IPU HANDS-ON SESSION

Reinforcement Learning Neural network visualization from POPLAR™

Day03
IPU optimization

BERT LARGE – POPLAR GRAPH VISUALIZATION FROM POPART

OVERVIEW

GOAL

Start with a configuration where Out Of Memory error occurs

Step-by-step optimization

Successfully run the same model with same batch size with high performance

- File Structure
  - main.py : main training file
  - utils.py : contains functions/classes used in main
  - requirements.txt : required packages (pip install –r requirements.txt)

- ResNet101 pipelined over 2 IPUs

```
model = torchvision.models.resnet101()
model.layer3[8] = poptorch.BeginBlock(model.layer3[8], ipu_id=1)
```

- CIFAR10 dataset

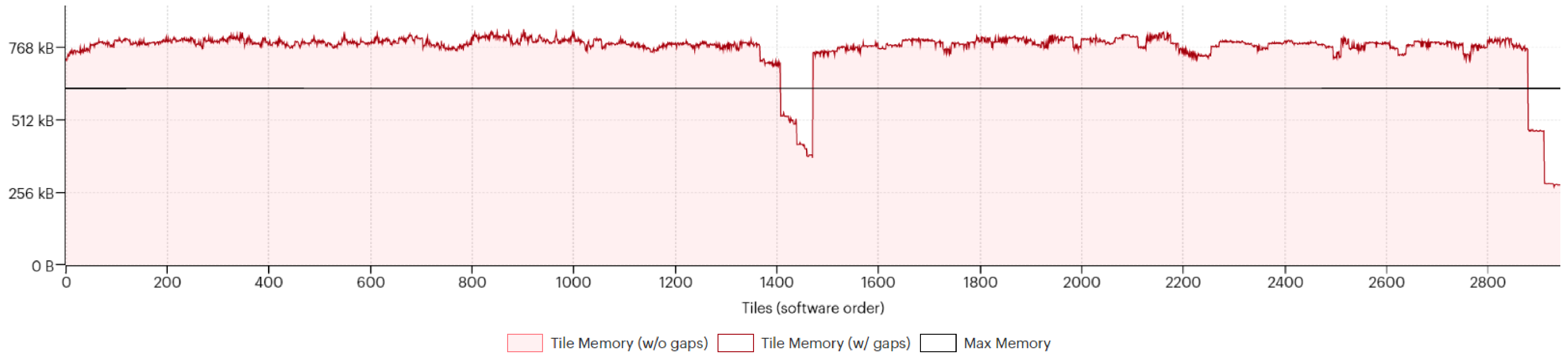
```
train_dataset = torchvision.datasets.CIFAR10(root='../datasets', \
    train=True, download=True, transform=transform)
```

BASE

- Configs where OOM occurs

```
python main.py --batch-size 64 --precision 32.32 --gradient-accumulation 3 --epochs 1
```

```
Namespace(async_dataloader=False, batch_size=64, device_iterations=1,
          eight_bit_io=False, epochs=1, gradient_accumulation=3, precision='32.32', replicas=1)
```



HALF PRECISION

- Graph analyzer provides some tips for reducing memory usage

Summary
Insights
Memory Report
Liveness Report
Program Tree

Reducing memory usage

You can reduce the memory requirements of your application in the following ways. More information on these can be found in the Graphcore Memory and Performance Optimisation Guide

These solutions may affect the performance, throughput, convergence and/or training characteristics of your model

Reducing the batch size | Using FP16 | Using recomputation | Using recomputation checkpoints | Setting the available memory proportion

Offloading variables | Reusing identical parts of your graph | Disable profiling on the IPU | + Using different optimizers

Top priority is to use FP16!

Using FP16 where appropriate

What is FP16?

Using FP16 instead of FP32 will reduce the memory required for a variable by half and can double the execution speed of operations that use them. For a model you can choose if you want to use FP16 for partials, compute and weights. FP16 may not be suitable in all cases due to a reduced numerical range. Techniques to mitigate this are described here.

How changing from FP32 to FP16 can fix OOM

Reducing the **partials** format also improves the performance (See Table 5.1 in the AI-FloatTM white paper). So, each time the numerical precision is halved, the execution speed of matmul and convolution operations is doubled.

Useful links

Memory and Performance Optimisation on the IPU

HALF PRECISION

Different precision formats during model execution

- Single precision : FP32 for both the data and the model
- Mixed precision : FP16 for the data and FP32 for the model
- Half precision : FP16 for both the data and the model

2 ways to set half precision

- model = model.to(torch.float16)
- model = model.half()

```
input_dtype = torch.float16 if args.precision[:2] == '16' else torch.float32
model_dtype = torch.float16 if args.precision[-2:] == '16' else torch.float32

if args.eight_bit_io:
    model = ModelWithNormalization(model, dtype=input_dtype)
model = ModelwithLoss(model, criterion).to(model_dtype)
```

```
class ModelWithNormalization(torch.nn.Module):
    def __init__(self, model: torch.nn.Module, dtype: torch.dtype):
        super().__init__()
        self.model = model
        self.dtype = dtype
        self.normalize = transforms.Normalize(**IMAGENET_STATISTICS)

    def forward(self, img):
        return self.model(self.normalize(img.to(self.dtype)))
```

HALF PRECISION

Poplar SDK contains many tools to support stable training with half precision, so it's highly recommended to use half precision on the IPU

```python
class AdamW(Optimizer, torch.optim.AdamW):
    """ Adam optimizer with true weight decay.

    This optimizer matches PyTorch's implementation
    (`torch.optim.AdamW <https://pytorch.org/docs/1.10.0/optim.htmltorch.optim.AdamW>`_)
    with optional loss scaling.

    AMSGrad is currently not supported."""

    # Variables which don't exist in the parent optimizer class and are
    # global (cannot be set per group).
    _child_vars = ["loss_scaling"]
    # All the attributes and variables which don't exist in the parent optimizer class.
    _child_only = _child_vars + [
        "bias_correction",
        "accum_type",
        "first_order_momentum_accum_type",
        "second_order_momentum_accum_type",
        "max_grad_norm",
    ]
```

- Poptorch optimizers are implemented by inheriting the original torch.optim
- Provides more flexible control over data types during execution

```python
if args.precision[-3:] == '.16':
    optimizer = poptorch.optim.AdamW(model.parameters(), lr=0.001,
                                     loss_scaling=1000,
                                     accum_type=torch.float16,
                                     first_order_momentum_accum_type=torch.float16,
                                     second_order_momentum_accum_type=torch.float32)
```

- Factor by which to scale the loss
- Automatic Loss Scaling is supported for SDK>=2.5

HALF PRECISION

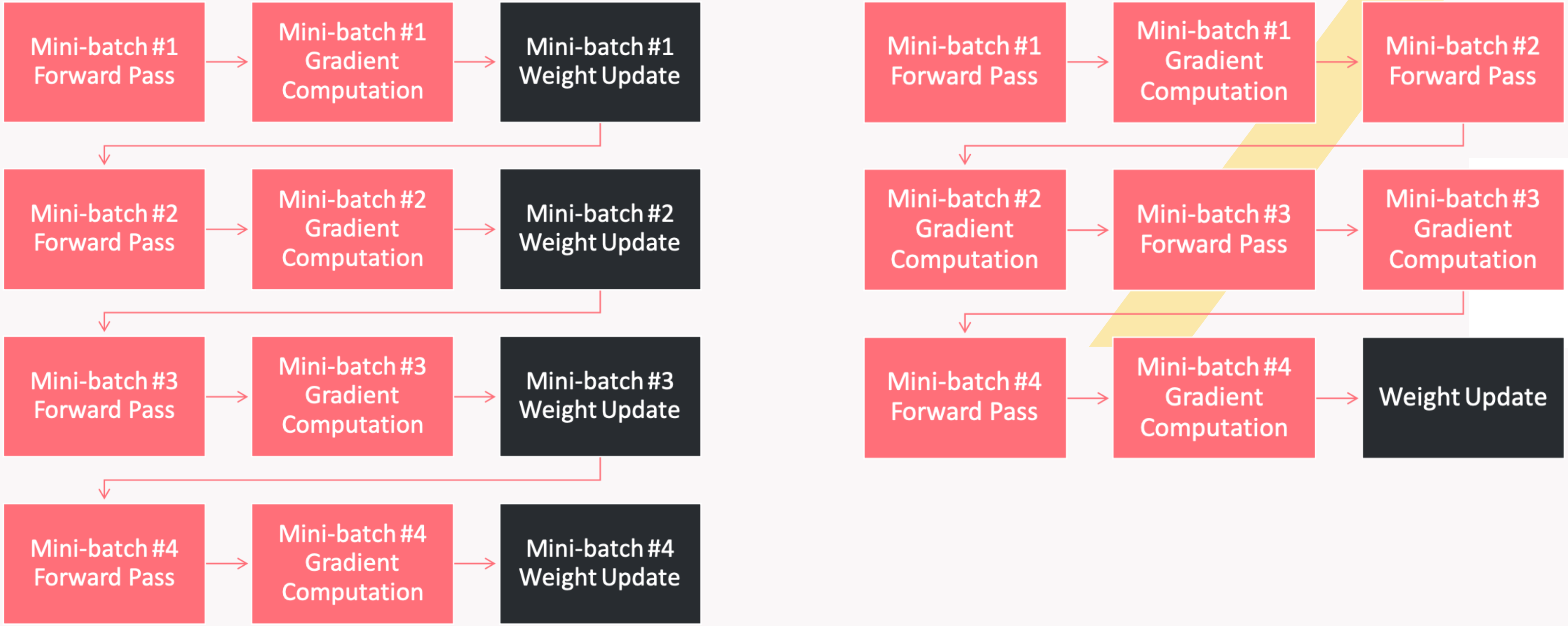
- Using half precision

```
python main.py --batch-size 64 --precision 16.16 --gradient-accumulation 3
```

```
Namespace(async_dataloader=False, batch_size=64, device_iterations=1,
          eight_bit_io=False, epochs=5, gradient_accumulation=3, precision='16.16', replicas=1)
```

