Refurbish Your Training Data: Reusing Partially Augmented Samples for Faster Deep Neural Network Training

Gyewon Lee[1,3], Irene Lee[2], Hyeonmin Ha[1],
Kyunggeun Lee[1], Hwarim Hyun[1], Ahnjae Shin[1,3], and Byung-Gon Chun[1,3]
Seoul National University[1], Georgia Institute of Technology[2], FriendliAI[3]

DNN Training Pipeline

DNN Training = Data Preparation + Gradient Computation

DNN Training Pipeline

DNN Training = Data Preparation + Gradient Computation

- Data read and preprocessing
- On CPU

DNN Training Pipeline

DNN Training = Data Preparation + Gradient Computation

Data Preparation:

- Data read and preprocessing
- On CPU

Gradient Computation:

- Forward and backward operations
- On DL accelerators (e.g., GPU, TPU)

DNN Training Pipeline

DNN Training = Data Preparation + Gradient Computation

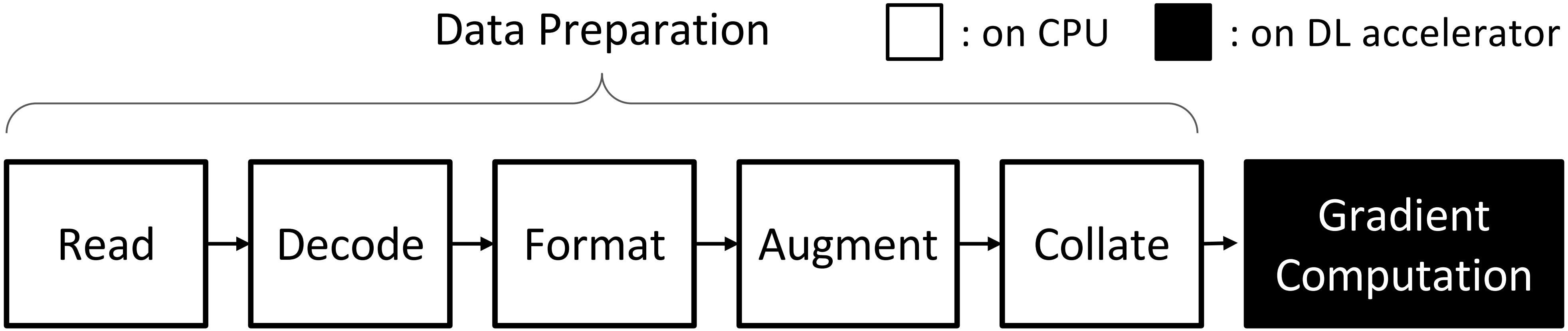
Data Preparation
- Data read and preprocessing
- On CPU

Bottleneck!

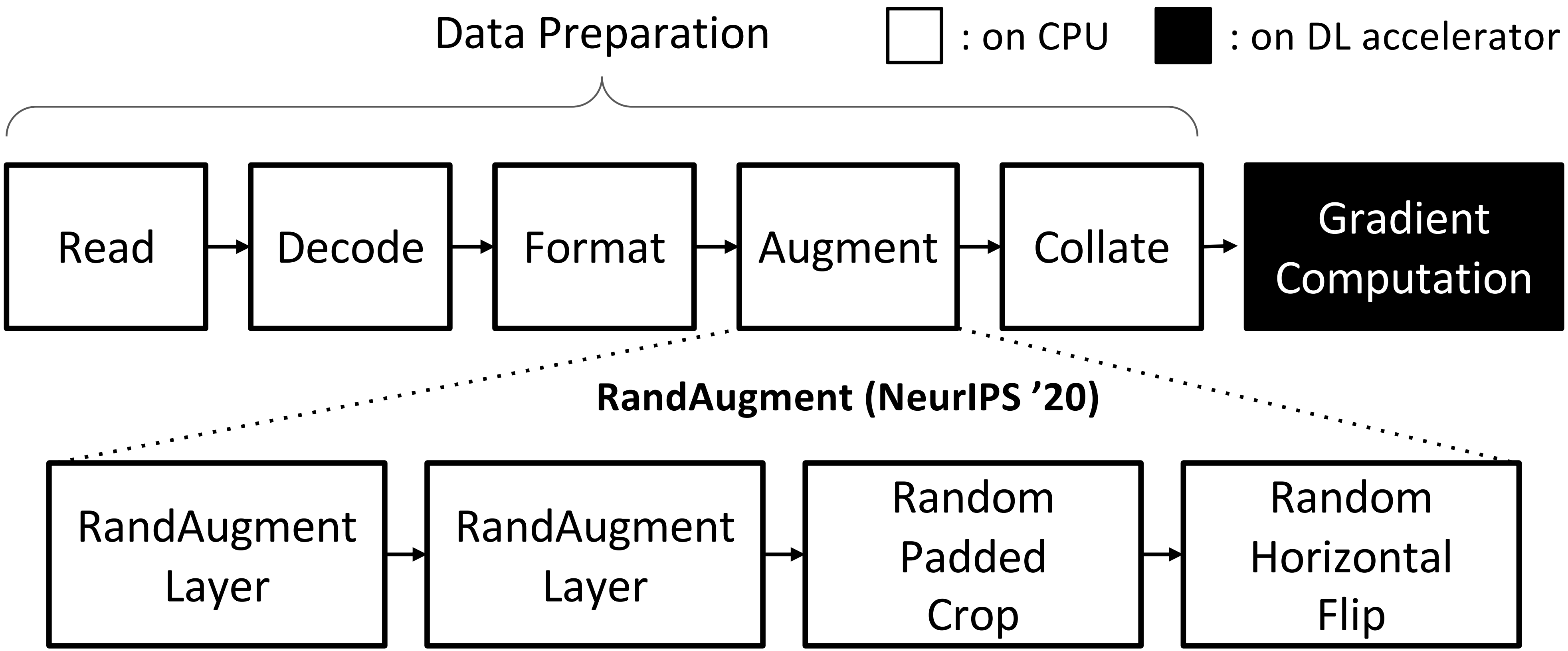
Gradient Computation
- Forward and backward operations
- On DL accelerators (e.g., GPU, TPU)

Getting faster: NVIDIA A100, Google TPU v3, ...

