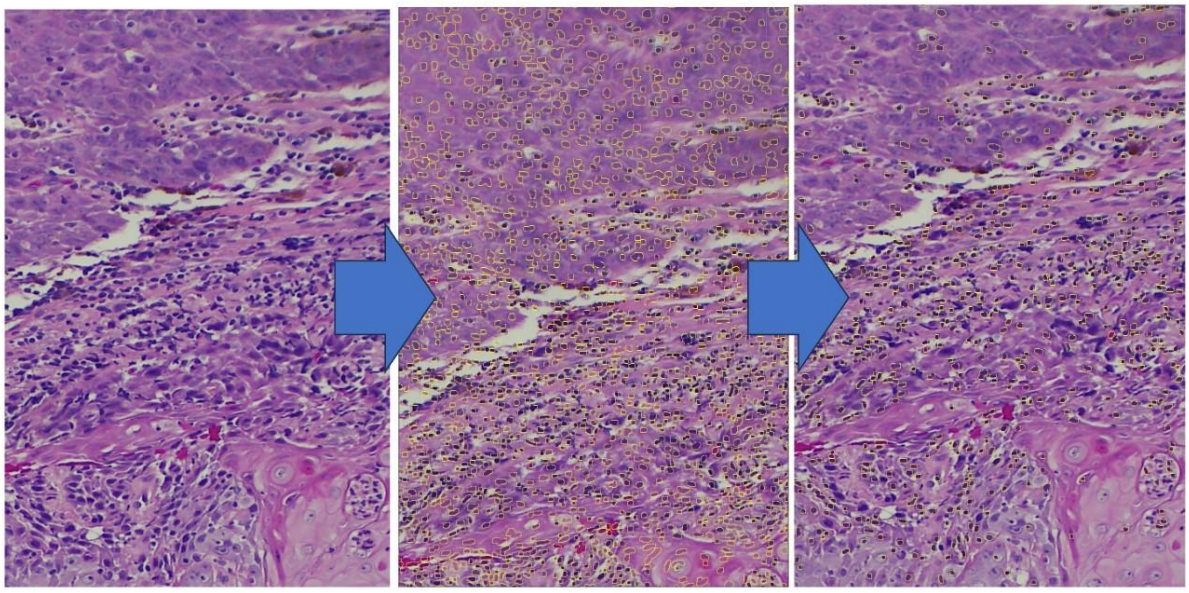
Supplementary Information

Supplementary material 1



Supplementary image: Illustration of the nuclei detection and segmentation pipeline used for Tumor-Infiltrating Lymphocyte (TIL) quantification. The process begins with maxima detection on H&E-stained tissue, followed by clustering and selection at higher magnification. Yellow outlines indicate regions identified as nuclei, forming the basis for subsequent k-means clustering stages in the unsupervised algorithm

Supplementary material 2

```
name=File.name;
run("Gaussian Blur...", "sigma=3");
run("Find Maxima...", "prominence=5 strict light output=[Maxima Within Tolerance]");
run("Dilate");
run("Dilate");
run("Dilate");
run("Dilate");
run("Create Selection");
roiManager("Add");
selectImage(name);
roiManager("Select", 0);
run("Make Inverse");
setBackgroundColor(0, 0, 0);
run("Clear", "slice");
selectImage(name);
```

```
run("Select None");
run("RGB Stack");
setThreshold(1, 255, "raw");
run("Analyze Particles...", "  show=[Count Masks] display exclude stack");
saveAs("Results", "E:/yourpath/measure/Results "+name+".csv");
run("Clear Results");
roiManager("Deselect");
roiManager("Delete");
selectImage("Count Masks of "+name);
saveAs("Tiff", "E:/yourpath/mask/Count Masks of "+name+".tif");
close;
selectImage(name+" Maxima");
close;
selectImage(name);
close;
```

Supplementary Text 1. ImageJ Macro for Pre-processing and Nuclei Segmentation. This text file (.txt) contains the automated script used in ImageJ (v1.54j) to perform initial image segmentation. The macro executes a standardized workflow including Gaussian blur (sigma=3) for noise reduction, Find Maxima detection (prominence=5) for identifying nuclei centroids, and iterative morphological dilation to approximate nuclear boundaries. The script concludes by running Analyze Particles to extract quantitative morphometric features (Area, RGB intensity, circularity, etc.) and exports binary mask images (.tif) and feature tables (.csv) for downstream analysis

Supplementary material 3

```
# -*- coding: utf-8 -*-
"""
Refactored on [Current Date]
Original Author: okky
"""

import os
import time
import pandas as pd
import numpy as np
import seaborn as sns
import matplotlib.pyplot as plt
import tifffile as tiff
from pathlib import Path
from sklearn.cluster import KMeans
from tqdm import tqdm
from PIL import Image
from matplotlib.colors import Normalize
```

```
# --- Configuration ---
BASE_PATH = Path(your_path)
DATA_PATH = BASE_PATH / "measure reordered"
MASK_PATH = BASE_PATH / "mask"
OUTPUT_IMG_PATH = BASE_PATH / "K means img"
OUTPUT_IMG_FULL_PATH = BASE_PATH / "K means img with non lymphocytes"
OUTPUT_CSV_PATH = BASE_PATH / "lymphocyte_counts.csv"

