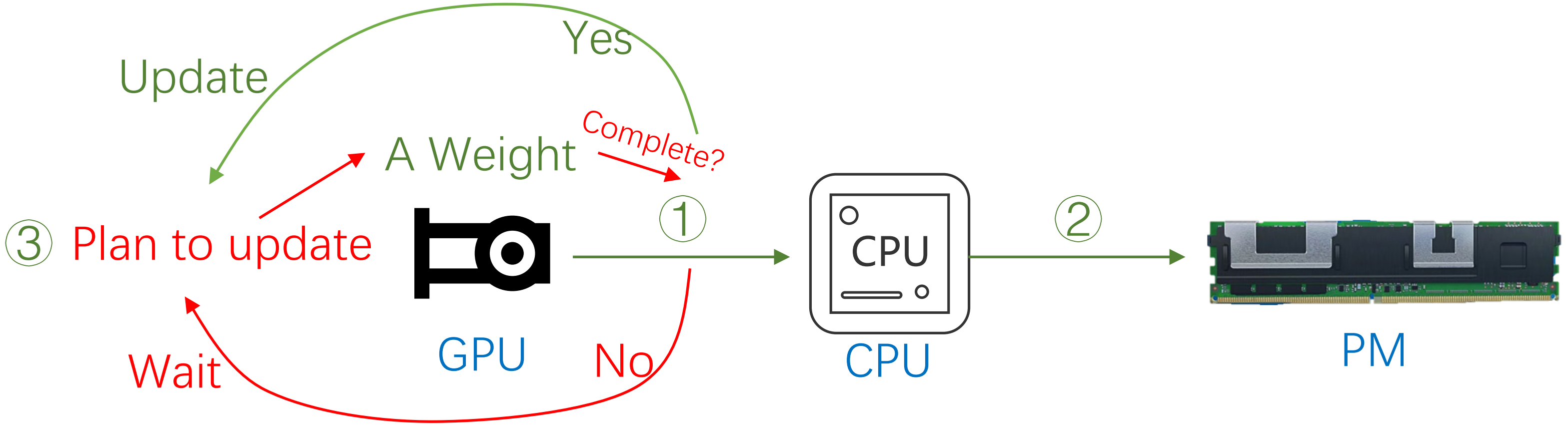


Handle Conflict

- Strategy 2:

①. Snapshot a weight from GPU to CPU (if in GPU);
②. Persist the weight;
③. If conflict occurs, wait the snapshot complete (needn't wait persist).

- Comments:
  - Pros: No additional GPU memory cost.
  - Cons: When GPU to CPU snapshot is slow, it will stall for a long time.



Handle Conflict

- Strategy 3:

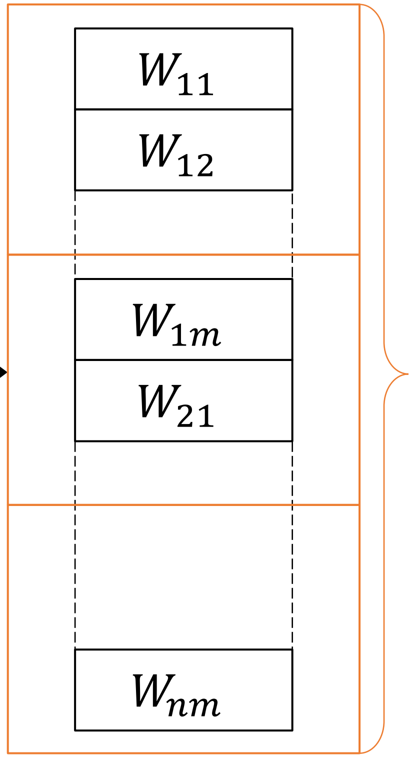
Divide a weight into different parts.
①. Snapshot parts of the weight from GPU to CPU (if in GPU) concurrently;
②. Persist the parts of the weight concurrently;
③. If conflict occurs, in-memory snapshot the remaining incomplete GPU to CPU snapshot parts of weight.

