

Delayed Data Processing

Robert Haase

Funded by



Bundesministerium
für Bildung
und Forschung

SACHSEN

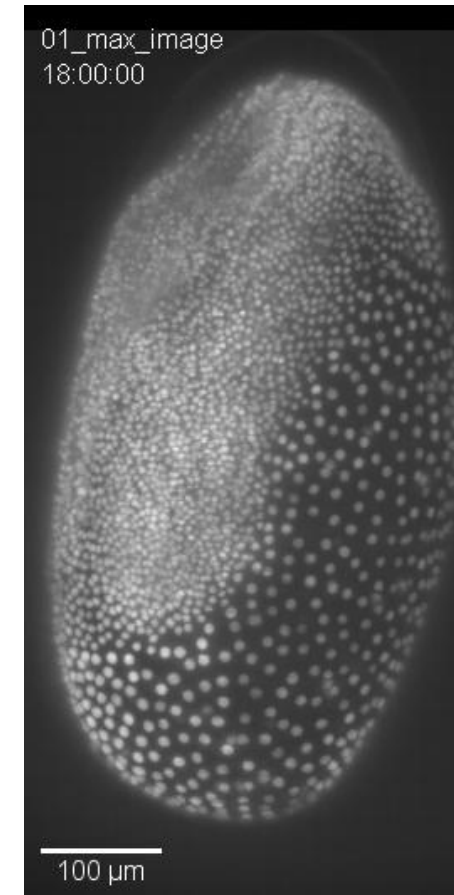


Diese Maßnahme wird gefördert durch die Bundesregierung
aufgrund eines Beschlusses des Deutschen Bundestages.
Diese Maßnahme wird mitfinanziert durch Steuermittel auf
der Grundlage des von den Abgeordneten des Sächsischen
Landtags beschlossenen Haushaltes.

Quiz: Memory constraints

- Assume your computer runs out of memory while processing this image dataset.
What can you do to avoid this?

- 1.
- 2.
3. Tiled image processing



The full data set is
about
1024 (width)
2048 (height)
100 (depth)
3700 (frames)
large

Optimal performance through smart memory management

- The classical way of dealing with large image stacks...

Load data

Preprocessing

Load data

Load data

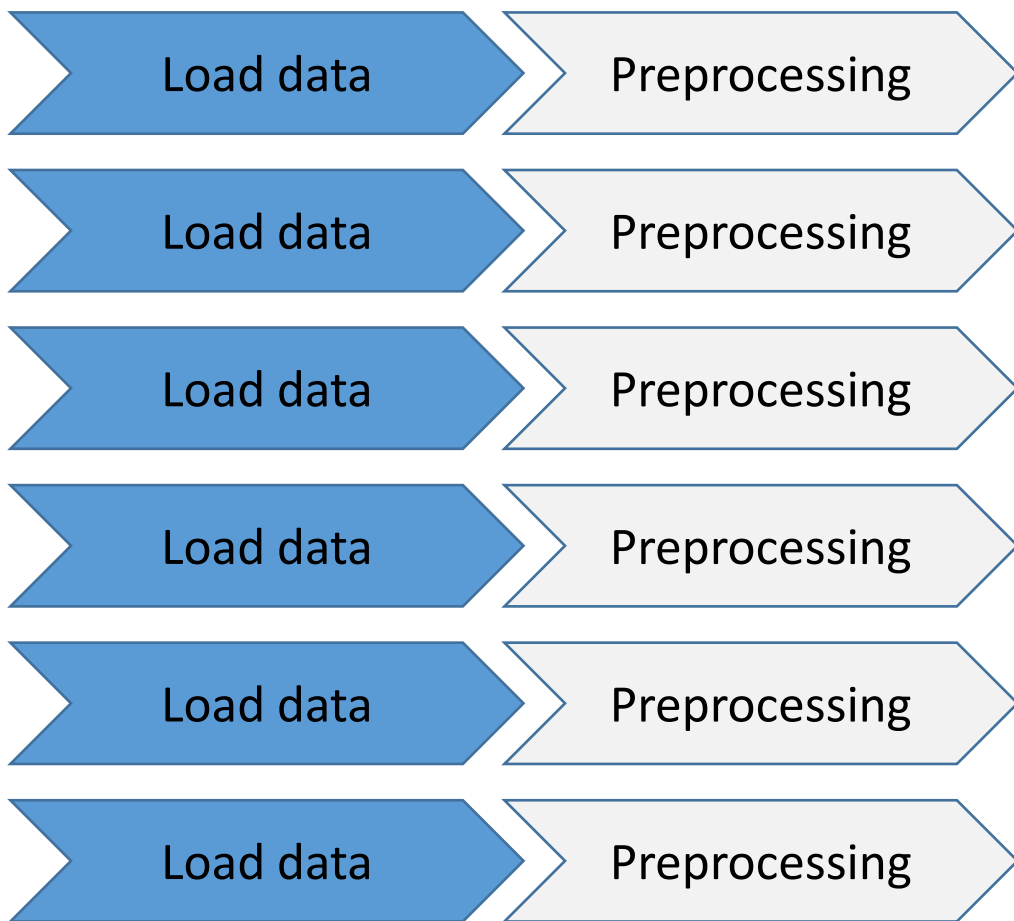
Load data

Load data

Load data

Optimal performance through smart memory management

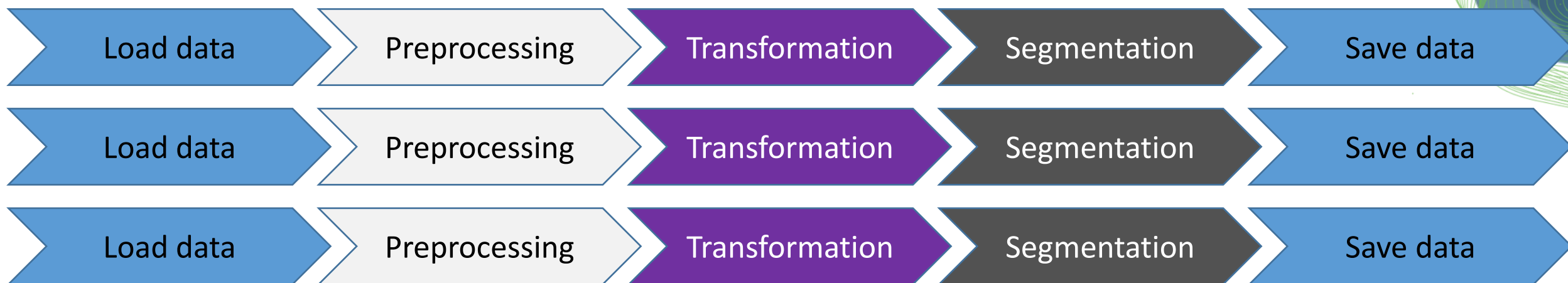
- The classical way of dealing with large image stacks... is suboptimal



This strategy does not just take long; it also costs a lot of memory!

