Run Full Pipeline on Tileset6 (difficult one) - October 2018

Created: 27 Oct 2018
Last update: 28 Oct 2018

Goal: Apply full pipeline on the hard sample, but then without black borders

(check the assumption that without black borders and smaller patches, the crystals are found)

---

1. Imports

In [1]:
```
# this will remove warnings messages
import warnings
warnings.filterwarnings('ignore')

import numpy as np
import pandas as pd

import matplotlib.pyplot as plt

%matplotlib inline

# import
from sklearn.cluster import KMeans
from sklearn.cluster import AgglomerativeClustering
from sklearn.preprocessing import LabelEncoder
from sklearn.preprocessing import StandardScaler
from sklearn.decomposition import PCA, TruncatedSVD
from sklearn.pipeline import Pipeline
from sklearn.metrics import silhouette_score

import imgutils
```

In [101]:
```
# Re-run this cell if you altered imgutils
import importlib
importlib.reload(imgutils)
```

Out[101]:
```
<module 'imgutils' from 'C:\\JADS\\SW\\Grad Proj\\sources\\imgutils.py'>
```

---

2. Data Definitions & Feature Specification

In [3]:
```
# Data:
datafolder = '../data/Crystals_Apr_12/Tileset6_Subset_NoBlack_2k'
# datafolder = '../data/Crystals_Apr_12/Tileset6_Subset_NoBlack_1k'

n_tiles_x = 2  # mostly for visualization
n_tiles_y = 2


# Features to use:
#feature_funcs = [imgutils.img_mean, imgutils.img_std, imgutils.img_median,
#                 imgutils.img_mode,
#                 imgutils.img_kurtosis, imgutils.img_skewness]
feature_funcs = [imgutils.img_std, imgutils.img_relstd, imgutils.img_mean,
                 imgutils.img_skewness,  imgutils.img_kurtosis, imgutils.img_mode]
feature_names = imgutils.stat_names(feature_funcs)

# Size of the grid, specified as number of slices per image in x and y direction:
default_grid_x = 4
default_grid_y = default_grid_x
```

---

3. Import Data & Extract Features

In [4]:
```
# image import:
print("Scanning for images in '{}'...".format(datafolder))
df_imgfiles = imgutils.scanimgdir(datafolder, '.tif')
imgfiles = list(df_imgfiles['filename'])
print("# of images: {} \n".format(len(imgfiles)))

# feature extraction:
print("Feature extraction...")
print("- Slicing up images in {} x {} patches. ".format(default_grid_y, default_grid_x))
print("- Extract statistics from each slice: {} ".format(', '.join(feature_names)))
print("...working...", end='\r')
df = imgutils.slicestats(imgfiles, default_grid_y, default_grid_x, feature_funcs)
print("# slices extracted: ", len(df))
```

```
Scanning for images in '../data/Crystals_Apr_12/Tileset6_Subset_NoBlack_2k'...
# of images: 4

Feature extraction...
- Slicing up images in 4 x 4 patches.
- Extract statistics from each slice: img_std, img_relstd, img_mean, img_skewness, img_kurtosis, img_mode
# slices extracted:  64
```

---

4. Machine Learning Pipeline

Hyper parameters

In [5]:
```
# data hyper-parameters
default_n_clusters = 3

# algorithm hyper-parameters:
kmeans_n_init = 20
```

In [6]:
```
def run_ml_pipeline2(X, ml_name, ml_algorithm, standardize=True, use_pca=True, n_pca=None):

    # Setup algorithmic pipeline, including standardization
    pipeline = Pipeline([(ml_name, ml_algorithm)])

    # watch the order, pca should happen after scaling, but we insert at 0
    if (use_pca):
        pipeline.steps.insert(0,('pca', PCA(n_components=n_pca)))
    if (standardize):
        pipeline.steps.insert(0, ('scaling_{0}'.format(ml_name), StandardScaler()))

    # run the pipelines
    y = pipeline.fit_predict(X) # calls predict on last step to get the labels

    # report score:
    score = silhouette_score(X, y)

    return score, y
```

In [7]:
```
def run_ml_pipelines2(df_data, feature_cols, n_clusters, standardize=True, use_pca=True, n_pca=None):
    global kmeans_n_init

    X = df_data.loc[:,feature_cols]

    # Setup ML clustering algorithms:
    kmeans = KMeans(algorithm='auto', n_clusters=n_clusters, n_init=kmeans_n_init, init='k-means++')
    agglomerative =  AgglomerativeClustering(n_clusters=n_clusters, affinity='euclidean', linkage='complete')

    # run the pipelines
    print("Executing clustering pipelines...")
    score_kmeans, y_kmeans = run_ml_pipeline2(X, 'kmeans', kmeans, standardize = standardize, use_pca = use_pca, n_pca=n_pca)
    score_hier, y_hier = run_ml_pipeline2(X, 'hierarchical', agglomerative, standardize = standardize, use_pca = use_pca, n_pca=n_pca)
    print("Done\n")

    # collect data
    df_data['kmeans']=y_kmeans
    df_data['hierarchical']=y_hier

    # report results:
    print("\nClustering Scores:")
    print("K-means: ", score_kmeans)
    print("Hierarchical: ", score_hier)
```

In [8]:
```
run_ml_pipelines2(df, feature_names, default_n_clusters, standardize=True, use_pca=True)
```

```
Executing clustering pipelines...
Done


Clustering Scores:
K-means:  0.33639024060299816
Hierarchical:  -0.1707082845689075
```

---

5. Alternave versions of the pipeline

Adjusting the ml_pipeline to use silhouette scoring based on it's last transformation: (later renamed the other ones to run_xxx2 to preserve them)

In [9]:
```
def run_ml_pipeline(X, ml_name, ml_algorithm, standardize=True, use_pca=True, n_pca=None):

    # Setup 'manual' pipeline (not using sklearn pipeline as intermediates are needed)
    feat_data = X
    if (standardize):
        standardizer = StandardScaler()
        X_norm = standardizer.fit_transform(X)
        feat_data = X_norm
    if (use_pca):
        pca = PCA(n_components=n_pca)
        X_pca = pca.fit_transform(feat_data)
        feat_data = X_pca

    # run the pipelines
    y = ml_algorithm.fit_predict(feat_data) # calls predict oto get the labels

    # report score:
    score = silhouette_score(feat_data, y)

    return score, y
```

In [10]:
```
def run_ml_pipelines(df_data, feature_cols, n_clusters, standardize=True, use_pca=True, n_pca=None):
    global kmeans_n_init

    X = df_data.loc[:,feature_cols]

    # Setup ML clustering algorithms:
    kmeans = KMeans(algorithm='auto', n_clusters=n_clusters, n_init=kmeans_n_init, init='k-means++')
    agglomerative =  AgglomerativeClustering(n_clusters=n_clusters, affinity='euclidean', linkage='complete')

    # run the pipelines
    print("Executing clustering pipelines...")
    score_kmeans, y_kmeans = run_ml_pipeline(X, 'kmeans', kmeans, standardize = standardize, use_pca = use_pca, n_pca=n_pca)
    score_hier, y_hier = run_ml_pipeline(X, 'hierarchical', agglomerative, standardize = standardize, use_pca = use_pca, n_pca=n_pca)
    print("Done\n")

    # collect data
    df_data['kmeans']=y_kmeans
    df_data['hierarchical']=y_hier

    # report results:
    print("\nClustering Scores:")
    print("K-means: ", score_kmeans)
    print("Hierarchical: ", score_hier)
```

In [11]:
```
run_ml_pipelines(df, feature_names, default_n_clusters, standardize=True, use_pca=True)
```

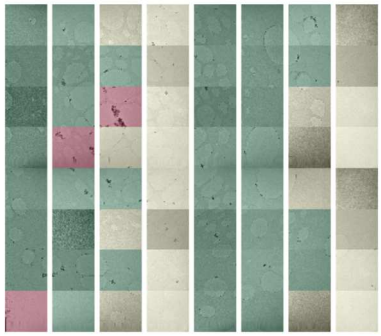
```
Executing clustering pipelines...
Done


Clustering Scores:
K-means:  0.3871490344399508
Hierarchical:  0.39321847388128733
```

5. Visualize results

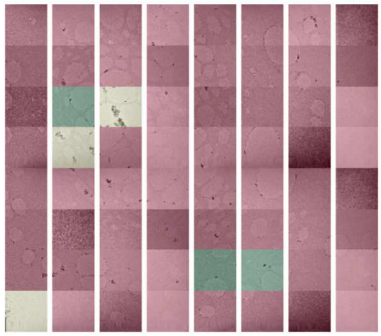
In [13]:
```
run_ml_pipelines(df, feature_names, default_n_clusters, standardize=True, use_pca=True)
s = (8,6)
imgutils.show_large_heatmap(df, 'kmeans', imgfiles, n_rows=n_tiles_y, n_cols=n_tiles_x, fig_size=s, opacity=0.3)
imgutils.show_large_heatmap(df, 'hierarchical', imgfiles, n_rows=n_tiles_y, n_cols=n_tiles_x, fig_size=s, opacity=0.3)
```

```
Executing clustering pipelines...
Done


Clustering Scores:
K-means:  0.3871490344399508
Hierarchical:  0.39321847388128733
```

Heats from: kmeans



Heats from: hierarchical

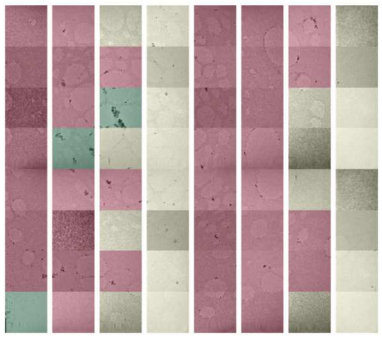


Run it again without PCA and/pr normalization compare results

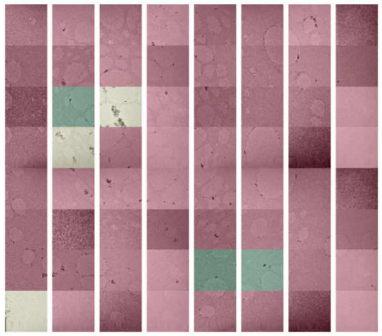
In [14]:
```
run_ml_pipelines(df, feature_names, default_n_clusters, standardize=True, use_pca=False)
imgutils.show_large_heatmap(df, 'kmeans', imgfiles, n_rows=n_tiles_y, n_cols=n_tiles_x, fig_size=s, opacity=0.3)
imgutils.show_large_heatmap(df, 'hierarchical', imgfiles, n_rows=n_tiles_y, n_cols=n_tiles_x, fig_size=s, opacity=0.3)
```

```
Executing clustering pipelines...
Done


Clustering Scores:
K-means:  0.38714903443995075
Hierarchical:  0.3932184738812874
```