GRADIENT ACCUMULATION

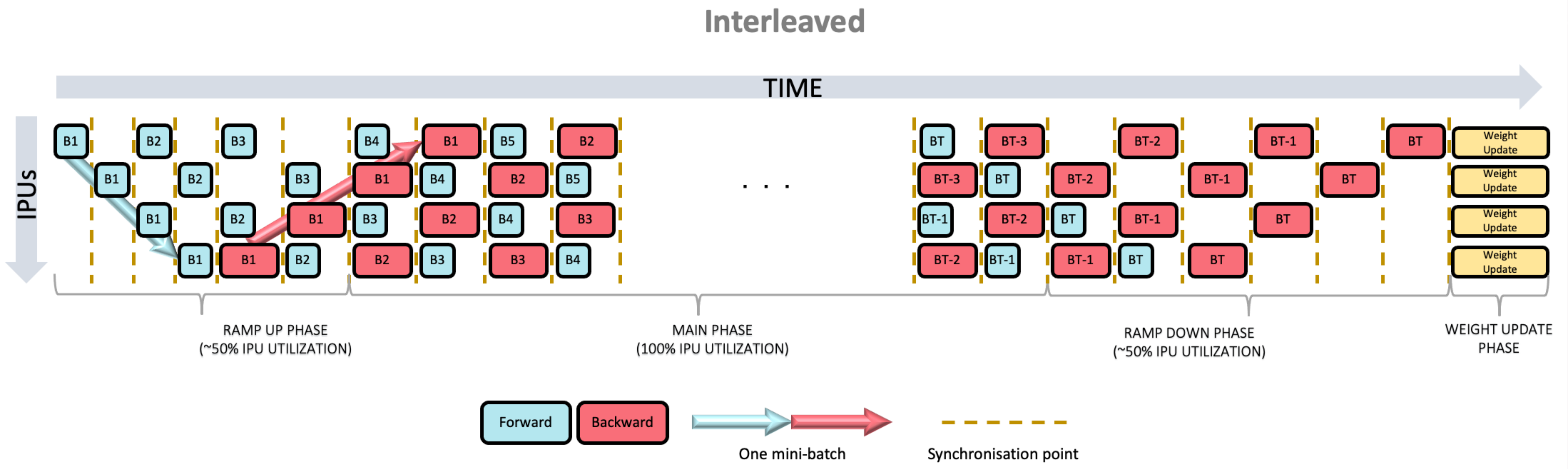
- More samples per a weight update without increase in memory usage
- Speed up due to less weight updates



https://docs.graphcore.ai/projects/tf-model-parallelism/en/latest/pipelining.html#increase-the-gradient-accumulation-count

GRADIENT ACCUMULATION

- Maximize the portion of main phase → higher throughput



https://docs.graphcore.ai/projects/tf-model-parallelism/en/latest/pipelining.html#pipeline-scheduling

GRADIENT ACCUMULATION

- How to set

```
opts = poptorch.Options()
opts.Training.gradientAccumulation(args.gradient_accumulation)
```

- Command using higher gradient accumulation factor

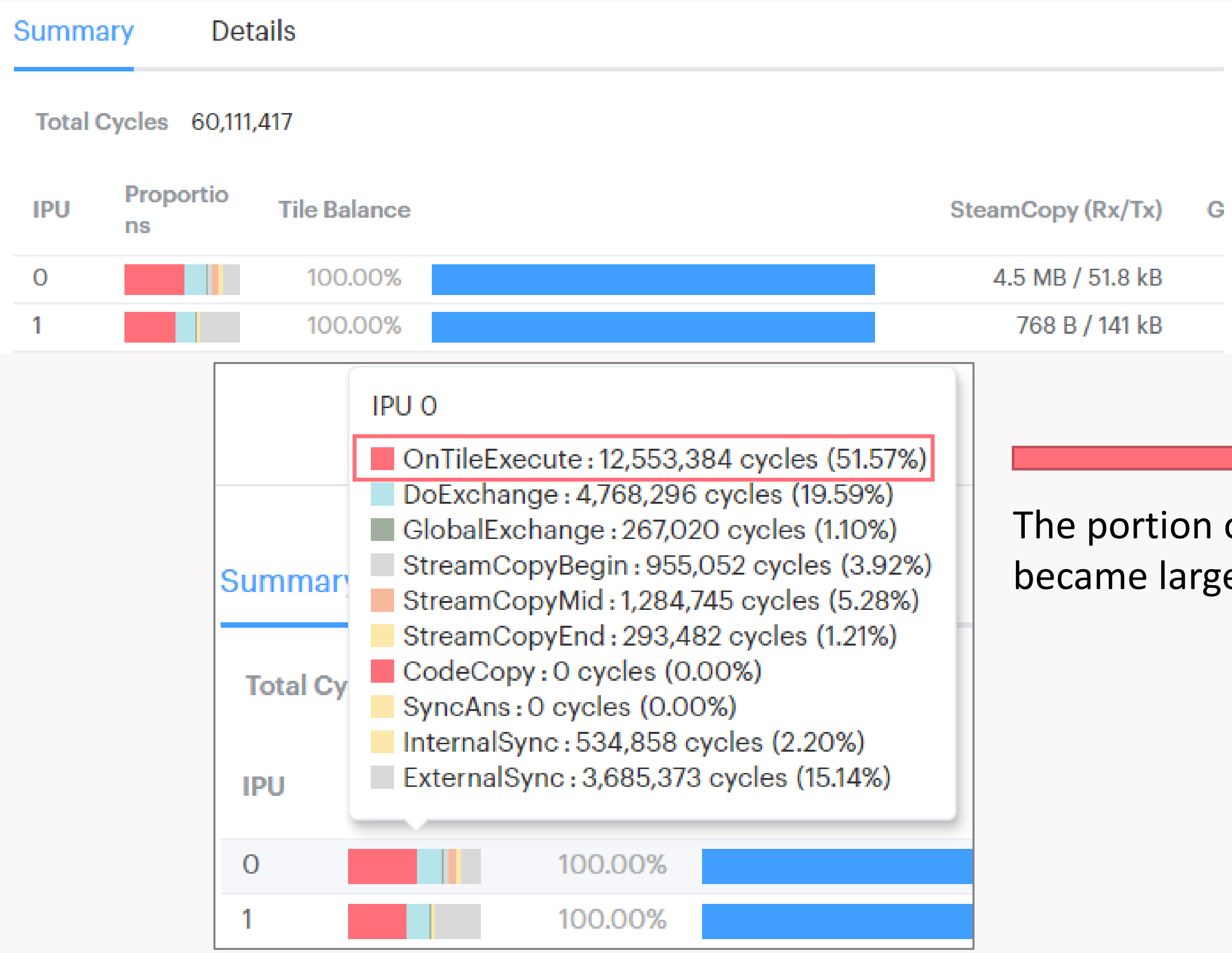
```
python main.py --batch-size 64 --precision 16.16 --gradient-accumulation 8
```