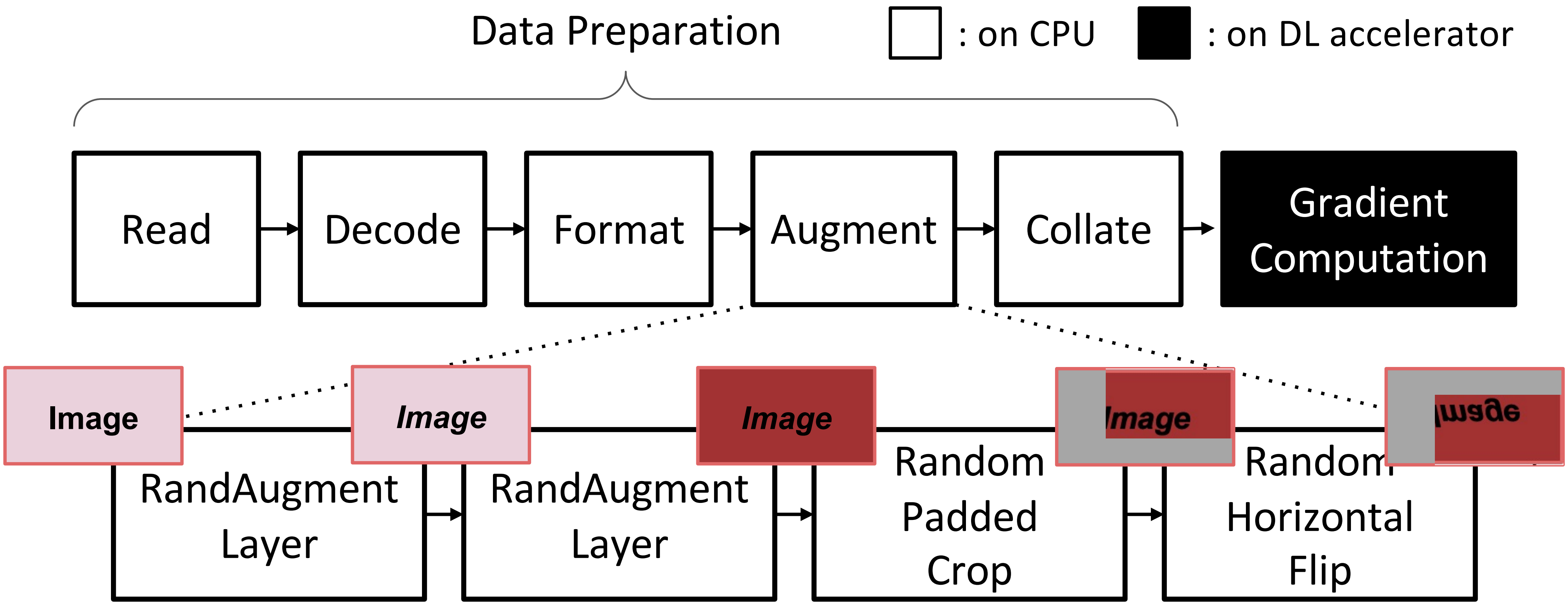
DNN Training Pipeline



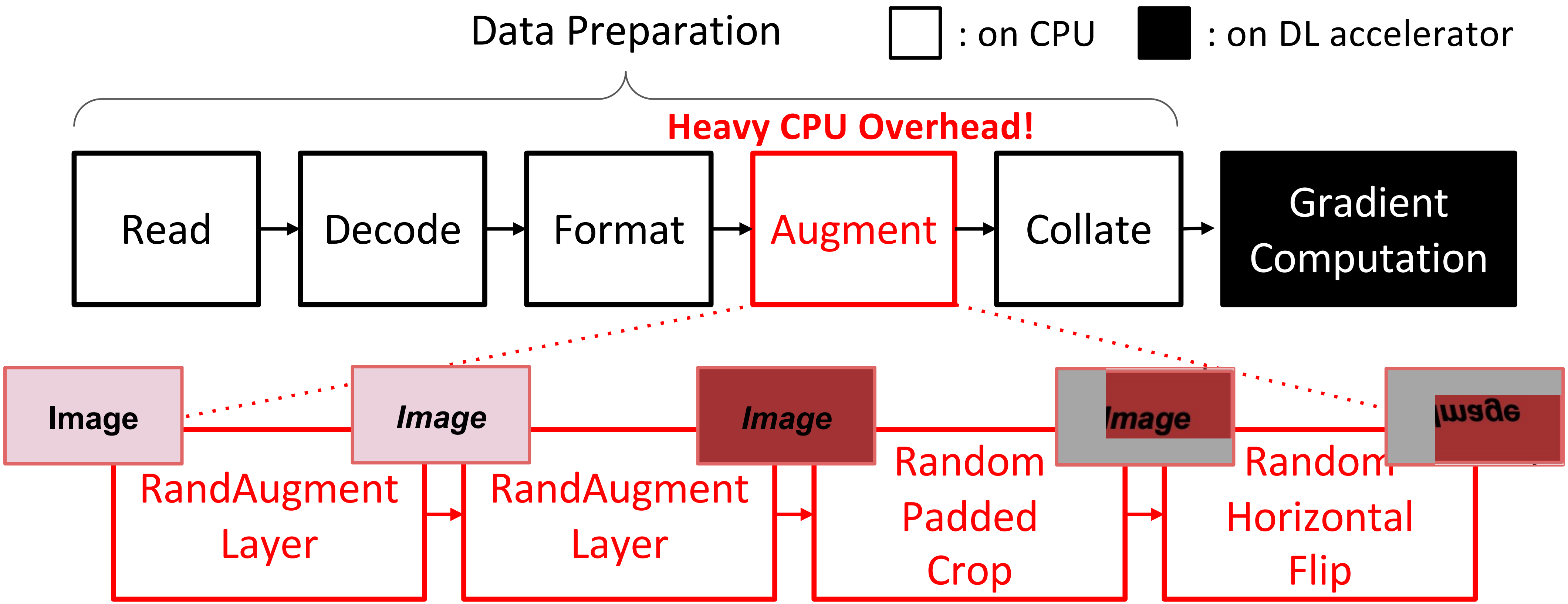
DNN Training Pipeline



DNN Training Pipeline



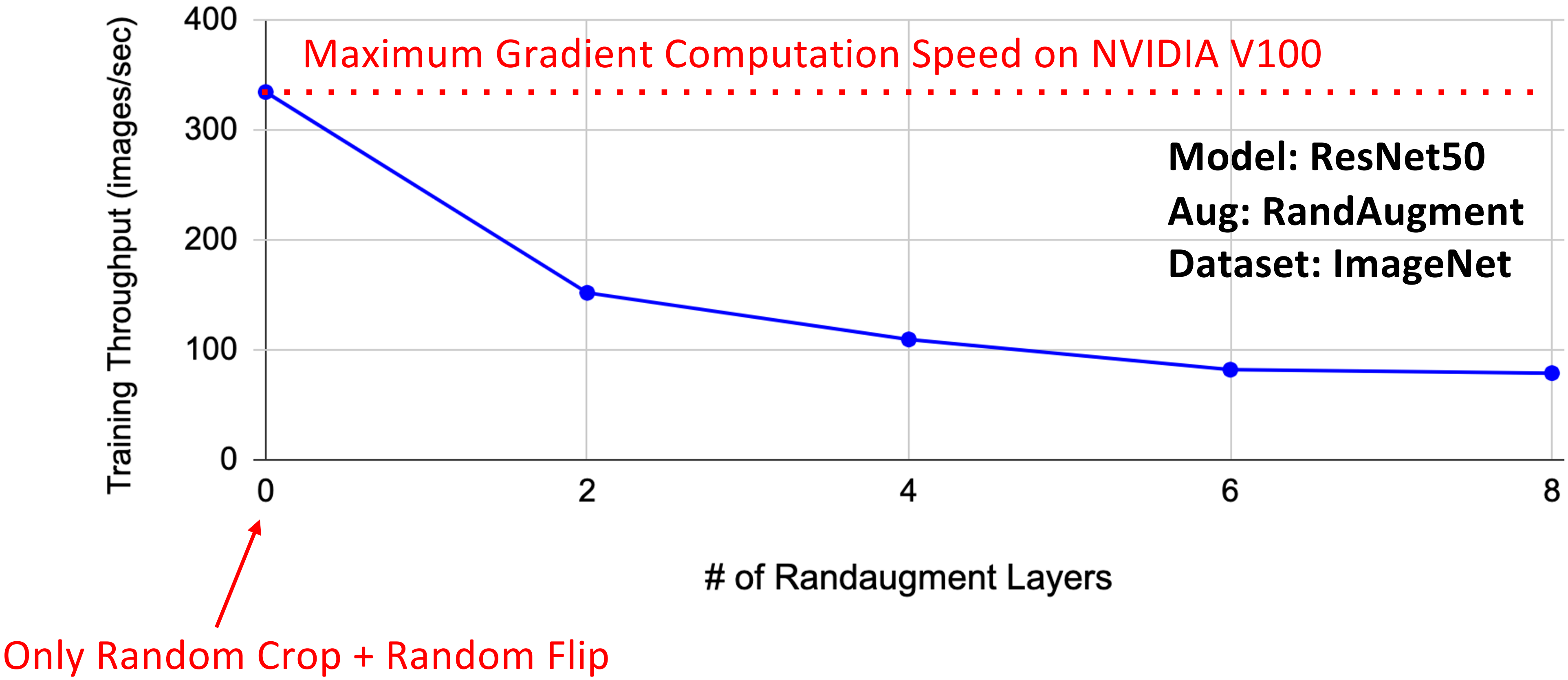
DNN Training Pipeline



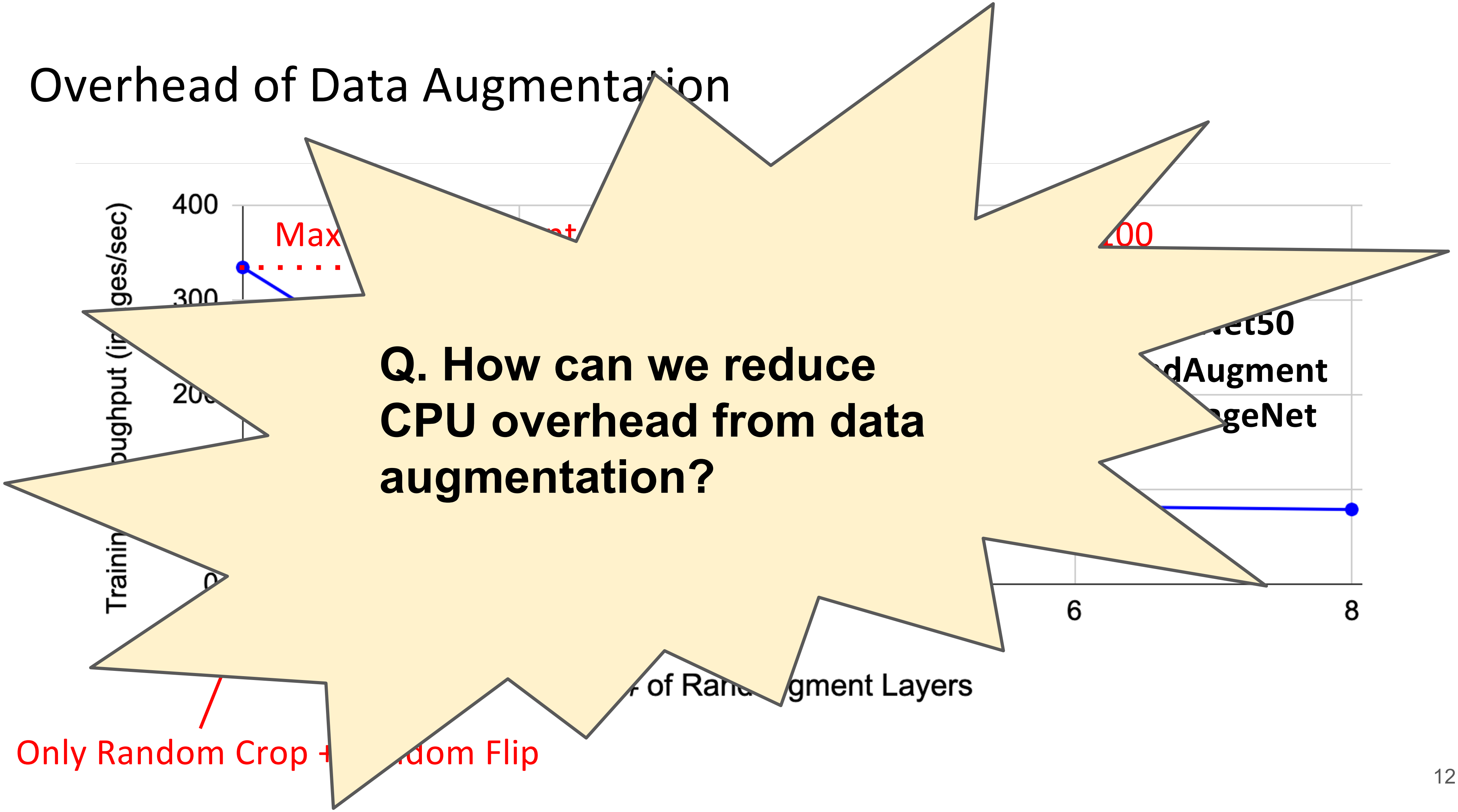
Overhead of Data Augmentation

- Investigate the impact of data augmentation overhead
- Workload: Training ResNet50 on ImageNet with RandAugment
  - Configuration: # of RandAugment Layers
- Environment: One NVIDIA V100 GPU with four physical CPU Cores
  - Same CPU-GPU ratio as cloud GPU VMs such as AWS P3 and GCP N1 instances

Overhead of Data Augmentation



Overhead of Data Augmentation



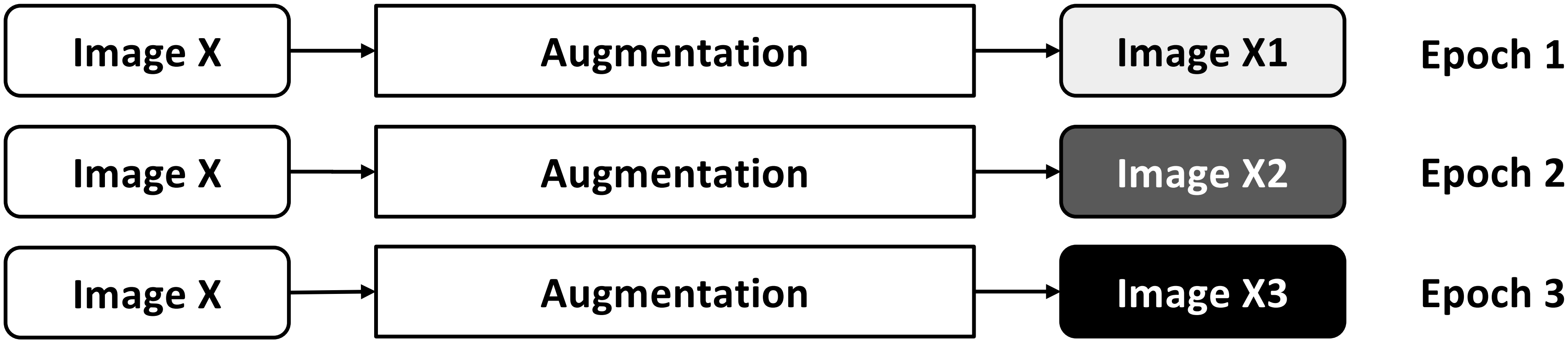
Q. How can we reduce CPU overhead from data augmentation?

