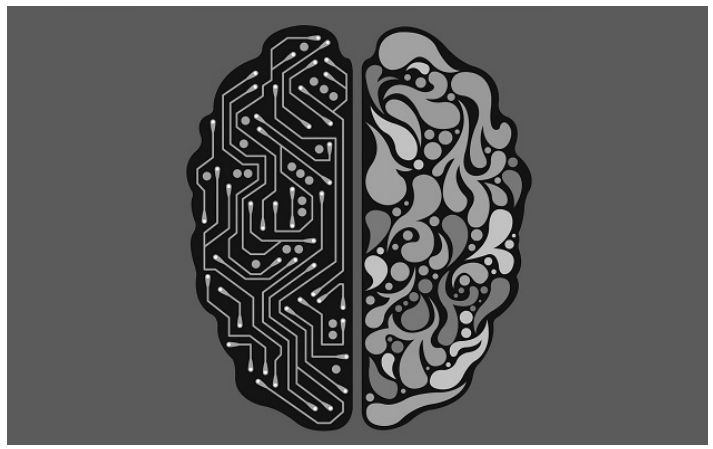
Projects Status Report

Deep Learning for Healthcare Imaging

Application support group meeting, Garching bei München, Mon 30.09.2019

Giuseppe Di Bernardo[1]

Data Analytics member @MPCDF



Slides: https://giuseppe82.github.io/myJupyterSlides/ (https://giuseppe82.github.io/myJupyterSlides/)

Agenda

1. First Study Case: Automatic Segmentation of 3D Medical Images

1. First Study Case: Automatic Segmentation of 3D Medical Images
- Scalable high performance for Deep Learning **Inferencing**

1. First Study Case: Automatic Segmentation of 3D Medical Images
- Scalable high performance for Deep Learning **Inferencing**
    - The Challenge

1. First Study Case: Automatic Segmentation of 3D Medical Images
- Scalable high performance for Deep Learning **Inferencing**
    - The Challenge
    - Our Tile-based Solution

1. First Study Case: Automatic Segmentation of 3D Medical Images
    - Scalable high performance for Deep Learning **Inferencing**
        - The Challenge
        - Our Tile-based Solution
2. Second Study Case: Denoising 3D Medical Images

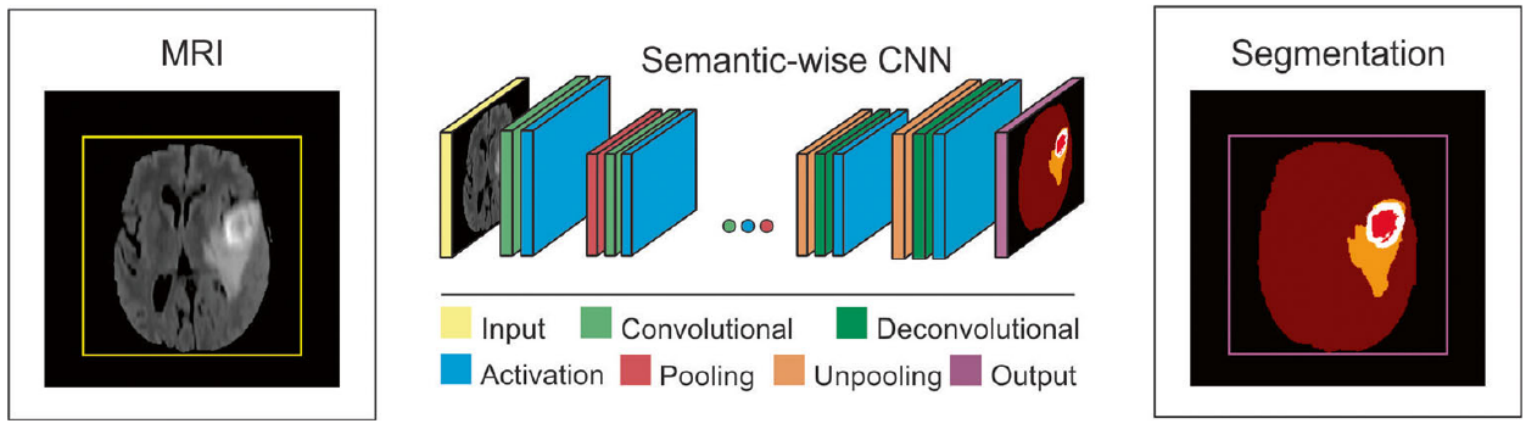
1. First Study Case: Automatic Segmentation of 3D Medical Images
    - Scalable high performance for Deep Learning **Inferencing**
        - The Challenge
        - Our Tile-based Solution
2. Second Study Case: Denoising 3D Medical Images
    - Scalable high performance for Deep Learning **Training**

1. First Study Case: Automatic Segmentation of 3D Medical Images
    - Scalable high performance for Deep Learning **Inferencing**
        - The Challenge
        - Our Tile-based Solution
2. Second Study Case: Denoising 3D Medical Images
    - Scalable high performance for Deep Learning **Training**
        - The Challenge

1. First Study Case: Automatic Segmentation of 3D Medical Images
    - Scalable high performance for Deep Learning **Inferencing**
        - The Challenge
        - Our Tile-based Solution
2. Second Study Case: Denoising 3D Medical Images
    - Scalable high performance for Deep Learning **Training**
        - The Challenge
        - Model Parallelism and Mixed Precision

Automatic Segmentation of 3D Medical Images

*Our present knowledge of cortical structure is based on the analysis of physical 2D sections.[...] Now with the combination of novel 3D imaging techniques and advanced images analysis methods, such as **deep neural networks**, the study of the full three-dimensional structure of the brain is within reach (K. Thierbach (HBCS) et al. 2019, publication in progress)*



(figure from Z. Akkus et al. 2017: [Deep Learning for Brain MRI Segmentation: State of the Art and Future Directions (https://web.stanford.edu/group/rubinlab/pubs/Akkus-2017.pdf)](https://web.stanford.edu/group/rubinlab/pubs/Akkus-2017.pdf))

... in collaboration with MPI for HBCS (and Intel®)

(Thierbach, K., Scharf, N., Weiskopf, N.)

- Challenges
    - Manual analysis of 3D data is not feasible
    - Image artefacts difficult to handle for classical image processing algorithms
- Goals
    - Detection and segmentation of thousands of cells
    - Minimal labeling/annotation effort
    - Easily adaptable method for varying protocols
- Solution(s)
    - Parallel tile-based inference (this talk)
    - (Intel® Distribution of OpenVINO™ toolkit)

MAX PLANCK INSTITUTE FOR HUMAN COGNITIVE AND BRAIN SCIENCES LEIPZIG

Inferencing

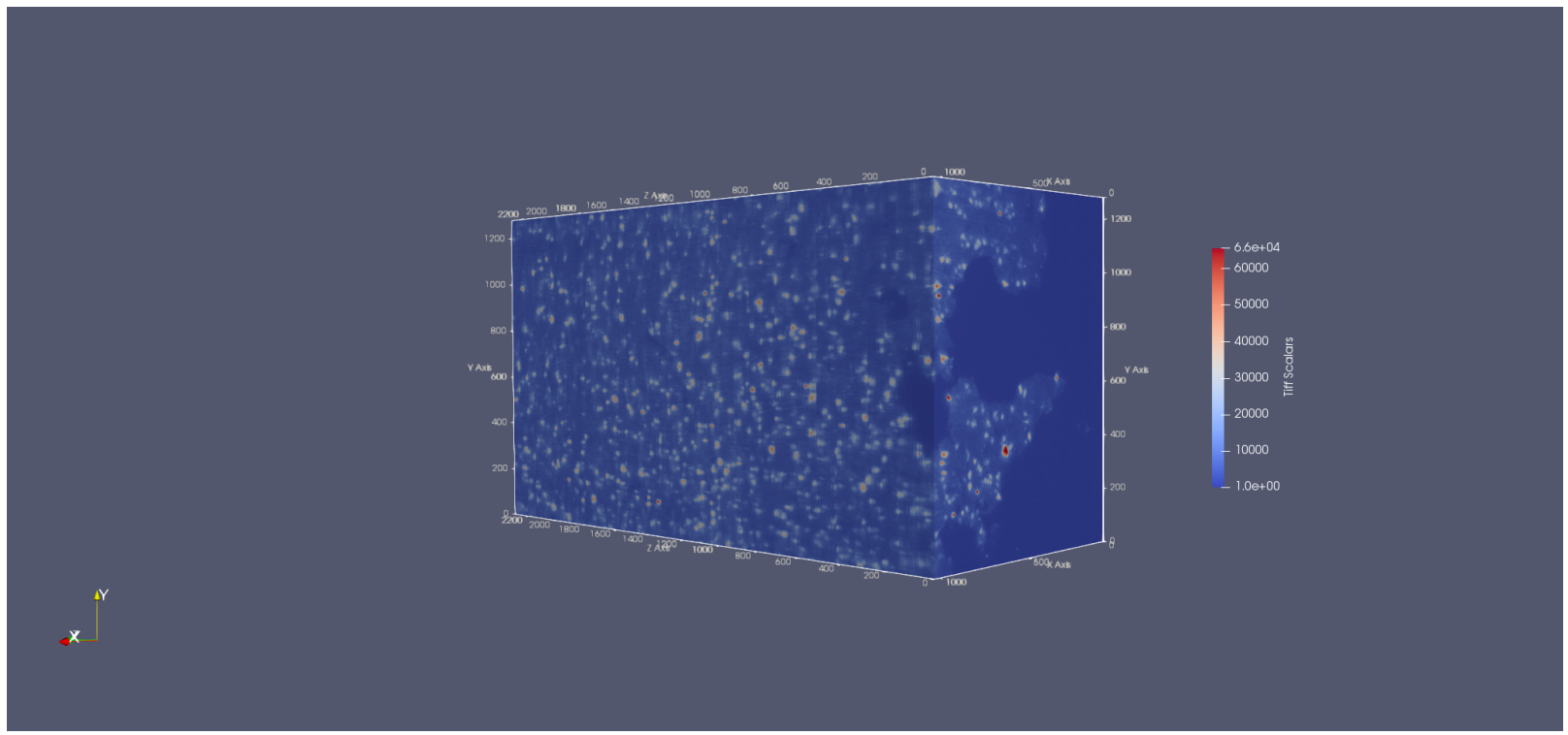
(from **Medium** (Stanford Healthcare for AI): [Don’t Just Scan This: Deep Learning Techniques for MRI (https://medium.com/stanford-ai-for-healthcare/dont-just-scan-this-deep-learning-techniques-for-mri-52610e9b7a85)](https://medium.com/stanford-ai-for-healthcare/dont-just-scan-this-deep-learning-techniques-for-mri-52610e9b7a85))