- Output

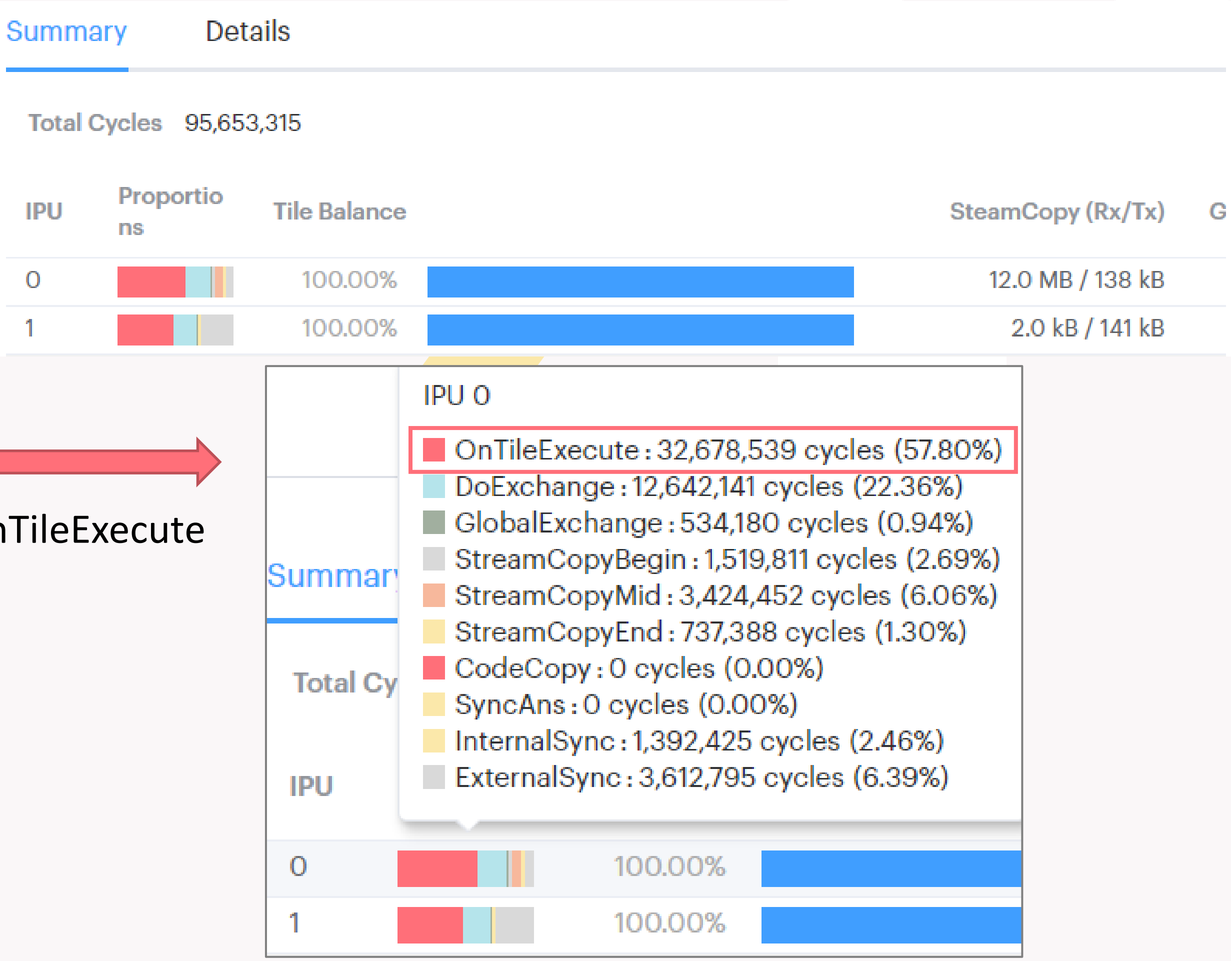
```
[Epoch 01]: 100%|          | 97/97 [09:20<00:00,  5.78s/it, loss=1.56, acc=39.1]
[Epoch 01] loss: 1.900, acc: 33.7
[Epoch 02]: 100%|          | 97/97 [00:46<00:00,  2.10it/s, loss=1.01, acc=67.2]
[Epoch 02] loss: 1.375, acc: 50.0
[Epoch 03]: 100%|          | 97/97 [00:46<00:00,  2.10it/s, loss=1.31, acc=54.7]
[Epoch 03] loss: 1.177, acc: 58.0
[Epoch 04]: 100%|          | 97/97 [00:46<00:00,  2.11it/s, loss=0.935, acc=67.2]
[Epoch 04] loss: 1.056, acc: 61.9
[Epoch 05]: 100%|          | 97/97 [00:46<00:00,  2.09it/s, loss=0.905, acc=64.1]
[Epoch 05] loss: 0.921, acc: 67.3
```