Heats from: kmeans



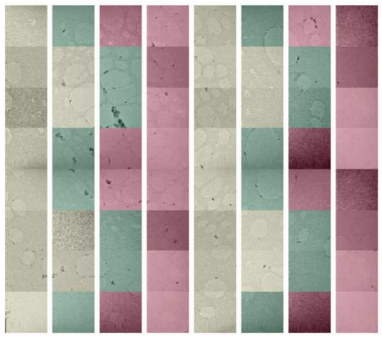
Heats from: hierarchical



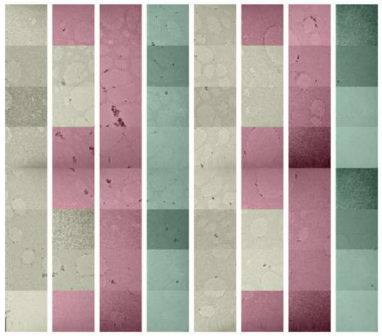
In [15]:
```
run_ml_pipelines(df, feature_names, default_n_clusters, standardize=False, use_pca=False)
imgutils.show_large_heatmap(df, 'kmeans', imgfiles, n_rows=n_tiles_y, n_cols=n_tiles_x, fig_size=s, opacity=0.3)
imgutils.show_large_heatmap(df, 'hierarchical', imgfiles, n_rows=n_tiles_y, n_cols=n_tiles_x, fig_size=s, opacity=0.3)
```

```
Executing clustering pipelines...
Done


Clustering Scores:
K-means:  0.6509288897521603
Hierarchical:  0.5548516497404329
```

Heats from: kmeans



Heats from: hierarchical



With this patch size, only the larger black spots are picked up

6. Combine import and pipeline:

In [16]:
```
def import_data(imagefolder):
    df_imgfiles = imgutils.scanimgdir(datafolder, '.tif')
    return list(df_imgfiles['filename'])

def extract_features(imgfiles, feature_funcs, n_grid_rows, n_grid_cols):
    df = imgutils.slicestats(imgfiles, n_grid_rows, n_grid_cols, feature_funcs)
    feature_names = imgutils.stat_names(feature_funcs)
    return df, feature_names
```

In [17]:
```
def run_kmeans_pipeline(df_data, feature_cols, n_clusters, standardize=True, use_pca=True, n_pca= None):
    global kmeans_n_init

    ml_name="kmeans"
    ml_algorithm = KMeans(algorithm='auto', n_clusters=n_clusters, n_init=kmeans_n_init, init='k-means++')

    X = df_data.loc[:,feature_cols]
    score, y = run_ml_pipeline(X, ml_name, ml_algorithm, standardize = standardize, use_pca = use_pca, n_pca=n_pca)
    df_data[ml_name]= y

    return score

def run_hierarchical_pipeline(df_data, feature_cols, n_clusters, standardize=True, use_pca=True, n_pca=None):

    ml_name="hierarchical"
    ml_algorithm =  AgglomerativeClustering(n_clusters=n_clusters, affinity='euclidean', linkage='complete')

    X = df_data.loc[:,feature_cols]
    score, y = run_ml_pipeline(X, ml_name, ml_algorithm, standardize = standardize, use_pca = use_pca, n_pca=n_pca)
    df_data[ml_name]= y

    return score
```

In [20]:
```
def run_fullpipeline(imagefolder, n_image_rows, n_image_cols,
                     n_grid_rows, n_grid_cols, feature_funcs, n_clusters,
                    fig_size = (10,8), return_df = False):
    """
    Run the full pipeline from import to visualization.
    """
    print("Working...\r")
    imgfiles = import_data(imagefolder)
    df, feature_names = extract_features(imgfiles, feature_funcs, n_grid_rows, n_grid_cols)
    score_kmeans = run_kmeans_pipeline(df, feature_names, n_clusters, standardize=True, use_pca=True )
    score_hier = run_hierarchical_pipeline(df, feature_names, n_clusters, standardize=False, use_pca=False)

    print('Results:')
    print('Score k-means:', score_kmeans)
    print('Score hierarchical:', score_hier)

    print('Visualizing...')
    s = (8,6)
    imgutils.show_large_heatmap(df, 'kmeans', imgfiles, n_rows=n_image_rows, n_cols=n_image_cols, fig_size=fig_size, opacity=0.3)
    imgutils.show_large_heatmap(df, 'hierarchical', imgfiles, n_rows=n_image_rows, n_cols=n_image_cols, fig_size=fig_size, opacity=0.3)

    if (return_df): return df
```

In [ ]:
```

```

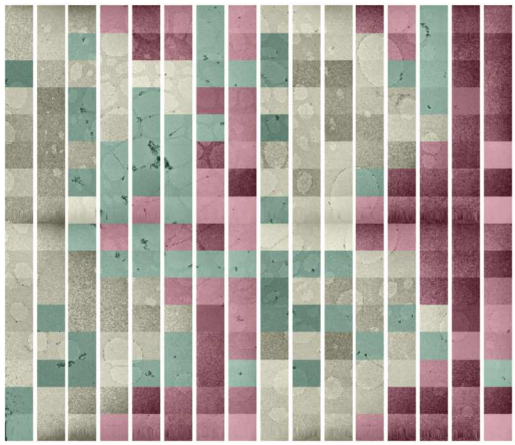
7. Try it out with different combinations of slices

8x8 - 3 clusters

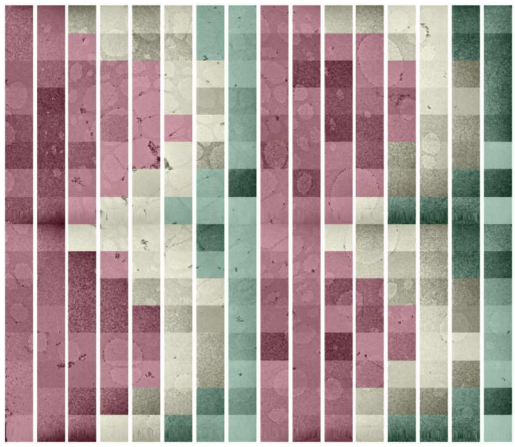
In [27]:
```
run_fullpipeline(datafolder, n_tiles_y, n_tiles_x, 8, 8, feature_funcs, 3)
```

```
Working...
Results:
Score k-means: 0.4040607007135775
Score hierarchical: 0.545888064776314
Visualizing...
```

Heats from: kmeans



Heats from: hierarchical



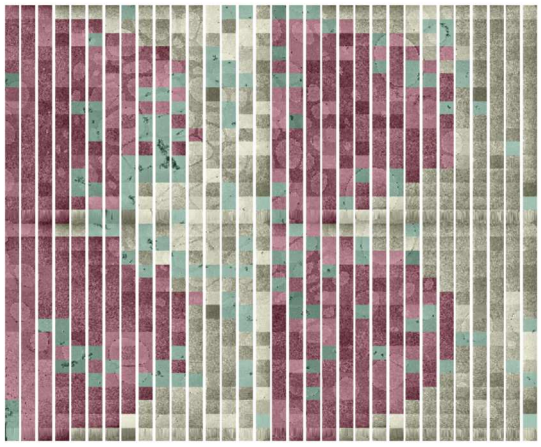
already somwhat better, but not goo enough (especially hierarchical is weird)

16x16, 3 clusters

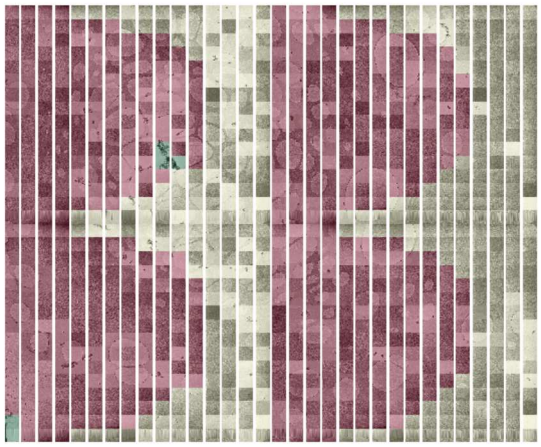
In [28]:
```
run_fullpipeline(datafolder, n_tiles_y, n_tiles_x, 16, 16, feature_funcs, 3)
```

```
Working...
Results:
Score k-means: 0.4763101434435471
Score hierarchical: 0.6198682234844993
Visualizing...
```

Heats from: kmeans



Heats from: hierarchical



Hierarchical is no good here. In general, weird background pattern is picked up which I think comes from uneven camera lighting / sensitiviy

---

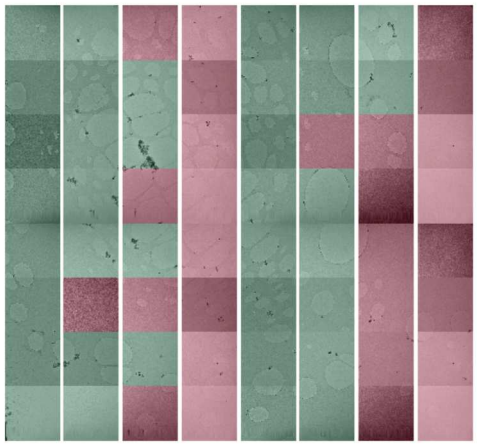
8. Try it out with different number of clusters

2 clusters (4x4 , 8x8, 16x16)

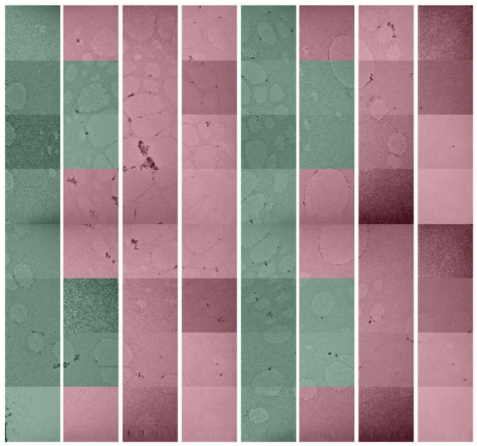
In [29]:
```
run_fullpipeline(datafolder, n_tiles_y, n_tiles_x, 4, 4, feature_funcs, 2)
```

```
Working...
Results:
Score k-means: 0.3425654277794109
Score hierarchical: 0.5396352292193908
Visualizing...
```