In [19]:
```
from IPython.display import Image
from IPython.core.display import HTML
from skimage import io # scikit-image for image processing
PATH="/home/gdib/Documents/PyCon-Italy"
IMAGE = PATH+ "/inputs/image_half.tif"
volume = io.imread(IMAGE)
print('dataset is a {}'.format(type(volume)))
print('... of type {}'.format(volume.dtype))
print('... and with shape {}'.format(volume.shape))
Image(filename="/home/gdib/Documents/applications-group-meeting/img/scikit.png", retina=True)
```

```
dataset is a <class 'numpy.ndarray'>
... of type uint16
... and with shape (2304, 1280, 1024, 1)
```

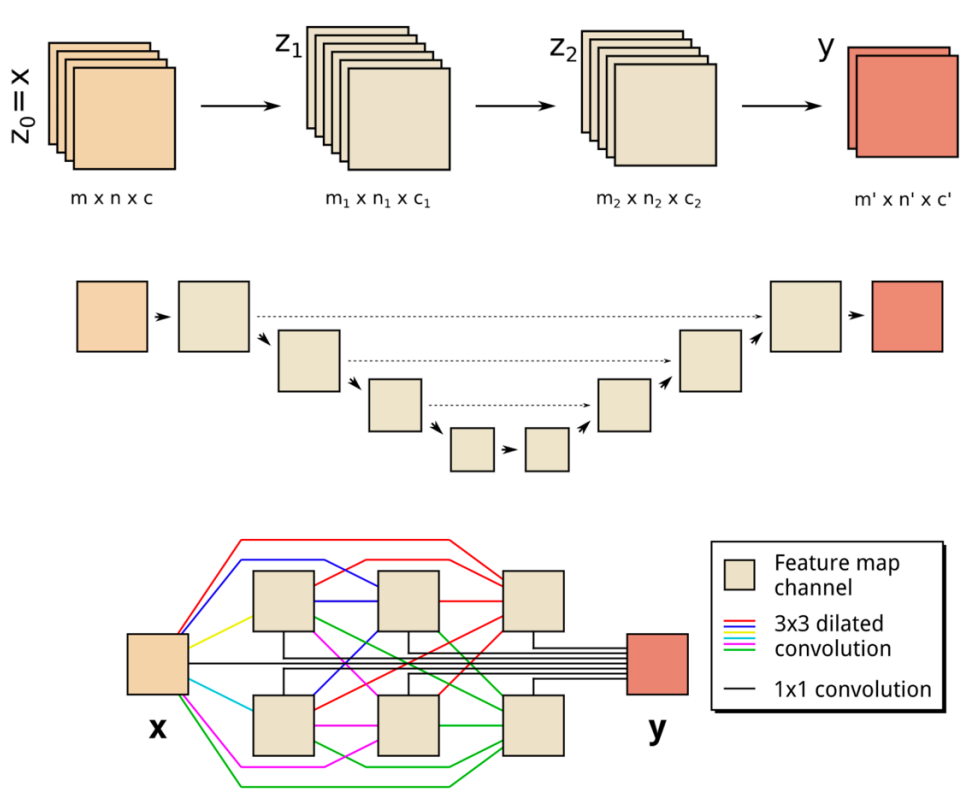
Out[19]:

Exploring MRI Volume Slices in Python



Mixed-Scale Dense Convolutional Neural Network

"In many scientific applications, tremendous manual labor is required to annotate and tag images — it can take weeks to produce a handful of carefully delineated images[...]. "Our goal was to develop a technique that learns from a very small data set."

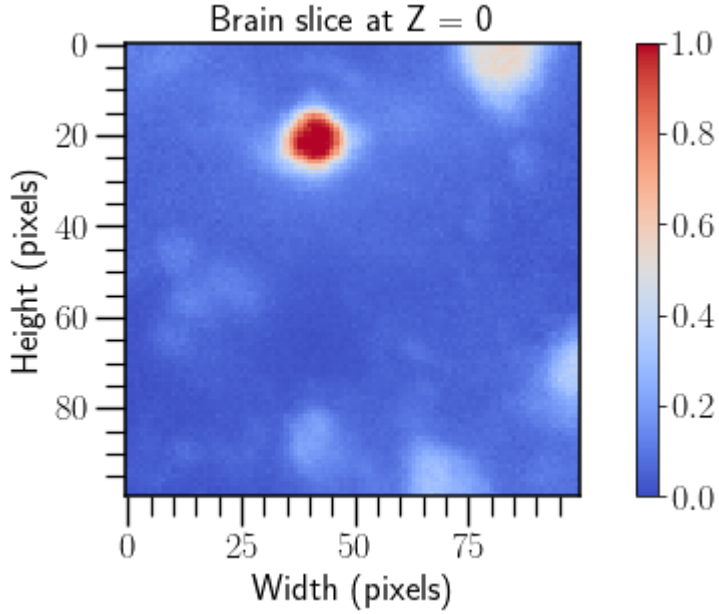


(Reference: *D.M.Pelt & J.A.Sethian, 2017*, **A mixed-scale dense convolutional neural networkfor image analysis** (https://www.pnas.org/content/pnas/early/2017/12/21/1715832114.full.pdf))

Computational Challenges

- MS-DNet Training:
    - Implemented with TensorFlow on small patches of 96^3
    - Model Size: 100K parameters (FCN for Semantic Segmentation)
- MS-DNet Inferencing:
    - Computational requirement: 16 Pflop
    - Memory requirement: 24 TB (with `TensorFlow`)
    - Standard DL frameworks do not (yet) provide *model-parallelism* at the inference step

In [8]:
```
img_spot = vol[0:100, 1000:1100, 750:850]
img_spot_YX = img_spot[0, :, :]
img_spot_YX = np.squeeze(img_spot_YX)
plot_slices(img_spot_YX)
```



Inferencing: our Tile-based Strategy

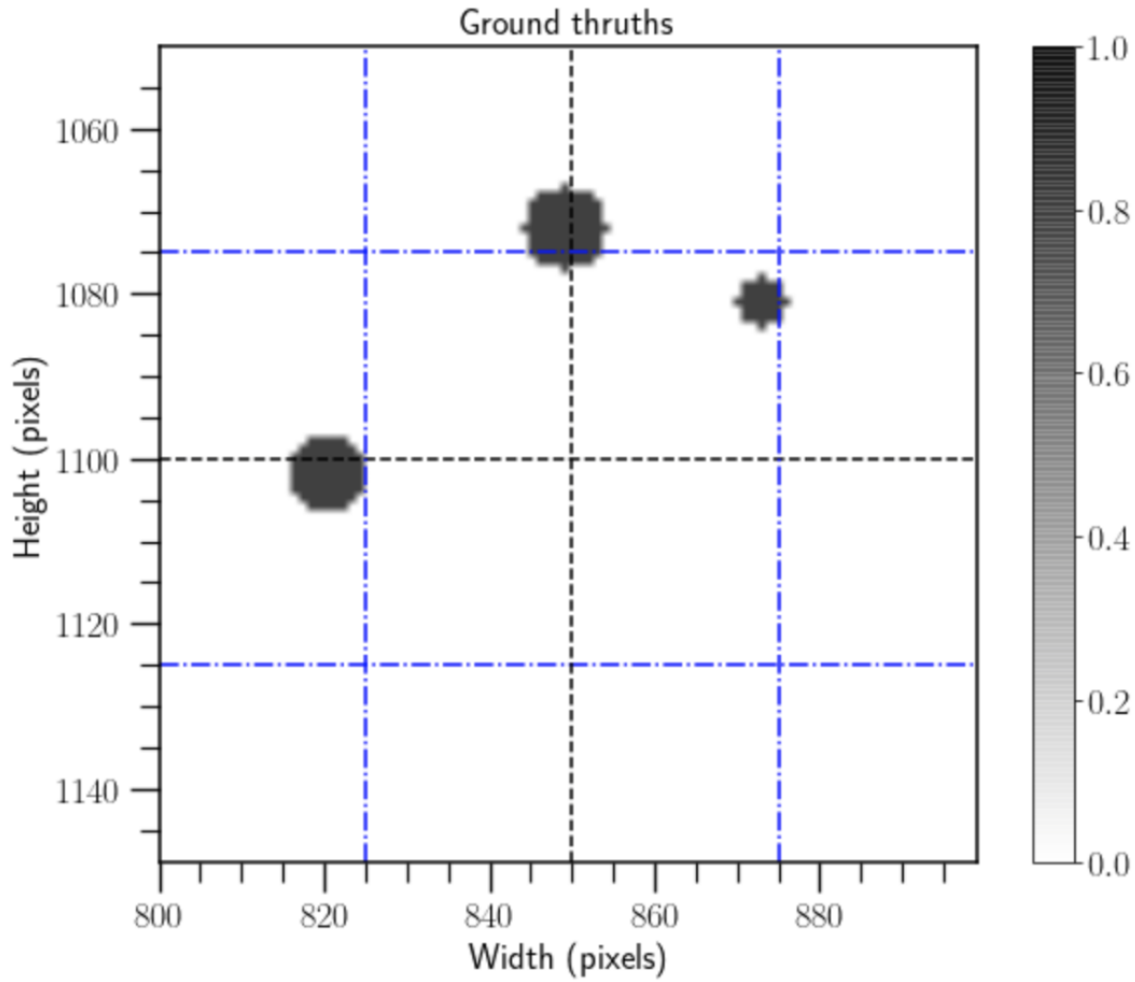
In [10]:
```
# Load image as Numpy array
labels = io.imread(GROUND_TRUTH)
parent = io.imread(PARENT)
child = io.imread(CHILD)
child_over = io.imread(CHILD_OVER)
```