Only Random Crop + … dom Flip

Existing Approach: Data Echoing

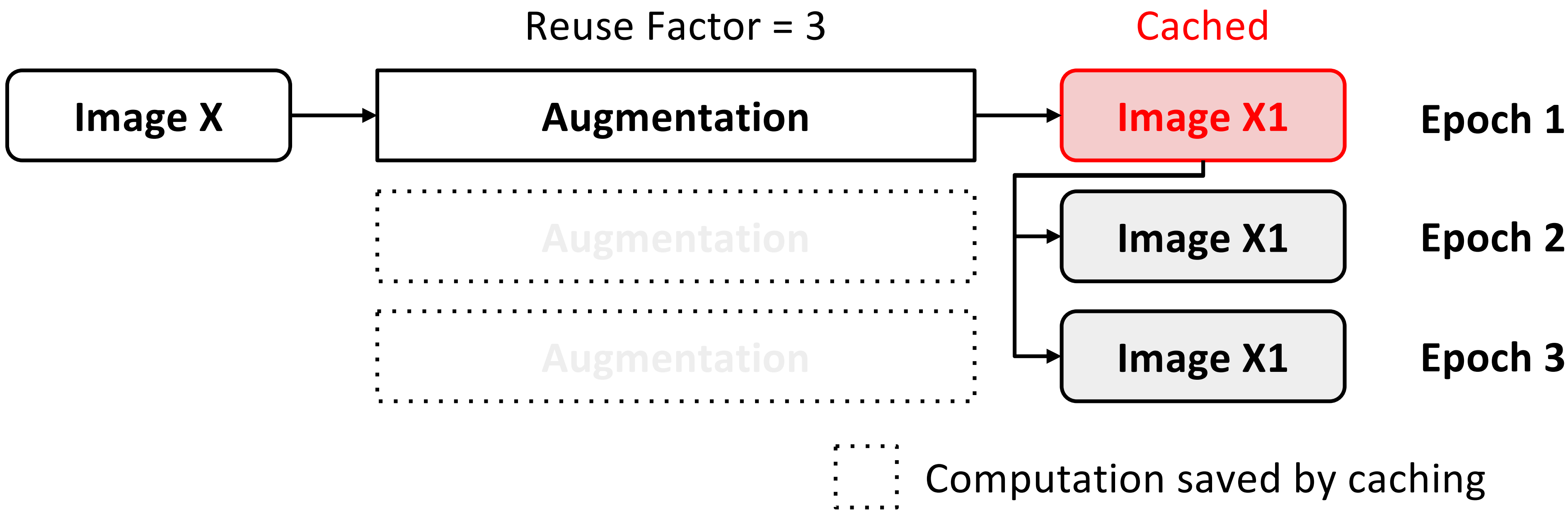
- Data echoing (arXiv ‘20, NeurIPS ‘20): Cache & reuse previously materialized samples
- Useful for training tasks with slow I/O
  - e.g., Training data on remote storage

Standard Training



Data Echoing

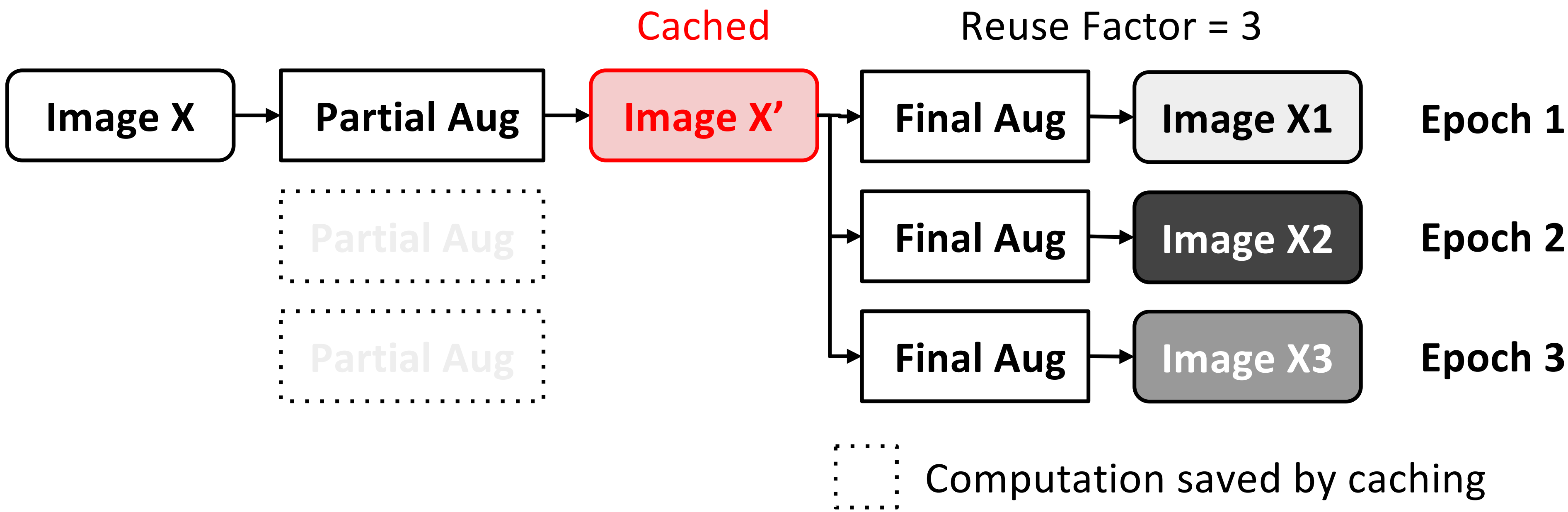
Problem: Sample diversity decreases to a great degree.
-> Low generalization of trained models



Contents

Our Approach: Data Refurbishing

Solution: Cache & reuse *partially augmented samples* by splitting augmentation pipelines



Analysis on Sample Diversity

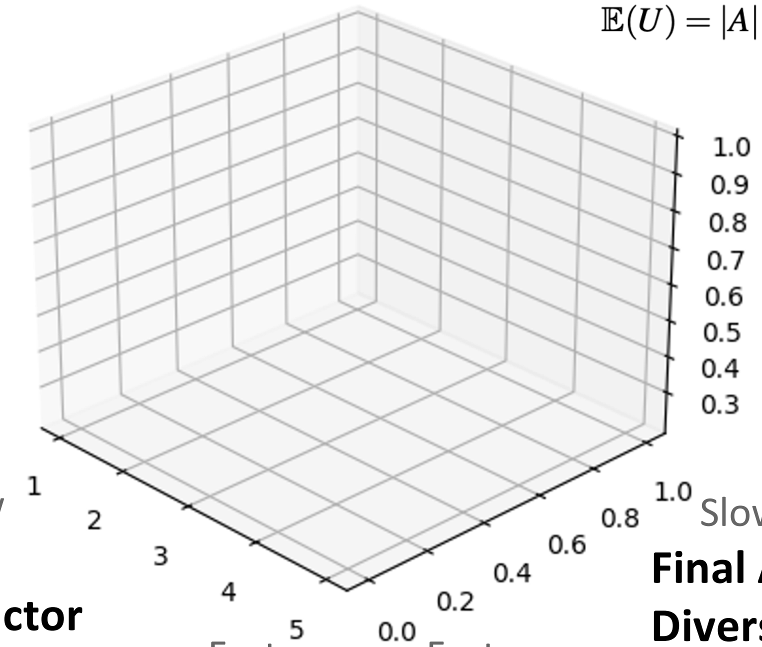
- Notations
  - Given a sample,
    - U (Sample Diversity): # of unique augmented samples during training
    - |A| (Augmentation Diversity): # of possible unique augmented samples by an augmentation pipeline A
    - $|A_F|$: The augmentation diversity of the final augmentation
  - r (Reuse Factor): # of reuses for each cached sample
  - k: The total number of training epochs

$$\mathbb{E}(U) = |A| \left(1 - \left(\left(1 - \frac{|A_F|}{|A|} + \frac{|A_F|}{|A|}\left(1 - \frac{1}{|A_F|}\right)\right)^{r}\right)^{\frac{k}{r}}\right)$$

Analysis on Sample Diversity

Aug: RandAugment
k (# of epochs) = 300

$$\mathbb{E}(U) = |A|\left(1 - \left(1 - \frac{|A_F|}{|A|} + \frac{|A_F|}{|A|}\left(1 - \frac{1}{|A_F|}\right)^r\right)^{\frac{k}{r}}\right)$$



High Accuracy

Sample Diversity

Low Accuracy

Slow

Reuse Factor

Fast

Fast

Slow

Final Augmentation Diversity (Split Strategy)

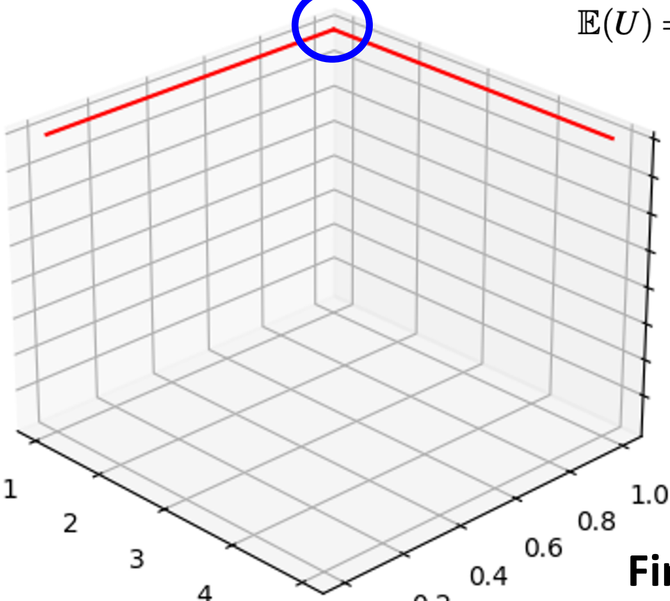
Case #1: Standard Training

$|A_F| = |A|$ or $r = 1$

Aug: RandAugment
k (# of epochs) = 300

High sample diversity but low throughput

$$\mathbb{E}(U) = |A|\left(1 - \left(\left(1 - \frac{|A_F|}{|A|} + \frac{|A_F|}{|A|}\left(1 - \frac{1}{|A_F|}\right)^r\right)^{\frac{k}{r}}\right)\right)$$