Heats from: kmeans



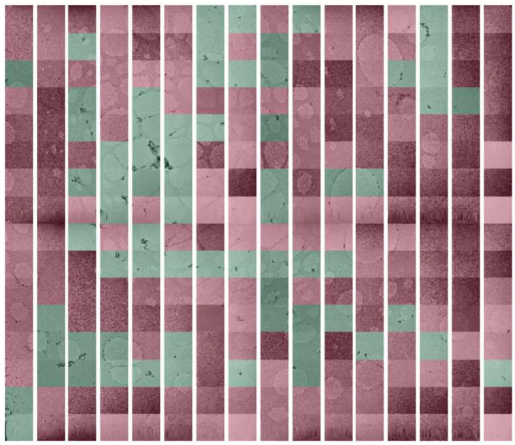
Heats from: hierarchical



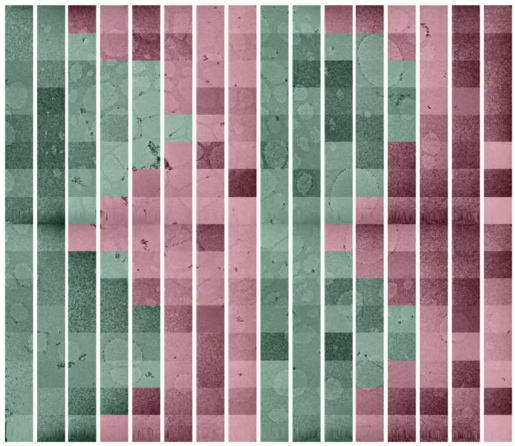
In [30]:
```
run_fullpipeline(datafolder, n_tiles_y, n_tiles_x, 8, 8, feature_funcs, 2)
```

```
Working...
Results:
Score k-means: 0.40768146154155516
Score hierarchical: 0.6033667732192352
Visualizing...
```

Heats from: kmeans



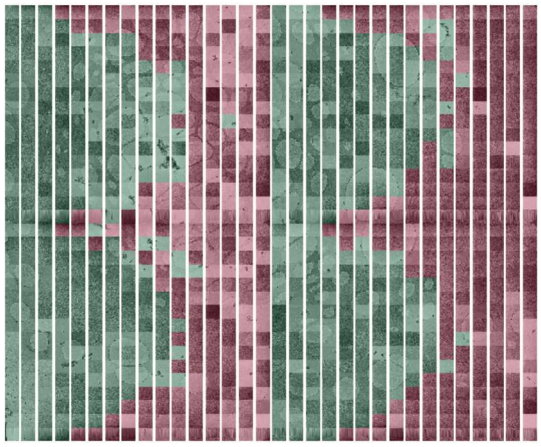
Heats from: hierarchical



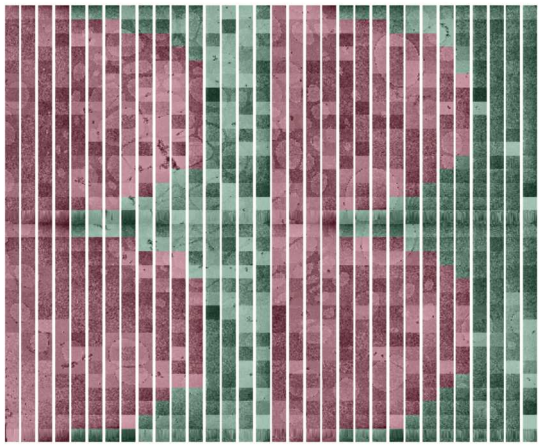
In [31]:
```
run_fullpipeline(datafolder, n_tiles_y, n_tiles_x, 16, 16, feature_funcs, 2)
```

```
Working...
Results:
Score k-means: 0.40428330918298055
Score hierarchical: 0.6264484710544231
Visualizing...
```

Heats from: kmeans



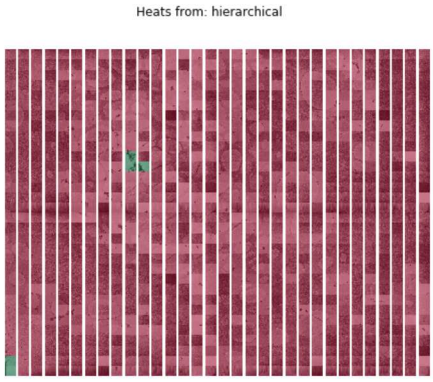
Heats from: hierarchical



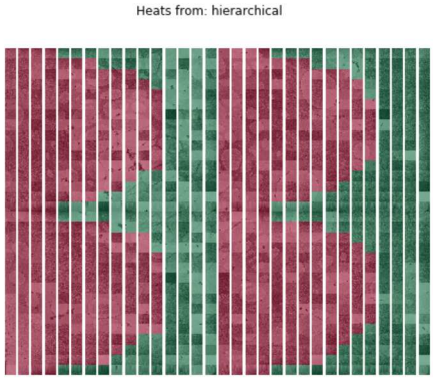
Look again at hierarchical with scaling and pca on:

In [32]:
```
imgfiles = import_data(datafolder)
df_temp, feature_names = extract_features(imgfiles, feature_funcs, 16, 16)
score = run_hierarchical_pipeline(df_temp, feature_names, 2, standardize=True, use_pca=True)
print("With scaling & pca:", score)
imgutils.show_large_heatmap(df_temp, 'hierarchical', imgfiles, n_rows=n_tiles_y, n_cols=n_tiles_x, fig_size=s)
score = run_hierarchical_pipeline(df_temp, feature_names, 2, standardize=False, use_pca=False)
print("Without scaling & pca:", score)
imgutils.show_large_heatmap(df_temp, 'hierarchical', imgfiles, n_rows=n_tiles_y, n_cols=n_tiles_x, fig_size=s)
```

```
With scaling & pca: 0.7917682403355812
```




```
Without scaling & pca: 0.6264484710544231
```



We need to drop the hierarchical for this set, it's not working well

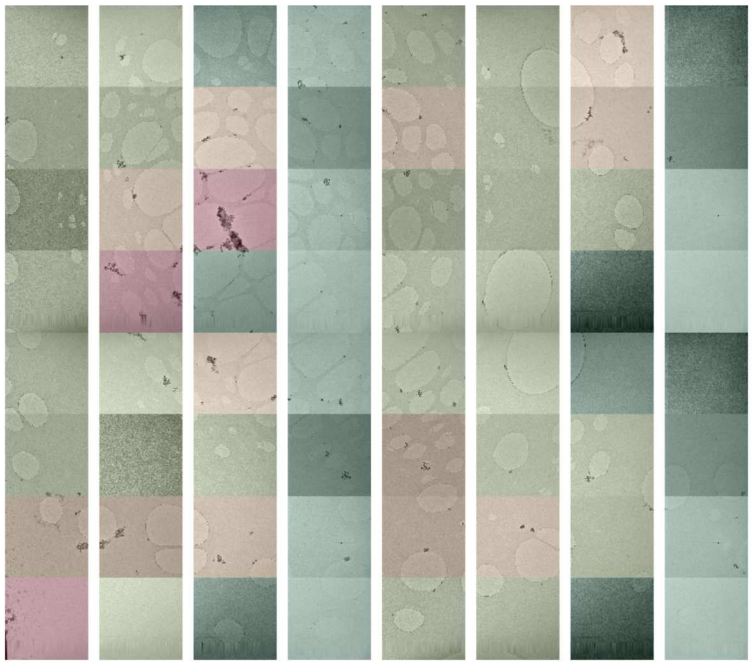
In [33]:
```
def run_fullpipeline_pca(imagefolder, n_image_rows, n_image_cols,
                         n_grid_rows, n_grid_cols, feature_funcs, n_clusters,
                        fig_size = (16,12), return_df = False):
    """
    Run the full pipeline from import to visualization.
    """
    imgfiles = import_data(imagefolder)
    df = run_fullpipeline_pca_filelist(imgfiles, n_image_rows, n_image_cols,
                                       n_grid_rows, n_grid_cols, feature_funcs, n_clusters, fig_size, return_df = True)
    if (return_df): return df

def run_fullpipeline_pca_filelist(imgfiles, n_image_rows, n_image_cols,
                         n_grid_rows, n_grid_cols, feature_funcs, n_clusters,
                        fig_size = (16,12), return_df = False, alpha=0.2):
    """
    Run the full pipeline from import to visualization.
    """
    print("Working...\r")
    df, feature_names = extract_features(imgfiles, feature_funcs, n_grid_rows, n_grid_cols)
    score_kmeans = run_kmeans_pipeline(df, feature_names, n_clusters, standardize=True, use_pca=True )

    print('Results:')
    print('Score k-means:', score_kmeans)

    print('Visualizing...')
    s = (8,6)
    imgutils.show_large_heatmap(df, 'kmeans', imgfiles, n_rows=n_image_rows, n_cols=n_image_cols, fig_size=fig_size, opacity=alpha)

    if (return_df): return df
```

4 clusters (4x4, 6x6, 8x8)

In [34]:
```
run_fullpipeline_pca(datafolder, n_tiles_y, n_tiles_x, 4, 4, feature_funcs, 4)
```

```
Working...
Results:
Score k-means: 0.3704474711797645
Visualizing...
```

Heats from: kmeans

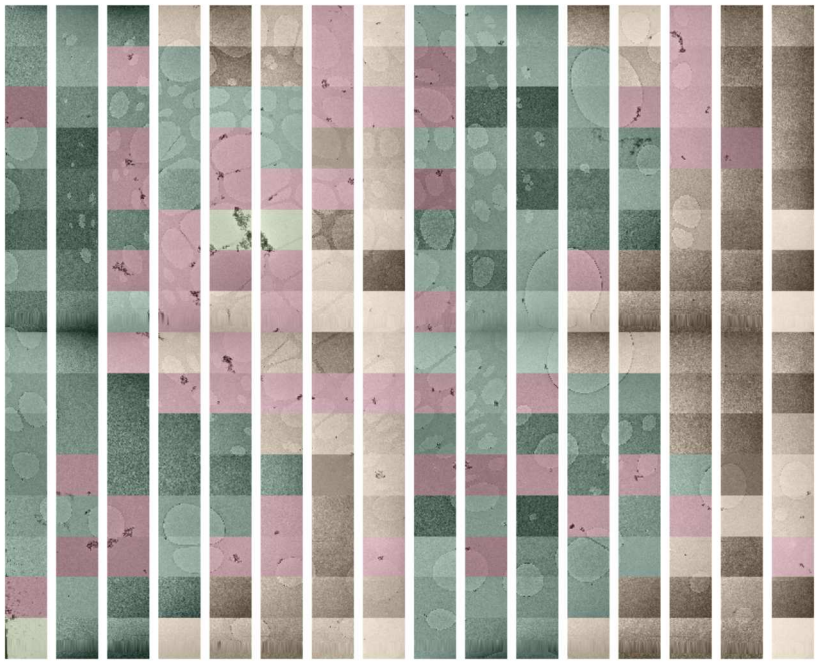


mwah, but colorcoding also makes it difficult to judge

In [35]:
```
run_fullpipeline_pca(datafolder, n_tiles_y, n_tiles_x, 8, 8, feature_funcs, 4)
```