GRADIENT ACCUMULATION

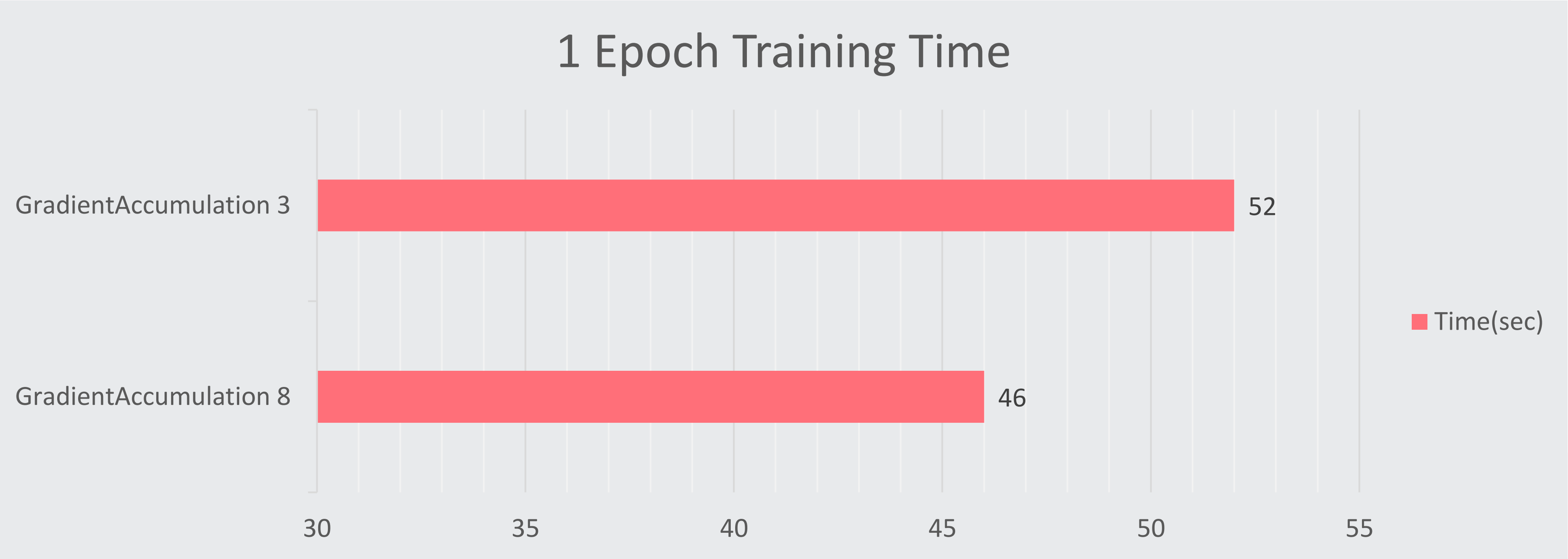
Execution Trace



The portion of OnTileExecute became larger

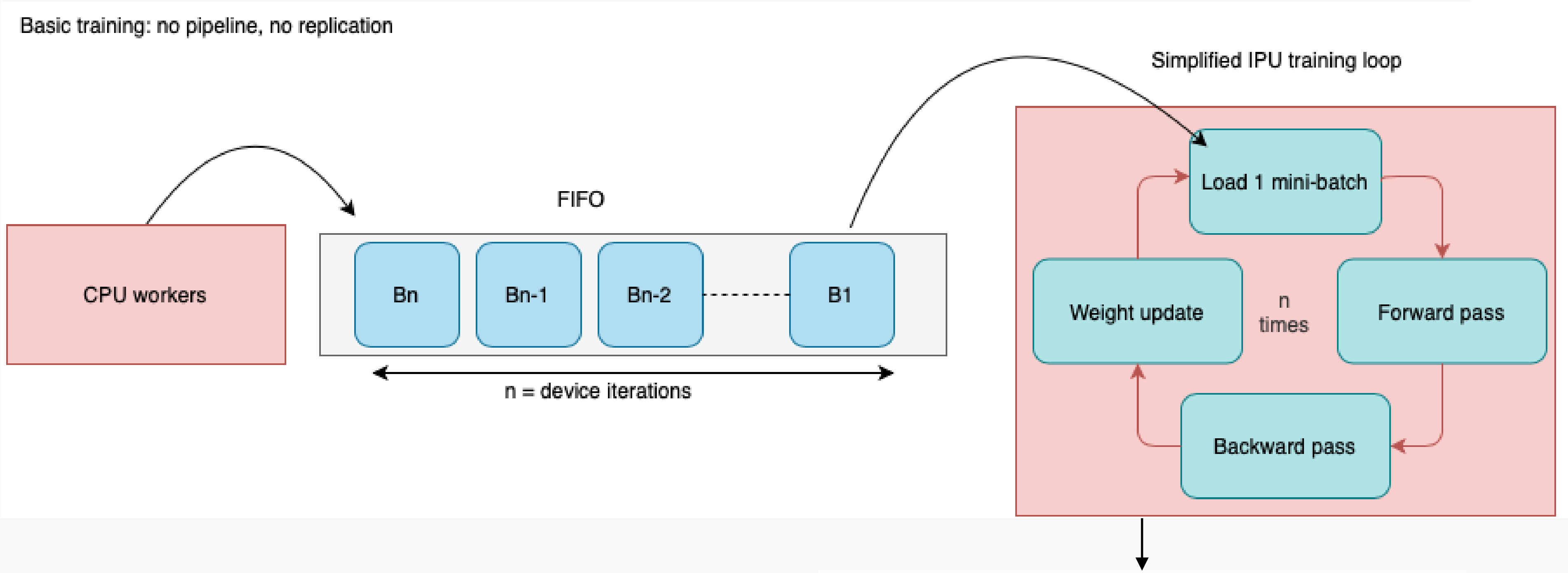


GRADIENT ACCUMULATION



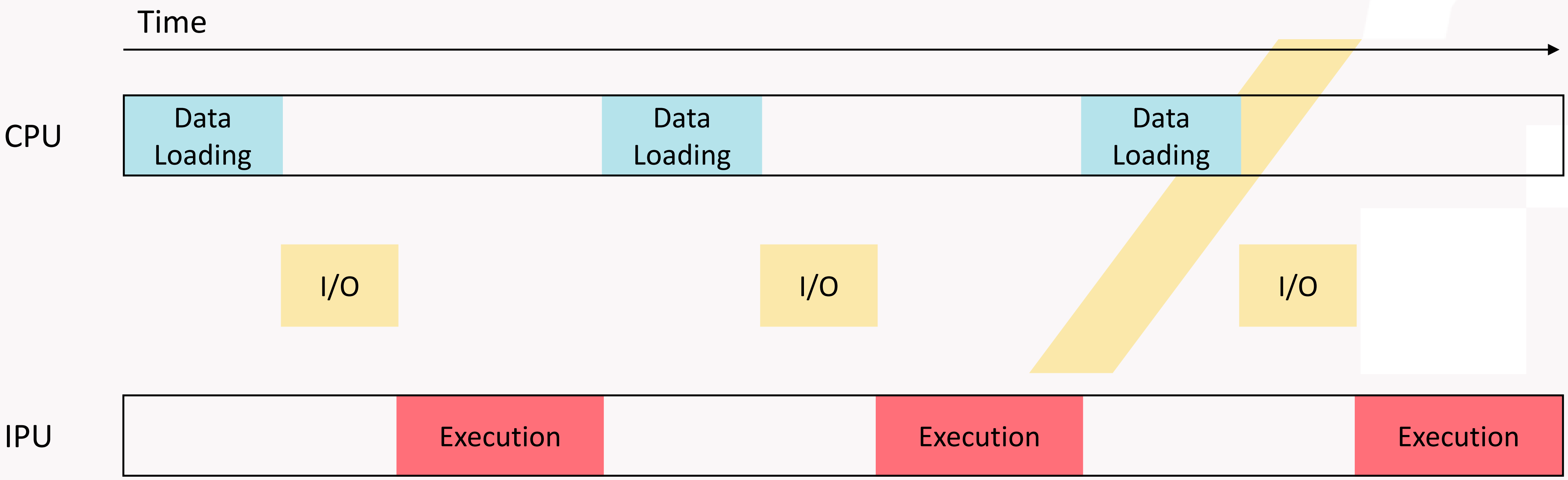
DEVICE ITERATION

- Iterate training loops over multiples batches from an input queue
- Accelerate training by reducing the number of host-device communications

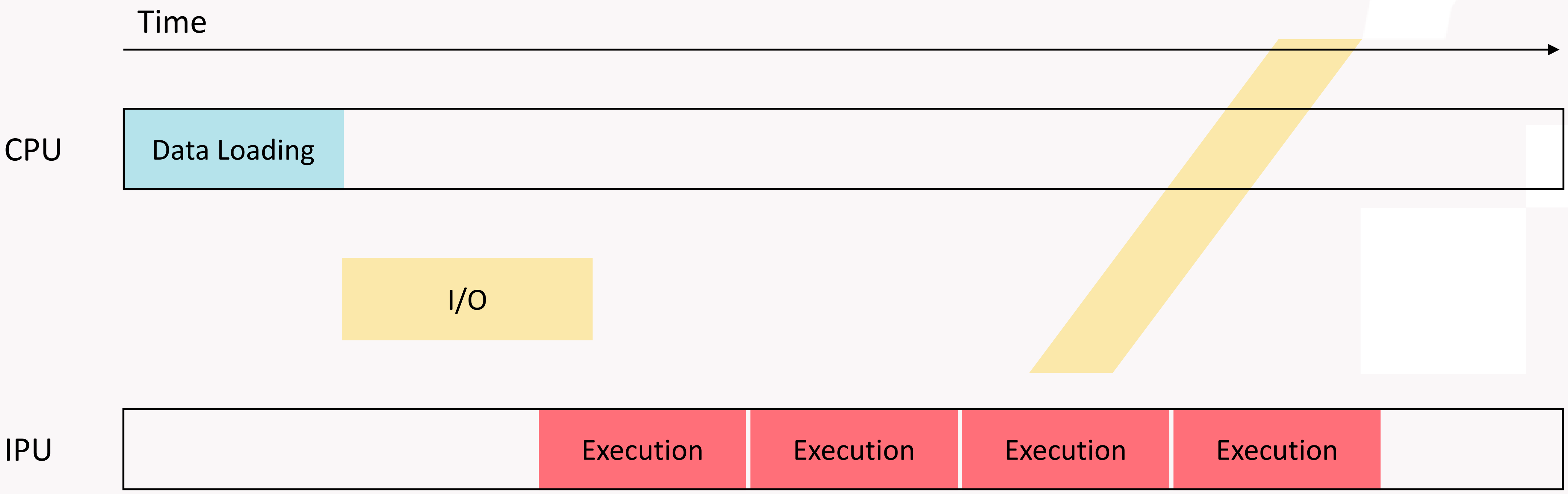


- **output_mode** (*poptorch.OutputMode*) –
  - `All`: Return a result for each batch.
  - `Sum`: Return the sum of all the batches.
  - `Final`: Return the last batch.
  - `EveryN`: Return every N batches: N is passed in as `output_return_period`.
  - Default: `All` for inference, `Final` for training.

