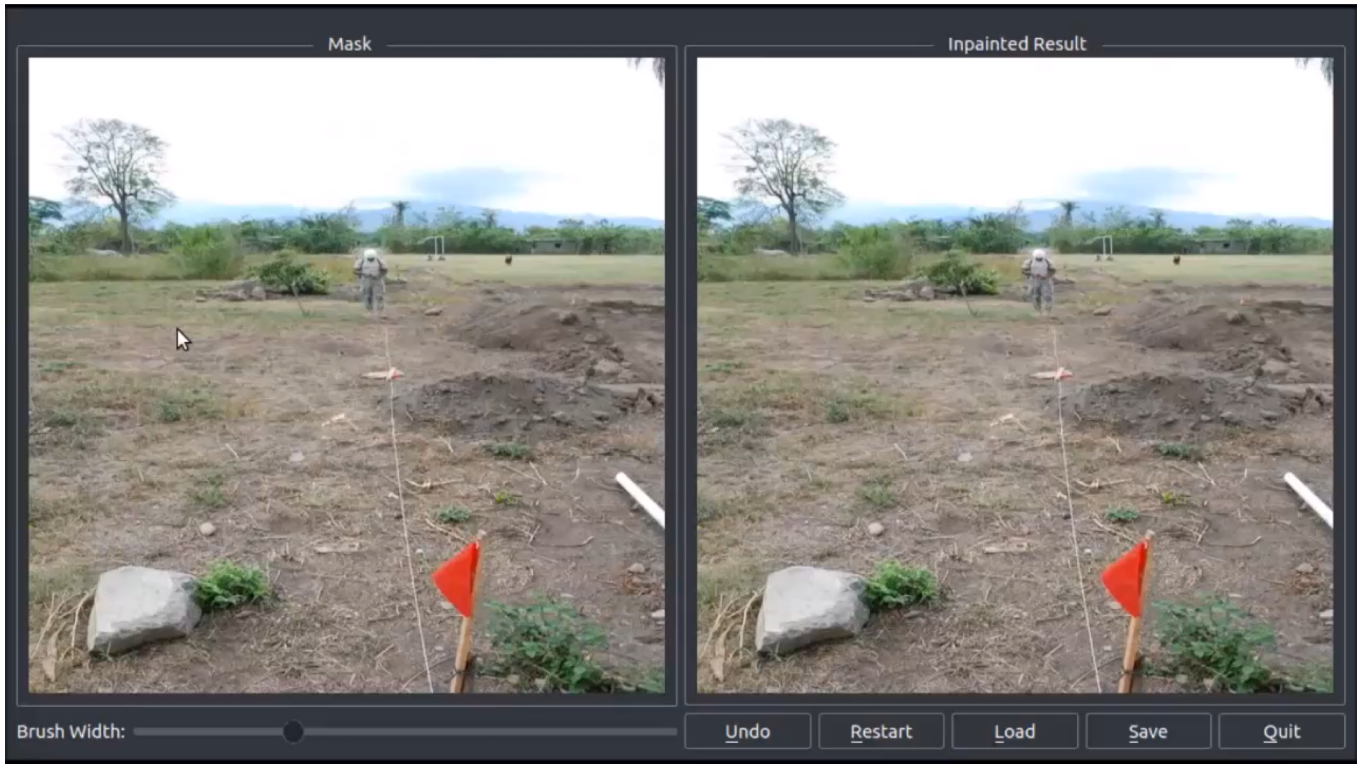
MIXED PRECISION TRAINING FOR IMAGE INPAINTING

Guilin Liu

IMAGE INPAINTING

Applications

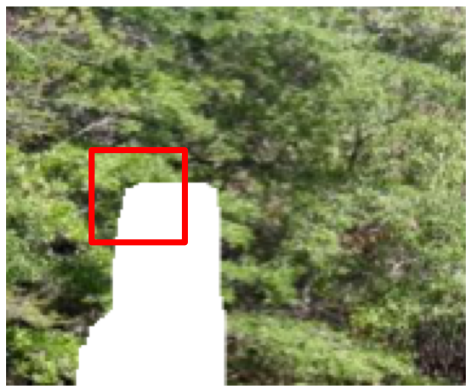
Image Inpainting: remove unwanted content and re-fill with reasonable content.