Optimal performance through smart memory management

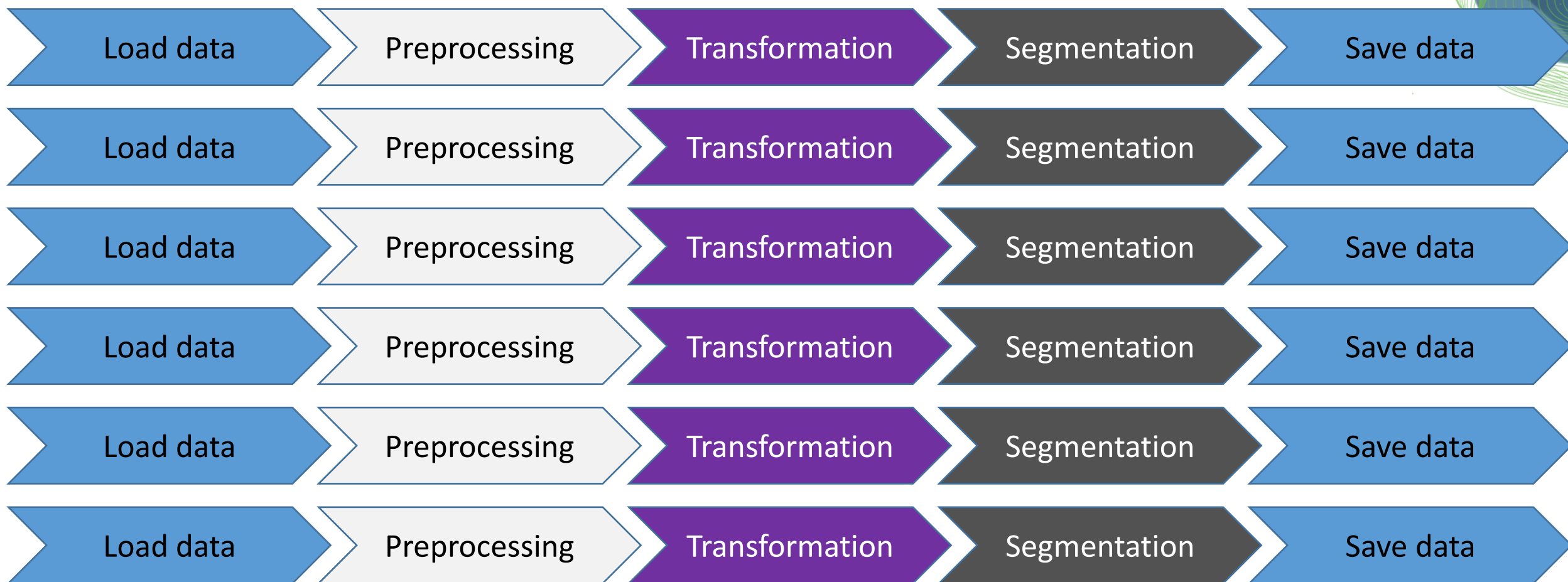
- Processing time-point by time-point is more efficient!



This strategy also works
tile-by-tile on large 3D
stacks and timelapse data!

Optimal performance through smart memory management

- Even better: Distribute tasks between parallelized computation systems

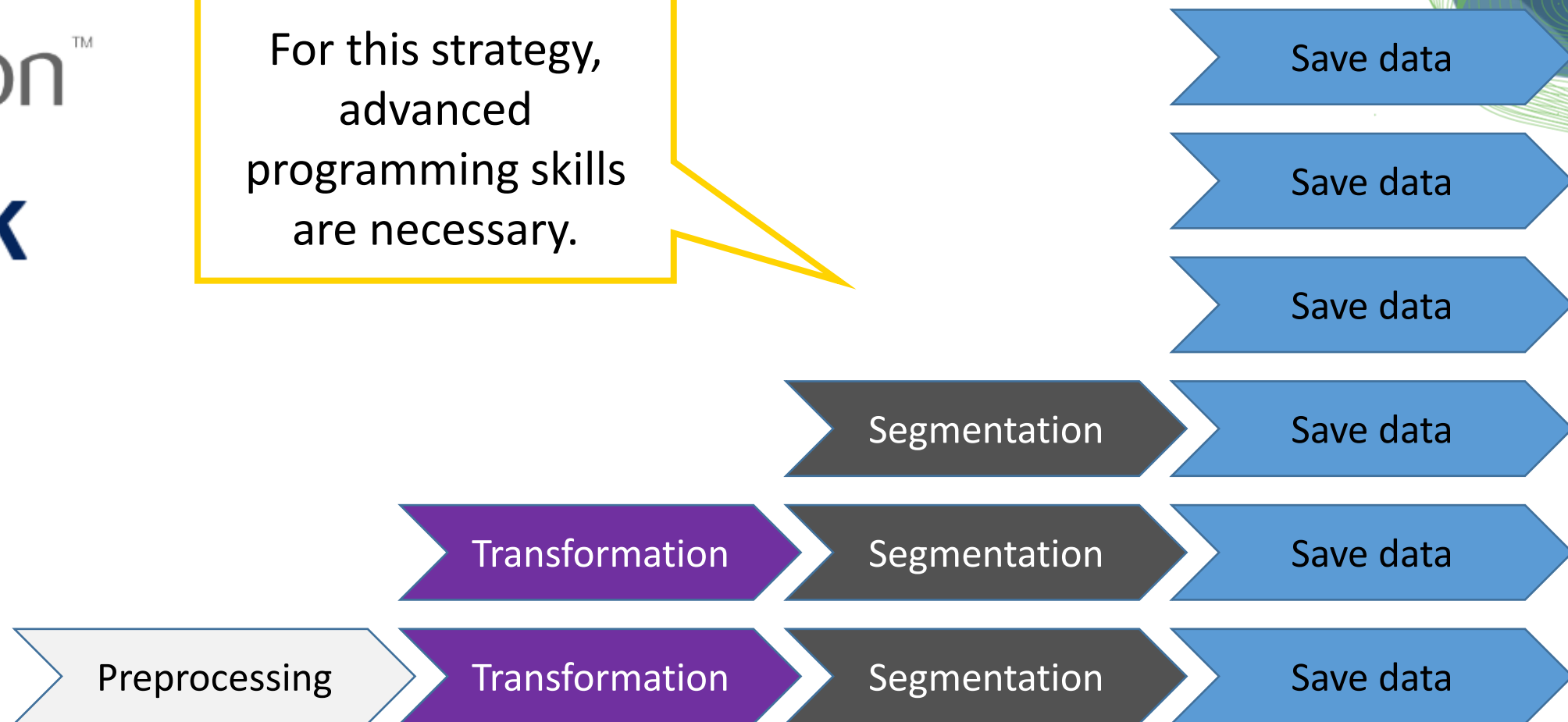


Optimal performance through smart memory management

- Even better: Distribute tasks between parallelized computation systems



For this strategy,
advanced
programming skills
are necessary.