DEVICE ITERATION



DEVICE ITERATION



DEVICE ITERATION

- How to set

```
opts = poptorch.Options()
opts.deviceIterations(args.device_iterations)
```

- Command using device iteration

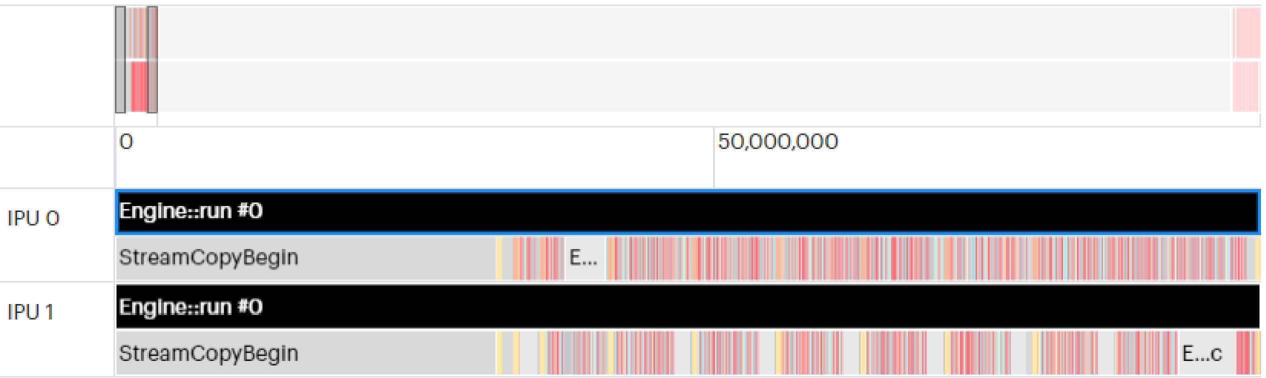
```
python main.py --batch-size 64 --precision 16.16 --gradient-accumulation 8 --device-iteration 4
```

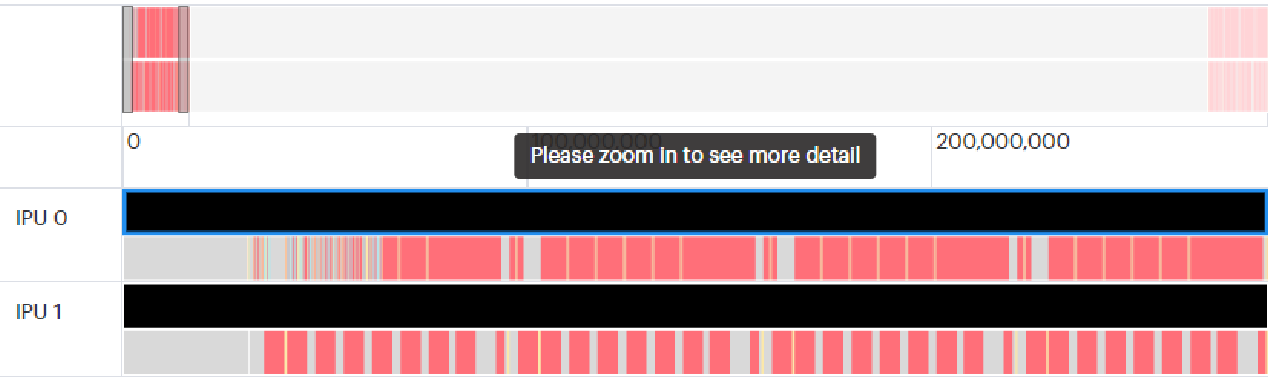
- Output

```
[Epoch 01]: 100%|██████████| 24/24 [09:11<00:00, 22.97s/it, loss=1.51, acc=45.3]
[Epoch 01] loss: 1.759, acc: 36.5
[Epoch 02]: 100%|██████████| 24/24 [00:45<00:00,  1.88s/it, loss=1.11, acc=57.8]
[Epoch 02] loss: 1.395, acc: 48.3
[Epoch 03]: 100%|██████████| 24/24 [00:45<00:00,  1.88s/it, loss=0.841, acc=75]
[Epoch 03] loss: 1.251, acc: 55.1
[Epoch 04]: 100%|██████████| 24/24 [00:45<00:00,  1.88s/it, loss=1.37, acc=54.7]
[Epoch 04] loss: 1.086, acc: 61.8
[Epoch 05]: 100%|██████████| 24/24 [00:45<00:00,  1.88s/it, loss=0.744, acc=73.4]
[Epoch 05] loss: 0.863, acc: 68.9
```

DEVICE ITERATION

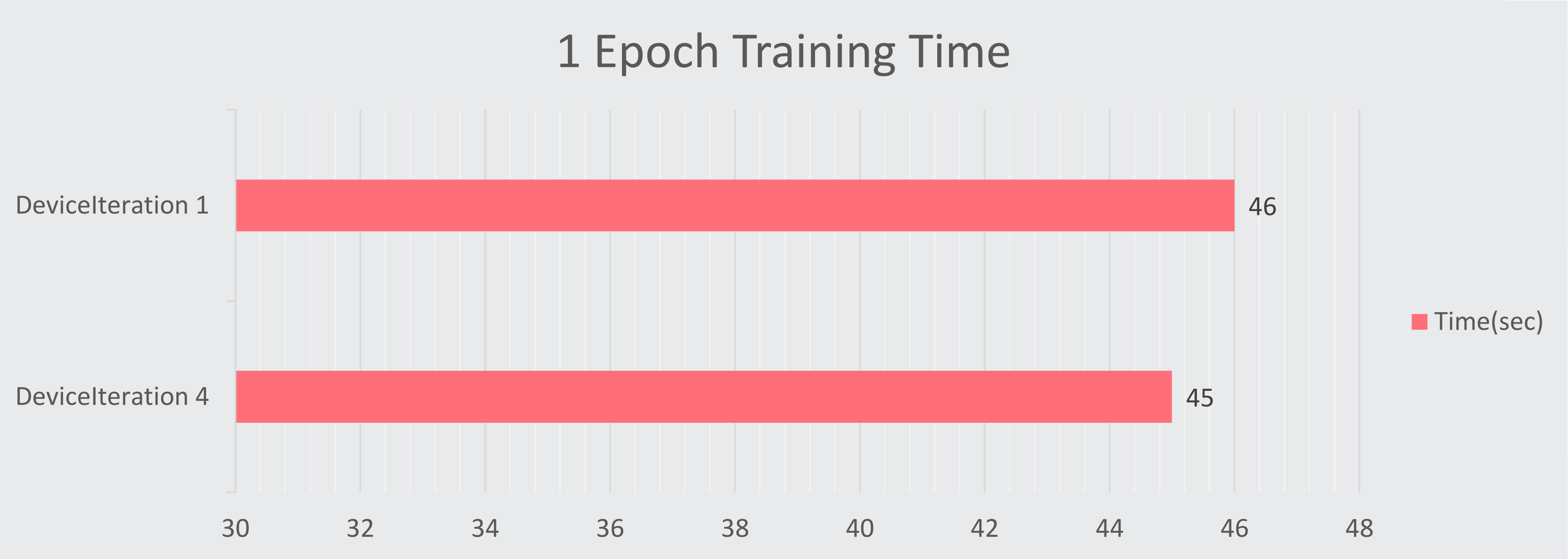
Execution Trace





Iterating 4 training loops on device

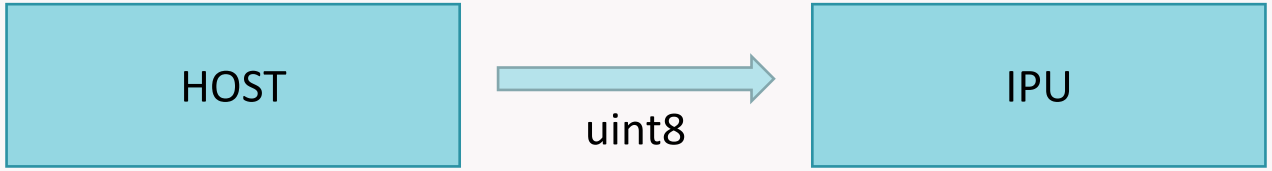
DEVICE ITERATION



Host overhead might be occurred while processing the larger data

8-BIT I/O

- Casting the input data to uint8 can reduce I/O overhead



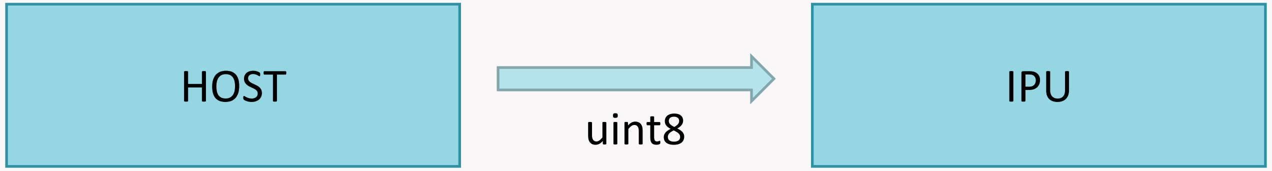
HOST side

```python
def get_transform(precision='16.16', eight_bit_io=False):
    preprocssing_steps = [
        transforms.RandomHorizontalFlip(),
        transforms.Resize((64, 64)),
        transforms.PILToTensor()
    ]

    if not eight_bit_io:
        dtype = torch.float16 if precision[:2] == '16' else torch.float32
        preprocssing_steps.extend([
            transforms.ConvertImageDtype(dtype),
            transforms.Normalize(**IMAGENET_STATISTICS)
        ])

    return transforms.Compose(preprocssing_steps)
```