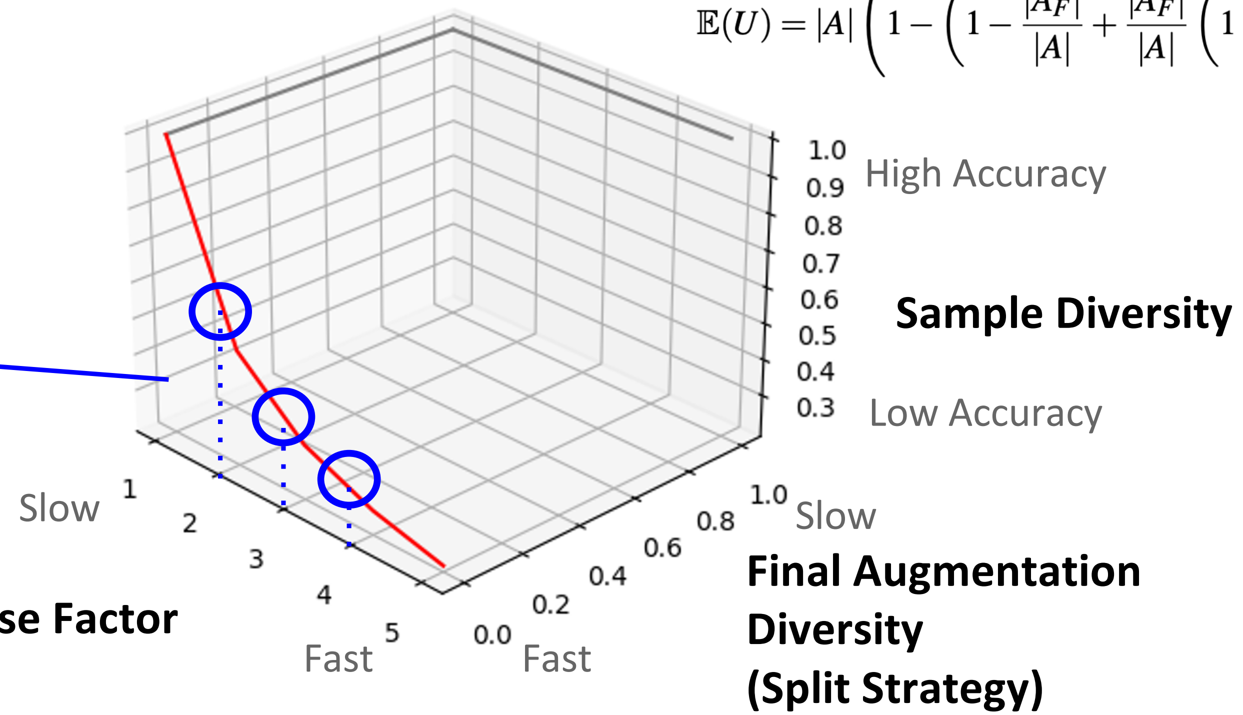
Case #2: Data Echoing

$|A_F| = 1$ and $r > 1$

Aug: RandAugment
k (# of epochs) = 300

$$\mathbb{E}(U) = |A| \left(1 - \left(1 - \frac{|A_F|}{|A|} + \frac{|A_F|}{|A|}\left(1 - \frac{1}{|A_F|}\right)^r\right)^{\frac{k}{r}}\right)$$

High throughput but low sample diversity



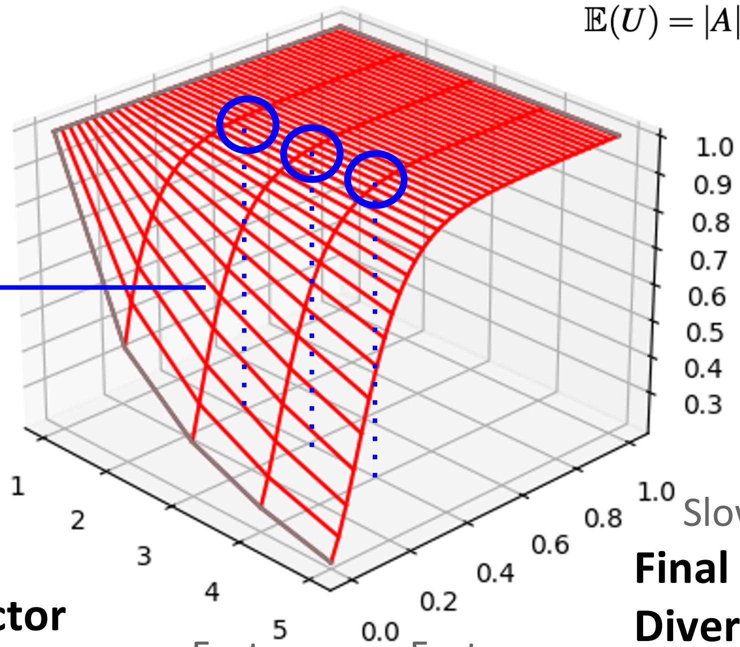
Case #3: Data Refurbishing

$1 < |A_F| < |A|$ and $r > 1$

Aug: RandAugment
k (# of epochs) = 300

$$\mathbb{E}(U) = |A|\left(1-\left(1-\frac{|A_F|}{|A|}+\frac{|A_F|}{|A|}\left(1-\frac{1}{|A_F|}\right)^{r}\right)^{\frac{k}{r}}\right)$$

Exploit "sweet spot"
=> High throughput &
high sample diversity



High Accuracy

Sample Diversity

Low Accuracy

Slow

Reuse Factor

Fast

Slow

Final Augmentation Diversity (Split Strategy)

Fast

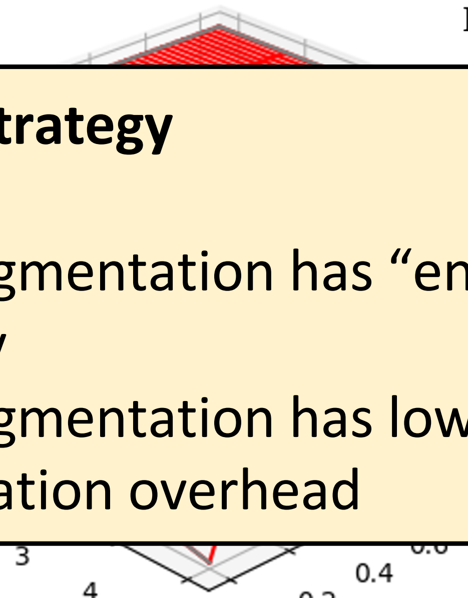
Case #3: Data Refurbishing

$1 < |A_F| < |A|$ and $r > 1$

Aug: RandAugment
k (# of epochs) = 300

$$\mathbb{E}(U) = |A|\left(1 - \left(1 - \frac{|A_F|}{|A|} + \frac{|A_F|}{|A|}\left(1 - \frac{1}{|A_F|}\right)^r\right)^{\frac{k}{r}}\right)$$

Exploit "sweet sp
=> High throughp
high sample dive



Good Split Strategy

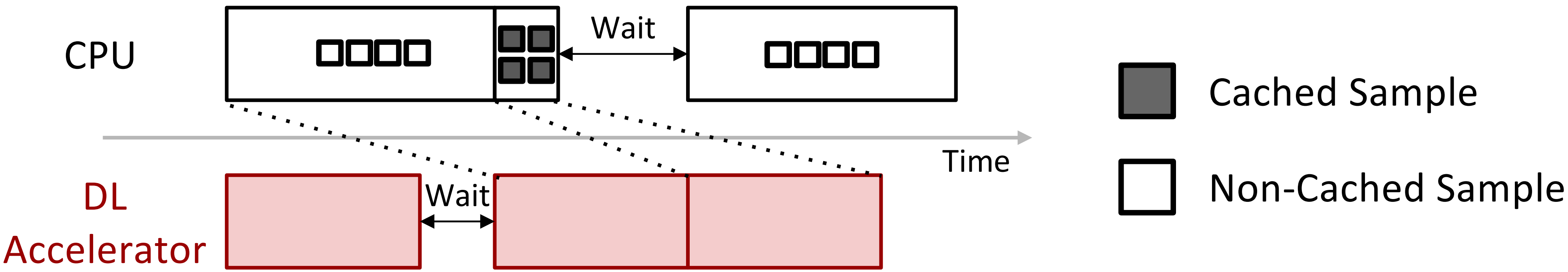
1. Final augmentation has "enough" diversity
2. Final augmentation has low computation overhead

Contents

Challenge: Inconsistent Batch Time

- Within a mini-batch,
  - CPU processing time fluctuates according to the # of cache misses
  - Gradient computation time on DL accelerator remains the same

=> Poor computation overlap



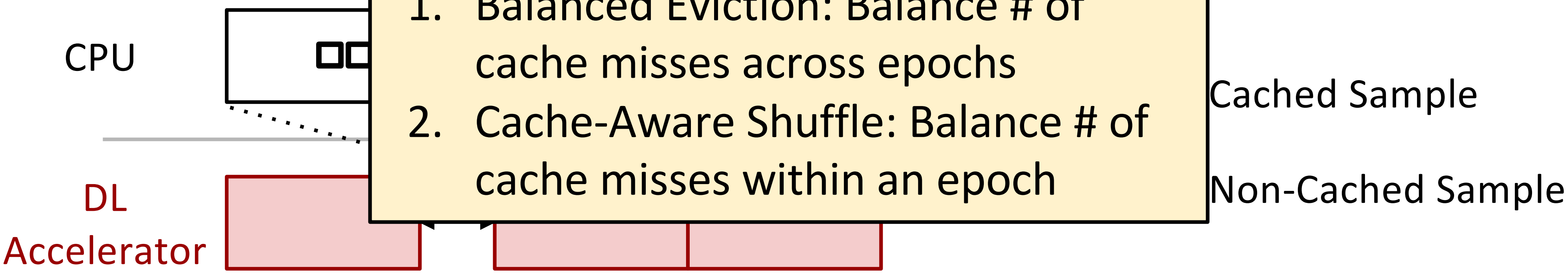
Challenge: Inconsistent Batch Time

- Within a mini-batch,
  - CPU processing time fluctuates according to the # of cache misses
  - Gradient computation time on DL accelerator remains the same