- Comments:
  - Pros: Compared to strategy 1 and 2, less GPU memory cost, less stall.
  - Cons: When GPU to CPU snapshot is close to complete, additional in-memory snapshot will be costly.

Handle Conflict

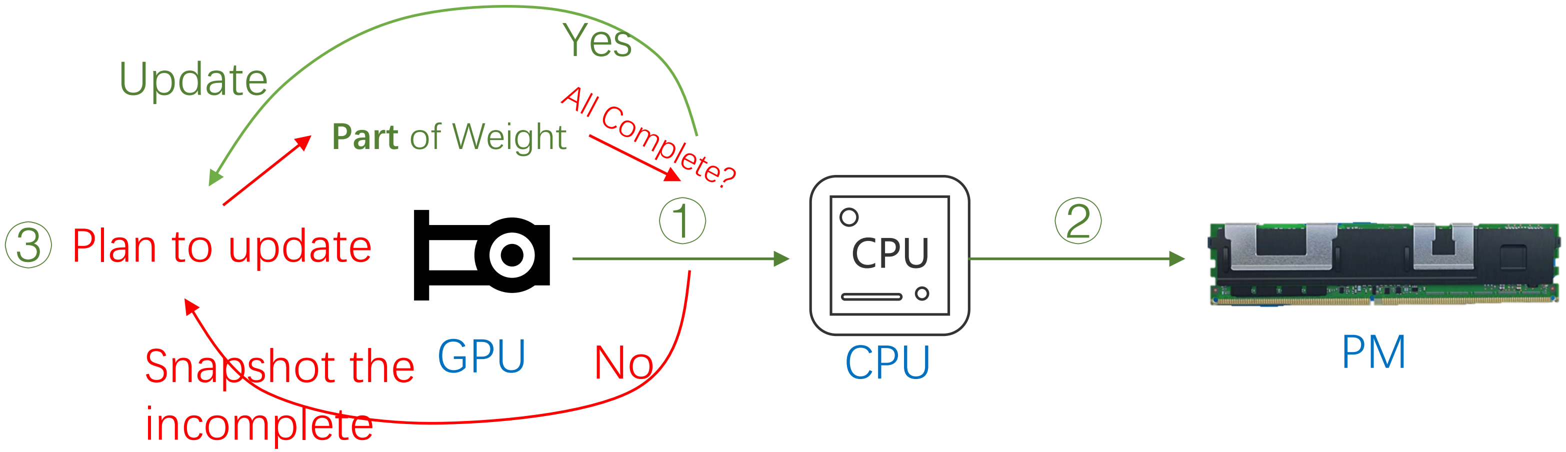
$$\begin{bmatrix} W_{11} & W_{12} & \cdots & W_{1m} \\ W_{21} & W_{22} & \cdots & W_{2m} \\ \vdots & \vdots & \ddots & \vdots \\ W_{n1} & W_{n2} & \cdots & W_{nm} \end{bmatrix}$$

Logic View of a Weight



Divided a weight into different parts

Storage View of a Weight



Adaptive Strategy Determination

- For weights that **can** be snapshotted without conflict:
  - Strategy 2
- For weights that **can’t** be snapshotted without conflict
  - Enough GPU memory: Strategy 1
  - Not Enough GPU memory: Strategy 3
- Adjust strategy for each weight according to

<u>last checkpoint iteration profile of snapshot</u> and <u>current GPU memory</u>.