Delayed processing with Dask

- Not not deal with details behind scheduling data loading and processing, we use Dask `delayed` and Dask `stacks`.

```
import dask
import dask.array as da
```

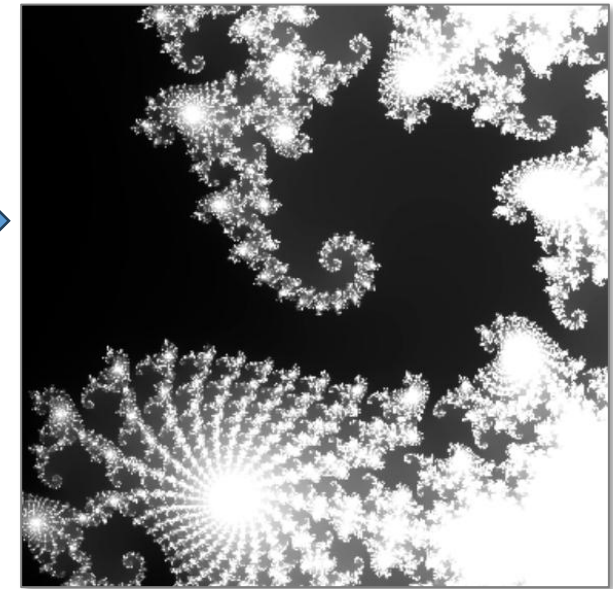
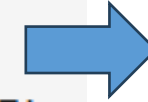
Decorator



Normal
Python
function



```
@delayed
def render_level(i):
    from mandelbrot import SCALE0, ZOOM_PER_LEVEL
    scale = SCALE0 / (ZOOM_PER_LEVEL ** i)
    return mandelbrot_array(scale=scale)
```



Delayed processing with Dask

- Not not deal with details behind scheduling data loading and processing, we use Dask `delayed` and Dask `stacks`.

```
levels = [  
    da.from_delayed(  
        render_level(i),  
        shape=(HEIGHT, WIDTH), dtype=DTYPE  
    )  
    for i in range(number_of_images)  
]  
stack = da.stack(levels, axis=0)
```

Delayed function calls,
NOT executed (yet)

Stack of *delayed* / not
yet computed data

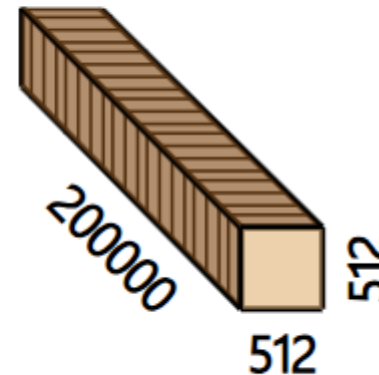
Delayed processing with Dask

- Not not deal with details behind scheduling data loading and processing, we use `Dask delayed` and `Dask stacks`.

```
[20]: stack
```

```
[20]:
```

	Array	Chunk
Bytes	97.66 GiB	512.00 kiB
Shape	(200000, 512, 512)	(1, 512, 512)
Dask graph	200000 chunks in 400001 graph layers	
Data type	uint16 numpy.ndarray	



At this point, no image has been computed yet.

Delayed processing with Dask

- Not not deal with details behind scheduling data loading and processing, we use Dask `delayed` and Dask `stacks`.

```
[22]: stackview.imshow(stack[20000])
```

Computation of
one chunk is
invoked when we
need the data

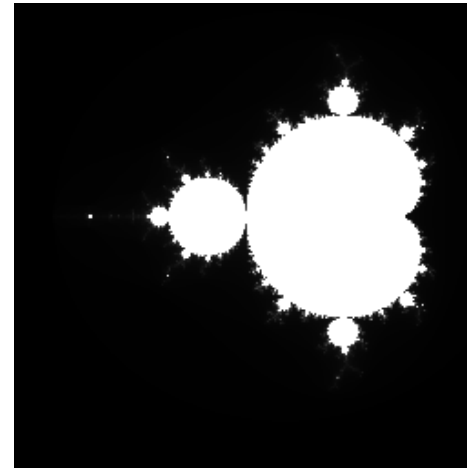


```
[11]: image = np.asarray(stack[0:20000:1000,::2,::2])  
image.shape
```

```
[11]: (20, 256, 256)
```

```
[17]: stackview.animate(image)
```

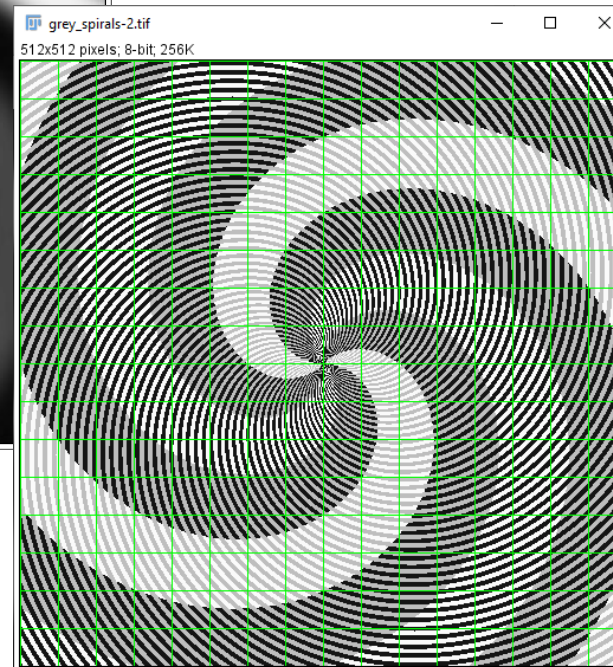
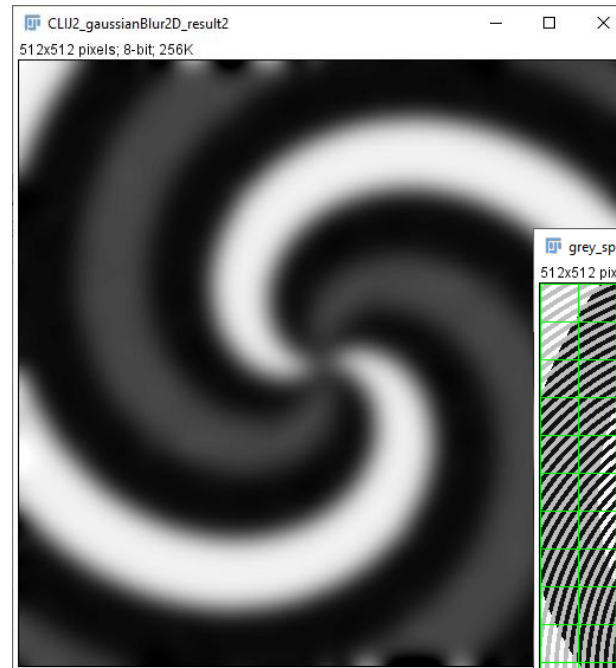
Also works with
multiple chunks



Tiling

- The **last** perimeter against processing big image data

If the image is too large for the computer memory, image processing as a whole is *not possible*.



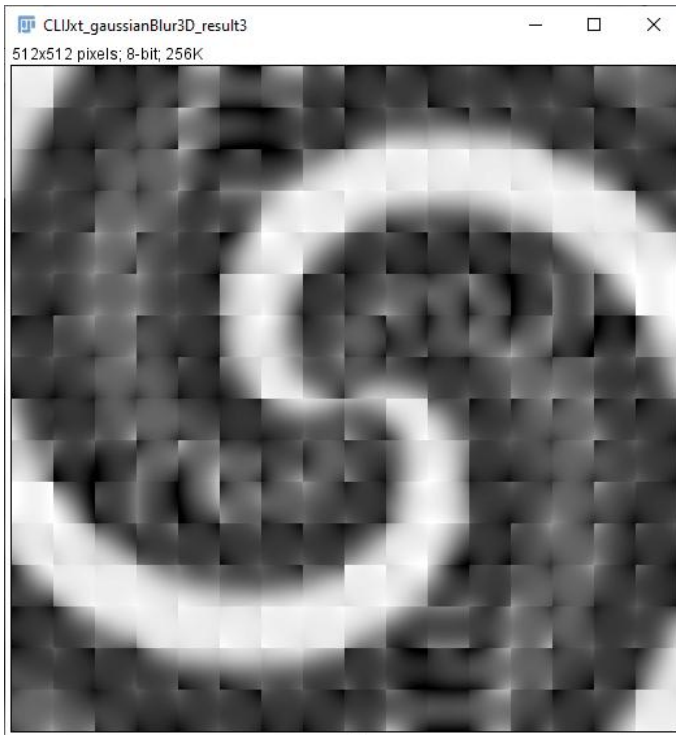
Processing tile-by-tile poses new challenges