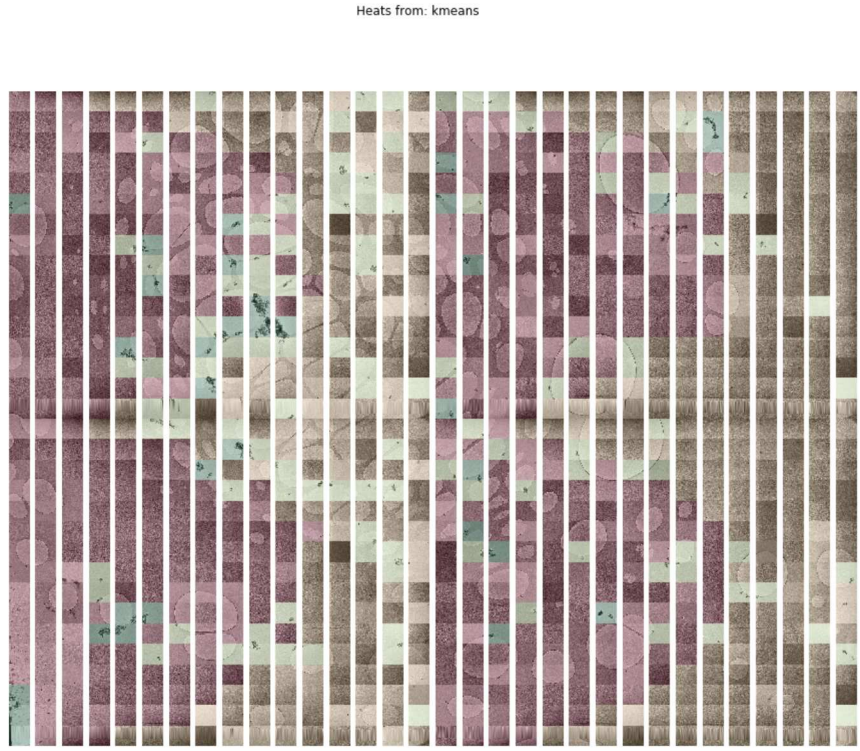
```
Working...
Results:
Score k-means: 0.4241338560199051
Visualizing...
```

Heats from: kmeans



In [36]:
```
run_fullpipeline_pca(datafolder, n_tiles_y, n_tiles_x, 16, 16, feature_funcs, 4)
```

```
Working...
Results:
Score k-means: 0.4978774492634023
Visualizing...
```



reasonable

Need 'grid search' versions to scan parameter space!!!

9. (Grid) Search for hyper-parameter optimizations

In [ ]:
```
default_grid_x = 10
default_grid_y = 10
```

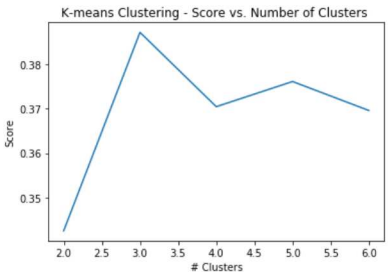
In [ ]:
```
imgfiles = import_data(datafolder)
df, feature_names = extract_features(imgfiles, feature_funcs, default_grid_y, default_grid_x)
```

9A. Number of clusters

In [37]:
```
def optimize_kmeans_nclusters(df_imgstats, feature_names, n_clusters_start, n_clusters_end):
    """
    Run the full pipeline (except for visualization) for different number of clusters
    and show graph with score vs number of clusters.
    """
    n_count = n_clusters_end + 1 - n_clusters_start
    result = np.empty(shape=(0,2), dtype=float)

    for n in range(n_clusters_start, n_clusters_end+1):
        score = run_kmeans_pipeline(df, feature_names, n, standardize=True, use_pca=True )
        result = np.append(result, np.array([[n, score]]), axis=0)

    plt.plot(result[:,0], result[:,1])
    plt.title('K-means Clustering - Score vs. Number of Clusters ')
    plt.xlabel('# Clusters')
    plt.ylabel('Score')
    plt.show()
```

In [ ]:
```
```

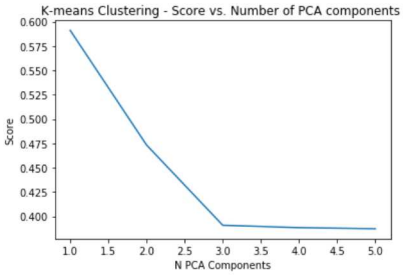
In [38]:
```
optimize_kmeans_nclusters(df, feature_names, 2, 6)
```



9B. Number of features / PCA Components

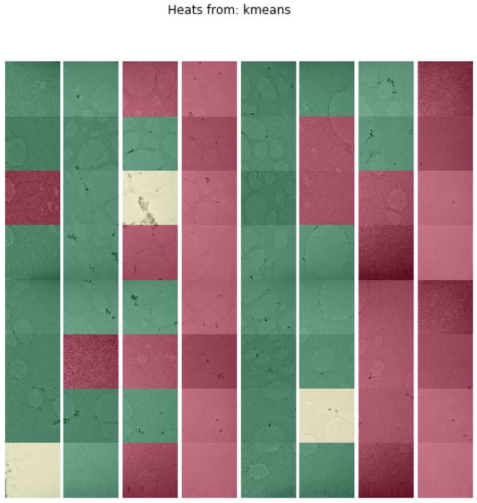
In [39]:
```
def optimize_kmeans_pcacomponents(df_imgstats, feature_names, n_clusters):
    """
    Run the full pipeline (except for visualization) for different number of PCA componenents
    and show graph with the score vs number of components.
    """
    n_count = len(feature_names) - 1
    result = np.empty(shape=(0,2), dtype=float)

    for n in range(1, len(feature_names)):
        score = run_kmeans_pipeline(df, feature_names, n_clusters, standardize=True, use_pca=True, n_pca = n )
        result = np.append(result, np.array([[n, score]]), axis=0)

    plt.plot(result[:,0], result[:,1])
    plt.title('K-means Clustering - Score vs. Number of PCA components ')
    plt.xlabel('N PCA Components')
    plt.ylabel('Score')
    plt.show()
```

In [40]:
```
optimize_kmeans_pcacomponents(df, feature_names, n_clusters=3)
```



Interesting. This graph says that a single component (after PCA transform) would already do a good job. Let's visualize

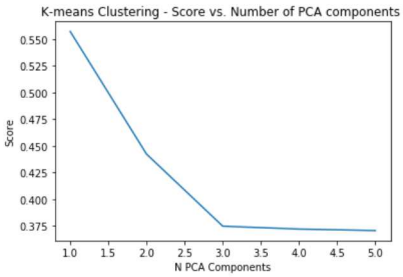
In [42]:
```
score = run_kmeans_pipeline(df, feature_names, 3, standardize=True, use_pca=True, n_pca=1)
imgutils.show_large_heatmap(df, 'kmeans', imgfiles, n_rows=n_tiles_y, n_cols=n_tiles_x, fig_size=(10,8))
```



Indeed not bad. Maybe not so surprise as in early EDA we saw that on this particular set the standard deviation is very informative (we used it for manual labelling)

A 'manual grid search' showed for 4 clusters the same

In [43]:
```
optimize_kmeans_pcacomponents(df, feature_names, 4)
score = run_kmeans_pipeline(df, feature_names, 4, standardize=True, use_pca=True, n_pca=1)
imgutils.show_large_heatmap(df, 'kmeans', imgfiles, n_rows=n_tiles_y, n_cols=n_tiles_x, fig_size=(10,8))
```



Heats from: kmeans

Actually this looks very good! It properly grouped 'fully occupied grid cells' from 'partially occupied'!

Don't know what these graphs really say; as the score is assessing cluster cohesion based on the last transformation in the pre-processing, it can be expected that less components are more separated. Maybe need to look at the 'elbows' here?

10. (Grid) Search for number of patches (expensive)

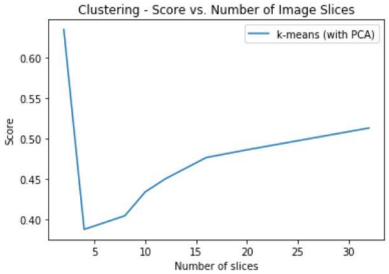
As the image slicing and feature extraction are the computationally expensive parts, both algorithms will be run in the search

10A. Optimize and plot graphs

In [47]:
```
import sys

def optimize_slices_pca(imagefolder, feature_funcs, n_clusters, list_n_slices):
    """
    Run the full pipeline (without visualization) for different number of image slices
    and show graph of score vs number of slices.
    Note that same number of slices is used in x and y direction.
    Returns a list of tuples of form (dataframe, n_slices, score_kmeans)
    """
    imgfiles = import_data(imagefolder)
    results = []

    print("Running slice optimization for {} clusters:".format(n_clusters))
    counter=0
    for n_slices in list_n_slices:
        progress = 100 * counter / len(list_n_slices)
        sys.stdout.write("{:.1f} %".format(progress) + "\r")

        df, feature_names = extract_features(imgfiles, feature_funcs, n_grid_rows=n_slices, n_grid_cols=n_slices)
        score_kmeans = run_kmeans_pipeline(df, feature_names, n_clusters, standardize=True, use_pca=True )
        results.append((df, n_slices, score_kmeans))

        counter += 1

    sys.stdout.write("Done.")
    #sys.stdout.flush()

    plotdata = list(zip(*results))

    plt.plot(plotdata[1], plotdata[2], label="k-means (with PCA)")
    plt.title('Clustering - Score vs. Number of Image Slices')
    plt.xlabel('Number of slices')
    plt.ylabel('Score')
    plt.legend(loc='upper right')
    plt.show()

    return results
```

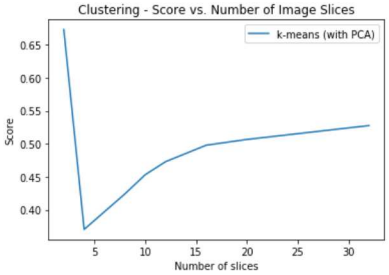
In [48]:
```
slice_opt_results_3clusters = optimize_slices_pca(datafolder, feature_funcs, n_clusters=3, list_n_slices=[2,4,8,10,12,16,20,32])
```

```
Running slice optimization for 3 clusters:
Done.%
```



In [49]:
```
slice_opt_results_4clusters = optimize_slices_pca(datafolder, feature_funcs, n_clusters=4, list_n_slices=[2,4,8,10,12,16,20,32])
```

```
Running slice optimization for 4 clusters:
Done.%
```



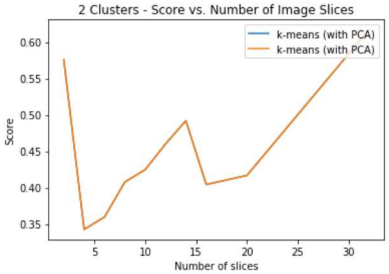
Interesting, optimum number of slices is around 16 for 3 clusters, and even higher number of patches for 4 clusters.

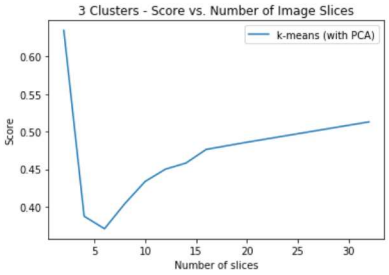
Also it is more efficient to do a **grid search** for clusters vs patches, as cluster evaluation is cheap while patches is expensive.