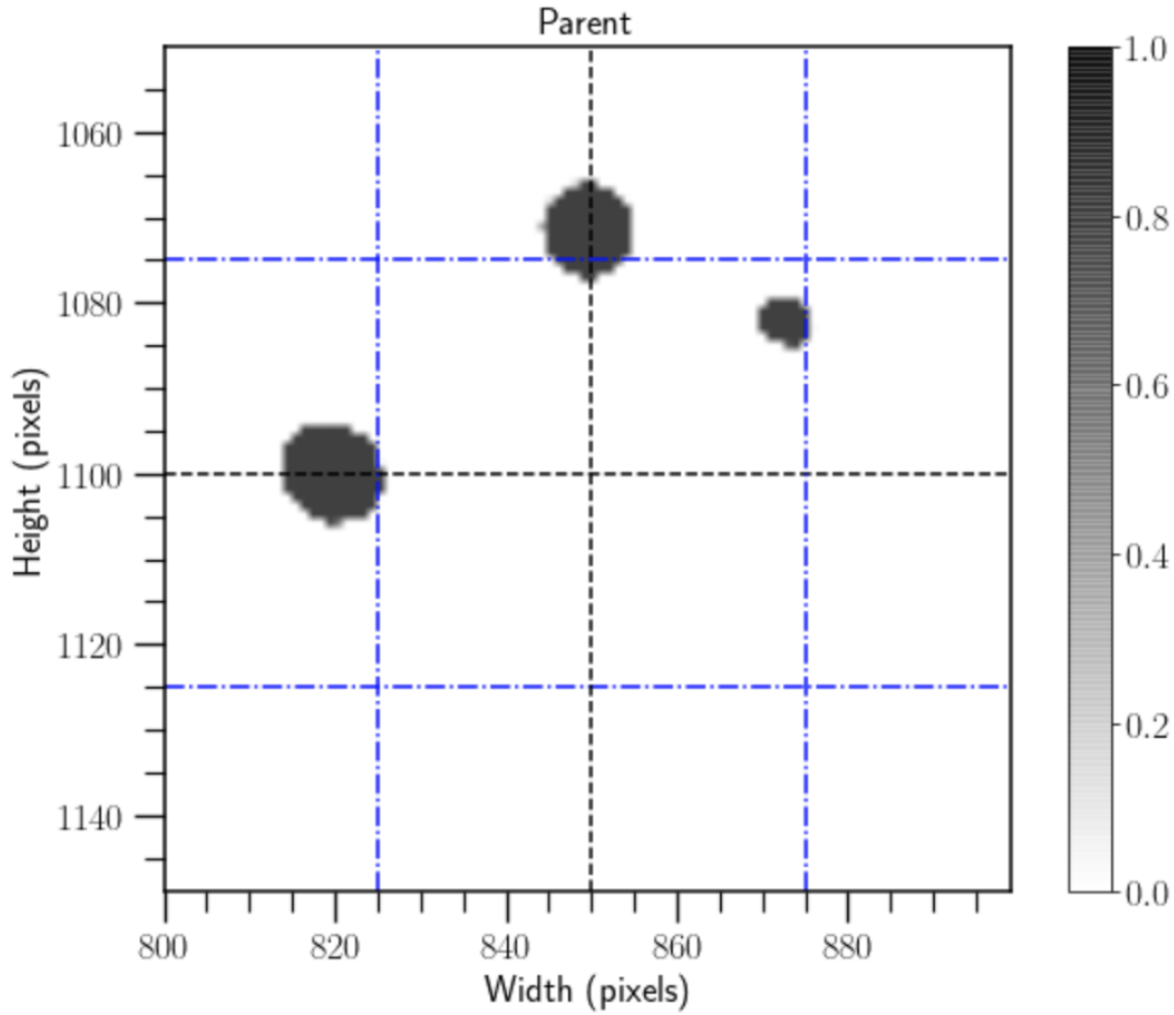
In [12]:
```
pixels_one = np.argwhere(labels_float == 1.0)
print("In the Ground Truths there are {} non-zero pixels".format(len(pixels_one)))
# Region selected to make inference
region = labels_float[0:100, 1050:1150, 800:900]
centroids_px = np.argwhere(region == 1.0)
centroids_px # in pixels
(zo, yo, xo) = (0, 1050, 800) # lower-left corner
```

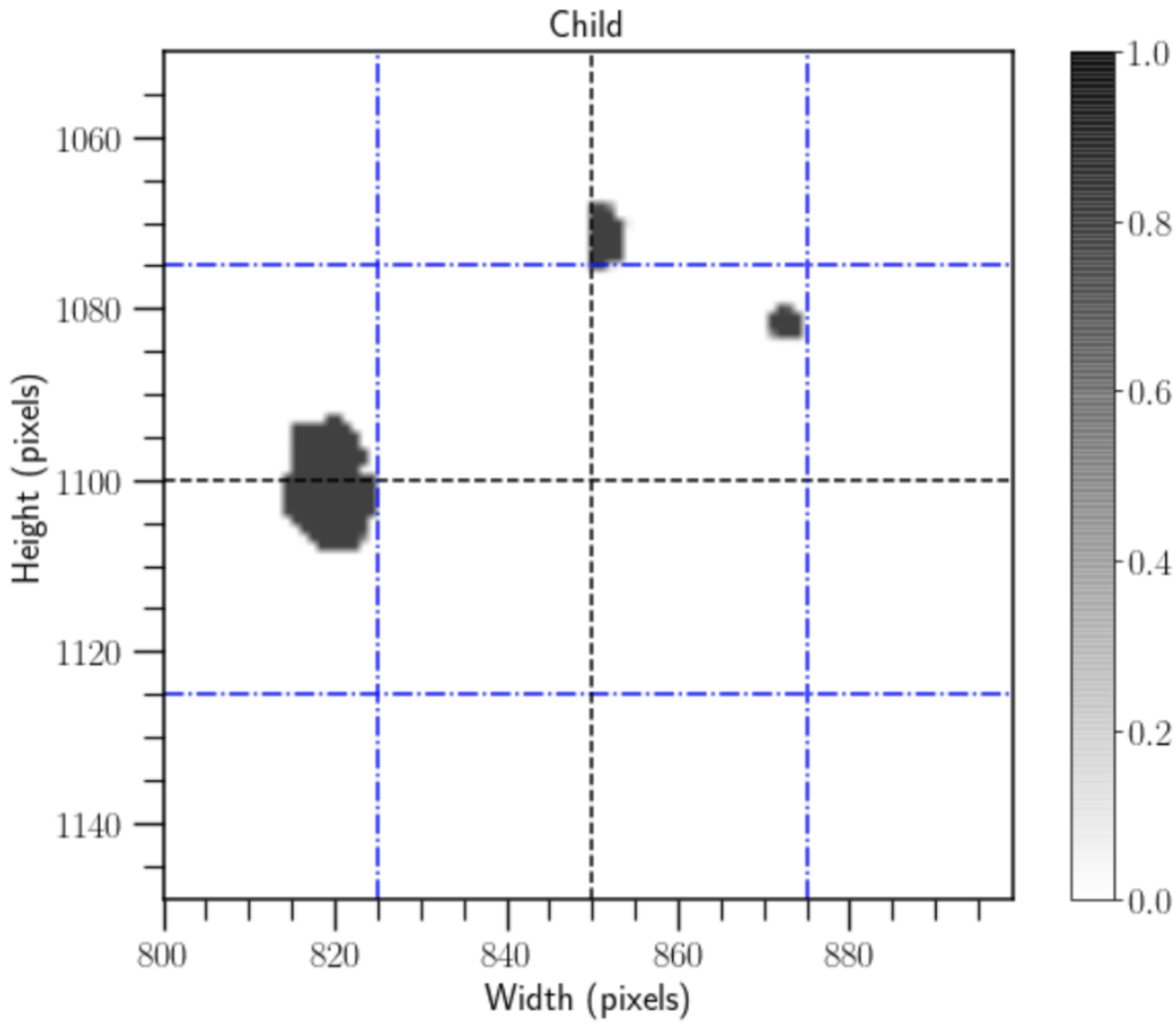
```
In the Ground Truths there are 13378 non-zero pixels
```