Tiling

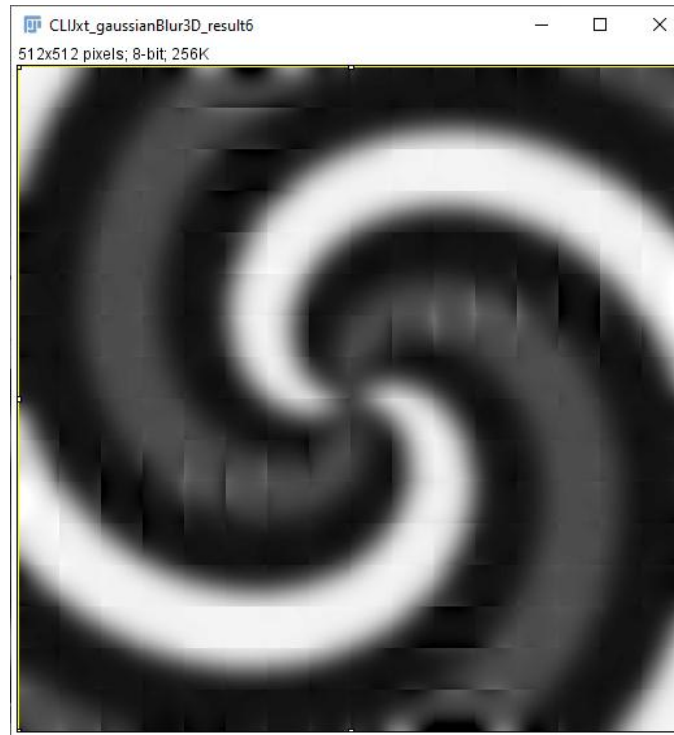
- Example: Gaussian blur (sigma = 20)
- Solution: Process with overlapping tiles (size + margin)

Optimal margin size depends
on algorithm and its
parameters

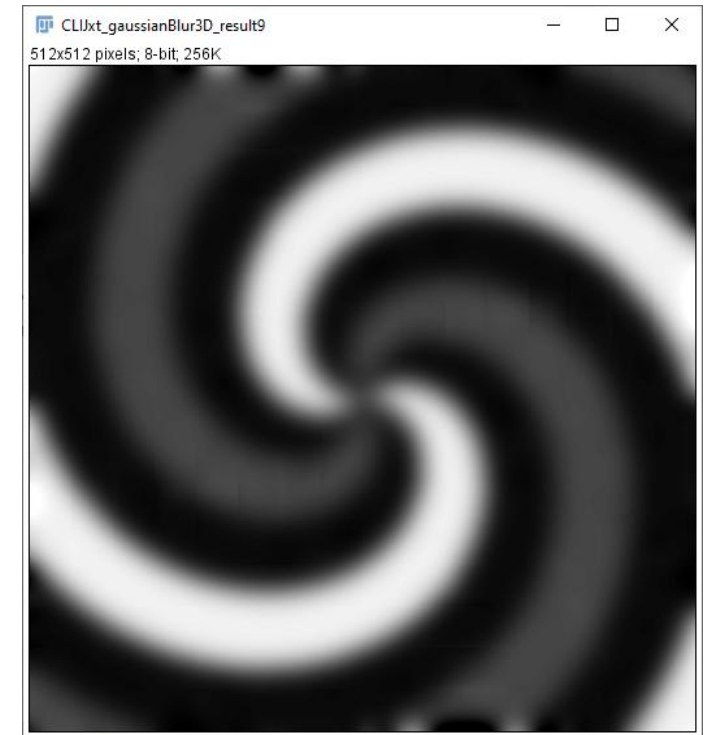
Margin: 0 pixels



Margin: 10 pixels



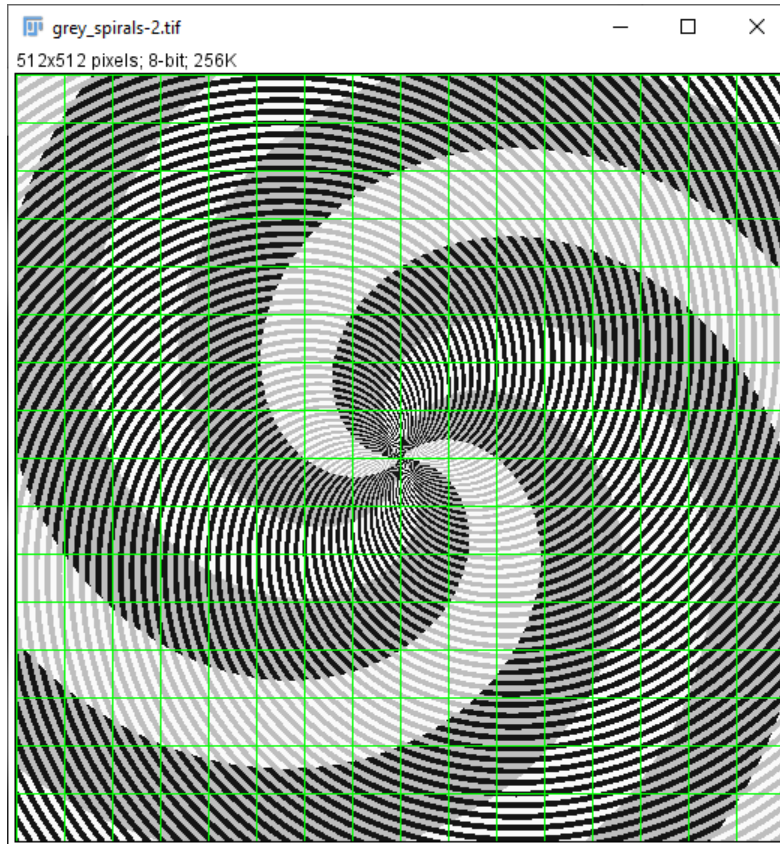
Margin: 20 pixels



Tiling

- Example: Gaussian blur (sigma = 20 pixels)
- Solution: Process with overlapping tiles (size + margin)