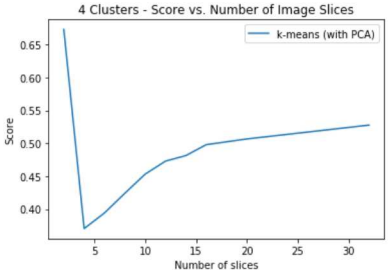
In [52]:
```
import sys

def gridsearch_slices_pca(imagefolder, feature_funcs, list_n_clusters, list_n_slices):
    """
    Run the full pipeline (without visualization) for different number of image slices
    and show graph of score vs number of slices.
    Note that same number of slices is used in x and y direction.
    Returns a list of tuples of form (dataframe, n_slices, n_clusters, score_kmeans)
    """
    imgfiles = import_data(imagefolder)
    results = []

    print("Running slice-#clusters grid-search...")
    counter=0
    for n_slices in list_n_slices:
        progress = 100 * counter / len(list_n_slices)
        sys.stdout.write("{:.1f} %".format(progress) + "\r")

        df, feature_names = extract_features(imgfiles, feature_funcs, n_grid_rows=n_slices, n_grid_cols=n_slices)

        for n_clust in list_n_clusters:
            df2 = df.copy()   # each result should have its own dataframe otherwise results are overwritten
            score_kmeans = run_kmeans_pipeline(df2, feature_names, n_clust, standardize=True, use_pca=True )
            results.append((df2, n_slices, n_clust, score_kmeans))

        counter += 1

    sys.stdout.write("Done.")

    for n_clust in list_n_clusters:
        results_for_cluster = [t for t in results if t[2]==n_clust]
        plotdata = list(zip(*results_for_cluster))

        plt.plot(plotdata[1], plotdata[3], label="k-means (with PCA)")
        plt.title('{} Clusters - Score vs. Number of Image Slices'.format(n_clust))
        plt.xlabel('Number of slices')
        plt.ylabel('Score')
        plt.legend(loc='upper right')
        plt.show()


    return results
```

In [53]:
```
slice_grid_search = gridsearch_slices_pca(datafolder, feature_funcs, list_n_clusters=[2,3,4], list_n_slices=[2,4,6,8,10,12,14,16,20,32])
```

```
Running slice-#clusters grid-search...
Done.%
```







(I one time ran it up to 64 slices, but that take forever. k-means flattens and hierarchical goes drops drastically)

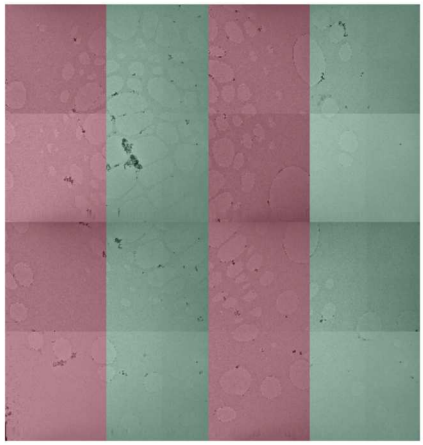
10B. Look at some heatmaps for 2 clusters

In [58]:
```
def get_df_from_slicegridsearch(results, num_clust, num_slices):
    results_for_cluster = [t for t in results if t[2]==num_clust]
    item_in_list = [t for t in results_for_cluster if t[1]==num_slices][0]
    return item_in_list[0]

def show_from_slicegridsearch(results, num_clust, num_slices, n_img_rows, n_img_cols, heat_col='kmeans', fig_size=(12,10)):
    df_toshow = get_df_from_slicegridsearch(results, num_clust, num_slices)
    imgfiles = df_toshow['filename'].unique().tolist()
    extratitle = "{}x{} slices, {} clusters".format(num_slices,num_slices, num_clust)
    imgutils.show_large_heatmap(df_toshow, heat_col, imgfiles, n_rows=n_img_rows, n_cols=n_img_cols, fig_size=fig_size, subtitle=extratitle, opacity=0.3)
```

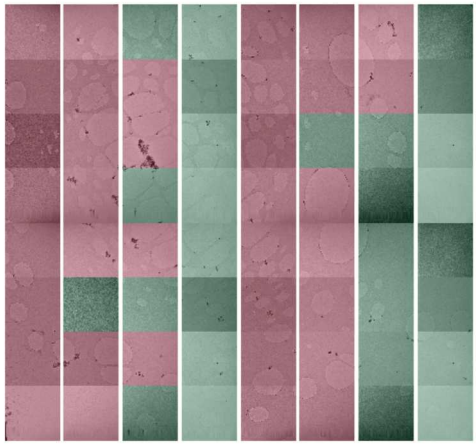
In [59]:
```
def show_multiple_from_slicegridsearch(results, list_num_clust, list_num_slices, n_img_rows, n_img_cols,
                                       heat_col='kmeans', fig_size=(12,10)):
    for n_slices in list_num_slices:
        for n_clust in list_num_clust:
            show_from_slicegridsearch(results, n_clust, n_slices, n_img_rows, n_img_cols, heat_col,  fig_size=fig_size)
```

In [61]:
```
show_multiple_from_slicegridsearch(slice_grid_search, [2], [2, 4, 8 ,10, 16], n_img_rows=n_tiles_y, n_img_cols=n_tiles_x, heat_col='kmeans',  fig_size=(10,8))
```

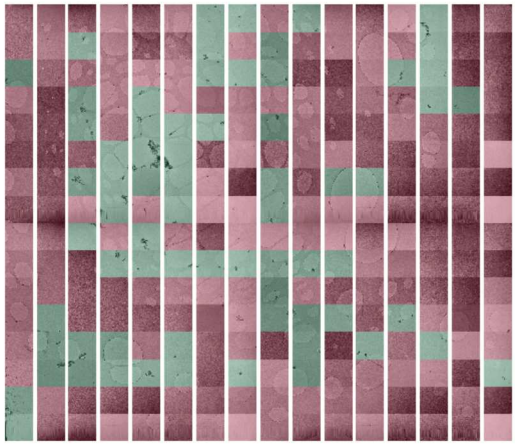
Heats from: kmeans - 2x2 slices, 2 clusters



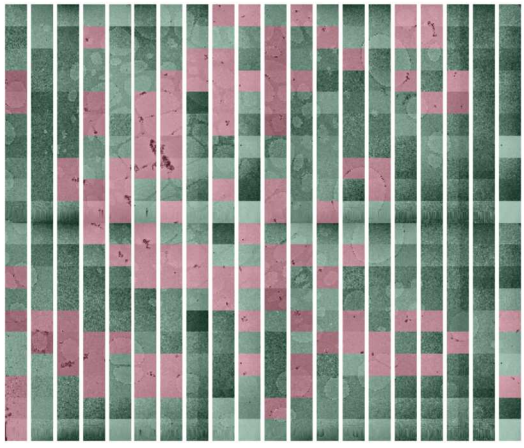
Heats from: kmeans - 4x4 slices, 2 clusters



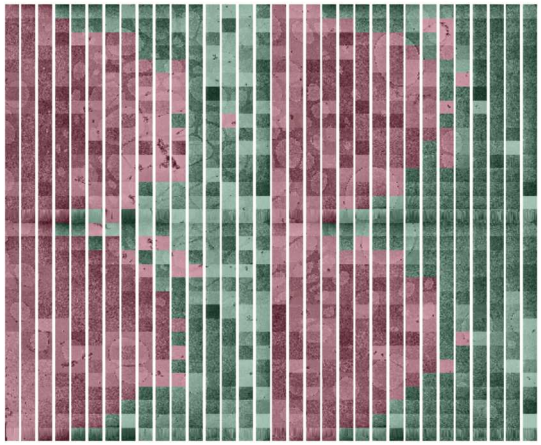
Heats from: kmeans - 8x8 slices, 2 clusters



Heats from: kmeans - 10x10 slices, 2 clusters



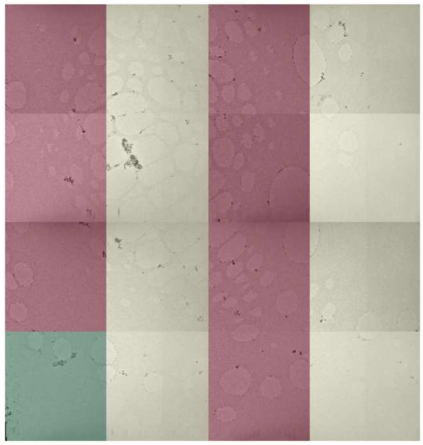
Heats from: kmeans - 16x16 slices, 2 clusters



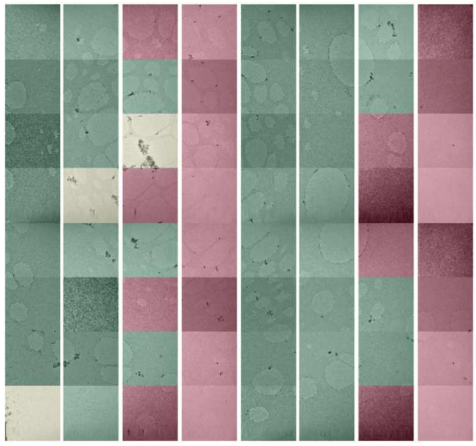
10c. Look at some heatmaps for3 clusters

In [62]:
```
show_multiple_from_slicegridsearch(slice_grid_search, [3], [2, 4, 8 ,10, 16], n_img_rows=n_tiles_y, n_img_cols=n_tiles_x, heat_col='kmeans',  fig_size=(10,8))
```

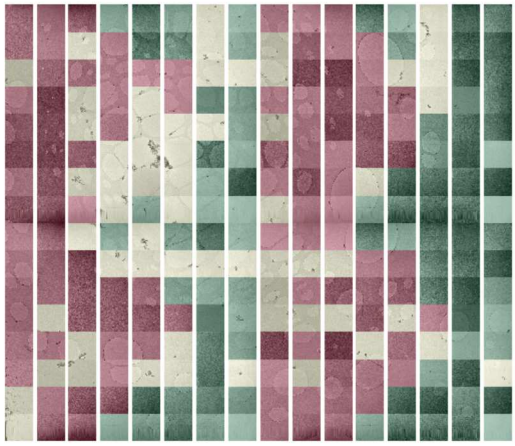
Heats from: kmeans - 2x2 slices, 3 clusters



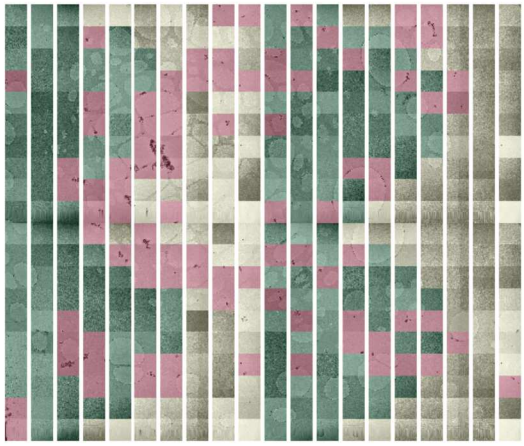
Heats from: kmeans - 4x4 slices, 3 clusters