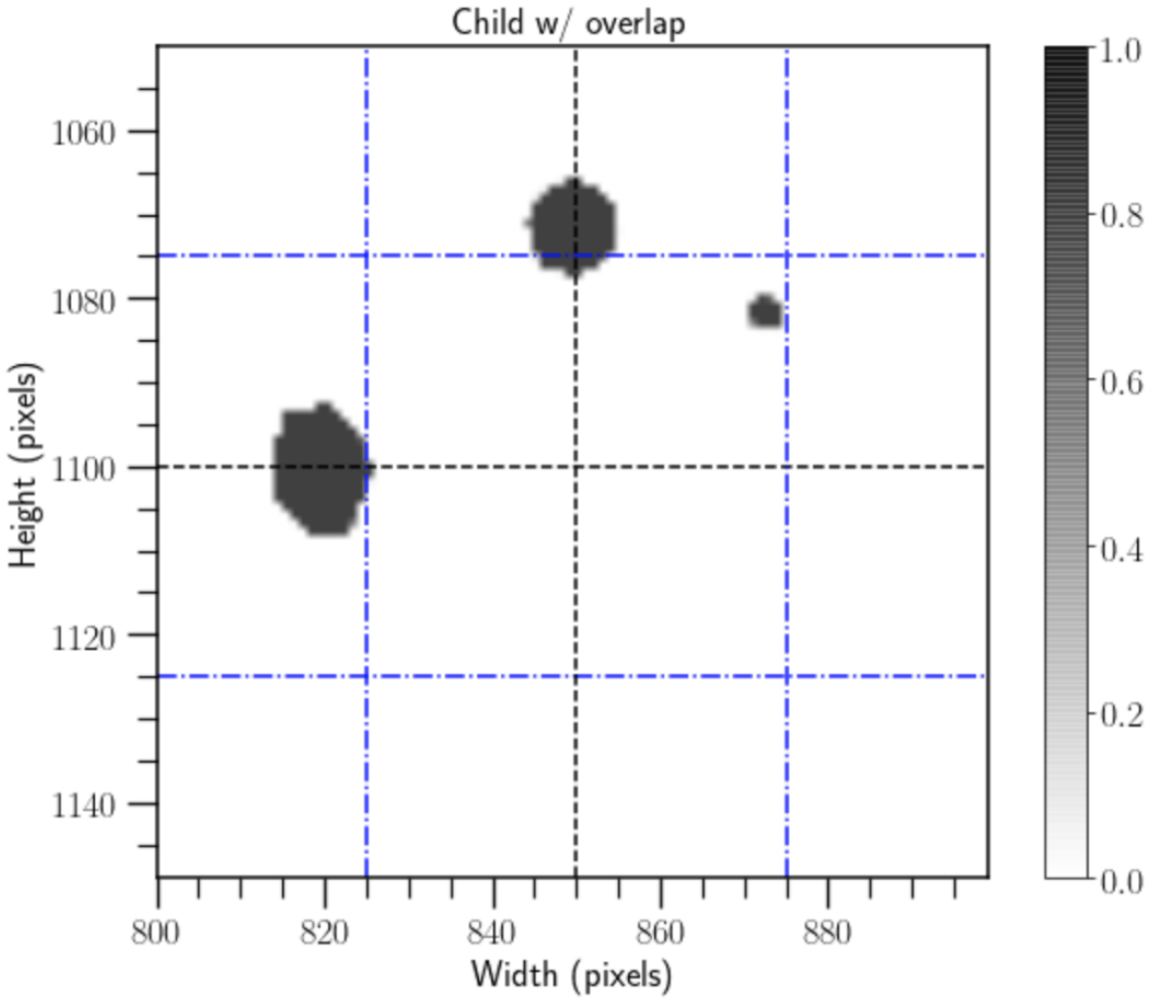
In [13]:
```
# in absolute coordinates
centroids = [centroids_px[i] + (zo, yo, xo) for i in range(len(centroids_px))]
centroids
```

Out[13]:
```
[array([   0, 1081,  873]),
 array([   4, 1072,  849]),
 array([   6, 1102,  820]),
 array([  38, 1104,  808]),
 array([  44, 1070,  827]),
 array([  44, 1122,  810]),
 array([  48, 1108,  885]),
 array([  60, 1105,  807]),
 array([  80, 1149,  876])]
```

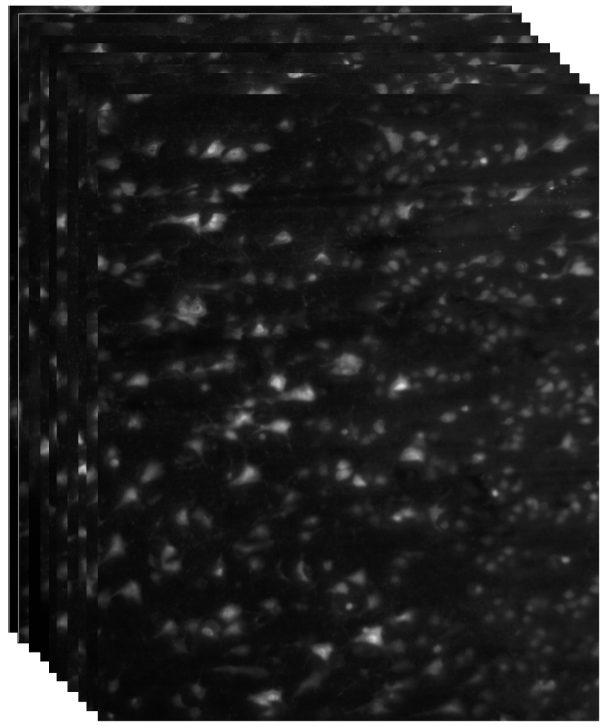

Ground thruths
1060
1080
1100
1120
1140
Height (pixels)
800
820
840
860
880
Width (pixels)
1.0
0.8
0.6
0.4
0.2
0.0


Parent
1060
1080
1100
1120
1140
Height (pixels)
800
820
840
860
880
Width (pixels)
1.0
0.8
0.6
0.4
0.2
0.0

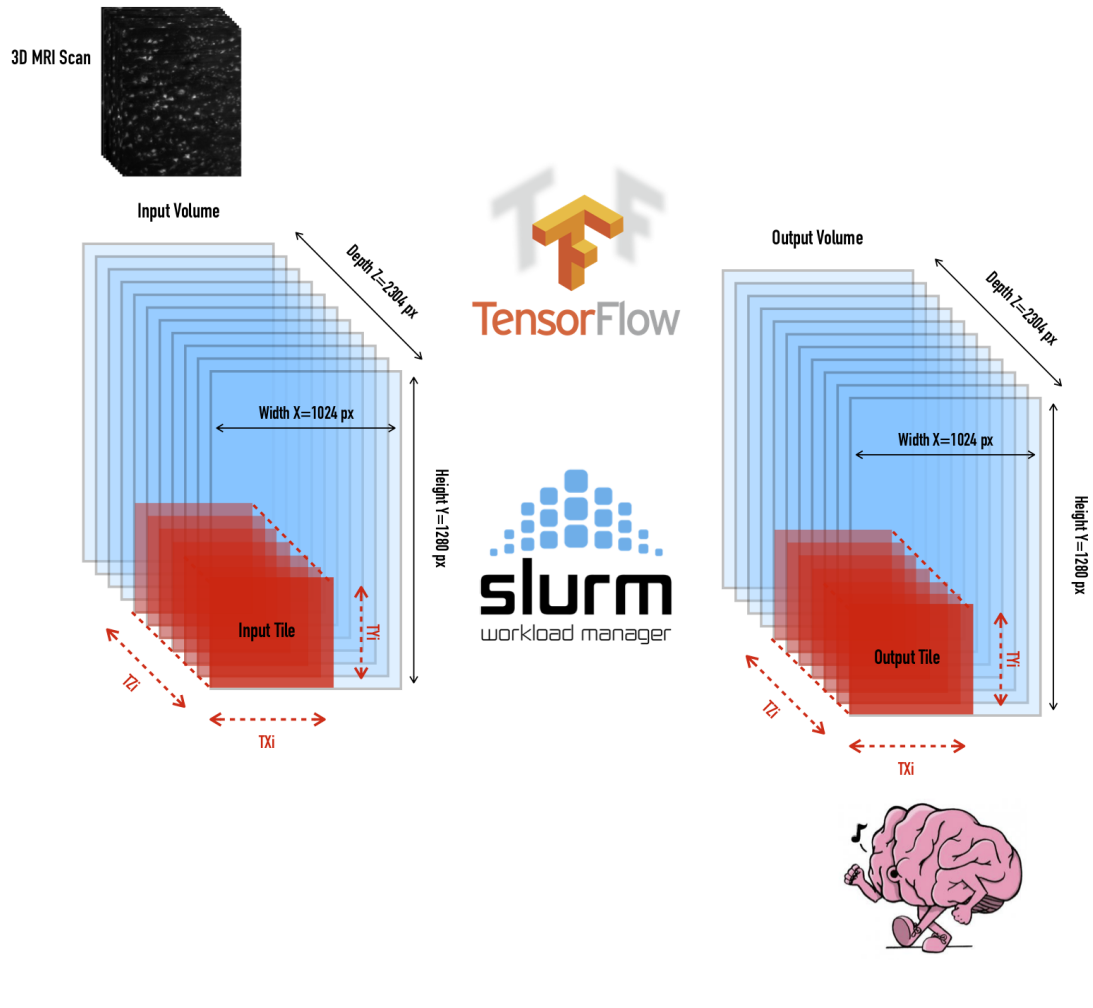

Child
Height (pixels)
1060
1080
1100
1120
1140
800
820
840
860
880
Width (pixels)
1.0
0.8
0.6
0.4
0.2
0.0


Child w/ overlap
Height (pixels)
1060
1080
1100
1120
1140
800
820
840
860
880
Width (pixels)
1.0
0.8
0.6
0.4
0.2
0.0

Tile-based strategy goes parallel on MPCDF's clusters

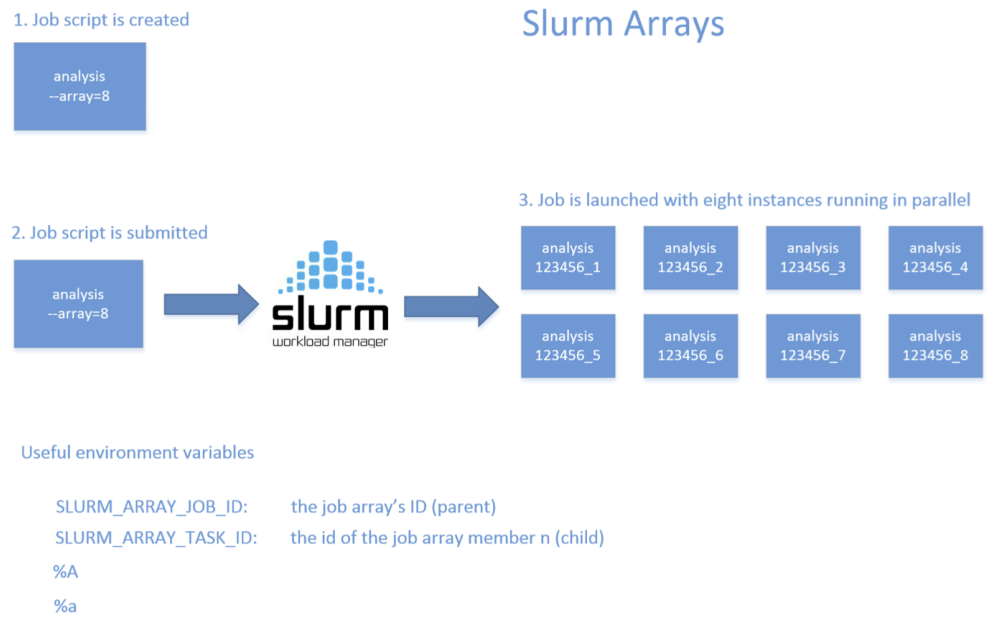


3D Tiling Solution Implemented on MPCDF's HPC Systems



Expanding Serial Analysis with SLURM Array

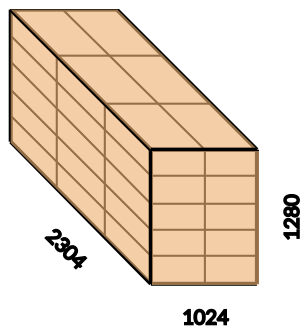
- Solution:
    - Submit a bunch of similar serial jobs each with their own job script
- Slurm **job arrays** can help!
    - Create many jobs from one job script! Even 100's of thousands!