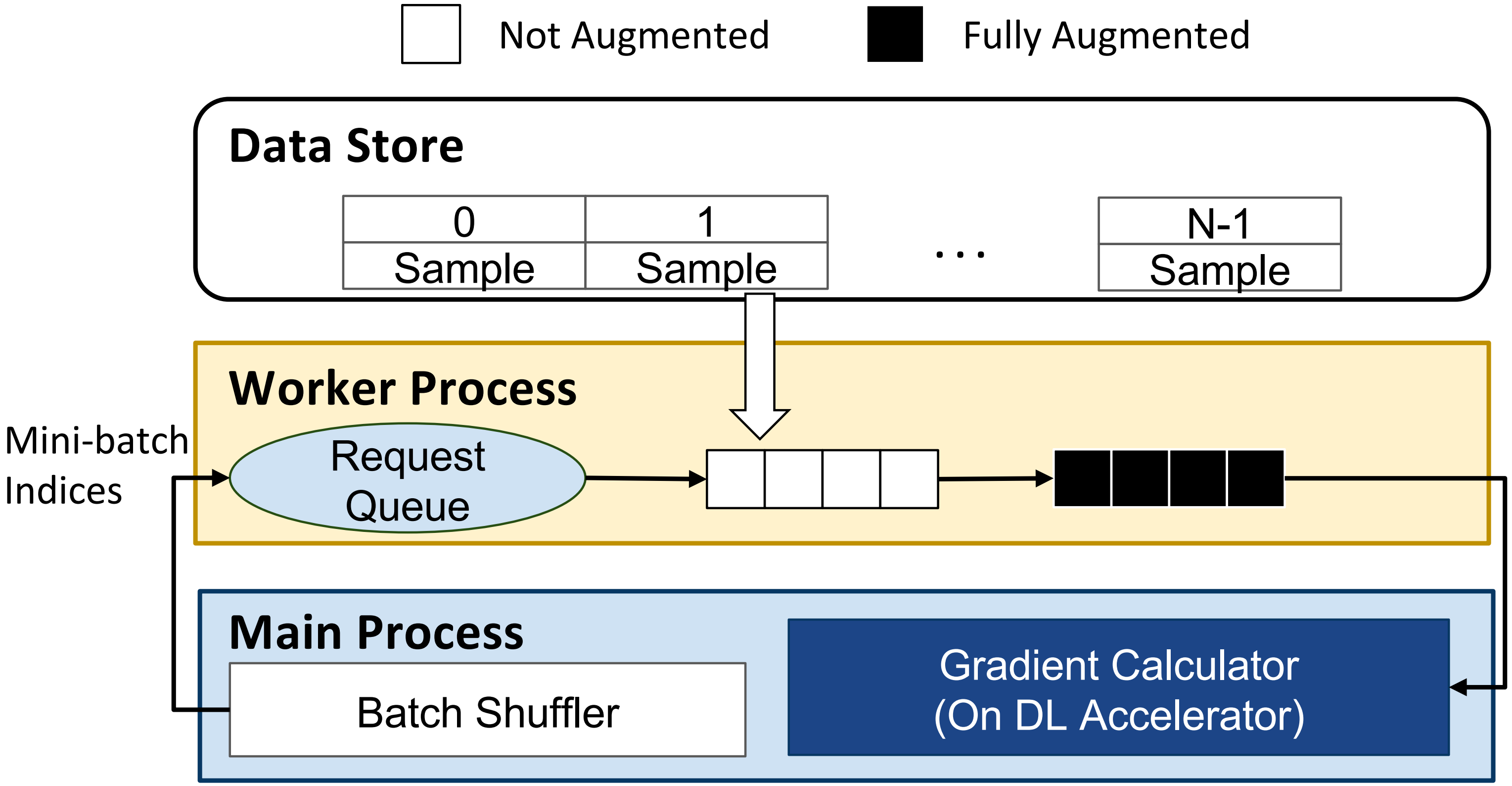
=> Poor computat

Solution: Revamper

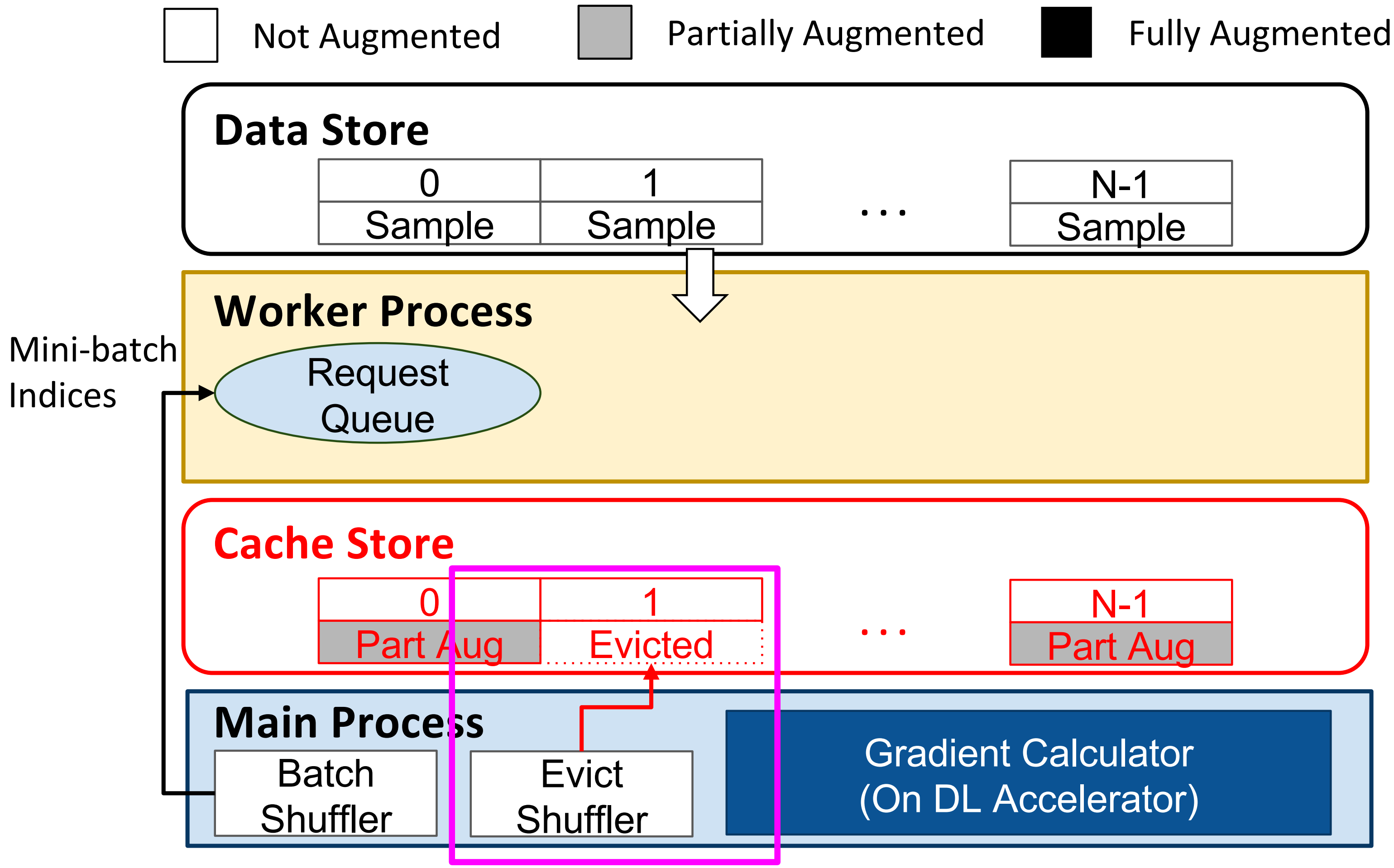
1. Balanced Eviction: Balance # of cache misses across epochs
2. Cache-Aware Shuffle: Balance # of cache misses within an epoch