In usual case (not using 8-bit I/O),
type casting and normalization conducted on CPU

8-BIT I/O

- Casting the input data to uint8 can reduce I/O overhead



IPU side

```python
if args.eight_bit_io:
    model = ModelWithNormalization(model, dtype=input_dtype)
```

```python
class ModelWithNormalization(torch.nn.Module):
    def __init__(self, model: torch.nn.Module, dtype: torch.dtype):
        super().__init__()
        self.model = model
        self.dtype = dtype
        self.normalize = transforms.Normalize(**IMAGENET_STATISTICS)

    def forward(self, img):
        # One-liner in Poplar SDK >= 2.6:
        # return self.model(self.normalize(img.to(self.dtype)))
        if self.dtype == torch.float32:
            img = img.float()
        elif self.dtype == torch.float16:
            img = img.half()
        else:
            raise Exception(f'Normalization dtype must be one of torch.float16 or torch.float32,'
                            f' but {self.dtype} is given')
        return self.model(self.normalize(img))
```

When using 8-bit I/O,
type casting and normalization conducted on IPU

8-BIT I/O

- Command using 8-bit I/O

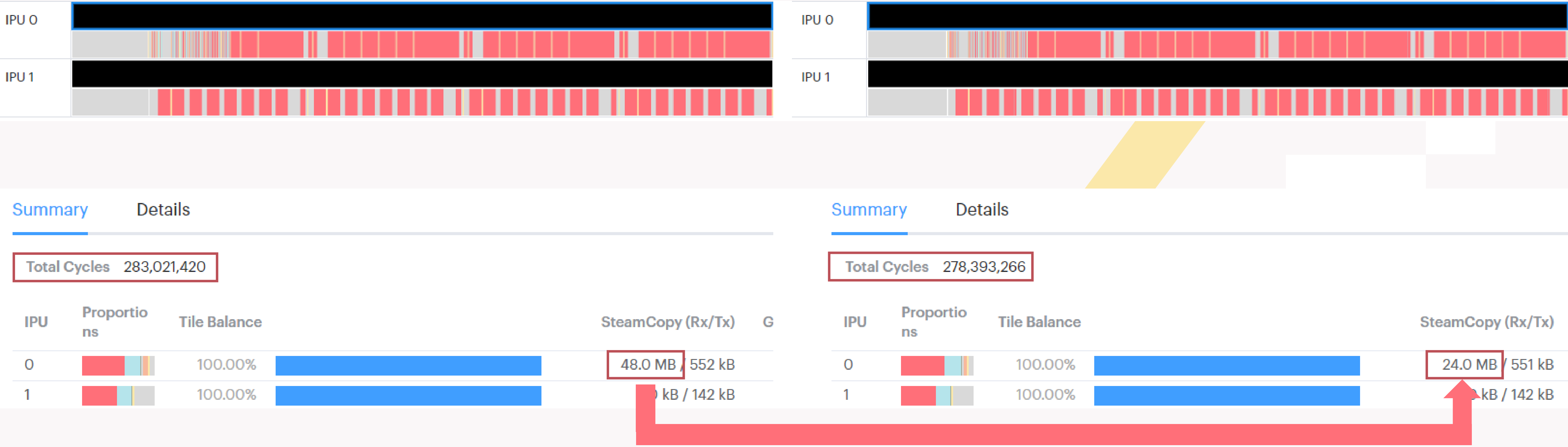
```
python main.py --batch-size 64 --precision 16.16 --gradient-accumulation 8 \
               --device-iteration 4 --eight-bit-io
```

- Output

```
[Epoch 01]: 100%|██████████| 24/24 [08:35<00:00, 21.48s/it, loss=1.65, acc=34.4]
[Epoch 01] loss: 1.848, acc: 32.6
[Epoch 02]: 100%|██████████| 24/24 [00:13<00:00,  1.77it/s, loss=1.33, acc=51.6]
[Epoch 02] loss: 1.334, acc: 51.8
[Epoch 03]: 100%|██████████| 24/24 [00:13<00:00,  1.77it/s, loss=1.44, acc=50]
[Epoch 03] loss: 1.205, acc: 56.3
[Epoch 04]: 100%|██████████| 24/24 [00:13<00:00,  1.77it/s, loss=0.639, acc=81.2]
[Epoch 04] loss: 1.004, acc: 63.4
[Epoch 05]: 100%|██████████| 24/24 [00:13<00:00,  1.77it/s, loss=0.903, acc=71.9]
[Epoch 05] loss: 0.926, acc: 67.2
```

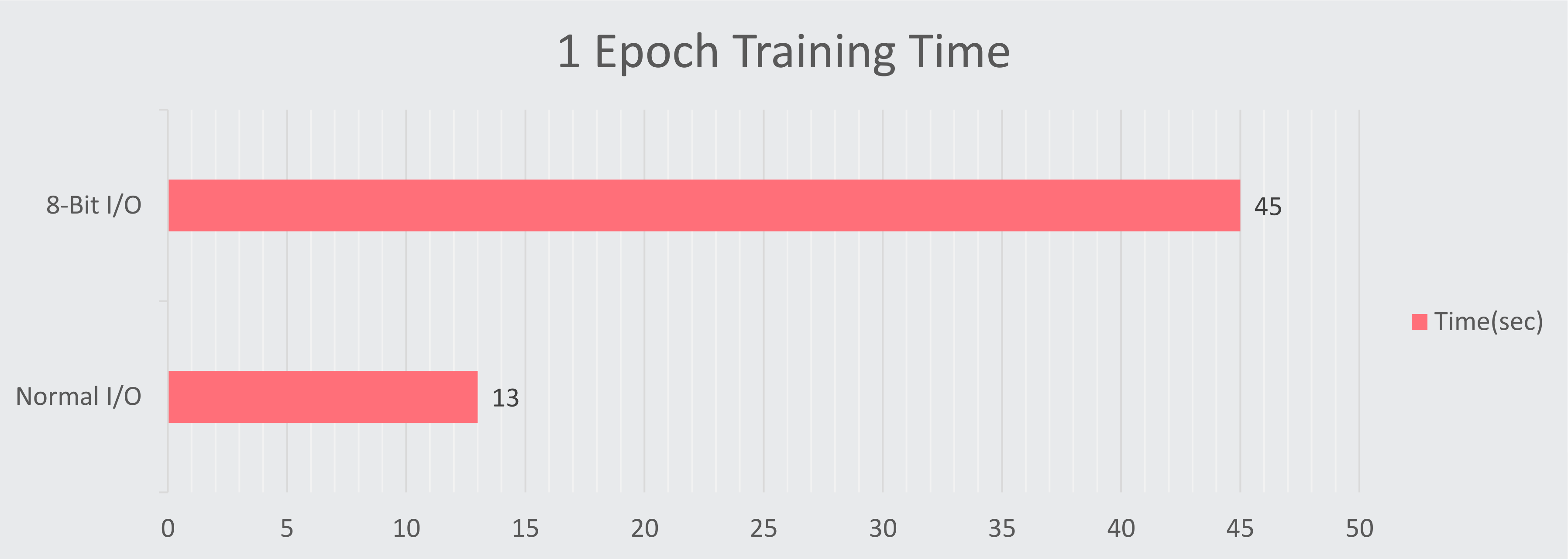
8-BIT I/O

Execution Trace



Size of the received data reduced to half

8-BIT I/O



1 Epoch Training Time
8-Bit I/O
Normal I/O
45
13
0
5
10
15
20
25
30
35
40
45
50
Time(sec)

ASYNCHRONOUS DATALOADER

Host starts loading the data asynchronously to the IPU

↓

Save the preprocessed data in buffer

↓

Transfer the data in the buffer immediately when IPU is requesting for the next batch