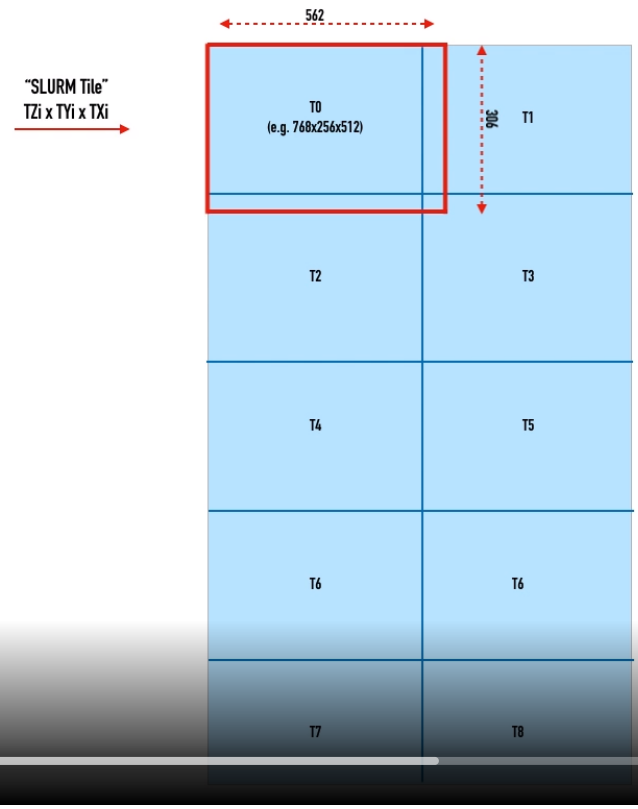
In [16]:
```
import dask.array as da
da.ones((2304,1280,1024), chunks=(768,256,512),dtype="float32")
```

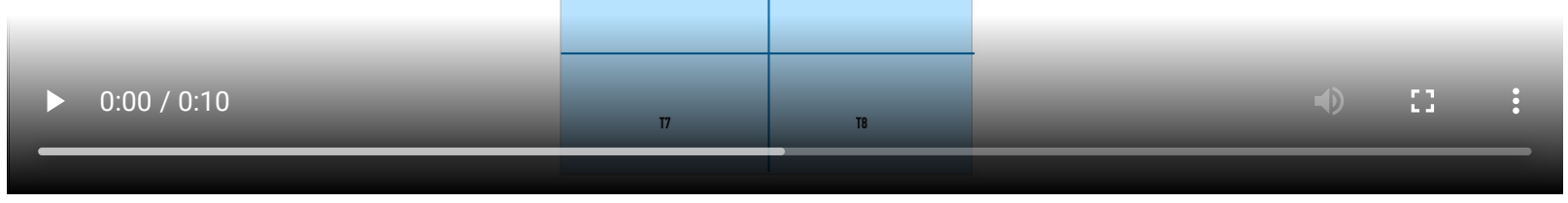
Out[16]:

	Array	Chunk
Bytes	12.08 GB	402.65 MB
Shape	(2304, 1280, 1024)	(768, 256, 512)
Count	30 Tasks	30 Chunks
Type	float32	numpy.ndarray



How it works

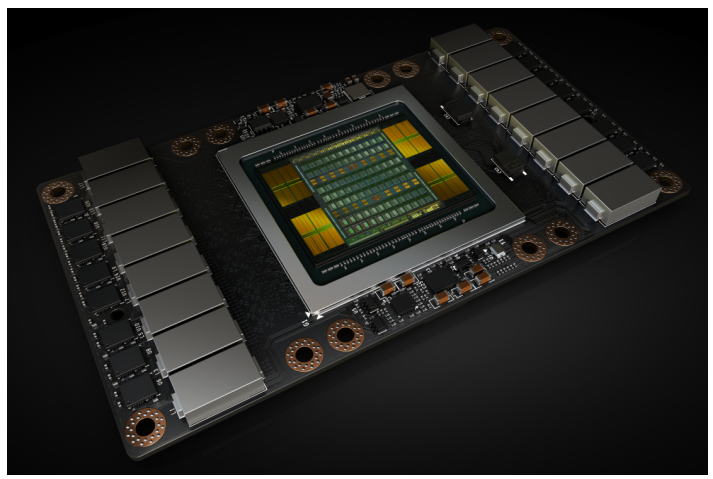

"SLURM Tile"
TZi x TYi x TXi
562
306
T0
(e.g. 768x256x512)
T1
T2
T3
T4
T5
T6
T6
T7
T8
e.g. @ page Z = 0


0:00 / 0:10

Results

30 SLURM Job Arrays, on 15 Computing Nodes

- 2 NVIDIA-V100-Tensor-Core GPUs, 32 GB (a job x each GPU): ~ 480 sec.
- 80 CPUs (hyperthreading enabled) Intel(R) Xeon(R) Gold 6126 CPU @ 2.60GHz: ~ 5000 sec.



Evaluation

Ground Truth

OpenVino

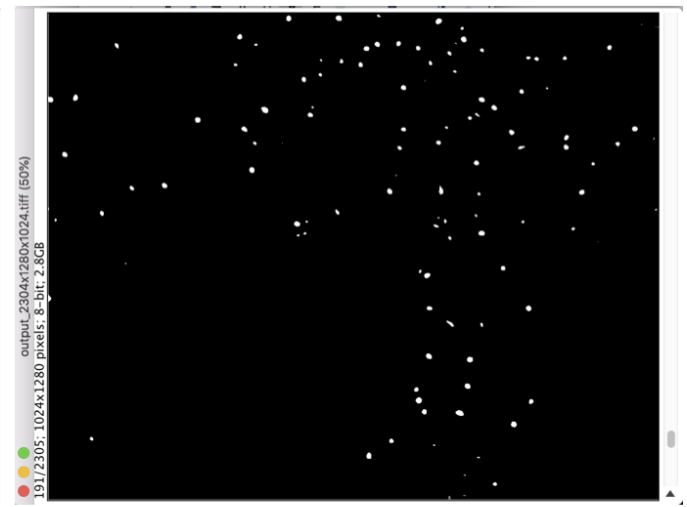
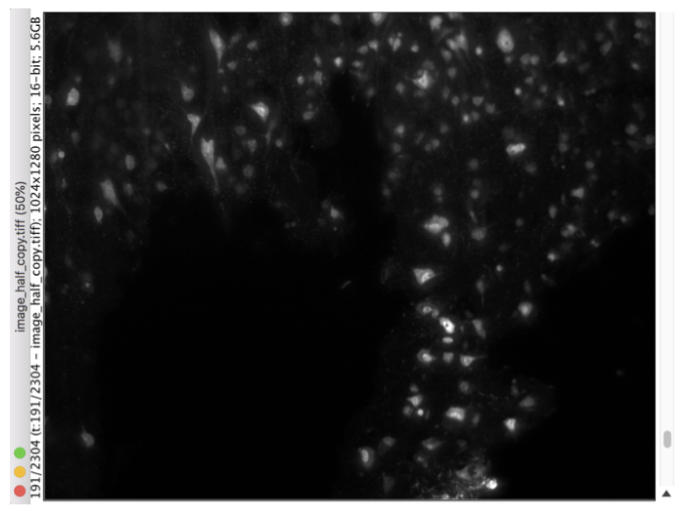
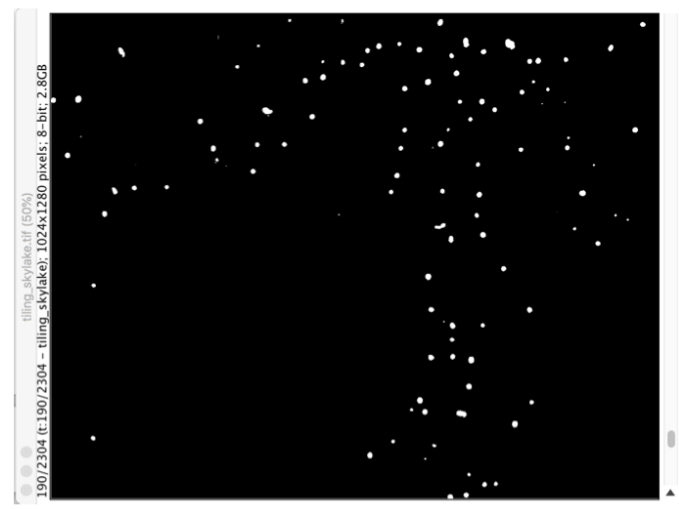