Computation time depends on tile size and margin width



Margin: 20 pixels
Size: 5x original

Margin: 10 pixels
Size: 2.7x original

Tile
32x32 pixels

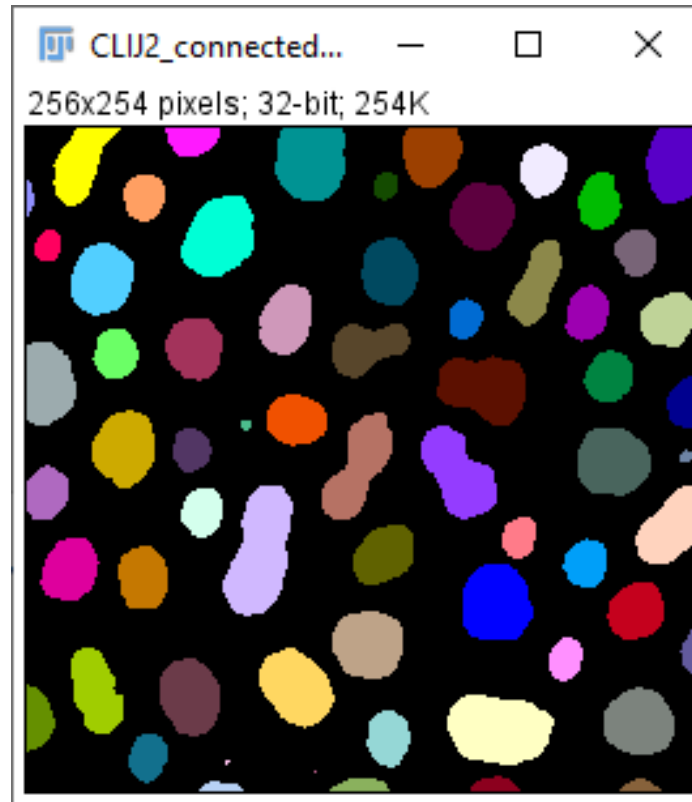
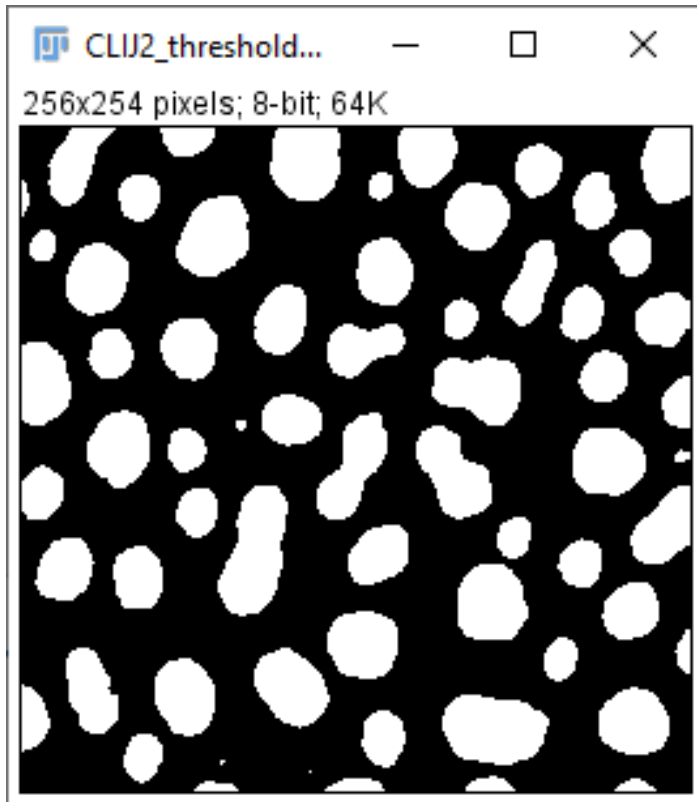
52x52 pixels

72x72 pixels

Tiling

- Some algorithms are hard to solve by processing tiles
- Example: Connected component analysis

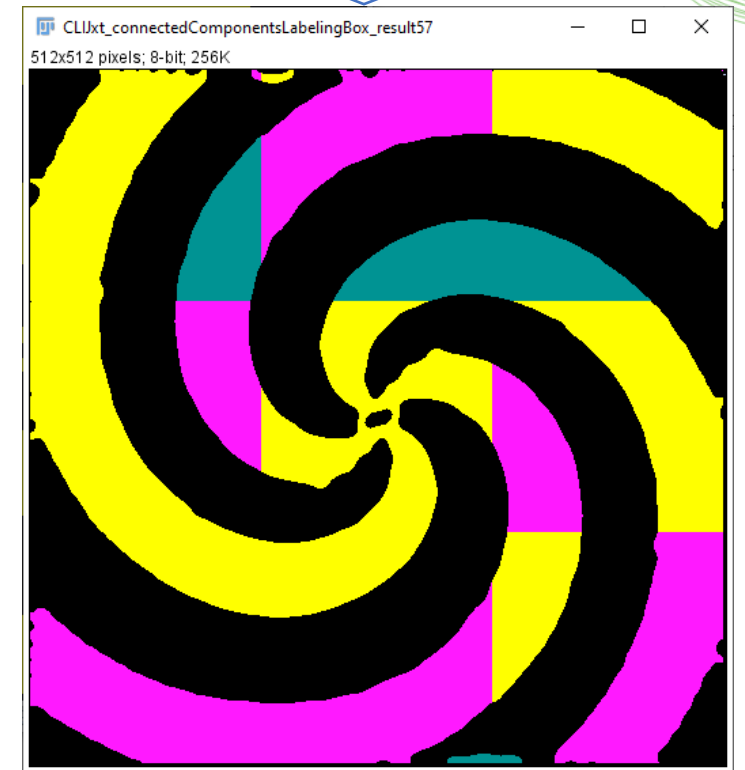
Checking which labels touch and combine them is feasible.



Tiling

- Some algorithms are hard to solve by processing tiles
- Example: Connected component analysis

There are algorithms for that, but hardly available tools.



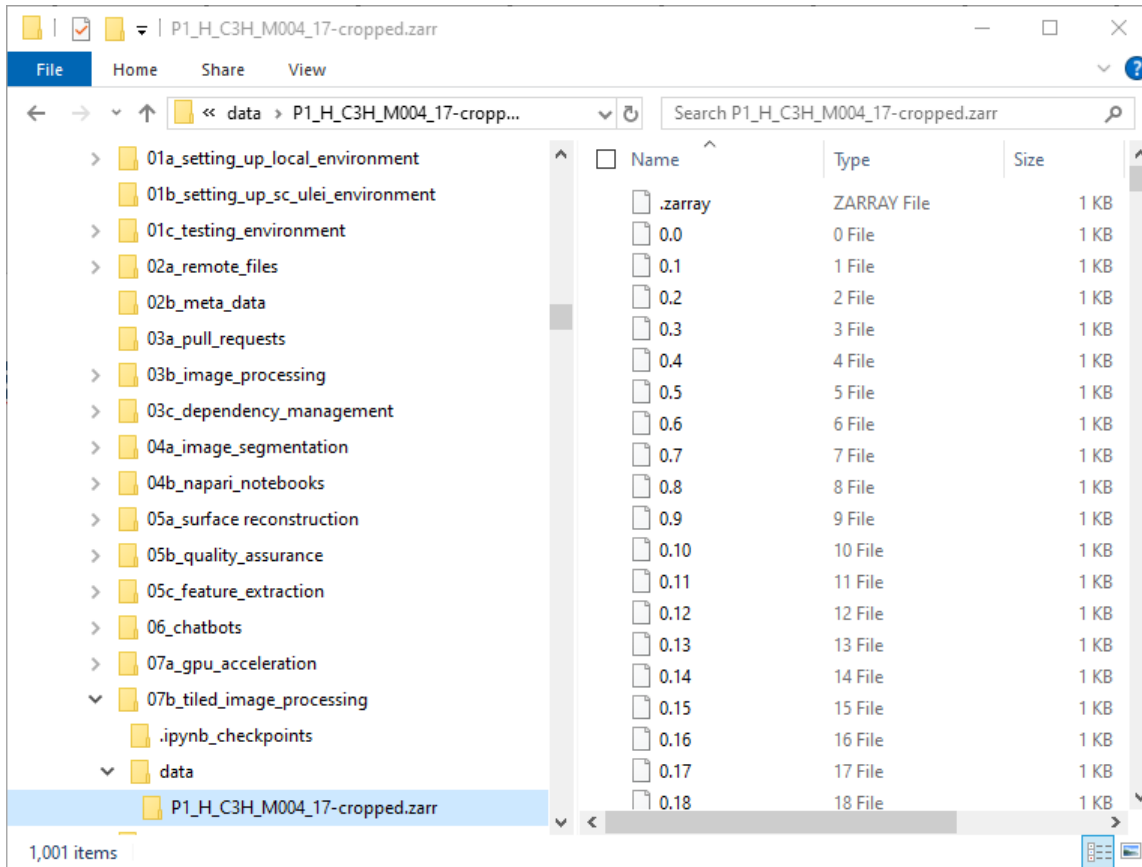
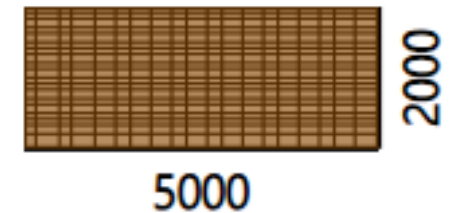
Tiled image processing in Python