MODEL DETAILS

- Partial Convolutions
- U-Net architecture with skip connections
- VGG loss (feature loss and gram matrix loss)

OUR CONTRIBUTION - PARTIAL CONVOLUTIONS



X: input

$$\left|\begin{matrix} 1 & 1 & 1 \\ 1 & 0 & 0 \\ 1 & 0 & 0 \end{matrix}\right|$$

M: mask

- Convolution (typical)

  Pixel update: $X_i' = W^T X + b$

- Partial Convolution

  K: conv kernel size

  normalization

  Pixel update: $X_i' = W^T (X \circ M) \cdot \frac{K^2}{sum(M)} + b$

  Mask update: $M_i' = \begin{cases} 1 & if\ sum(M) > 0 \\ 0 & if\ sum(M) = 0 \end{cases}$

mask updating after several partial conv layers

As the receptive field becomes larger, mask will become all 1

OUR SOLUTION: PARTIAL CONVOLUTION

- Advantages
  - Initial value in holes doesn’t matter
  - U-Net structure is possible
  - Another padding scheme
  - Possibly useful for other incomplete data

CHANGES REQUIRED

```
from apex import amp
```

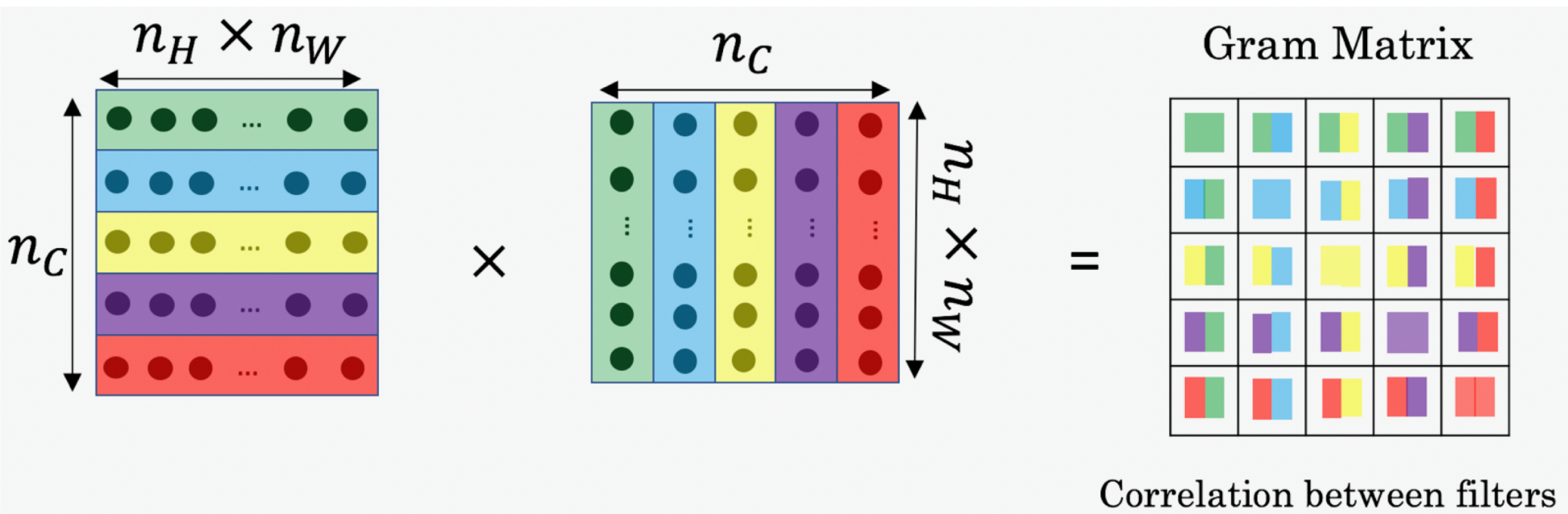
```
self.generator, self.optimizer = amp.initialize(self.generator,
    self.optimizer, opt_level=args.fp16_opt_level)
```

```
self.vgg_feat_loss = amp.initialize(self.vgg_feat_loss,
    opt_level=args.fp16_opt_level)
```

```
with amp.scale_loss(total_loss, self.optimizer) as scaled_loss:
    scaled_loss.backward()
```