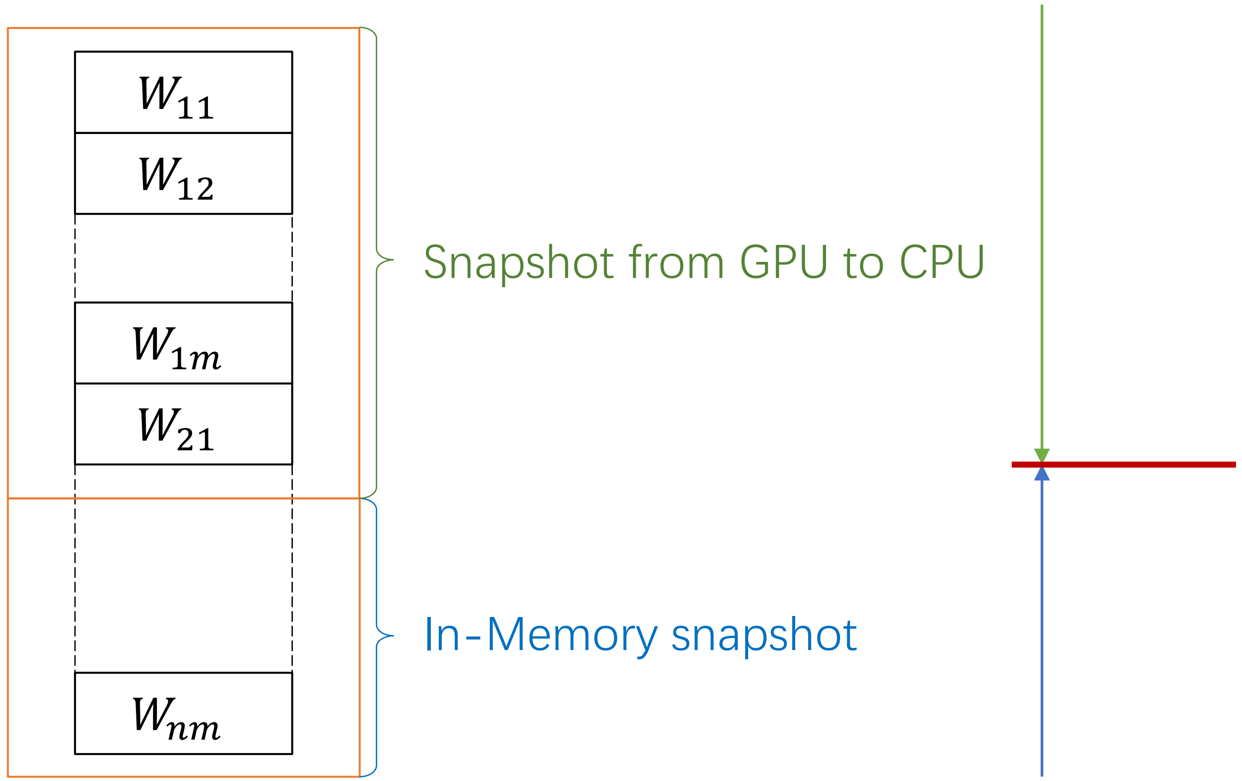
Handle Conflict

- Strategy 4:

Divide a weight into two groups: in-memory snapshot group(1) & gpu to cpu snapshot group(2)
①. Snapshot group (1) from GPU to CPU (if in GPU) concurrently
while Snapshot group (2) in-memory concurrently;
②. Persist the parts of the weight concurrently;
③. If conflict occurs, wait ① to complete.

- Comments:
  - Pros: If (1) and (2) are grouped properly, there will be no stall, less memory cost
  - Cons: Weights in two groups must be carefully selected, an adaptive group determination algorithm may be designed.

Handle Conflict



Storage View of a Weight

These two snapshots run in parallel.

Ideally, weight update in the next iteration will occur when two processes meet at this point.

It will be no stall and minimum memory consumption of in-memory snapshot.

Maintain Consistency

- Write two checkpoint files in turn, ensuring that there is at least one consistent checkpoint.
- Ensure that there is only one checkpoint process in parallel with DNN training process.

More Checkpoints

- Ensure that there is always a checkpoint process in each DNN training iteration.
- That is, doing another checkpoint in the next iteration once completing the checkpoint in the current iteration.