- Key: tiled file formats, for parallel, distributed, lazy loading

After executing this,
no pixel has been
read yet.

```
zarr_image = da.from_zarr(zarr_filename)
zarr_image
```

	Array	Chunk
Bytes	9.54 MiB	9.77 kiB
Shape	(2000, 5000)	(100, 100)
Dask graph	1000 chunks in 2 graph layers	
Data type	uint8 numpy.ndarray	



Tiled image processing in Python

- Delayed processing

After executing this, no pixel has been read or processed yet.

```
[5]: tile_map = da.map_blocks(count_nuclei, zarr_image)
```

```
tile_map
```

```
Processing image of size (0, 0)
```

```
(1, 1)
```

```
Processing image of size (1, 1)
```

```
(1, 1)
```

```
[5]:
```

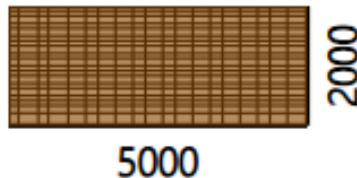
	Array	Chunk
--	-------	-------

Bytes	76.29 MiB	78.12 kiB
-------	-----------	-----------

Shape	(2000, 5000)	(100, 100)
-------	--------------	------------

Dask graph 1000 chunks in 3 graph layers

Data type	float64 numpy.ndarray
-----------	-----------------------



After that, results are available

```
result = tile_map.compute()
```

```
Processing image of size (100, 100)
```

```
Processing image of size (100, 100)
```

```
Processing image of size (100, 100)
```

```
Processing image of size (100, 100)
```

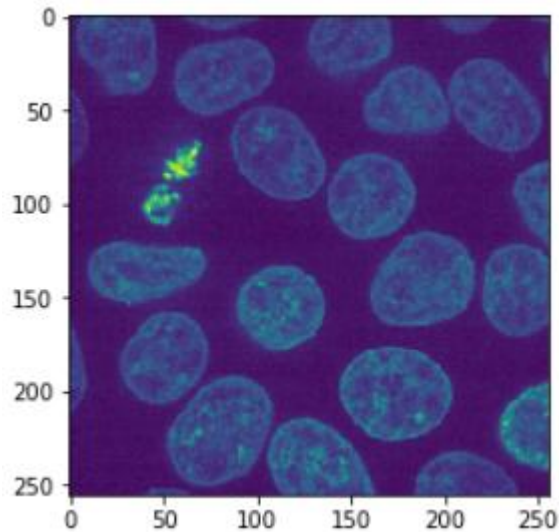
```
Processing image of size (100, 100)
```

```
Processing image of size (100, 100)
```

```
Processing image of size (100, 100)
```

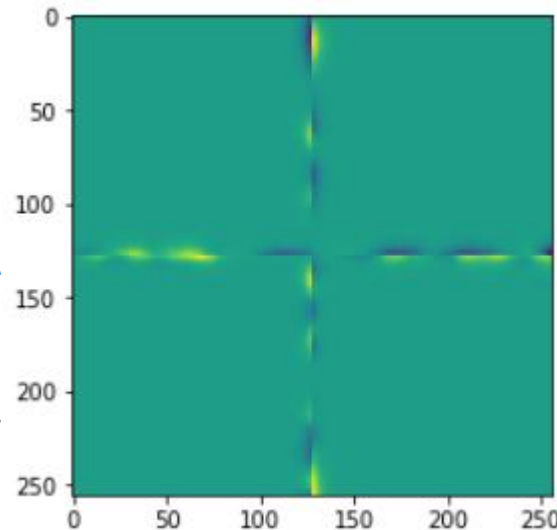
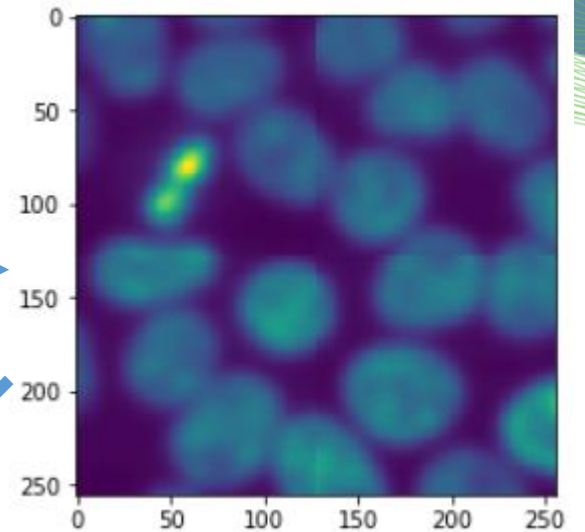
Tiling with/out overlap

- Processing of images in tiles: artifacts at tile borders



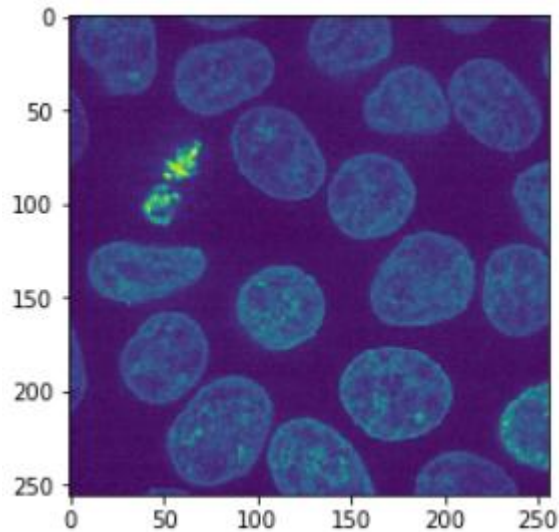
```
tile_map = da.map_blocks(procedure, tiles)
```

```
result = tile_map.compute()
```