PyTorch Dataloader

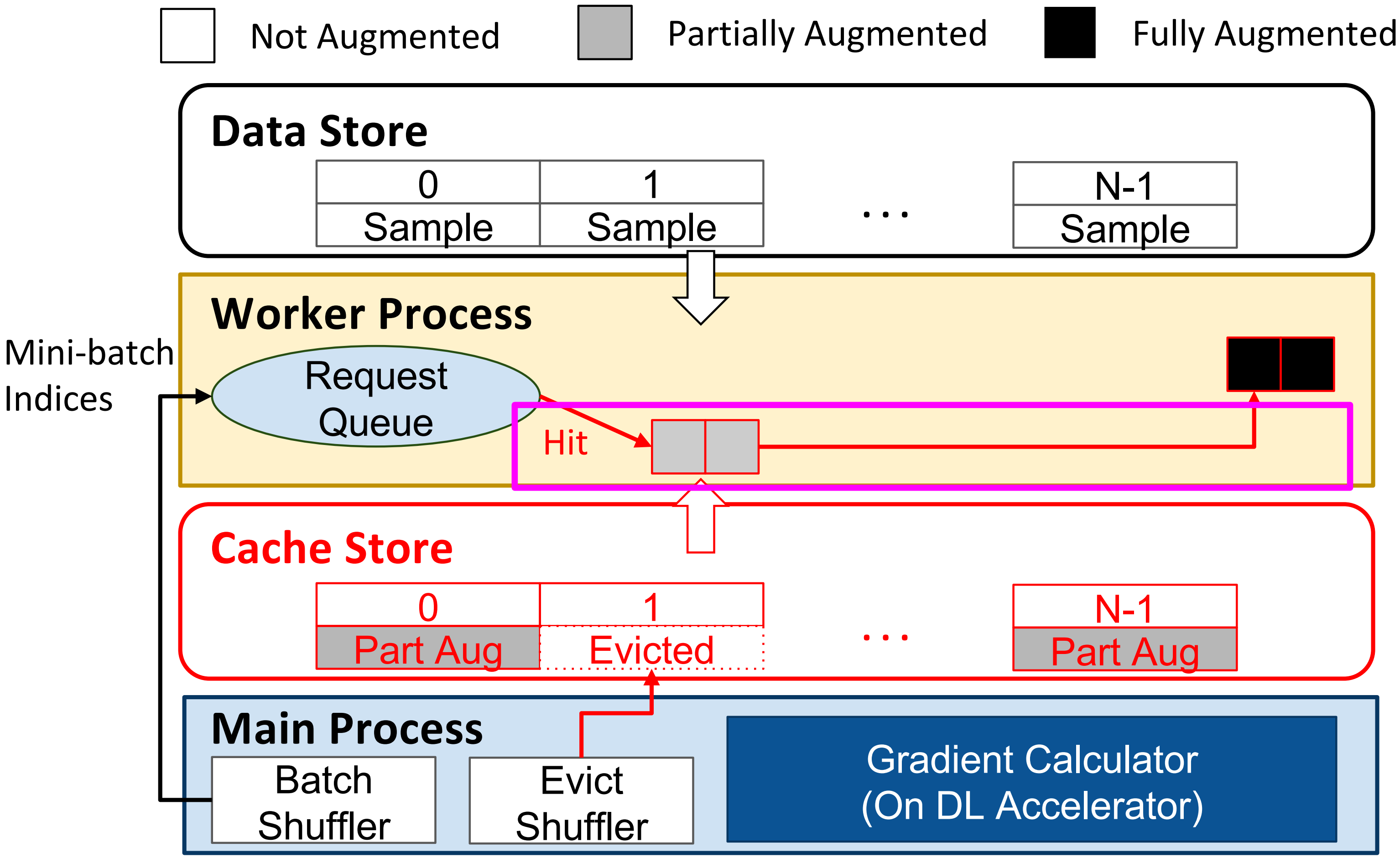


Revamper



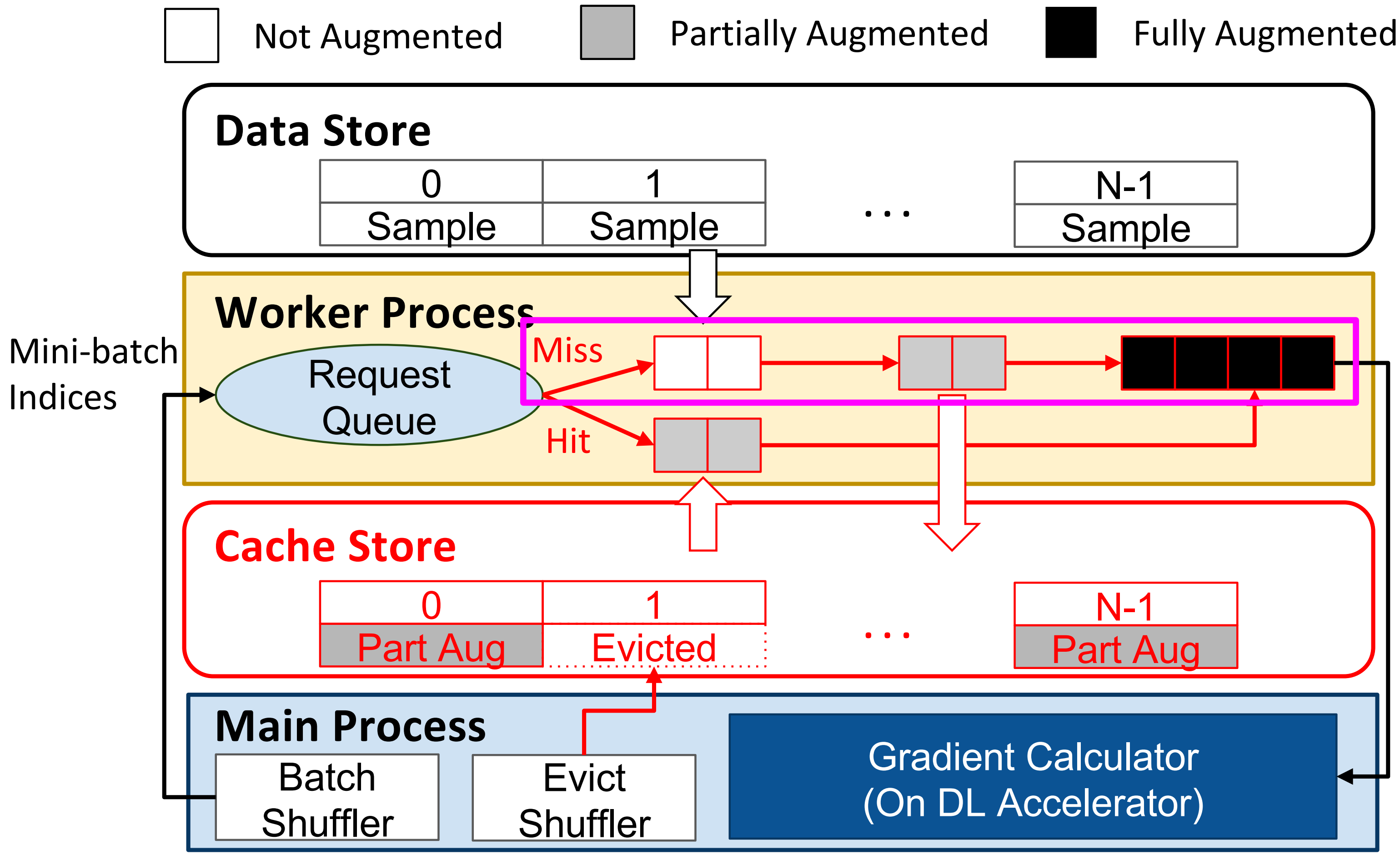
Not Augmented
Partially Augmented
Fully Augmented
Data Store
0
Sample
1
Sample
. . .
N-1
Sample
Worker Process
Mini-batch
Indices
Request
Queue
Cache Store
0
Part Aug
1
Evicted
. . .
N-1
Part Aug
Main Process
Batch
Shuffler
Evict
Shuffler
Gradient Calculator
(On DL Accelerator)

Revamper



Not Augmented
Partially Augmented
Fully Augmented
Data Store
0
Sample
1
Sample
. . .
N-1
Sample
Worker Process
Mini-batch
Indices
Request
Queue
Hit
Cache Store
0
Part Aug
1
Evicted
. . .
N-1
Part Aug
Main Process
Batch
Shuffler
Evict
Shuffler
Gradient Calculator
(On DL Accelerator)

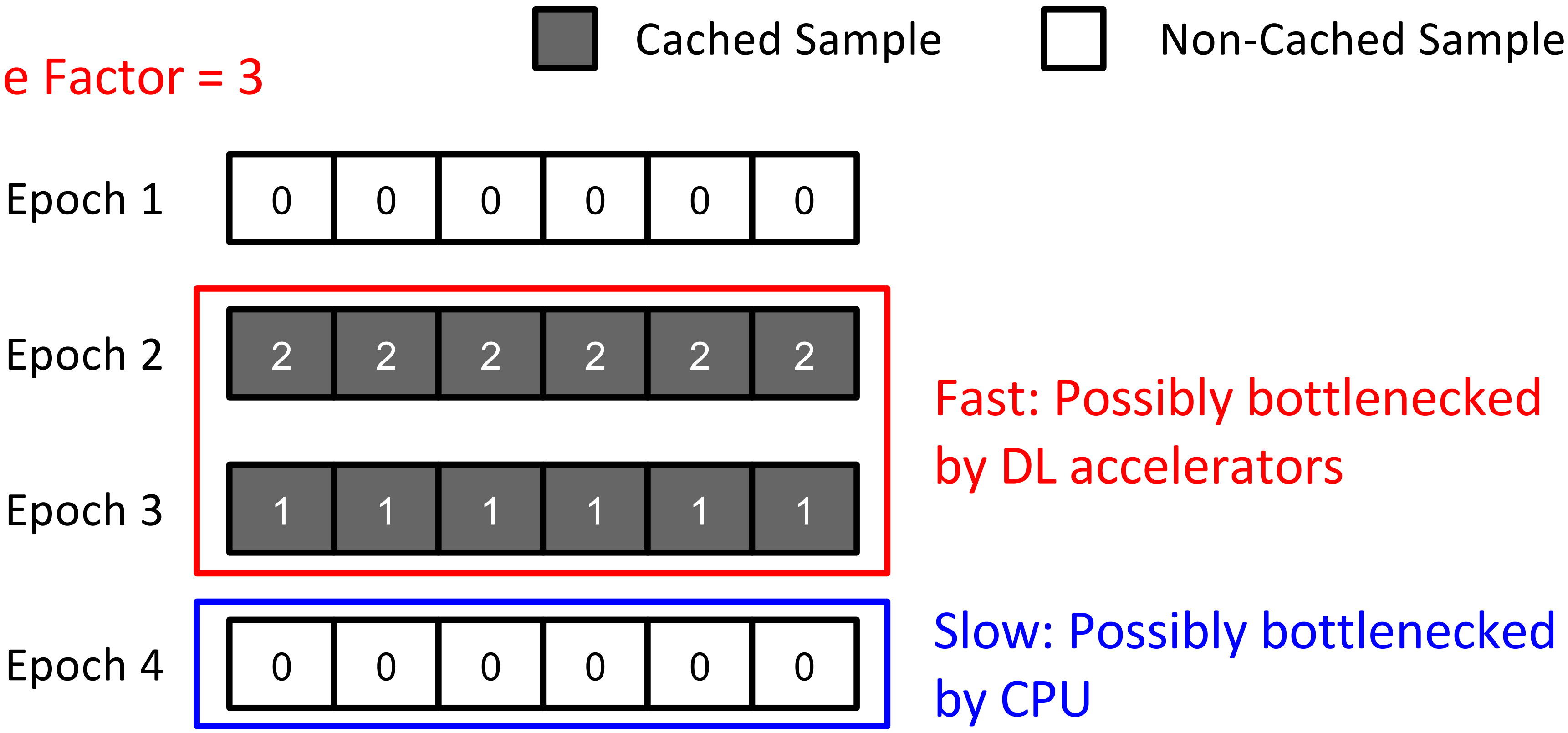
Revamper



Not Augmented
Partially Augmented
Fully Augmented
Data Store
0
Sample
1
Sample
. . .
N-1
Sample
Worker Process
Mini-batch Indices
Request Queue
Miss
Hit
Cache Store
0
Part Aug
1
Evicted
. . .
N-1
Part Aug
Main Process
Batch Shuffler
Evict Shuffler
Gradient Calculator (On DL Accelerator)