↓

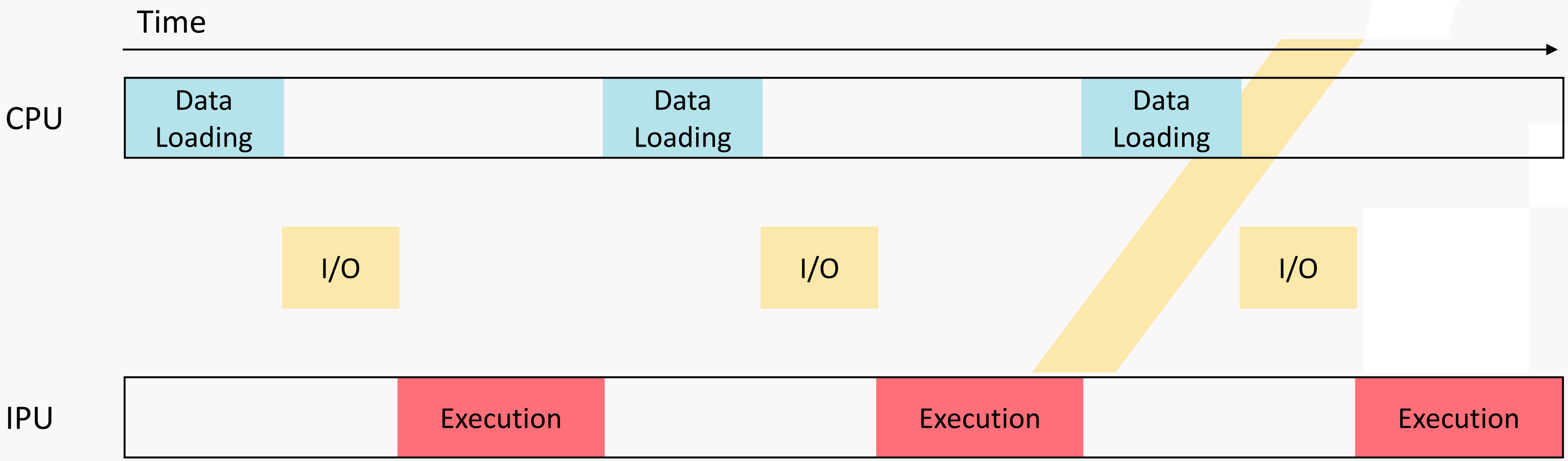
Minimize the host overhead

- How to set

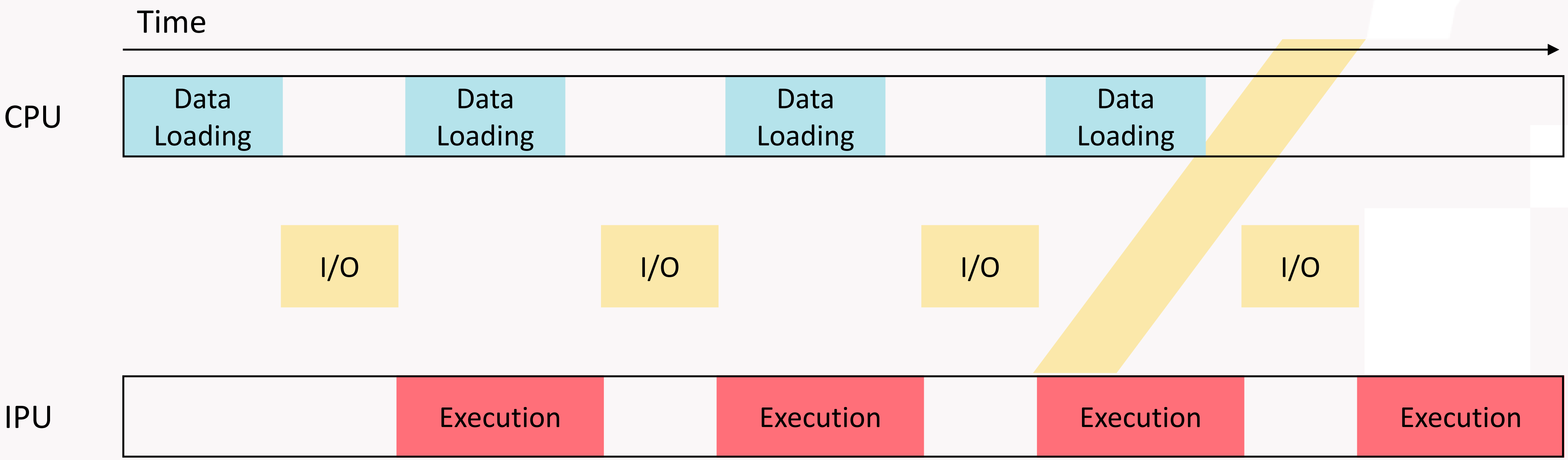
```python
mode = poptorch.DataLoaderMode.Async if args.async_dataloader \
    else poptorch.DataLoaderMode.Sync

train_dataloader = poptorch.DataLoader(opts,
                                       train_dataset,
                                       batch_size=args.batch_size,
                                       shuffle=True,
                                       mode=mode)
```

https://docs.graphcore.ai/projects/poptorch-user-guide/en/latest/batching.html#poptorch-asynchronousdataaccessor

ASYNCHRONOUS DATALOADER



ASYNCHRONOUS DATALOADER



ASYNCHRONOUS DATALOADER

- Command using asynchronous dataloader

```
python main.py --batch-size 64 --precision 16.16 --gradient-accumulation 8 \
               --device-iteration 4 --eight-bit-io --async-dataloader
```