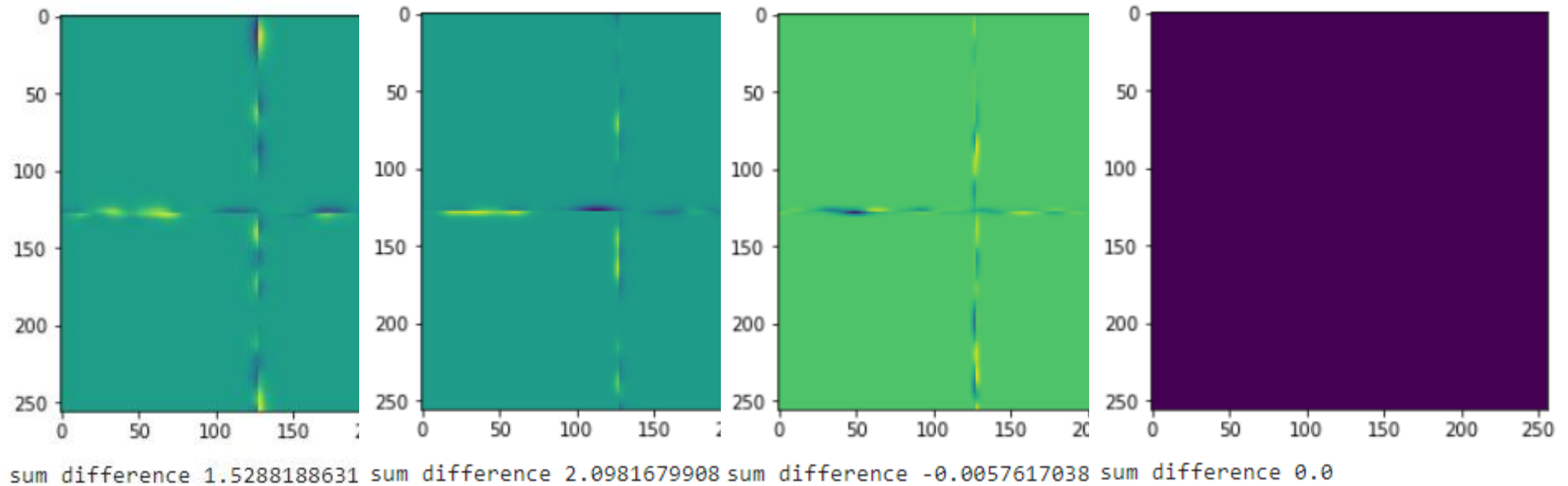


Tiling with/out overlap

- Processing of images in tiles: artifacts at tile borders

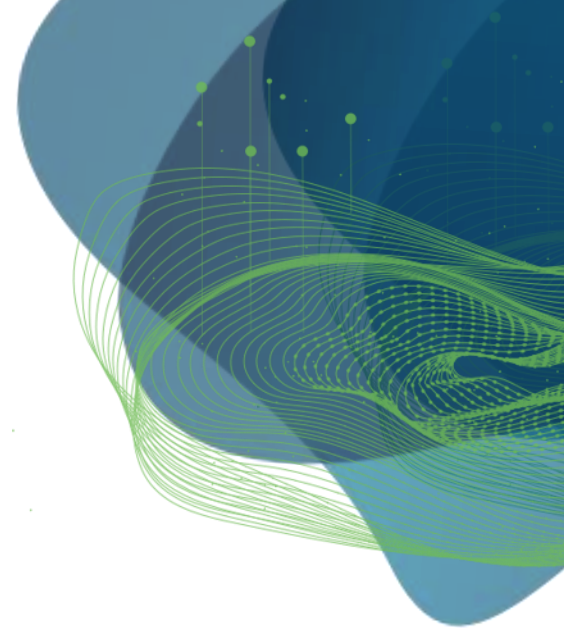


```
overlap_width = 1  
tile_map = da.map_overlap(procedure, tiles, depth=overlap_width)  
  
result = tile_map.compute()
```



Exercises

Robert Haase



GEFÖRDERT VOM



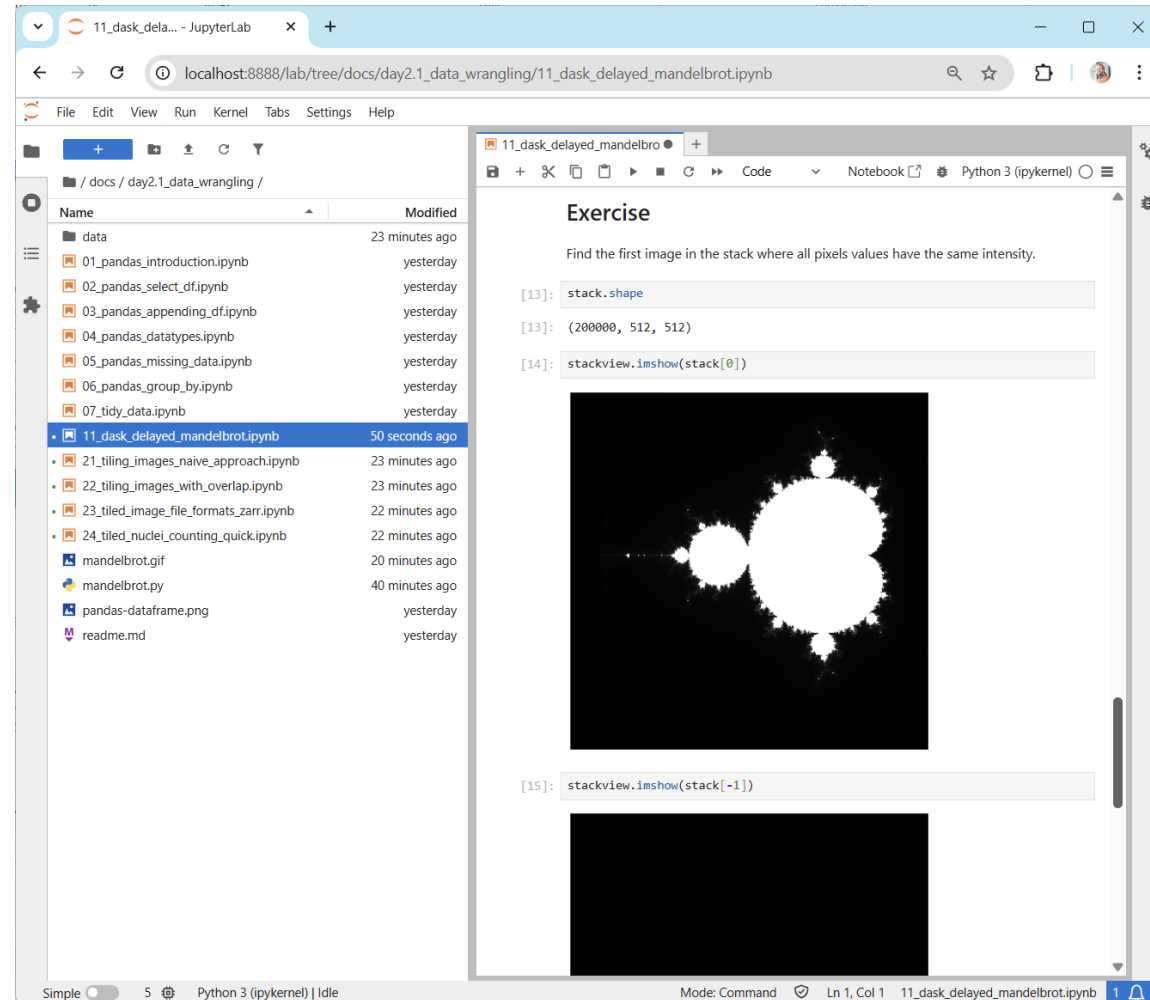
Bundesministerium
für Bildung
und Forschung



Diese Maßnahme wird gefördert durch die Bundesregierung aufgrund eines Beschlusses des Deutschen Bundestages. Diese Maßnahme wird mitfinanziert durch Steuermittel auf der Grundlage des von den Abgeordneten des Sächsischen Landtags beschlossenen Haushaltes.