GRAM MATRIX COMPUTATION

$$G^l_{ij} = \sum_k F^l_{ik} F^l_{jk}.$$



Source: https://www.deeplearning.ai/

EFFICIENT GRAM MATRIX COMPUTATION

Same code implementation for FP32/FP16

- More performant version for batch matrix-matrix product of matrices
  - `baddbmm` $\text{out}_i = \beta\ \text{input}_i + \alpha\ (\text{batch1}_i\ @\ \text{batch2}_i)$
  - saves memory traffic on reads/writes
- Factor `1./(c * h * w)` can be too small for FP16
  - large channel count or image resolution, e.g. when value less than 2^-24
  - change one-step big division to two-step smaller divisions to avoid underflow

```
def gram_matrix(input_tensor):
  """Compute gram matrix."""
  b, c, h, w = input_tensor.size()
  features = input_tensor.view(b, c, h * w)
  features_t = features.transpose(1, 2)
  inputs = torch.zeros(b, c, c).type(features.type())
  gram = torch.baddbmm(inputs, features, features_t,
                       beta=0, alpha=1./(c * h * w), out=None)
  return gram
```

DEALING WITH MACHINE EPSILON

- Meant to avoid division by zero
  - make your small constant number a bit bigger for FP16
  - e.g. **1e-8** ⟶ **1e-6**

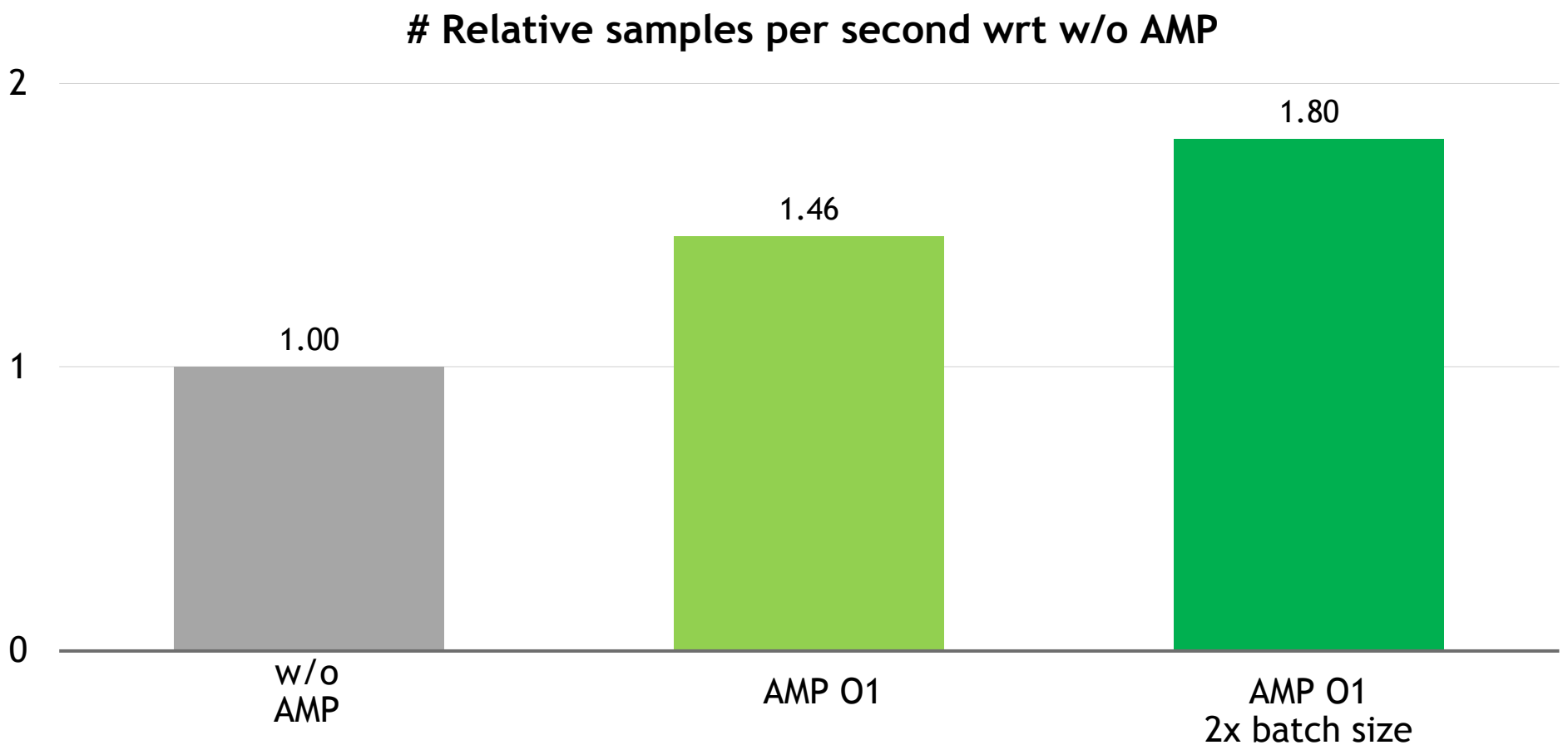
```
1 self.mask_ratio = self.slide_winsize/(self.update_mask + 1e-8)
```

```
1 self.mask_ratio = self.slide_winsize/(self.update_mask + 1e-6)
```