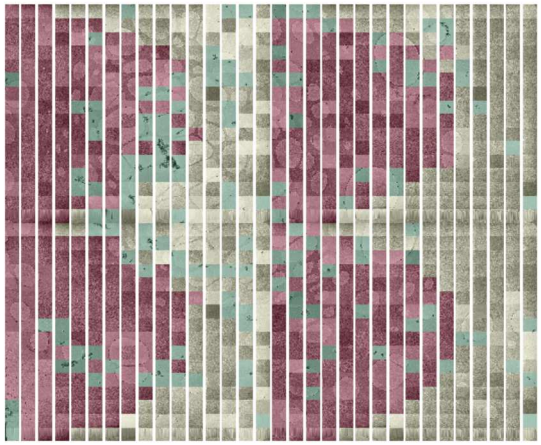
Heats from: kmeans - 8x8 slices, 3 clusters



Heats from: kmeans - 10x10 slices, 3 clusters



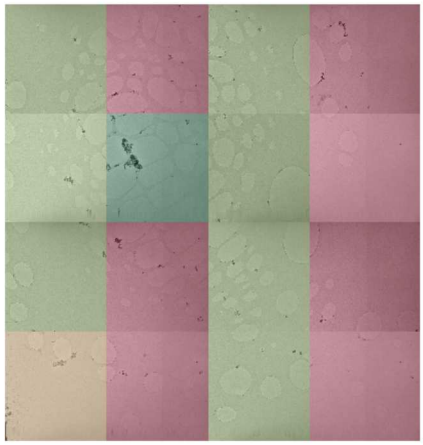
Heats from: kmeans - 16x16 slices, 3 clusters



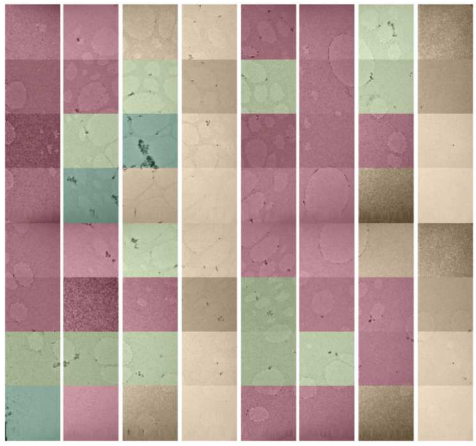
10D. Look at some heatmaps for 4 clusters

In [63]:
```
show_multiple_from_slicegridsearch(slice_grid_search, [4], [2, 4, 8 ,10, 16], n_img_rows=n_tiles_y, n_img_cols=n_tiles_x, heat_col='kmeans',  fig_size=(10,8))
```

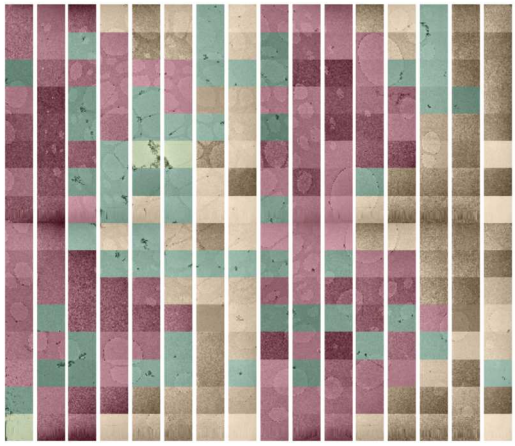
Heats from: kmeans - 2x2 slices, 4 clusters



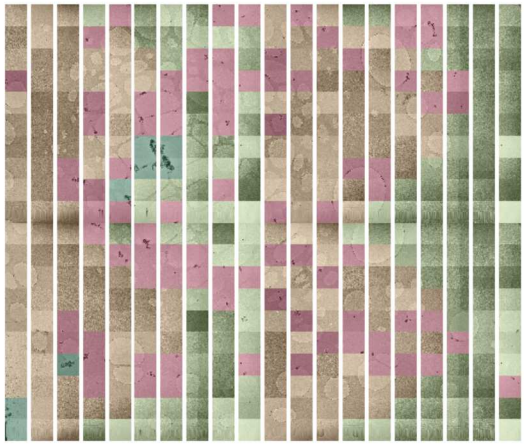
Heats from: kmeans - 4x4 slices, 4 clusters



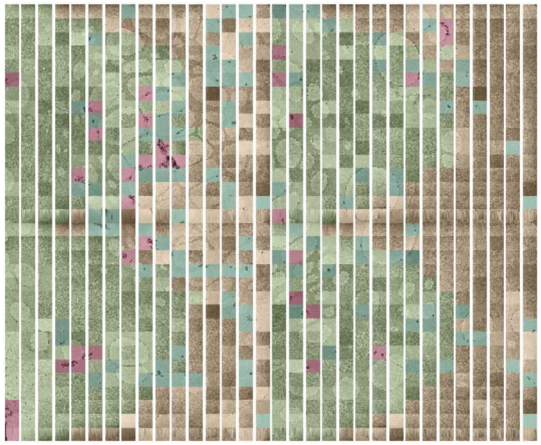
Heats from: kmeans - 8x8 slices, 4 clusters



Heats from: kmeans - 10x10 slices, 4 clusters



Heats from: kmeans - 16x16 slices, 4 clusters

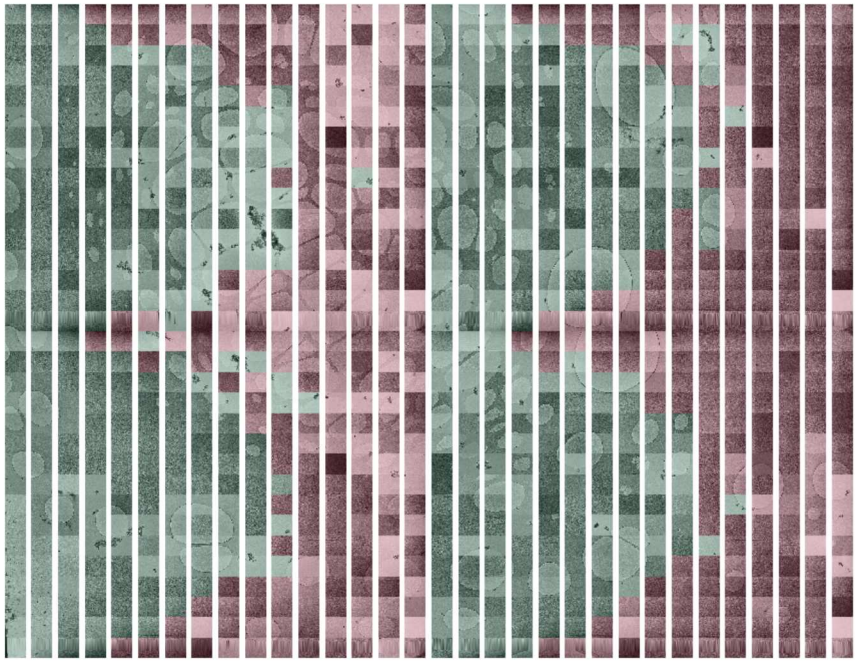


11. Finally, run it once more with most optimal settings

In [64]:
```
run_fullpipeline_pca(datafolder, n_tiles_y, n_tiles_x, 16, 16, feature_funcs, 2, fig_size=(16,12))
```

```
Working...
Results:
Score k-means: 0.40428330918298055
Visualizing...
```

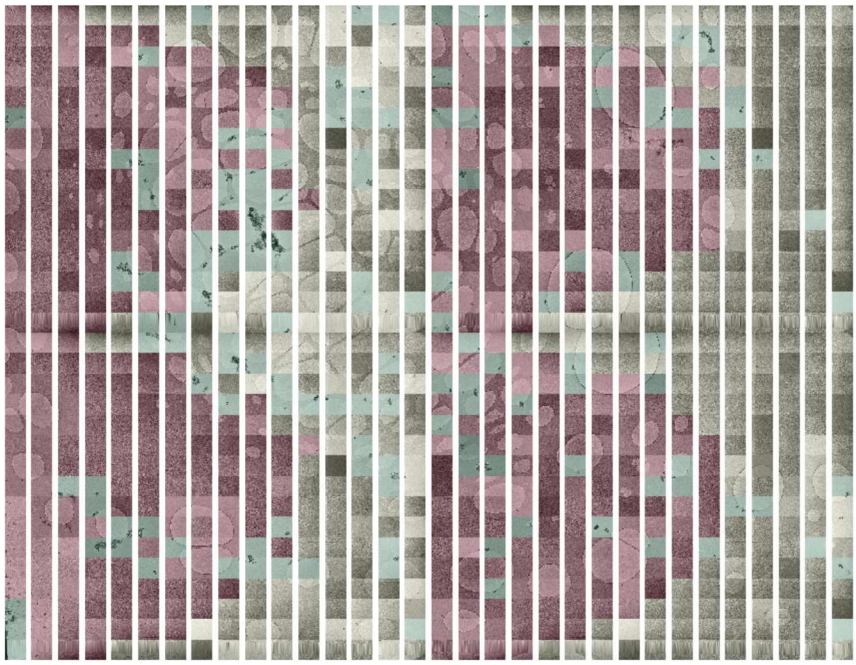
Heats from: kmeans



In [65]:
```
run_fullpipeline_pca(datafolder, n_tiles_y, n_tiles_x, 16, 16, feature_funcs, 3, fig_size=(16,12))
```

```
Working...
Results:
Score k-means: 0.4763101434435471
Visualizing...
```

Heats from: kmeans



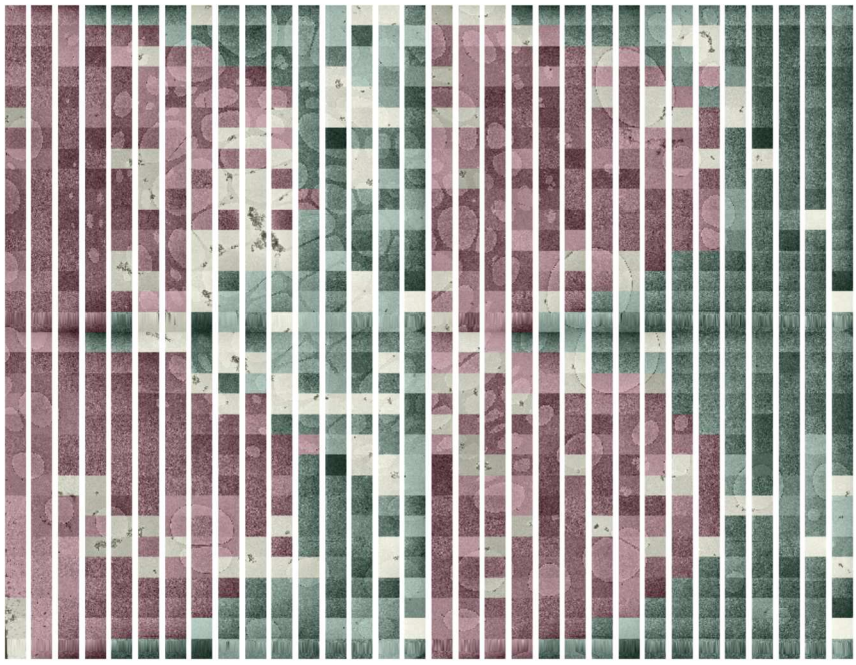
Looks like the mean has to much impact, maybe exclude it in the features

