Performance Optimizations for Deep Image Matting in Photoshop

Betty Leong  | Photoshop Engineering Manager, Adobe
Salil Tambe  | Computer Vision Engineer, Adobe
Chris Hebert | DevTech Engineer, Nvidia
Topics

- Introduction to Matting
- Deep Matting vs Photoshop Matting
- Deployment Challenges
- Optimization
- Results and Demo
- Conclusion
Select and Mask in Photoshop
Select and Mask in Photoshop
Matting in Photoshop
Deep Matting
Brian Price (GTC 2017)

Topics

- Introduction to Matting
- Deep Matting vs Photoshop Matting
- Deployment Challenges
- Optimization
- Results
- Conclusion
Correct matting for hair

Trimap

Photoshop Matte

Hair, should be white

Deep Matte

Image
Topics

- Introduction to Matting
- Deep Matting vs Photoshop Matting
- Deployment Challenges
- Optimization
- Results and Demo
- Conclusion
Tech Transfer Challenges

- Resolution (320 x 320)
- Model size (80 MB)
- Memory
- Run time performance
- Cross platform support

Image Credits: Deep Image Matting [Xu et. al; CVPR 2017]
Challenges

- Interactive editing with Deep Matting should be possible, but it is computationally expensive!

  Matting uses a VGG-Net based Encoder-Decoder network that requires > 1sec and 600mb memory for inferencing a 320 x 320 image on Intel i-7 8650K CPU using caffe

- It should be deployable on all platforms supported by Photoshop
  i.e. all combinations of Intel, AMD and Nvidia hardware on Mac and Windows
Deep Matting Deep Dive

Input

trimap
Deep Matting Deep Dive

Input

+ trimap
Inference Per Tile

Framework used: Caffe (with CUDA and CUDNN)

Image Credits: Deep Image Matting [Xu et. al; CVPR 2017]
Fine Network

Refinement Network

Final Matte
Explorations/Experiments

- Collaborate with **Nvidia DevTech ProVis** Team to come up with better per tile inference performance
- **Chris Hebert** – DevTech Engineer
- Inference customization with CuDNN kernels for optimal performance
Topics

- Introduction to Matting
- Deep Matting vs Photoshop Matting
- Deployment Challenges
- Optimization
- Results and Demo
- Conclusion
Caffe: Inference Timeline (nv prof)

160ms

Titan-X
Caffe: CPU Overhead

Layer weights transferred one at a time

Titan-X
Caffe: Uses FFT (memory intensive)
cuDNN – A bit like OpenGL for Neural Networks

- Networks for inferencing are not difficult to implement with cuDNN

- cuDNN Provides a set of common network operations
  - Convolution
  - Activation
  - Tensor Ops – Add, multiply etc

- Highly optimized for respective HW architectures

- cuDNN is the backend for most frameworks that target NVIDIA Hardware
Optimization 1: Better memory management

- Pre-allocate the max buffer size required for the workspace
Output layers size

<table>
<thead>
<tr>
<th>Layer</th>
<th>Size</th>
<th>Memory</th>
</tr>
</thead>
<tbody>
<tr>
<td>Conv1_1</td>
<td>320 x 320 x 64</td>
<td>25mb</td>
</tr>
<tr>
<td>Conv1_2</td>
<td>320 x 320 x 64</td>
<td>25mb</td>
</tr>
<tr>
<td>Conv2_1</td>
<td>160 x 160 x 128</td>
<td>12.5mb</td>
</tr>
<tr>
<td>Conv2_2</td>
<td>160 x 160 x 128</td>
<td>12.5mb</td>
</tr>
<tr>
<td>Conv3_1</td>
<td>80 x 80 x 256</td>
<td>6.25mb</td>
</tr>
<tr>
<td>Conv3_2</td>
<td>80 x 80 x 256</td>
<td>6.25mb</td>
</tr>
<tr>
<td>Conv3_3</td>
<td>80 x 80 x 256</td>
<td>6.25mb</td>
</tr>
<tr>
<td>Conv4_1</td>
<td>40 x 40 x 512</td>
<td>3.125mb</td>
</tr>
<tr>
<td>Conv4_2</td>
<td>40 x 40 x 512</td>
<td>3.125mb</td>
</tr>
<tr>
<td>Conv4_2</td>
<td>40 x 40 x 512</td>
<td>3.125mb</td>
</tr>
<tr>
<td>Conv4_3</td>
<td>20 x 20 x 512</td>
<td>0.78125mb</td>
</tr>
<tr>
<td>Conv5_1</td>
<td>20 x 20 x 512</td>
<td>0.78125mb</td>
</tr>
<tr>
<td>Conv5_2</td>
<td>20 x 20 x 512</td>
<td>0.78125mb</td>
</tr>
<tr>
<td>Deconv1</td>
<td>40 x 40 x 256</td>
<td>1.5625mb</td>
</tr>
<tr>
<td>Deconv2</td>
<td>80 x 80 x 128</td>
<td>3.125mb</td>
</tr>
<tr>
<td>Deconv3</td>
<td>160 x 160 x 64</td>
<td>6.25mb</td>
</tr>
<tr>
<td>Deconv4</td>
<td>320 x 320 x 64</td>
<td>25mb</td>
</tr>
<tr>
<td>Deconv5</td>
<td>320 x 320 x 1</td>
<td>0.390625mb</td>
</tr>
</tbody>
</table>
Optimization 1(a): Pre-allocate the max buffer size required