Specs:
Machine: DGX1 V100 16GB

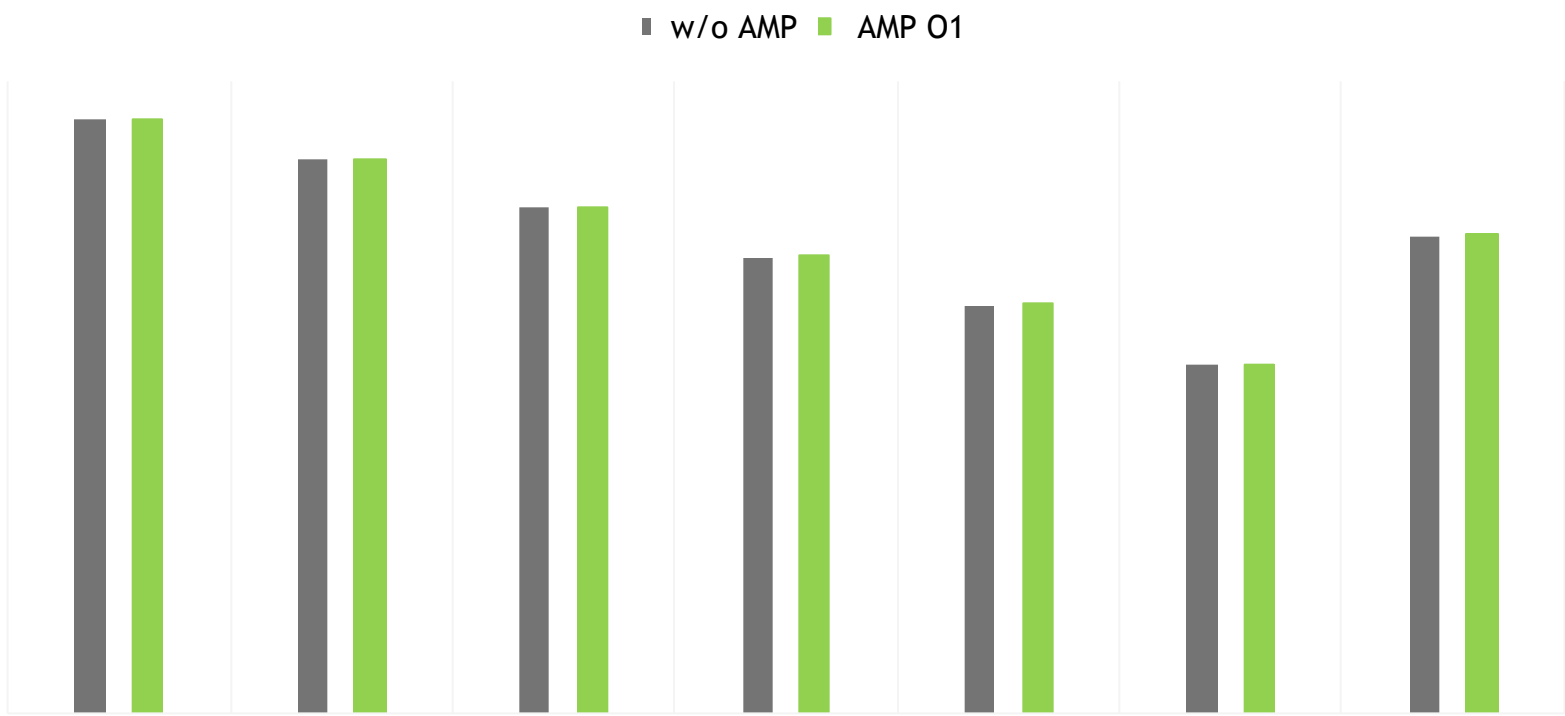
OBTAINED SPEEDUPS

1.4x faster (~1.8x faster with doubled batch size)