# Ensure output directories exist
OUTPUT_IMG_PATH.mkdir(parents=True, exist_ok=True)
OUTPUT_IMG_FULL_PATH.mkdir(parents=True, exist_ok=True)

def plot_clusters(df, title_suffix):
    """Helper function to generate scatter plots."""
    sample = df.sample(frac=0.05, random_state=1)

    pairs = [
        ("mean luminance", "log2_Area"),
        ("mean luminance", "Mean_Red"),
        ("mean luminance", "Mean_Blue"),
        ("Mean_Red", "Mean_Blue"),
        ("Min_Blue", "Max_Blue"),
    ]

    if "log2_Area" not in df.columns:
        pairs[0] = ("mean luminance", "Area")

    for x_col, y_col in pairs:
        plt.figure()
        sns.scatterplot(data=sample, x=x_col, y=y_col, hue="K Cluster")
        plt.title(f'{x_col} vs {y_col} - {title_suffix}')
        plt.show()

def process_images_vectorized(file_list, df_data, output_folder, mask_folder):
    """
    Generates images using NumPy vectorization instead of loops.
    This is significantly faster than iterating through object IDs.
    """
    colormap = plt.cm.viridis

    for file_name in tqdm(file_list, desc=f"Generating Images in {output_folder.name}"):
        image_name_clean = file_name.replace("Mask - ", "").replace(".tif", "")

        # Filter data for this specific image
        img_data = df_data[df_data["image"] == image_name_clean]

        if img_data.empty:
            continue
```

```
output_file = output_folder / f"{image_name_clean}.png"

# Skip if already exists
if output_file.exists():
    continue

start_time = time.time()

# Load Tiff Mask (Assuming single channel)
try:
    mask_arr = tiff.imread(mask_folder / file_name)
    if mask_arr.ndim == 3:
        mask_arr = mask_arr[0, :, :] # Take first channel if 3D
except Exception as e:
    print(f"Error reading {file_name}: {e}")
    continue

# --- Vectorized Mapping Strategy ---
# 1. Create a lookup array (mapping) where index = object_ID and value = cluster_ID
# We need the max ID in the mask to size the array correctly
max_id = mask_arr.max()

# Default value is 0 (background)
# We assume cluster IDs are 1-based integers for visualization
mapping = np.zeros(max_id + 1, dtype=np.uint8)

# Map object indices to their Cluster ID (+1 to distinguish from background 0)
# Ensure indices exist in the mask range
valid_indices = img_data["idx"][img_data["idx"] <= max_id]
valid_clusters = img_data.loc[img_data["idx"] <= max_id, "K Cluster"]

# Assign clusters to the mapping array
mapping[valid_indices] = valid_clusters + 1

# 2. Apply mapping: This replaces all pixels at once
# clustered_img will have values 0 (bg), 1, 2, 3 etc. corresponding to clusters
clustered_img = mapping[mask_arr]

# Normalize and Colorize
norm = Normalize(vmin=0, vmax=np.max(clustered_img))
colored_array = colormap(norm(clustered_img))