<table>
<thead>
<tr>
<th>Module</th>
<th>Dimensions</th>
<th>Memory</th>
</tr>
</thead>
<tbody>
<tr>
<td>Conv1_1</td>
<td>320 x 320 x 64</td>
<td>25mb</td>
</tr>
<tr>
<td>Conv1_2</td>
<td>320 x 320 x 64</td>
<td>25mb</td>
</tr>
<tr>
<td>Conv2_1</td>
<td>160 x 160 x 128</td>
<td>12.5mb</td>
</tr>
<tr>
<td>Conv2_2</td>
<td>160 x 160 x 128</td>
<td>12.5mb</td>
</tr>
<tr>
<td>Conv3_1</td>
<td>80 x 80 x 256</td>
<td>6.25mb</td>
</tr>
<tr>
<td>Conv3_2</td>
<td>80 x 80 x 256</td>
<td>6.25mb</td>
</tr>
<tr>
<td>Conv3_3</td>
<td>80 x 80 x 256</td>
<td>6.25mb</td>
</tr>
<tr>
<td>Conv4_1</td>
<td>40 x 40 x 512</td>
<td>3.125mb</td>
</tr>
<tr>
<td>Conv4_2</td>
<td>40 x 40 x 512</td>
<td>3.125mb</td>
</tr>
<tr>
<td>Conv4_3</td>
<td>20 x 20 x 512</td>
<td>0.78125mb</td>
</tr>
<tr>
<td>Conv5_1</td>
<td>20 x 20 x 512</td>
<td>0.78125mb</td>
</tr>
<tr>
<td>Conv5_2</td>
<td>20 x 20 x 512</td>
<td>0.78125mb</td>
</tr>
<tr>
<td>Deconv1</td>
<td>40 x 40 x 256</td>
<td>1.5625mb</td>
</tr>
<tr>
<td>Deconv2</td>
<td>80 x 80 x 128</td>
<td>3.125mb</td>
</tr>
<tr>
<td>Deconv3</td>
<td>160 x 160 x 64</td>
<td>6.25mb</td>
</tr>
<tr>
<td>Deconv4</td>
<td>320 x 320 x 64</td>
<td>25mb</td>
</tr>
<tr>
<td>Deconv5</td>
<td>320 x 320 x 1</td>
<td>0.390625mb</td>
</tr>
</tbody>
</table>
Optimization 1: Better memory management

- Pre-allocate the max buffer size required for the workspace

```
Optimization 1: Better memory management

- Pre-allocate the max buffer size required for the workspace
```

```
Conv1_1
Conv1_2
Conv1_3
Pool1
Conv2_1
```

```
Memory Pool
2x Largest Tensor
```

```
A 25mb
B 25mb
A
B
A
B
```
Optimization 1: Better memory management

- Pre-allocate the max buffer size required for the workspace
- Carefully choose convolution algorithm for performance and memory requirements
  - FFT – Fast, but requires a lot of device workspace memory
  - GEMM – In place general matrix multiply
  - Winograd – fast, but unstable for large filter sizes.
Optimization 1: Better memory management

- Pre-allocate the max buffer size required for the workspace
- Carefully choose convolution algorithm for performance and memory requirements
  - FFT – Fast, but requires a lot of device workspace memory
  - GEMM – In place general matrix multiply
  - Winograd – fast, but unstable for large filter sizes.
- Convolution algorithm chosen on a per layer basis
  - According to per layer constraints
Optimization 1: Better memory management

- Pre-allocate the max buffer size required for the workspace
- Carefully choose convolution algorithm for performance and memory requirements
  - FFT – Fast, but requires a lot of device workspace memory
  - GEMM – In place general matrix multiply
  - Winograd – fast, but unstable for large filter sizes.