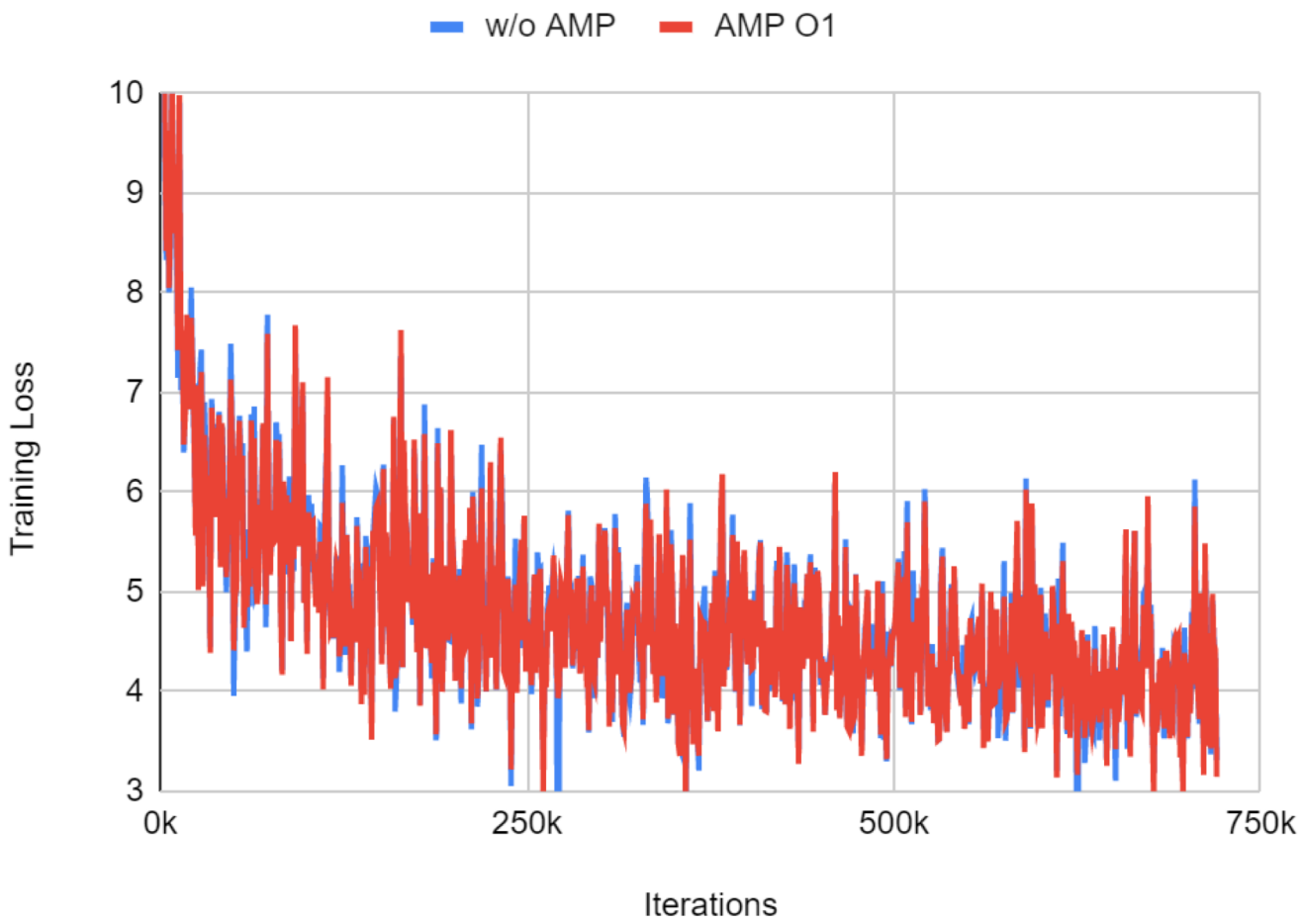


EFFECT ON ACCURACY

Same SSIM scores



METRICS OVER TIME



TAKEAWAYS

1. AMP is easy to integrate with very few changes
   - only 4 lines of code
   - use baddbmm for perf and watch out for too small normalization factor
   - avoid division by zero in reduced precision, e.g. **1e-8** ⟶ **1e-6**

2. For the same model and batch size, we can get 1.4x speed up

3. For 2x larger batch size, we can get 1.8x speed up


NVIDIA