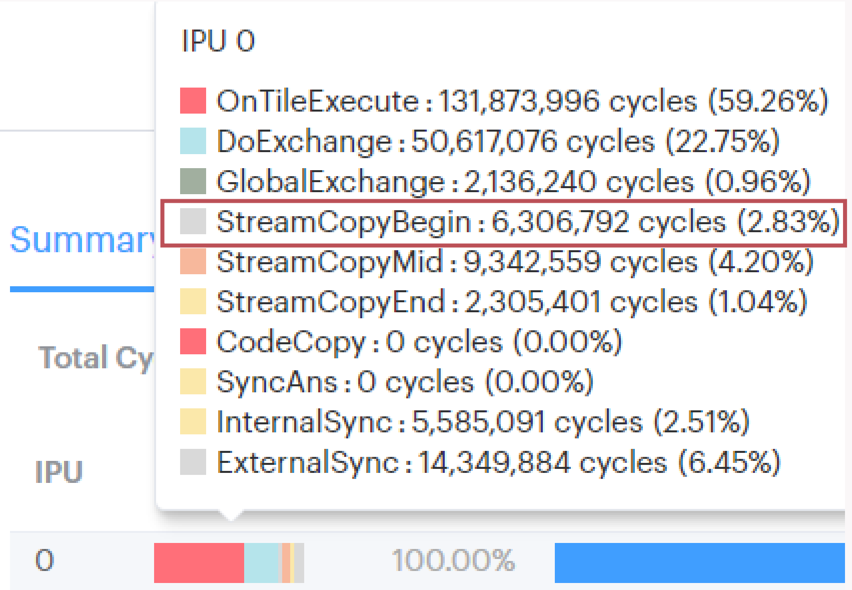
- Output

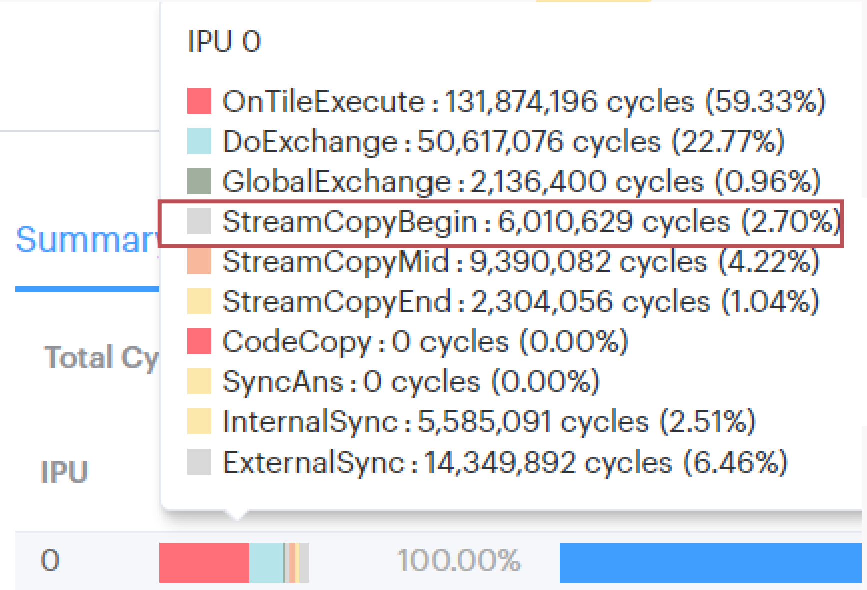
```
[Epoch 01]: 100%|██████████| 24/24 [08:32<00:00, 21.35s/it, loss=1.27, acc=50]
[Epoch 01] loss: 1.758, acc: 34.2
[Epoch 02]: 100%|██████████| 24/24 [00:08<00:00,  2.75it/s, loss=1.4, acc=50]
[Epoch 02] loss: 1.352, acc: 49.4
[Epoch 03]: 100%|██████████| 24/24 [00:08<00:00,  2.71it/s, loss=1.03, acc=65.6]
[Epoch 03] loss: 1.200, acc: 56.9
[Epoch 04]: 100%|██████████| 24/24 [00:08<00:00,  2.78it/s, loss=0.954, acc=65.6]
[Epoch 04] loss: 1.039, acc: 64.2
[Epoch 05]: 100%|██████████| 24/24 [00:08<00:00,  2.76it/s, loss=1.1, acc=62.5]
[Epoch 05] loss: 1.003, acc: 65.1
```

ASYNCHRONOUS DATALOADER

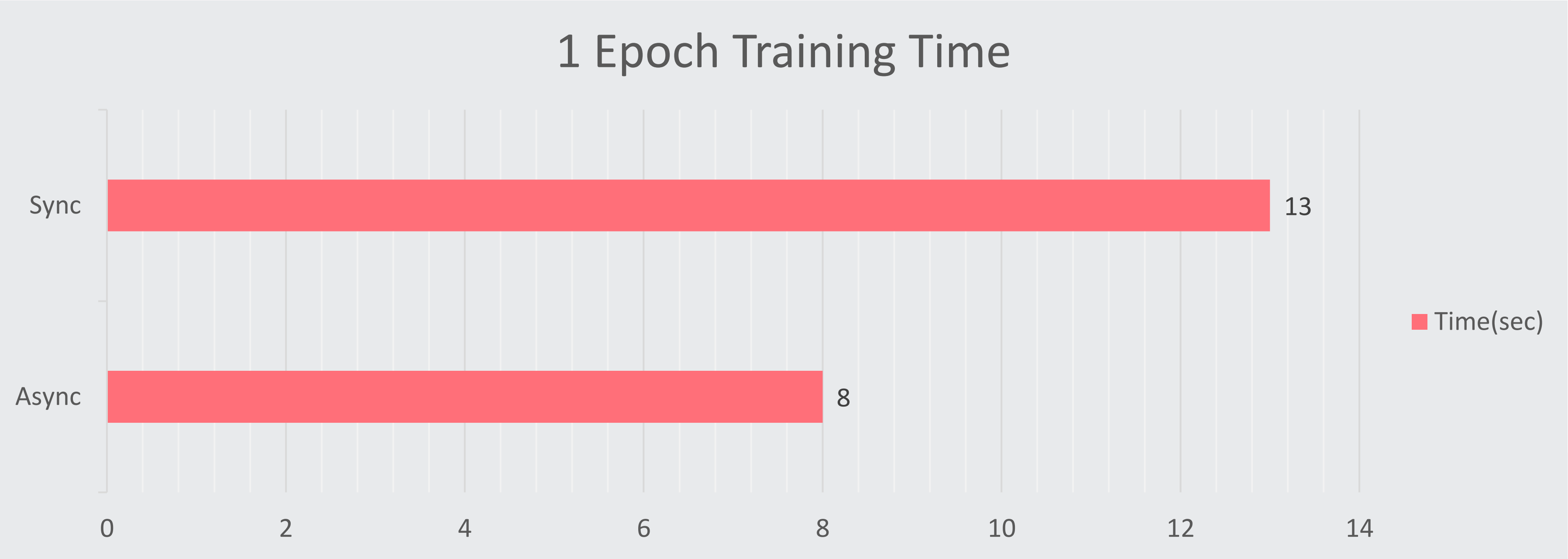
Execution Trace

Time spent for waiting CPU to prepare the data is reduced

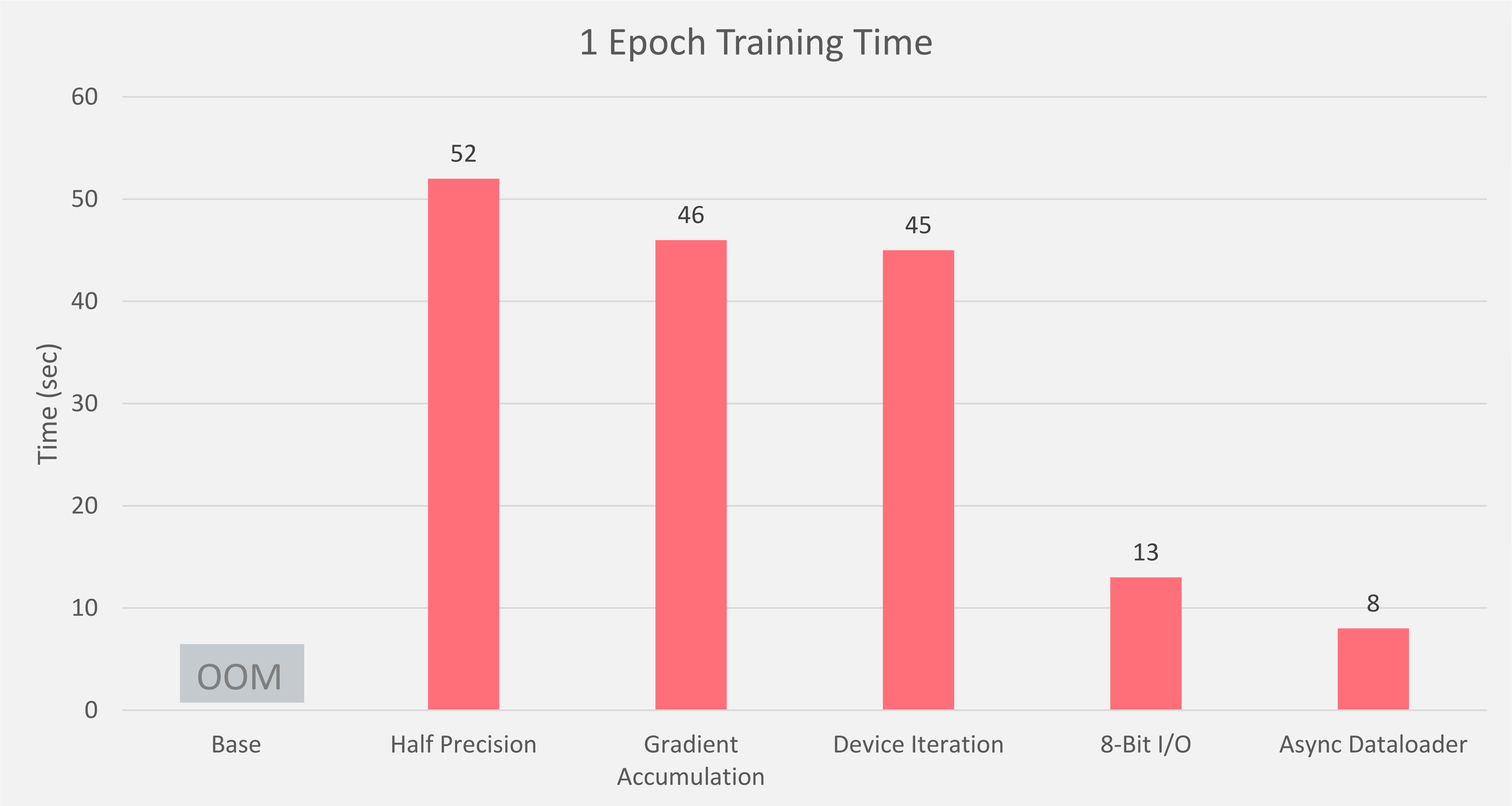




ASYNCHRONOUS DATALOADER



SUMMARY



Performance gain from each techniques might differ for different applications

THANK YOU