In [66]:
```
feature_funcs2 = [imgutils.img_std, imgutils.img_relstd,
                  imgutils.img_skewness,  imgutils.img_kurtosis, imgutils.img_mode]
feature_names2 = imgutils.stat_names(feature_funcs)
```

In [67]:
```
run_fullpipeline_pca(datafolder, n_tiles_y, n_tiles_x, 16, 16, feature_funcs, 3, fig_size=(16,12))
```

```
Working...
Results:
Score k-means: 0.4763101434435471
Visualizing...
```

Heats from: kmeans

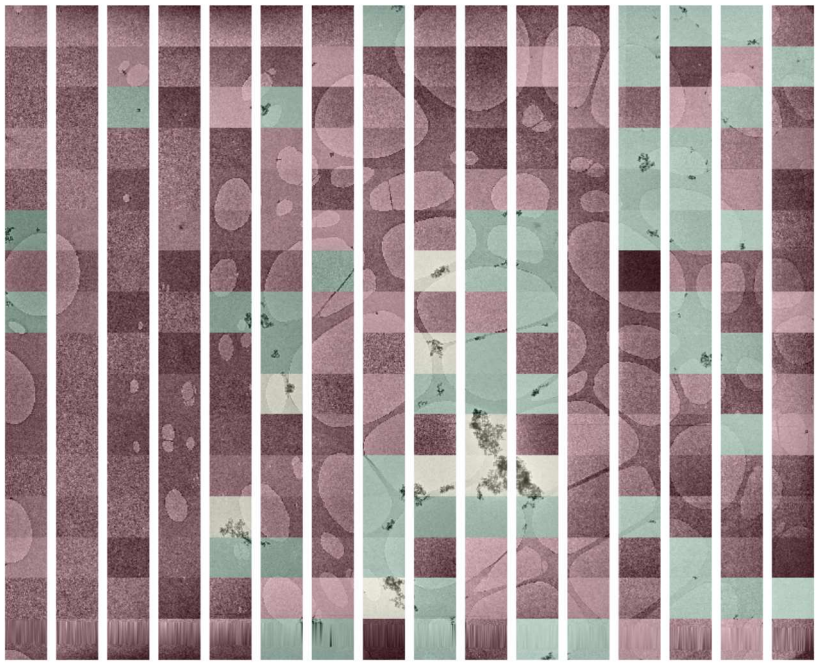


This finally starts to look like it (not perfect yet)

But it's also hard to see, lets look on a single file

In [68]:
```
run_fullpipeline_pca_filelist([imgfiles[0]], 1, 1, 16, 16, feature_funcs2, 3, fig_size=(16,12))
```

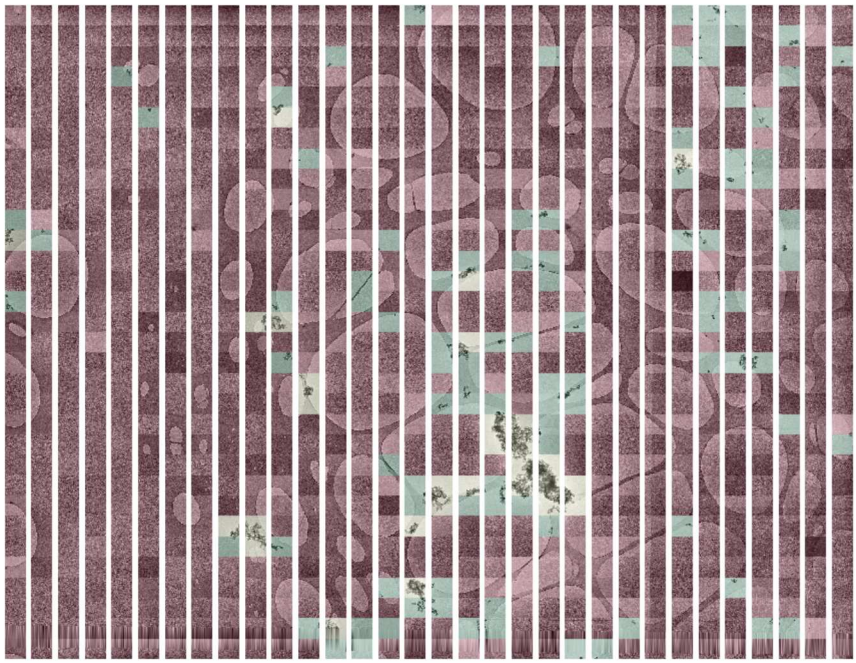
```
Working...
Results:
Score k-means: 0.523070123155694
Visualizing...
```



Yep, looks reasonable. Maybe tryout with even smaller patches

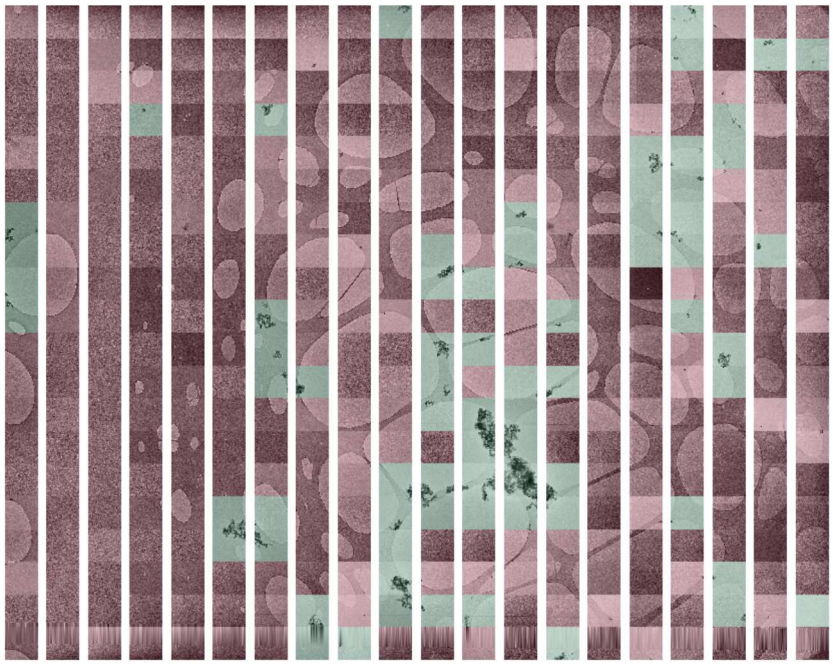
In [69]:
```
run_fullpipeline_pca_filelist([imgfiles[0]], 1, 1, 32, 32, feature_funcs2, 3, fig_size=(16,12))
```

```
Working...
Results:
Score k-means: 0.6524583895470091
Visualizing...
```



In [70]:
```
run_fullpipeline_pca_filelist([imgfiles[0]], 1, 1, 20, 20, feature_funcs2, 2, fig_size=(16,12), alpha = 0.2)
```

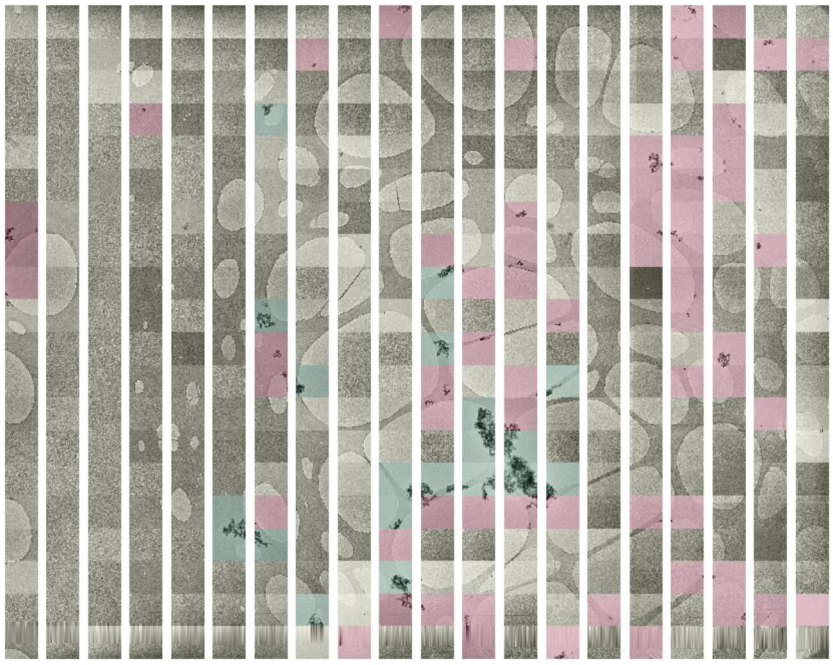
```
Working...
Results:
Score k-means: 0.5710543727985685
Visualizing...
```



Look good as far as I can judge. Now also try with 3

In [71]:
```
run_fullpipeline_pca_filelist([imgfiles[0]], 1, 1, 20, 20, feature_funcs2, 3, fig_size=(16,12), alpha = 0.2)
```