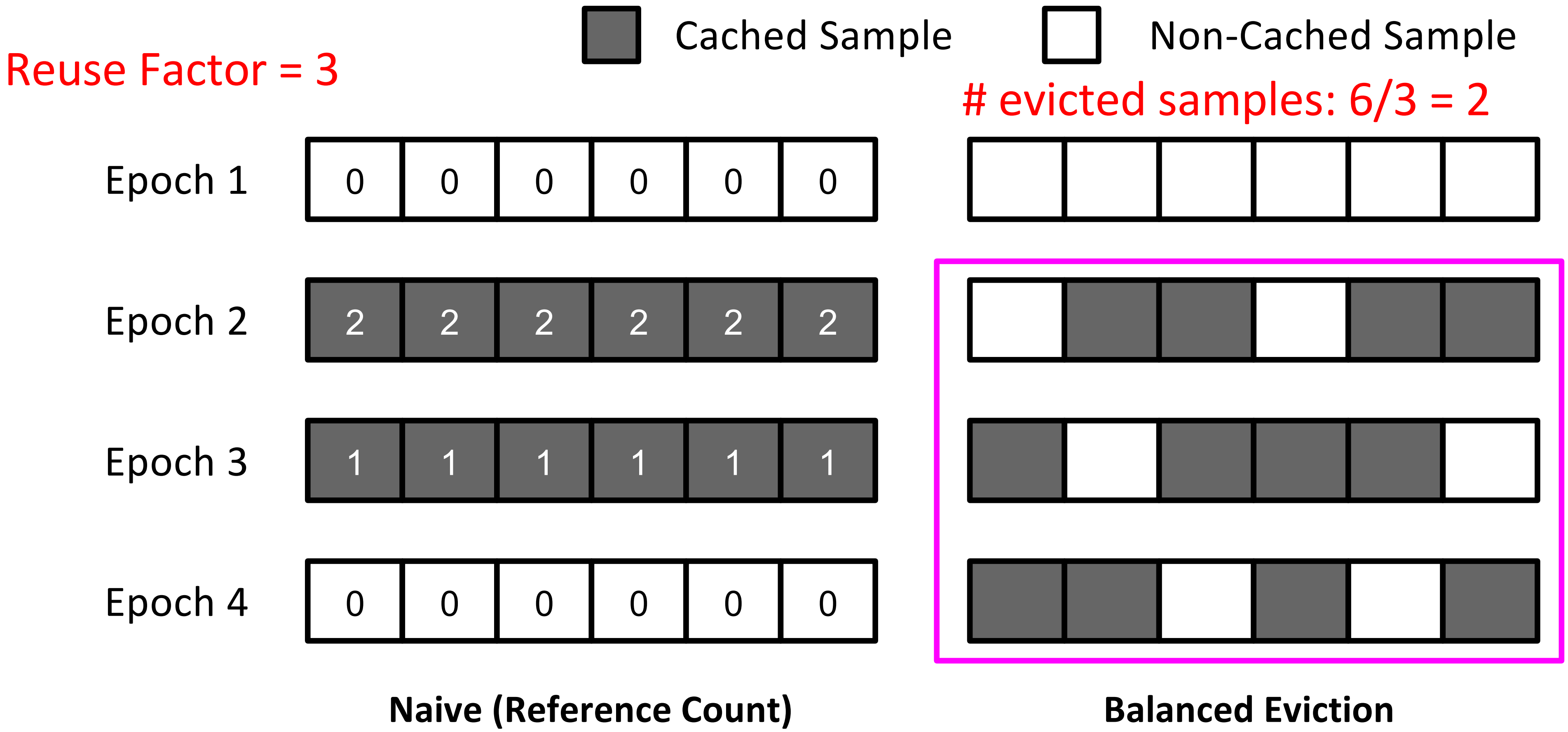
Balanced Eviction

Reuse Factor = 3



Naive (Reference Count)

Balanced Eviction

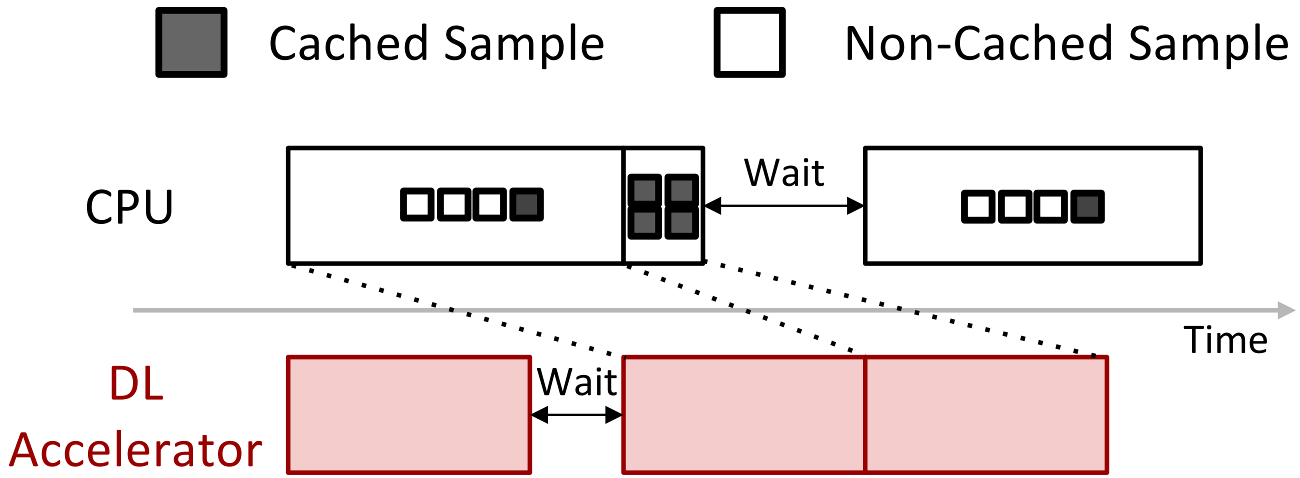


Cache-Aware Shuffle

Batch Size = 4

Reuse Factor = 2

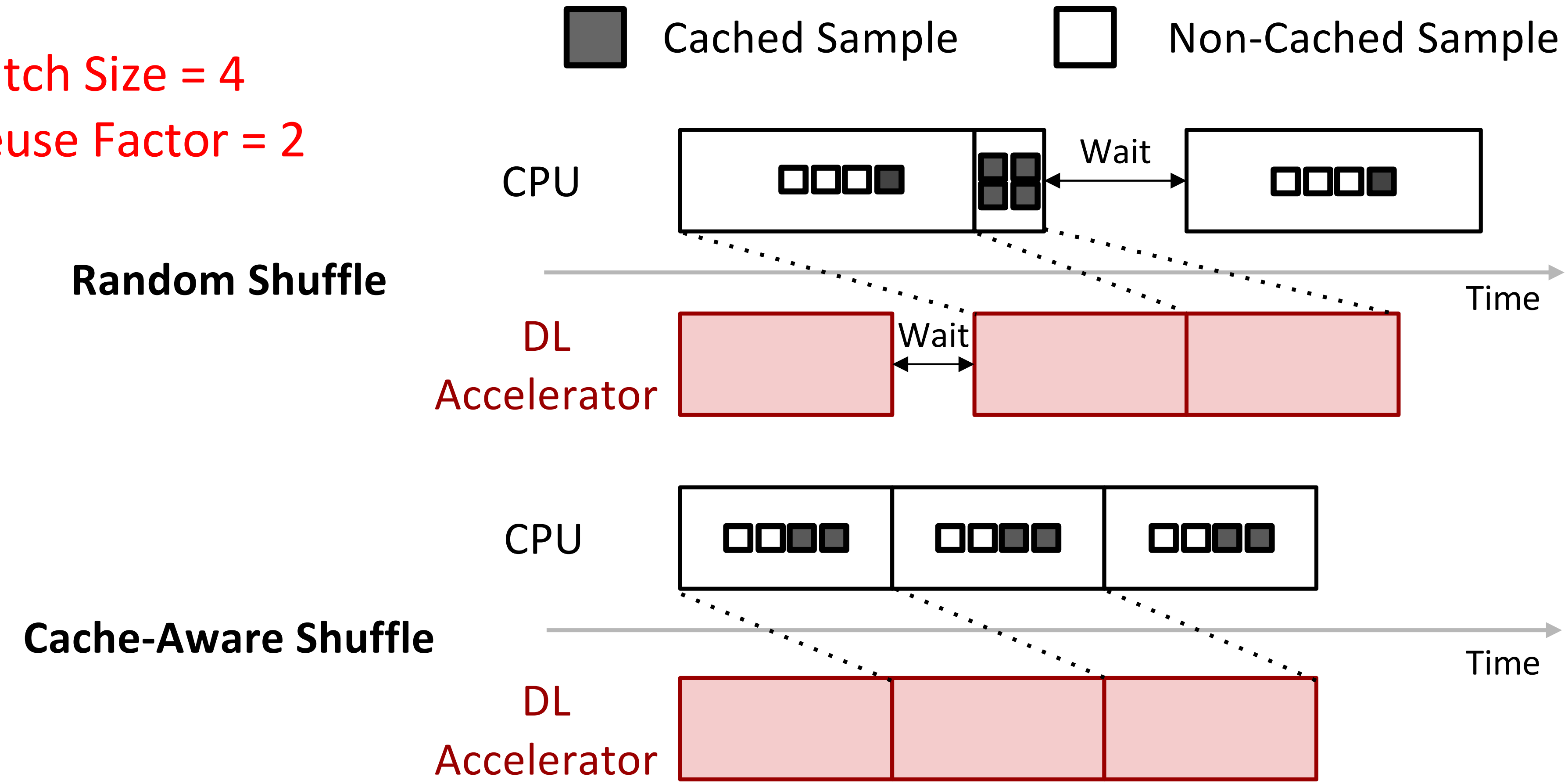
Random Shuffle



Cache-Aware Shuffle

Batch Size = 4

Reuse Factor = 2



Contents

Implementation

- Implemented in 2000+ lines of Python code based on PyTorch 1.6
- Identical interface to the PyTorch dataloader except for some additional parameters
  - e.g., reuse factor and split strategy