Image 2304x1280x1024



Tiling approach



Denosing of 3D Brain MRI Images with CNN

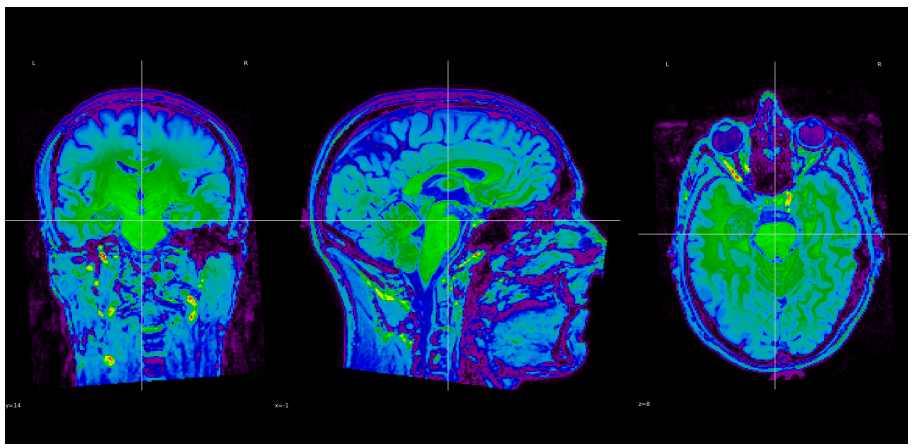
In collaboration with MPI for Human Cognitive and Brain Sciences (HBCS)
(Podranski, K., Scharf, N., Weiskopf, N.)

MAX PLANCK INSTITUTE FOR HUMAN COGNITIVE AND BRAIN SCIENCES LEIPZIG

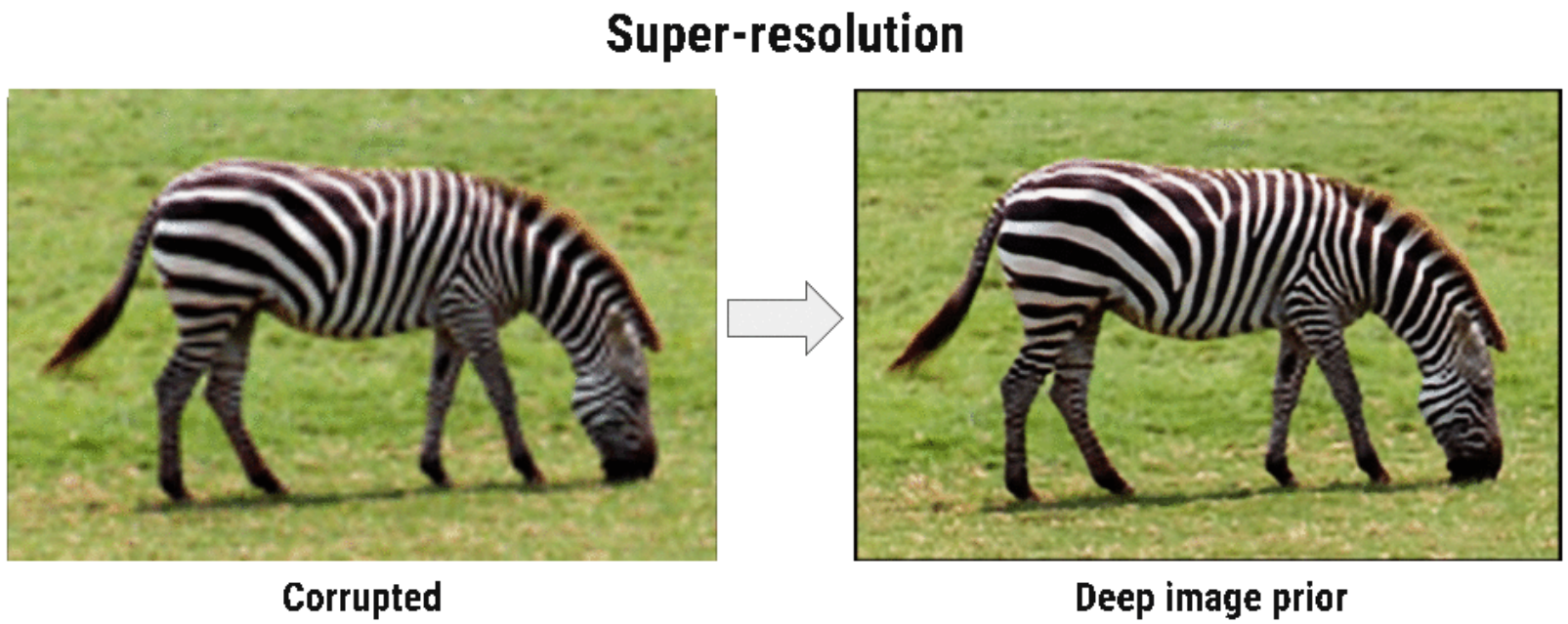
- Goal: Denoising and generic reconstruction: filter out noise from an image
- Method: given a noisy image x_0, we want to recover a clean image $x^* = f_{\theta^*}(z)$, after substituting the minimizer θ^* obtained using an optimizer such as gradient descent starting from a *random initialization of the parameters*

Neuroimaging in Python

- **NiBabel**: Read +/- write access to some common medical and neuroimaging file formats
- Each data set includes magnetic resonance imaging data in *Neuroimaging Informatics Technology Initiative* (NIFTI, `.nii.gz`) file format. E.g. a 256×256×150 MRI image from :http://brain-development.org/ixi-dataset (http://brain-development.org/ixi-dataset)



Deep Image Prior



(Web page: https://dmitryulyanov.github.io/deep_image_prior (https://dmitryulyanov.github.io/deep_image_prior))

Deep Image Prior Architecture

- Fully-convolutional architectures
    - Input: a random code tensor/vector $z \in \mathbb{R}^{C' \times W \times H}$
    - Output: image $x = f_\theta(z) \in \mathbb{R}^{3 \times W \times H}$

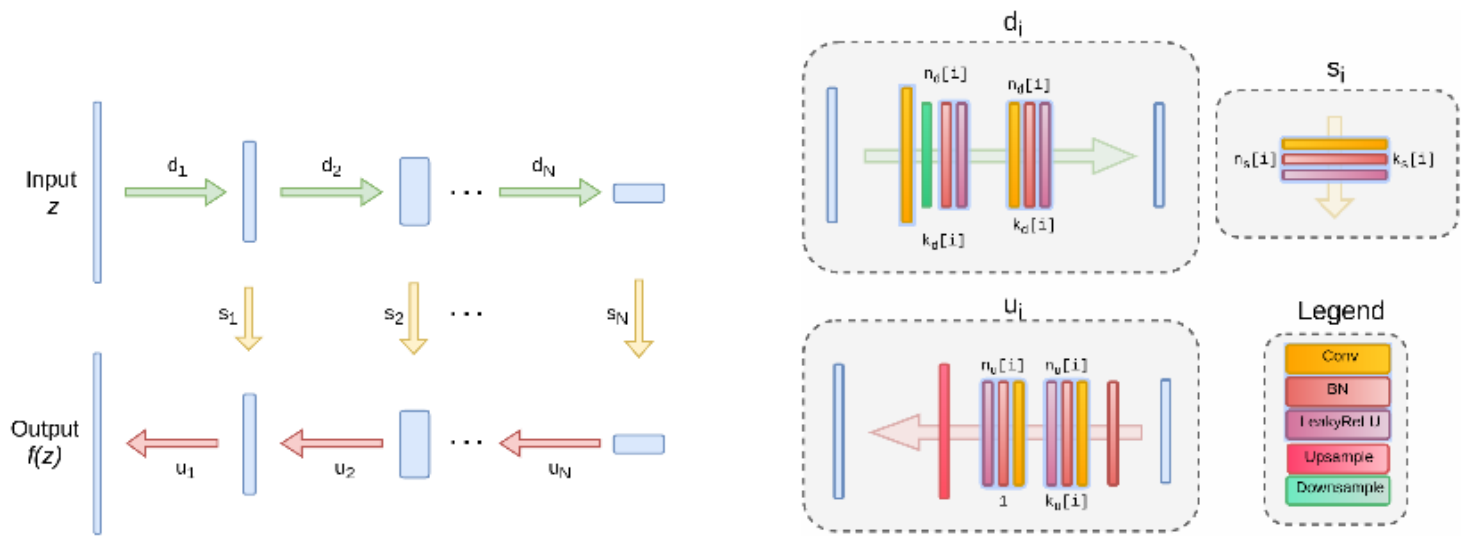


Figure 1: **The architecture used in the experiments.** We use "hourglass" (also known as "decoder-encoder") architecture. We sometimes add skip connections (yellow arrows). $n_u[i]$, $n_d[i]$, $n_s[i]$ correspond to the number of filters at depth i for the upsampling, downsampling and skip-connections respectively. The values $k_u[i]$, $k_d[i]$, $k_s[i]$ correspond to the respective kernel sizes.