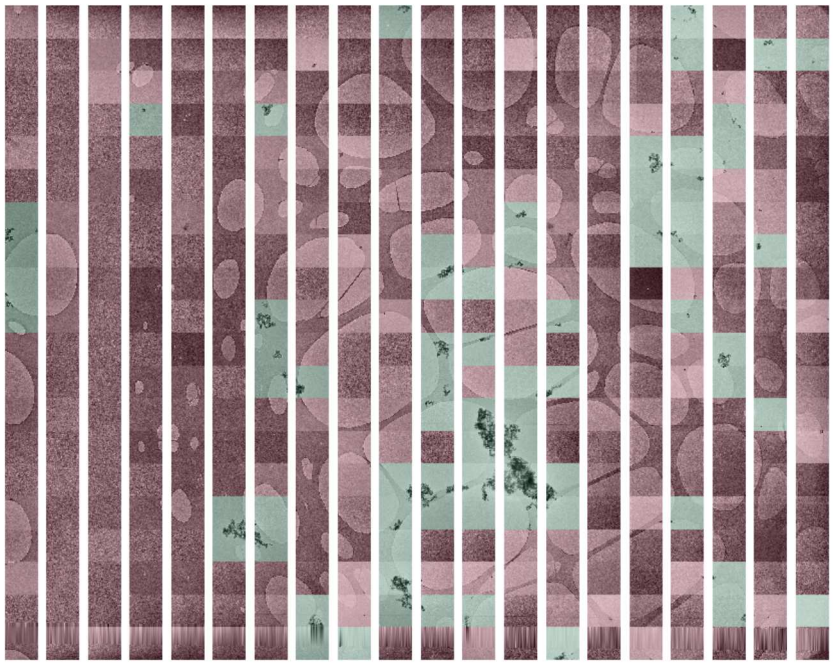
```
Working...
Results:
Score k-means: 0.553756604958431
Visualizing...
```



and try once more with mean include

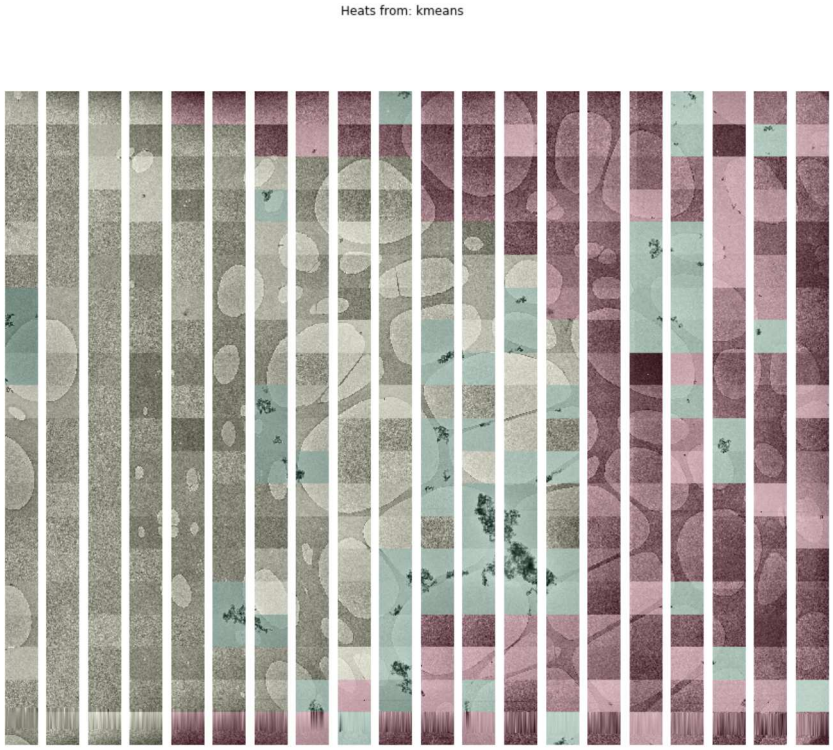
In [72]:
```
run_fullpipeline_pca_filelist([imgfiles[0]], 1, 1, 20, 20, feature_funcs, 2, fig_size=(16,12), alpha = 0.2)
```

```
Working...
Results:
Score k-means: 0.5084303261094089
Visualizing...
```



In [73]:
```
run_fullpipeline_pca_filelist([imgfiles[0]], 1, 1, 20, 20, feature_funcs, 3, fig_size=(16,12), alpha = 0.2)
```

```
Working...
Results:
Score k-means: 0.4781325672994294
Visualizing...
```

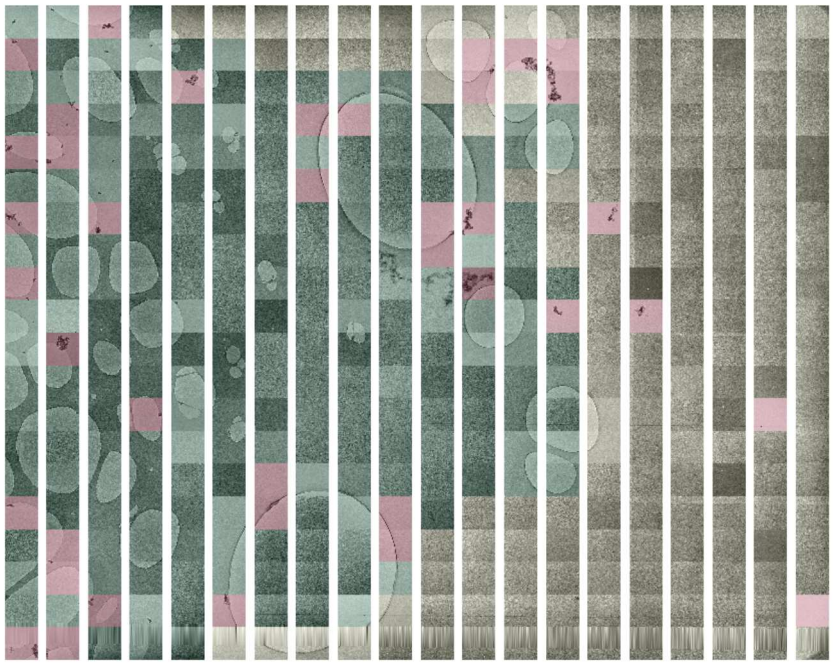


So, both 2 and 3 clusters will work, with 3 being a bit more strict (missing more but also less mistakes)

Also check some of the other images

In [80]:
```
run_fullpipeline_pca_filelist([imgfiles[1]], 1, 1, 20, 20, feature_funcs, 3, fig_size=(16,12), alpha = 0.2)
```

```
Working...
Results:
Score k-means: 0.49236846496408515
Visualizing...
```

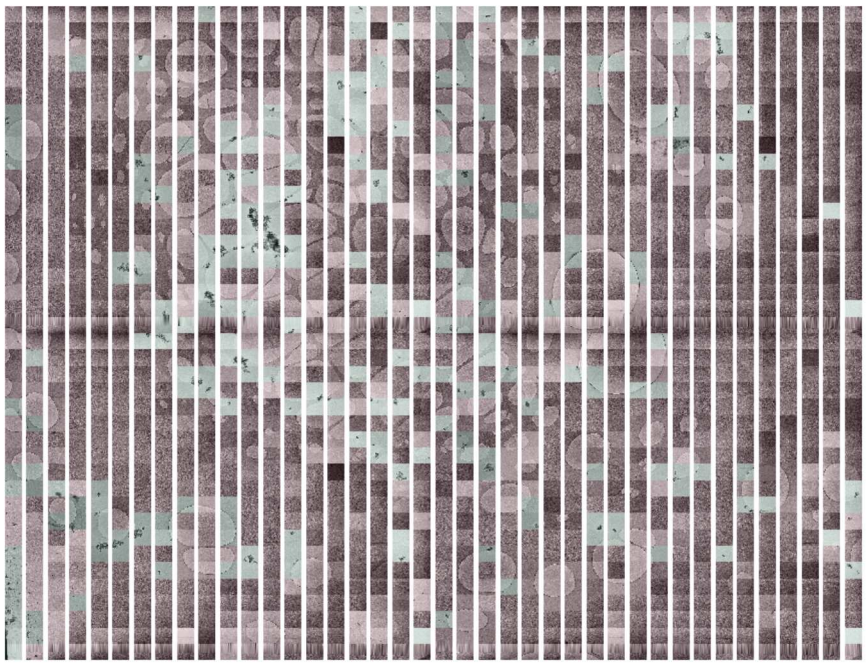


Not so good: some black borders also picked up and some blacks crystals are missed.

A final runs on the entire set

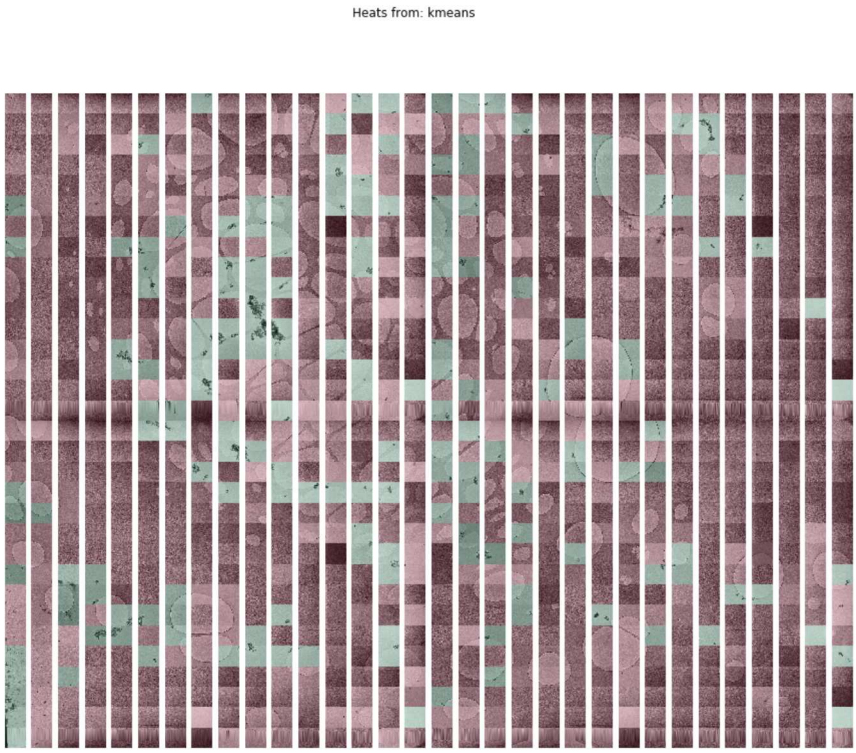
In [88]:
```
run_fullpipeline_pca_filelist(imgfiles, n_tiles_y, n_tiles_x, 20, 20, feature_funcs2, 2 ,fig_size=(16,12), alpha=0.1)
```

```
Working...
Results:
Score k-means: 0.5933803319669712
Visualizing...
```



In [83]:
```
run_fualpipeline_pca_filelist(imgfiles, n_tiles_y, n_tiles_x, 16, 16, feature_funcs2, 2 ,fig_size=(16,12), alpha=0.2)
```

```
Working...
Results:
Score k-means: 0.5653552598835074
Visualizing...
```



In [105]:
```
run_fullpipeline_pca_filelist(imgfiles, n_tiles_y, n_tiles_x, 10, 10, feature_funcs2, 2 ,fig_size=(14,12), alpha=0.2)
```

```
Working...
Results:
Score k-means: 0.4849711067294329
Visualizing...
```

In [110]:
```
run_fullpipeline_pca_filelist(imgfiles, n_tiles_y, n_tiles_x, 10, 10, feature_funcs2, 3 ,fig_size=(14,12), alpha=0.2)
```

```
Working...
Results:
Score k-means: 0.4894880485271768
Visualizing...
```

Reasonable result (but not perfect) (from visual inspection)

12. Conclusions:

- Could get it working on this data set if..
- ... black tiles are ommitted
- ... patch size is small enough
- It looks like for this set, the 'mean' is not the best statistic (without results were slightly better)
- Needed to improve the large heatmap visualization a bit in order to properly see it
- As results are sensitive to the parametrizations 'Path Size' and 'number of clusters', in a final implementation one should consider making these available to the end-user (e.g. as sliders). In such case it is key though that performance is good enough

Michael Janus, 27 October 2018