- Convolution algorithm chosen on a per layer basis
  - According to per layer constraints

- Share max per layer workspace memory between all layers
- Re-use the buffer for the computation of each layer
Optimization 1(b): Load the entire model in one go

Weights + biases ~ 87 MB
Optimization 1(c): Choose optimal convolution algorithm

Conv1_1[3 x 3 x 4 x 64]  Memory: 0.59mb  CUDNN_CONVOLUTION_FWD_ALGO_IMPLICIT_PRECOMP_GEMM
Conv1_2[3 x 3 x 64 x 64]  Memory: 0.25mb  CUDNN_CONVOLUTION_FWD_ALGO_WINOGRAD
Conv2_1[3 x 3 x 64 x 128]  Memory: 0.50mb  CUDNN_CONVOLUTION_FWD_ALGO_WINOGRAD
Conv2_2[3 x 3 x 128 x 128]  Memory: 1mb  CUDNN_CONVOLUTION_FWD_ALGO_WINOGRAD
Conv3_1[3 x 3 x 128 x 256]  Memory: 2mb  CUDNN_CONVOLUTION_FWD_ALGO_WINOGRAD
Conv3_2[3 x 3 x 256 x 256]  Memory: 4mb  CUDNN_CONVOLUTION_FWD_ALGO_WINOGRAD
Conv3_3[3 x 3 x 256 x 256]  Memory: 4mb  CUDNN_CONVOLUTION_FWD_ALGO_WINOGRAD
Conv4_1[3 x 3 x 256 x 512]  Memory: 8mb  CUDNN_CONVOLUTION_FWD_ALGO_WINOGRAD
Conv4_2[3 x 3 x 512 x 512]  Memory: 16mb  CUDNN_CONVOLUTION_FWD_ALGO_WINOGRAD
Conv4_3[3 x 3 x 512 x 512]  Memory: 16mb  CUDNN_CONVOLUTION_FWD_ALGO_WINOGRAD
Conv5_1[3 x 3 x 512 x 512]  Memory: 0mb  CUDNN_CONVOLUTION_FWD_ALGO_IMPLICIT_GEMM
Conv5_2[3 x 3 x 512 x 512]  Memory: 0mb  CUDNN_CONVOLUTION_FWD_ALGO_IMPLICIT_GEMM
Deconv5[5 x 5 x 512 x 512]  Memory: 0mb  CUDNN_CONVOLUTION_FWD_ALGO_IMPLICIT_PRECOMP_GEMM
Deconv4[5 x 5 x 512 x 256]  Memory: 0mb  CUDNN_CONVOLUTION_FWD_ALGO_IMPLICIT_GEMM
Deconv3[5 x 5 x 256 x 128]  Memory: 0.04mb  CUDNN_CONVOLUTION_FWD_ALGO_IMPLICIT_PRECOMP_GEMM
Deconv2[5 x 5 x 128 x 64]  Memory: 0.15mb  CUDNN_CONVOLUTION_FWD_ALGO_IMPLICIT_PRECOMP_GEMM
Deconv1[5 x 5 x 64 x 64]  Memory: 0.59mb  CUDNN_CONVOLUTION_FWD_ALGO_IMPLICIT_PRECOMP_GEMM
## Optimization 1: Results

### Performance

<table>
<thead>
<tr>
<th>Image size</th>
<th>Caffe</th>
<th>Optimization 1</th>
<th>%Reduction</th>
</tr>
</thead>
<tbody>
<tr>
<td>320 x 320</td>
<td>210ms</td>
<td>20ms</td>
<td>95%</td>
</tr>
<tr>
<td>640 x 640</td>
<td>540ms</td>
<td>68ms</td>
<td>87.4%</td>
</tr>
<tr>
<td>960 x 960</td>
<td>1.04sec</td>
<td>153ms</td>
<td>85%</td>
</tr>
<tr>
<td>1280 x 1280</td>
<td>cannot run</td>
<td>261ms</td>
<td>-</td>
</tr>
</tbody>
</table>