# Convert to 8-bit RGB
rgb_array = (colored_array[:, :, :3] * 255).astype(np.uint8)

# Save
Image.fromarray(rgb_array).save(output_file)
```

```
        end_time = time.time()
        # Optional: Print only if it takes too long, otherwise it spams the console
        if (end_time - start_time) > 1.0:
            print(f"Processed {image_name_clean} in {int(end_time - start_time)}s")

def main():
    # --- 1. Data Loading ---
    start_global = time.time()
    files = [f for f in os.listdir(DATA_PATH) if f.endswith('.csv')]

    df_list = []

    for file in tqdm(files, desc="Sampling Phase"):
        temp_df = pd.read_csv(DATA_PATH / file)

        # Create 'idx' column (1-based index)
        temp_df.insert(1, "idx", range(1, len(temp_df) + 1))

        # Track file name and length
        temp_df['source_file'] = file # Optional: keep track of source
        df_list.append(temp_df)

    # Concatenate once (Faster than concat in loop)
    sample_df = pd.concat(df_list, ignore_index=True)

    # Clean NaN values
    original_len = len(sample_df)
    # Assuming clustering columns start from 5th column roughly, adjusting logic to match original
    # Dropping rows where specific numeric columns might be NaN
    sample_df = sample_df.dropna()
    print(f"Dropped {original_len - len(sample_df)} rows containing NaN values.")
    print(f"Total objects: {len(sample_df)}")

    # Feature Engineering
    # Calculate sum of RGB means for luminance
    sample_df["mean luminance"] = sample_df.loc[:, "Mean_Red":"Mean_Blue"].sum(axis=1)
    sample_df['log2_Area'] = np.log2(sample_df['Area'])

    # --- 2. Clustering Phase 1: Flukes vs Cells ---
    print("\n--- Clustering Phase 1 (Flukes vs Cells) ---")
    start_c1 = time.time()

    # Using 'Area' and subsequent columns for features
    features_c1 = sample_df.loc[:, "Area":].columns

    kmeans_1 = KMeans(n_clusters=2, random_state=42)
```

```
    sample_df["K Cluster"] = kmeans_1.fit_predict(sample_df.loc[:, "Area":]) # Implicitly using numeric cols

    # Visualization Phase 1
    plot_clusters(sample_df, "Phase 1")

    # Separation: Cluster 1 = Flukes, Cluster 0 = Cells (based on user logic)
    df_flukes = sample_df[sample_df["K Cluster"] == 1].copy()
    sample_df = sample_df[sample_df["K Cluster"] == 0].copy()

    print(f"Phase 1 Runtime: {int(time.time() - start_c1)} seconds. Flukes removed.")

    # --- 3. Clustering Phase 2: Refinement ---
    print("\n--- Clustering Phase 2 ---")

    # Features from 'Circ.' to 'StdDev_Blue'
    # Note: Ensure columns exist. Adapting slice logic from original code.
    features_c2 = sample_df.loc[:, "Circ.":"StdDev_Blue"]

    kmeans_2 = KMeans(n_clusters=2, random_state=77)
    sample_df["K Cluster"] = kmeans_2.fit_predict(features_c2)