Seminal paper: **Deep Image Prior**, *Dmitry Ulianov, Andrea Vedaldi & Victor Lemptisky, 2017*: https://sites.skoltech.ru/app/data/uploads/sites/25/2018/04/deep_image_prior.pdf (https://sites.skoltech.ru/app/data/uploads/sites/25/2018/04/deep_image_prior.pdf)

Deep Image Prior: 3D Implementation of the Original Paper

```
############################## NEURAL NETWORK ##############################
def skip3d(arguments):
    """Assembles encoder-decoder CNN with skip connections.

    Arguments:
        act_fun: Either string 'LeakyReLU|Swish|ELU|none' or module (e.g. nn.ReLU)
        pad (string): zero|reflection (default: 'zero')
        upsample_mode (string): 'nearest|bilinear' (default: 'nearest')
        downsample_mode (string): 'stride|avg|max|lanczos2' (default: 'stride')


    Returns:
        model according to given arguments
    """
```

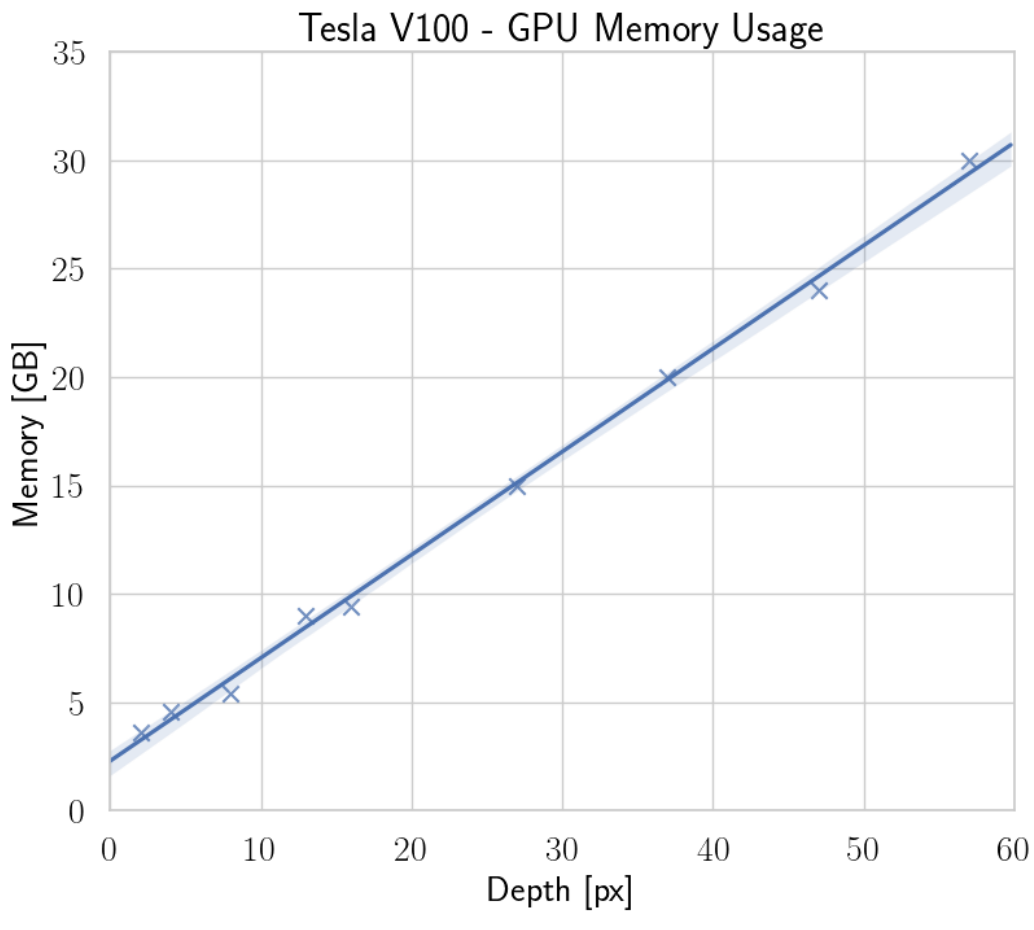
Computational Challenges

- Deep Image Prior Training - Denoising of 3D Medical Images

- Deep Image Prior Training - Denoising of 3D Medical Images
  - Number of network parameters: ≥ 6M

- Deep Image Prior Training - Denoising of 3D Medical Images
  - Number of network parameters: ≥ 6M
  - `RuntimeError: CUDA out of memory. Tried to allocate [...]`

- Deep Image Prior Training - Denoising of 3D Medical Images
  - Number of network parameters: ≥ 6M
  - `RuntimeError: CUDA out of memory. Tried to allocate [...]`



Solution Implemented on MPCDF's HPC Systems

1. Using Tensor Cores for Mixed-Precision Scientific Computing

1. Using Tensor Cores for Mixed-Precision Scientific Computing

    - NVIDIA/Apex

1. Using Tensor Cores for Mixed-Precision Scientific Computing

    - NVIDIA/Apex
        - more memory-efficient

1. Using Tensor Cores for Mixed-Precision Scientific Computing

    - NVIDIA/Apex
        - more memory-efficient
        - improve training speed

1. Using Tensor Cores for Mixed-Precision Scientific Computing

    - NVIDIA/Apex
        - more memory-efficient
        - improve training speed
        - with no achitecture change

1. Using Tensor Cores for Mixed-Precision Scientific Computing
    - NVIDIA/Apex
        - more memory-efficient
        - improve training speed
        - with no achitecture change
2. The model is (**still!**) too large to fit into a single GPU: distributed training technique

1. Using Tensor Cores for Mixed-Precision Scientific Computing

    - NVIDIA/Apex
        - more memory-efficient
        - improve training speed
        - with no achitecture change
2. The model is (**still!**) too large to fit into a single GPU: distributed training technique

    - Model Parallelism: split a single model on (two) separate (V100) GPUs

What is Mixed Precision?

- PyTorch has a comprehensive support for FP16
    - Convert a tensor to FP16: `my_tensor.half()`
    - Convert a module to FP16: `my_module.half()`
- Mixed precision
    - Most of the network in FP16
    - A few carefully selected `ops` in the FP32

Why Mixed Precision?

1. Weight updates benefit from the precision of FP32

`1 + 0.0001 = ???`

```
>>> param = torch.cuda.HalfTensor([1.0])
>>> update = torch.cuda.HalfTensor([0.0001])
>>> print(param + update)
>>> 1
```

In FP16, when update/param < 2e-11 ~ 0.00049, update has no effect.

```
>>> param = torch.cuda.FloatTensor([1.0])
>>> update = torch.cuda.FloatTensor([0.0001])
>>> print(param + update)
>>> 1.0001
```

Why Mixed Precision?

2. Certain ops benefit from the precision of FP32

`BatchNorm, reductions, exponentiations`

```
>>> a = torch.cuda.HalfTensor(4096)
>>> a.fill_(16)
tensor([16., 16., 16.,  ..., 16., 16., 16.], device='cuda:0',
       dtype=torch.float16)
>>> a.sum()
tensor(inf, device='cuda:0', dtype=torch.float16)
>>>

>>> b = torch.cuda.FloatTensor(4096)
>>> b.fill_(16)
>>> b.sum()
tensor(65536., device='cuda:0')
>>>
```

Maximizing Model Performance

2.1 Assign each operation its optimal precision

- FP16
    - GEMMs + Conv. can use Tensor Cores
    - 2X memory throughput
- FP32
    - Weight updates benefit from `fp32` precision
    - Loss functions benefit from precision and range
    - `Softmax`, norms, some other ops benefit from precision and range

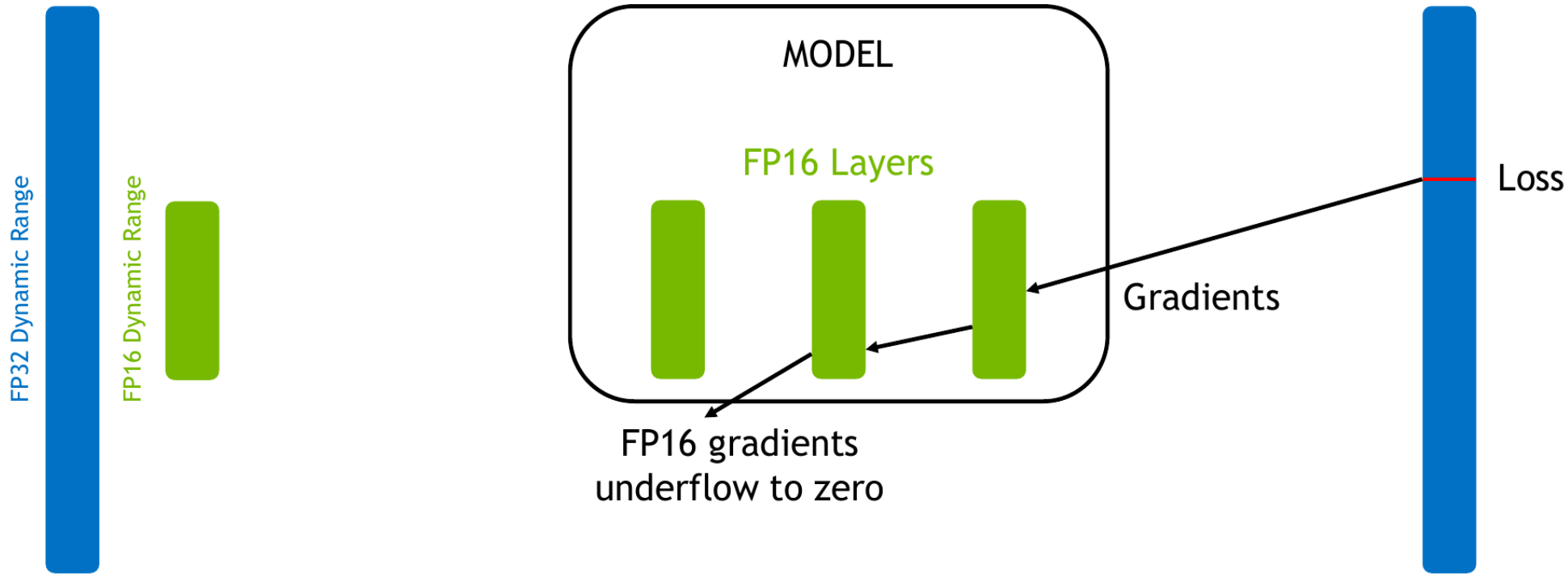
Why Mixed Precision?

3. Small gradients may underflow in FP16 regions of the network

- Loss scaling
    - Multiply the loss by some constant S.
        - `scaled_loss = loss*S`
    - Call `backward()` on `scaled loss`.
        - `scaled_loss.backward()` By the chain rule, gradients will also be scaled by S. This preserve gradient values.
    - Unscale gradients before `optimizer.step()`.