### Memory

<table>
<thead>
<tr>
<th>Image size</th>
<th>Caffe</th>
<th>Optimization 1</th>
<th>%Reduction</th>
</tr>
</thead>
<tbody>
<tr>
<td>320 x 320</td>
<td>588mb</td>
<td>159mb</td>
<td>73%</td>
</tr>
<tr>
<td>640 x 640</td>
<td>4113mb</td>
<td>323mb</td>
<td>92%</td>
</tr>
<tr>
<td>960 x 960</td>
<td>8778mb</td>
<td>643mb</td>
<td>92.7%</td>
</tr>
<tr>
<td>1280 x 1280</td>
<td>Cannot run</td>
<td>977mb</td>
<td>-</td>
</tr>
</tbody>
</table>
Optimization 2: Use FP16 instead of FP32

- All the weights for convolution and de-convolution were converted to float16.
  - Volta has hardware for FAST fp16 – TRUE_HALF_CONFIG
  - On Pascal and below, store in fp16 but process in fp32 – PSEUDO_HALF_CONFIG

- Pooling and un-pooling indices were stored with 8 bits (for 2 x 2 kernel size).
Optimization 2: Use FP16 instead of FP32

- Results slightly different -> retrain with FP16
Optimization 2: Results

### Performance

<table>
<thead>
<tr>
<th>Image size</th>
<th>FP32</th>
<th>FP16</th>
<th>%Reduction</th>
</tr>
</thead>
<tbody>
<tr>
<td>320 x 320</td>
<td>20ms</td>
<td>13ms</td>
<td>35%</td>
</tr>
<tr>
<td>640 x 640</td>
<td>68ms</td>
<td>35ms</td>
<td>50.7%</td>
</tr>
<tr>
<td>960 x 960</td>
<td>153ms</td>
<td>87ms</td>
<td>43.1%</td>
</tr>
<tr>
<td>1280 x 1280</td>
<td>261ms</td>
<td>153ms</td>
<td>41.4%</td>
</tr>
</tbody>
</table>

### Memory

<table>
<thead>
<tr>
<th>Image size</th>
<th>FP32</th>
<th>FP16</th>
<th>%Reduction</th>
</tr>
</thead>
<tbody>
<tr>
<td>320 x 320</td>
<td>159mb</td>
<td>99mb</td>
<td>37.7%</td>
</tr>
<tr>
<td>640 x 640</td>
<td>323mb</td>
<td>210mb</td>
<td>35%</td>
</tr>
<tr>
<td>960 x 960</td>
<td>643mb</td>
<td>361mb</td>
<td>43.8%</td>
</tr>
<tr>
<td>1280 x 1280</td>
<td>977mb</td>
<td>559mb</td>
<td>42.8%</td>
</tr>
</tbody>
</table>
Optimization 3: Use Tensor Core on Volta

- Tensor Core performs half matrix multiply accumulate (HMMA)
- cuDNN 7.0 has optimizations for HMMA
- Convolutions must use
  - CUDNN_CONVOLUTION_FWD_ALGO_WINOGRAD_NONFUSED
  - CUDNN_CONVOLUTION_FWD_ALGO_IMPLICIT_PRECOMP_GEMM
    - X,y,w tensors must be FP16
    - Input and output filter maps must be multiple of 8 for alignment
Optimization 3: Use Tensor Core on Volta

Mixed Precision Matrix Math
4x4 matrices

\[
D = AB + C
\]
# Optimization 3: Results

## Performance

<table>
<thead>
<tr>
<th>Image size</th>
<th>FP16</th>
<th>FP16(with TensorCore)</th>
<th>%Reduction</th>
</tr>
</thead>
<tbody>
<tr>
<td>320 x 320</td>
<td>13ms</td>
<td>5ms</td>
<td>61.5%</td>
</tr>
<tr>
<td>640 x 640</td>
<td>35ms</td>
<td>15ms</td>
<td>57.4%</td>
</tr>
<tr>
<td>960 x 960</td>
<td>87ms</td>
<td>32ms</td>
<td>63.2%</td>
</tr>
<tr>
<td>1280 x 1280</td>
<td>153ms</td>
<td>53ms</td>
<td>65.3%</td>
</tr>
</tbody>
</table>