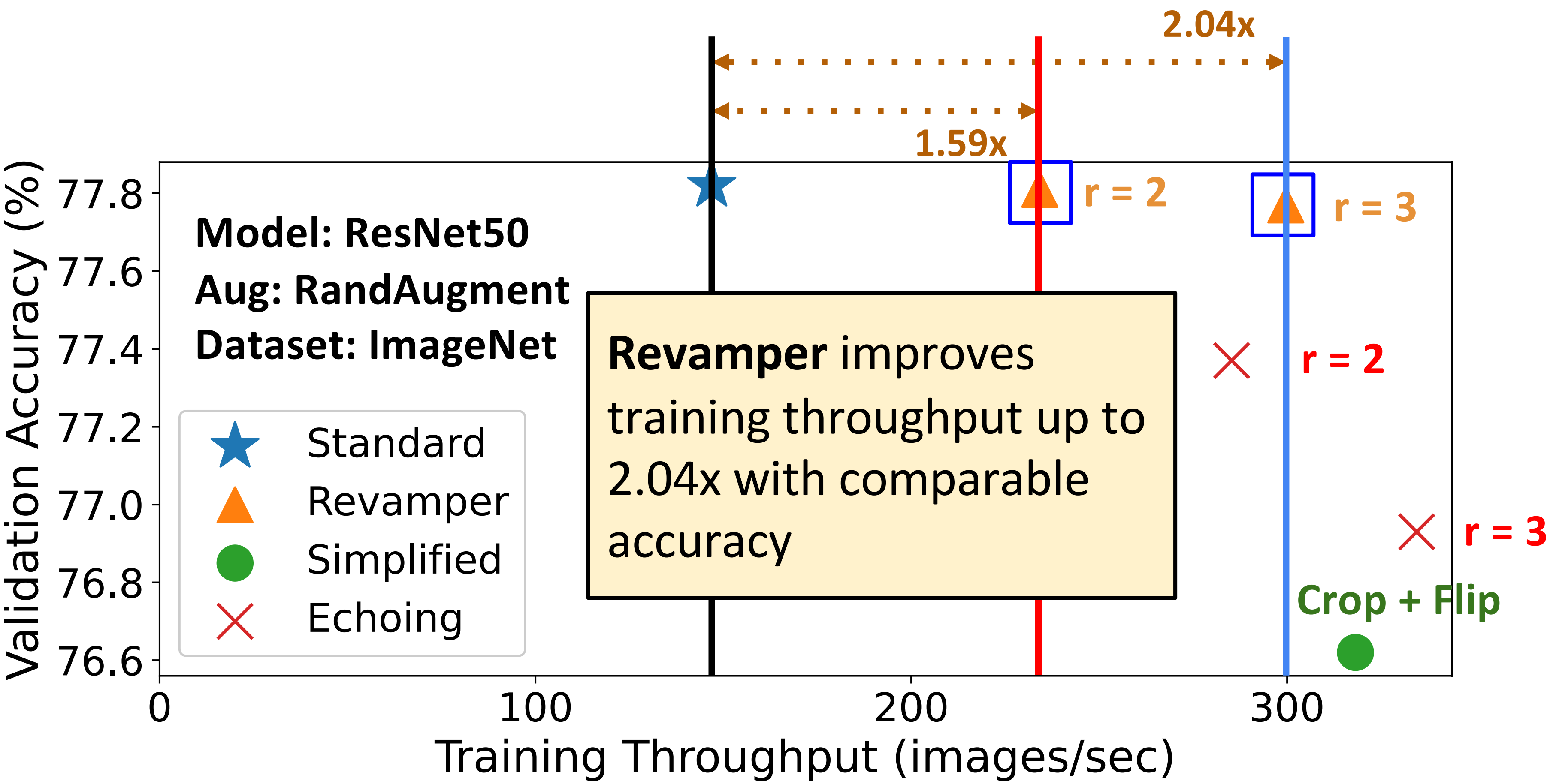
Evaluation: Environments

- Training server specification
  - CPU: Intel Xeon E5-2695v4 (18 cores, 2.10GHz, 45MB Cache)
  - RAM: 256GB DRAM
  - GPU: NVIDIA V100
  - Disk: Samsung 970 Pro 1TB NVMe SSDs
- We adjust CPU-GPU ratios using a Docker container (Default = 4:1)
- Workload: Image Classification

Evaluation: Baselines

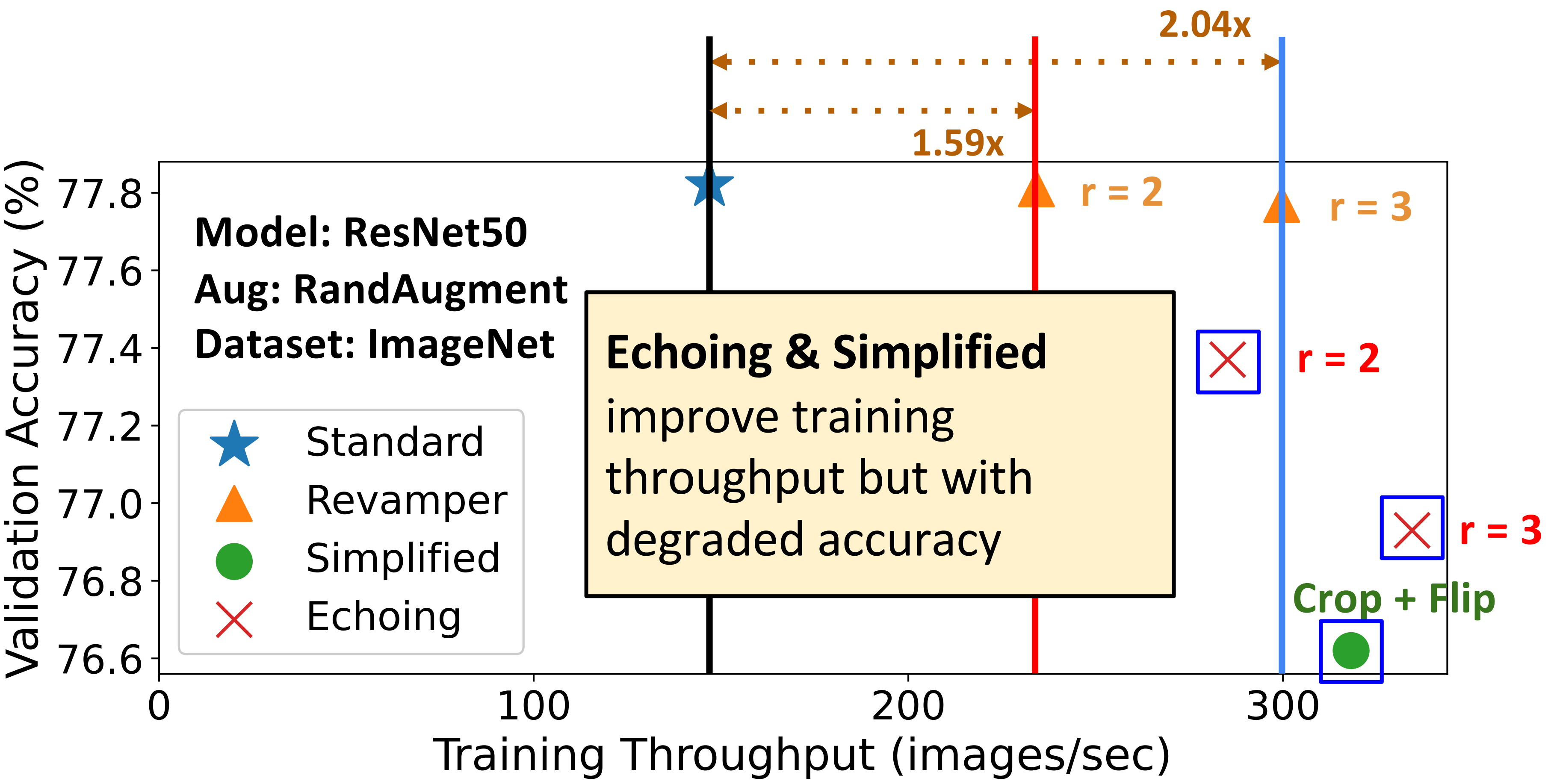
- Standard: DNN training without adopting data reusing mechanism
- Data Echoing: Cache & reuse fully augmented samples
- Simplified: Simply removing one or more transformation layers
- Same hyperparameters for all the training settings
  - Revamper does not require additional hyperparameter tuning

Evaluation: Accuracy & Throughput

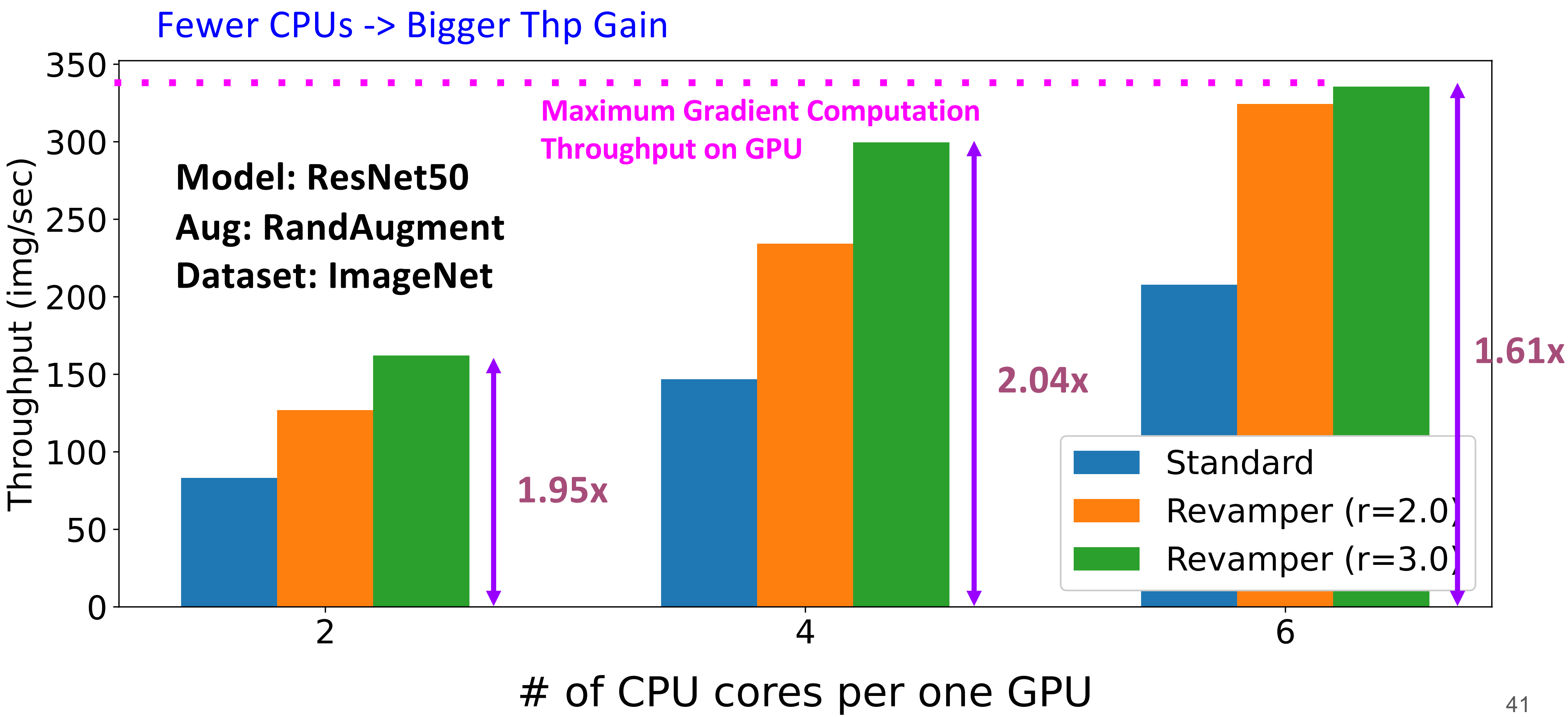


Revamper improves training throughput up to 2.04x with comparable accuracy

Evaluation: Accuracy & Throughput



Evaluation: CPU-GPU Ratio



Conclusion

- **Data refurbishing** is a new intermediate data caching technique for DNN training that accelerates data augmentation while preserving diversity of augmented samples.
- **Revamper** realizes data refurbishing by maximizing computation overlap between CPU and DL accelerators with carefully-designed cache eviction and shuffle strategies.
- Revamper improves training throughput of DNN models by **1.03x-2.04x** while maintaining comparable accuracy.