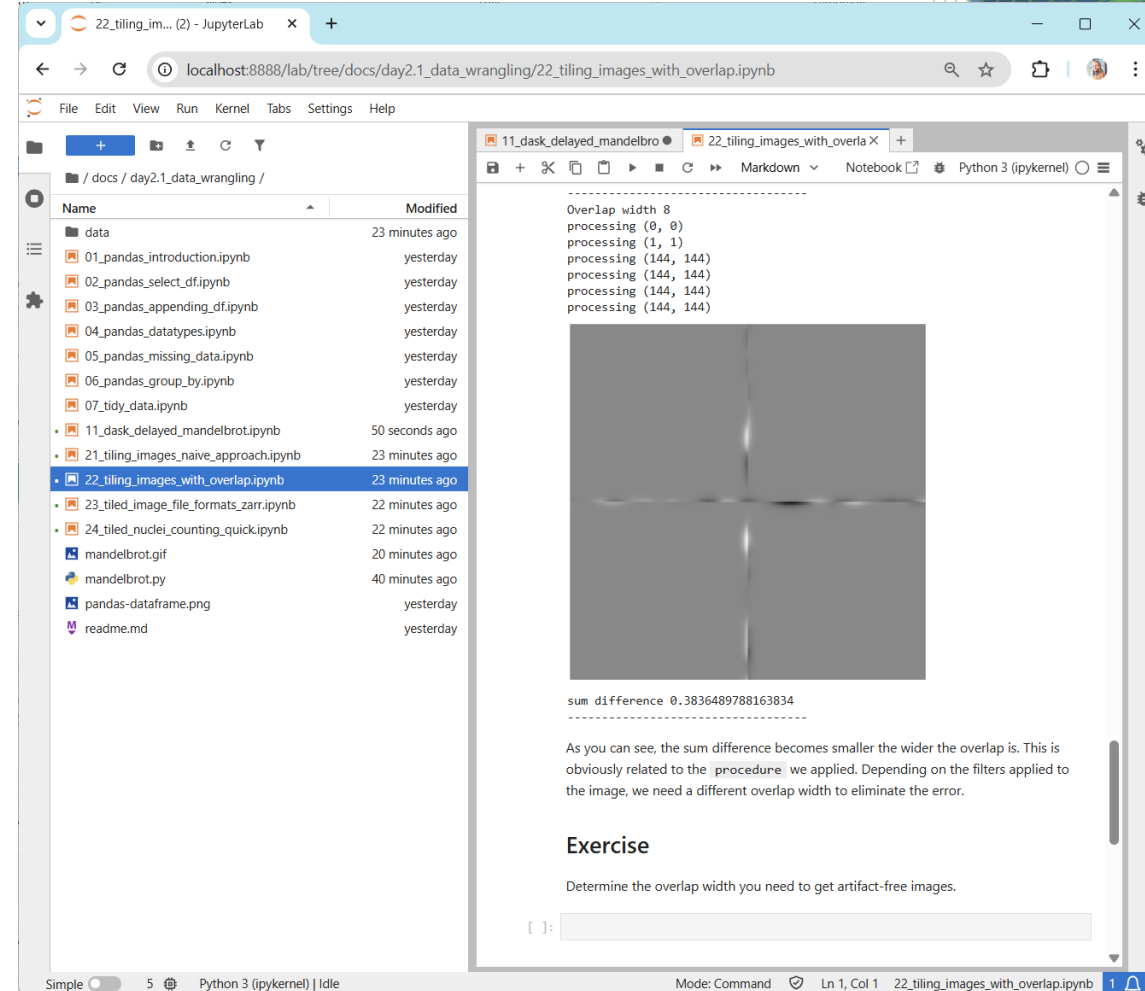
Exercise: Delayed processing

- Determine the first image index when all pixels have the same intensity (approximately).



Exercise: Tiled image processing

- Apply background-removal to an image in tiles. Determine the overlap width that's necessary to have artifact-free results.



The screenshot shows a JupyterLab environment. On the left is a file browser for the directory `/docs/day2.1_data_wrangling/`. It lists several IPython notebooks, with `22_tiling_images_with_overlap.ipynb` selected. On the right is the notebook interface for `22_tiling_images_with_overlap.ipynb`. The notebook content shows the output of a processing function with an overlap width of 8. The output includes a list of processing steps, a visual representation of a tiled image with overlap, and a sum difference value of 0.3836489788163834. Below the visual output, there is a paragraph explaining that the sum difference becomes smaller as the overlap width increases, and a section titled "Exercise" asking the user to determine the overlap width needed to get artifact-free images.

```
Overlap width 8
processing (0, 0)
processing (1, 1)
processing (144, 144)
processing (144, 144)
processing (144, 144)
processing (144, 144)
```

sum difference 0.3836489788163834

As you can see, the sum difference becomes smaller the wider the overlap is. This is obviously related to the procedure we applied. Depending on the filters applied to the image, we need a different overlap width to eliminate the error.

Exercise

Determine the overlap width you need to get artifact-free images.

[]:

Exercise: Counting Nuclei in tiles

- After segmenting and counting nuclei in tiles, compare the result to processing the whole image. Try out strategies for reducing the error.

24_tiled_nuc... (4) - JupyterLab

localhost:8888/lab/tree/docs/day2.1_data_wrangling/24_tiled_nuclei_counting_quick.ipynb

File Edit View Run Kernel Tabs Settings Help

docs / day2.1_data_wrangling /

Name Modified

- data 23 minutes ago
- 01_pandas_introduction.ipynb yesterday
- 02_pandas_select_df.ipynb yesterday
- 03_pandas_appending_df.ipynb yesterday
- 04_pandas_datatypes.ipynb yesterday
- 05_pandas_missing_data.ipynb yesterday
- 06_pandas_group_by.ipynb yesterday
- 07_tidy_data.ipynb yesterday
- 11_dask_delayed_mandelbrot.ipynb now
- 21_tiling_images_naive_approach.ipynb 23 minutes ago
- 22_tiling_images_with_overlap.ipynb 23 minutes ago
- 23_tiled_image_file_formats_zarr.ipynb 22 minutes ago
- 24_tiled_nuclei_counting_quick.ipynb 22 minutes ago
- mandelbrot.gif 20 minutes ago
- mandelbrot.py 40 minutes ago
- pandas-dataframe.png yesterday
- readme.md yesterday

[10]: np.float64(5545.0)

Again, as the result map is small, we can just visualize it.

[11]: stackview.insight(result)

[11]:

shape (20, 50)
dtype float64
size 7.8 kB
min 0.0
max 20.5

Sanity check

Applying the `count_nuclei` function to the whole zarr image allows us to check for the error of the computation above.

Note that this would not work with *big* data.

[12]: count_nuclei(np.asarray(zarr_image))

[12]: array([[5423.5]])

Exercise

Use the function `da.from_zarr(zarr_image).rechunk((500, 500))` to apply `count_nuclei` to differently chunked data. Depending on the chunk size, the error of the method seems different. why?