## Memory

<table>
<thead>
<tr>
<th>Image size</th>
<th>FP16</th>
<th>FP16(with TensorCore)</th>
<th>%Increase</th>
</tr>
</thead>
<tbody>
<tr>
<td>320 x 320</td>
<td>99mb</td>
<td>111mb</td>
<td>12%</td>
</tr>
<tr>
<td>640 x 640</td>
<td>210mb</td>
<td>262mb</td>
<td>24.7%</td>
</tr>
<tr>
<td>960 x 960</td>
<td>361mb</td>
<td>533mb</td>
<td>47.6%</td>
</tr>
<tr>
<td>1280 x 1280</td>
<td>559mb</td>
<td>914mb</td>
<td>63.5%</td>
</tr>
</tbody>
</table>
Optimization 4: Network Fusing

- Original Caffe implementation used 2 networks:
  - Coarse matting: VGG16 autoencoder
  - Fine matting: shallow 4 layer cnn
    - Input A: Original mean subtracted RGB as per coarse network
    - Input B: Output of coarse network scaled back to 0:255 and mean subtracted.
Optimization 4: Network Fusing

- Original Caffe implementation used 2 networks:
  - Coarse matting: VGG16 autoencoder
  - Fine matting: shallow 4 layer CNN
    - Input A: Original mean subtracted RGB as per course network
    - Input B: Output of coarse network scaled back to 0:255 and mean subtracted.

- Causes unnecessary driver overhead copying to and from the CPU
- Pre and post processing can be done on the GPU
Optimization 4: Network Fusing

- Treat both networks as a single network.
- Keep mean subtracted RGB on GPU.
- Treat coarse output post processing as a custom network layer.
Optimization 5: Layer fusing

- Some layer operations can be fused
  - Convolution
  - Bias Add
  - Activation (eg. Relu)

- Advantages
  - Reduces kernel launch overhead
  - Some arithmetic operations can be combined (eg. FMAD)

- cuDNN as a combined version of Convolution+Bias+Activation
  - \texttt{cudnnStatus\_t cudnnConvolutionBiasActivationForward(...)}

- TensorRT will find the best fused configuration at serialization time
Topics

- Introduction to Matting
- Deep Matting vs Photoshop Matting
- Deployment Challenges
- Optimization
- Results and Demo
- Conclusion
Caffe vs Caffe2 vs Our Optimizations

Performance

<table>
<thead>
<tr>
<th>Resolution</th>
<th>Caffe</th>
<th>Caffe2</th>
<th>Cudnn Optimized</th>
</tr>
</thead>
<tbody>
<tr>
<td>320 x 320</td>
<td>210</td>
<td>76</td>
<td>5</td>
</tr>
<tr>
<td>640 x 640</td>
<td>540</td>
<td>195</td>
<td>15</td>
</tr>
<tr>
<td>960 x 960</td>
<td>1040</td>
<td>375</td>
<td>32</td>
</tr>
<tr>
<td>1280 x 1280</td>
<td>605</td>
<td>0</td>
<td>53</td>
</tr>
</tbody>
</table>
Caffe vs Caffe2 vs Our Optimizations

Memory

<table>
<thead>
<tr>
<th>Size</th>
<th>Caffe</th>
<th>Caffe2</th>
<th>Cudnn optimized</th>
</tr>
</thead>
<tbody>
<tr>
<td>320 x 320</td>
<td>588</td>
<td>235</td>
<td>111</td>
</tr>
<tr>
<td>640 x 640</td>
<td>4113</td>
<td>1645</td>
<td>262</td>
</tr>
<tr>
<td>960 x 960</td>
<td>8778</td>
<td>3582</td>
<td>533</td>
</tr>
<tr>
<td>1280 x 1280</td>
<td>6369</td>
<td>0</td>
<td>914</td>
</tr>
</tbody>
</table>
Summary

✓ Do better memory management
✓ Use optimal algorithm for convolution based on image and filter size (gemm/fft/winograd)
✓ Do inference at lower precision (fp16 or uint8) if possible
✓ Use hardware-specific optimizations (ex. HMMA on TensorCore)
✓ Do layer fusion
Conclusion and Future Work

- Explore WindowsML/DirectML for optimized cross platform inference
- Try TensorRT, Nvidia’s latest solution for optimized high performance inference