    # Save the non-cluster-0 cells aside (assuming cluster 0 is the one we keep drilling down?)
    # Original code logic: df_cells_1 are those NOT 0
    df_cells_1 = sample_df[sample_df["K Cluster"] != 0].copy()
    sample_df = sample_df[sample_df["K Cluster"] == 0].copy() # Keep working on cluster 0

    plot_clusters(sample_df, "Phase 2")

    # --- 4. Clustering Phase 3: Final Lymphocyte ID ---
    print("\n--- Clustering Phase 3 ---")

    # Move log2_Area position if needed (simulating original pop/insert)
    # Ideally, just select columns explicitly rather than moving them in the DF
    features_c3 = sample_df.loc[:, "log2_Area":"StdDev_Blue"]

    kmeans_3 = KMeans(n_clusters=6, random_state=77)
    sample_df["K Cluster"] = kmeans_3.fit_predict(features_c3)

    # Visualization Phase 3
    plot_clusters(sample_df, "Phase 3")

    # Based on original code: Clusters 1, 4, 5 are Lymphocytes
    lymphocytes = sample_df[sample_df["K Cluster"].isin([1, 4, 5])].copy()

    # Others are saved as 'cells_2'
    df_cells_2 = sample_df[~sample_df["K Cluster"].isin([1, 4, 5])].copy()
```

```
    # Reset final cluster ID for lymphocytes to 0 (for counting/csv)
    lymphocytes["K Cluster"] = 0

    # --- 5. Export Counts ---
    print("\n--- Exporting Counts ---")
    # Parsing slide name and image number.
    # Warning: Slicing [-5:-4] is risky. Assuming format ends in "...N.csv" or similar

    # Better to use Regex or split, but keeping logic close to original with safety:
    lymphocytes["slide"] = lymphocytes["image"].apply(lambda x: x[:-5])

    # Extract number (robustly)
    import re
    def extract_img_num(s):
        match = re.search(r'(\d+)$', s)
        return int(match.group(1)) if match else 0

    lymphocytes["no img"] = lymphocytes["image"].apply(lambda x: x[:-5]).apply(extract_img_num) #
Logic adapted

    # Group by slide
    count_df = lymphocytes.groupby("slide").agg(
        lymphocyte_count=("image", "count"),
        no_img=("no img", "max")
    ).reset_index()

    count_df.to_csv(OUTPUT_CSV_PATH, index=False)
    print("Lymphocyte counts saved.")

    # --- 6. Image Generation 1: Lymphocytes Only ---
    print("\n--- Generating Images (Lymphocytes Only) ---")

    # Prepare list of mask files
    mask_files = [f for f in os.listdir(MASK_PATH) if f.endswith(('.tif', '.tiff'))]

    # Process images using the fast vectorized function
    # Only passing the lymphocyte dataframe
    process_images_vectorized(mask_files, lymphocytes, OUTPUT_IMG_PATH, MASK_PATH)

    # --- 7. Prepare Full Dataset for Second Image Set ---
    # Re-assemble the dataset with distinct IDs for visualization

    # 1. Flukes -> Cluster 1
    df_flukes = df_flukes.sample(frac=0.4, random_state=1)
    df_flukes["K Cluster"] = 1

    # 2. Other Cells -> Cluster 2
    df_cells_combined = pd.concat([df_cells_1, df_cells_2])
```

```
    df_cells_combined = df_cells_combined.sample(frac=0.4, random_state=1)
    df_cells_combined["K Cluster"] = 2

    # 3. Lymphocytes -> Cluster 0 (Already set)
    # Combine all
    final_df = pd.concat([
        lymphocytes[["image", "idx", "K Cluster"]],
        df_flukes[["image", "idx", "K Cluster"]],
        df_cells_combined[["image", "idx", "K Cluster"]]
    ])

    # --- 8. Image Generation 2: All Objects ---
    print("\n--- Generating Images (All Objects) ---")
    process_images_vectorized(mask_files, final_df, OUTPUT_IMG_FULL_PATH, MASK_PATH)

    end_global = time.time()
    print(f"\nTotal Runtime: {int(end_global - start_global)} seconds")

if __name__ == "__main__":
    main()
```

Supplementary text 2. Python Script for Serial K-Means Clustering. This Python script (.py) implements the unsupervised serial K-means clustering pipeline described in the study. Utilizing the scikit-learn, pandas, and matplotlib libraries, the script processes the feature data generated by the ImageJ macro. The code performs the three-stage hierarchical clustering workflow: (1) separation of artifacts based on object area, (2) isolation of lymphocyte candidates based on mean luminance, and (3) final population refinement into six sub-clusters. The script outputs the final lymphocyte counts (.csv) and generates visualization overlays (.png) that map the identified clusters back onto the original cell masks for validation.