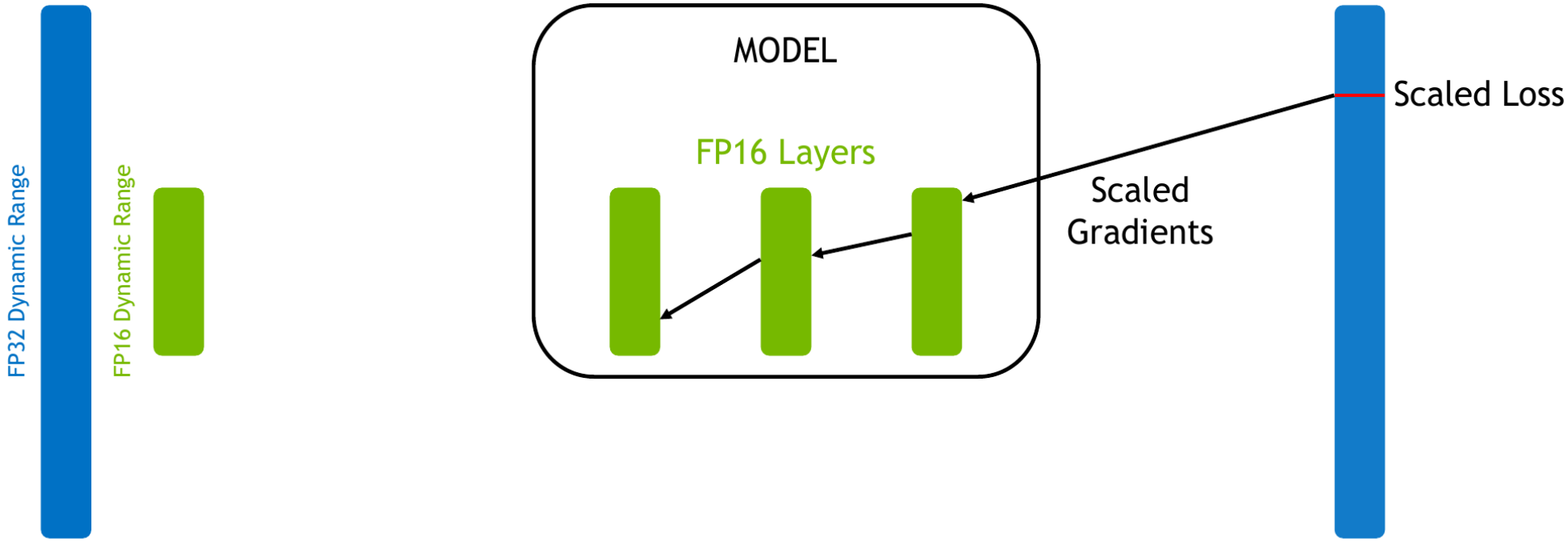
Why Mixed Precision?

3. Small gradients may underflow in FP16 regions of the network



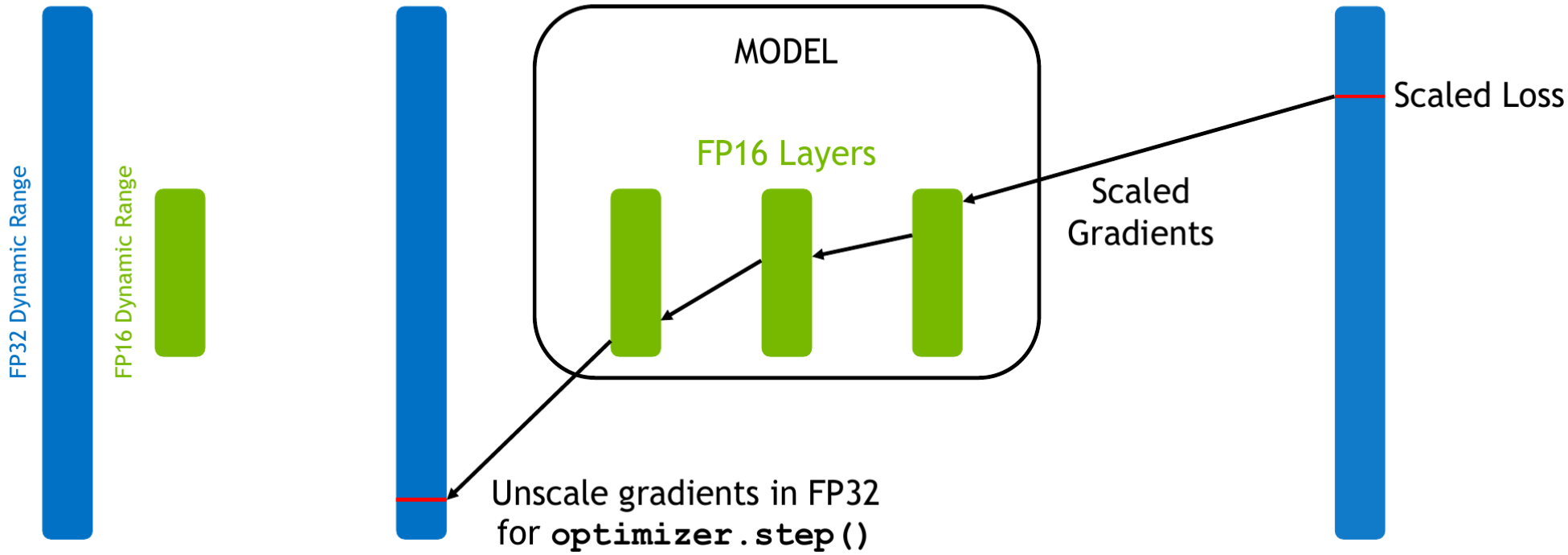
Why Mixed Precision?

3. Small gradients may underflow in FP16 regions of the network



Why Mixed Precision?

3. Small gradients may underflow in FP16 regions of the network

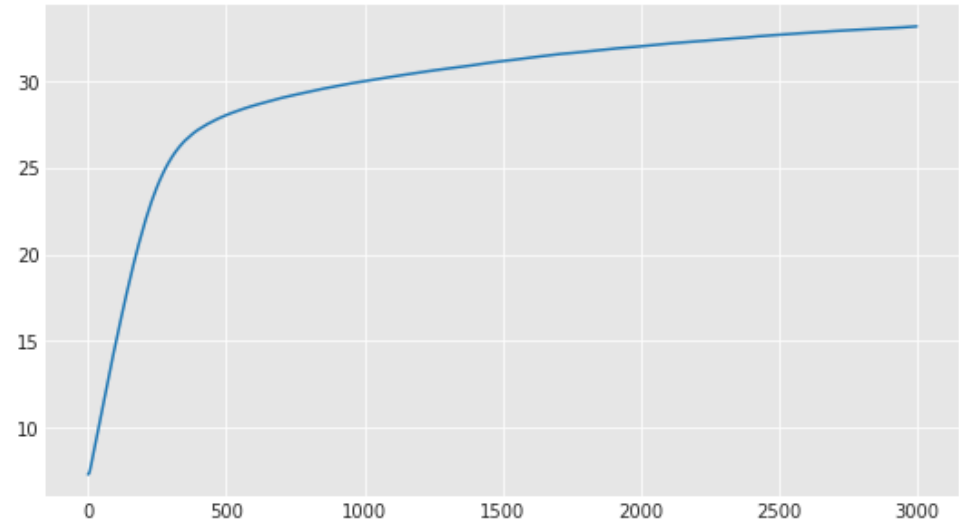


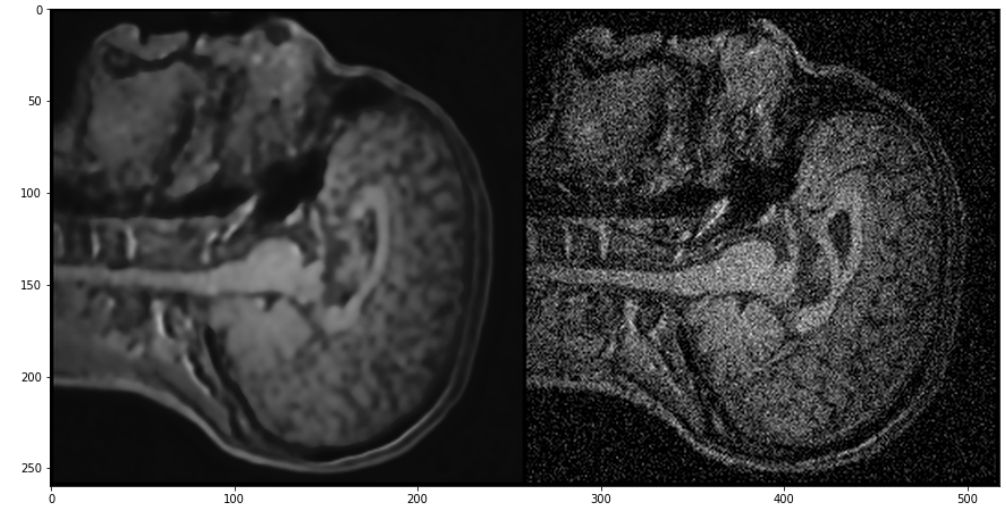
The (Distributed) Mixed Precision Recipe for Deep Image Prior

- Master weights in FP32
- Leave some ops in FP32
- Loss (gradient) scaling

```
import apex
# 1."Model" parallelism
down, up = get_net(args)
down = down.cuda(0).half()
up = up.cuda(1).half()

# 2. Mixed precision Training with FP16_Optimizer Apex class
optimizer = torch.optim.Adam(model.parameters(), lr=LR)

# 2.1 Construct FP16_Optimizer
optimizer = FP16_Optimizer(optimizer, dynamic_loss_scale=True)

# 2.2 loss.backward() becomes:
loss = torch.nn.MSELoss(output, target)
optimizer.backward(loss) # backpropagation, formerly loss.backward()
optimizer.step()
```

Results





Thank you!

Questions: giuseppe.di-bernardo@mpcdf.mpg.de