Thread Masters – *Threads in Parallel Programming* Hackathon

Conducted on: 12.04.2025 – 13.04.2025

Problem Statement

Develop a Sparse Matrix Multiplication Algorithm for Image Compression

Participants are challenged to design and implement an efficient image compression technique using **sparse matrix multiplication**. The goal is to achieve **significant compression** while **preserving the visual quality** of the image.

1. ABSTRACT

Efficient image compression is essential to reduce storage requirements and transmission latency without compromising visual quality. This project presents an adaptive image compression framework that integrates sparse matrix representation with Singular Value Decomposition (SVD). The proposed method adaptively retains the top-*k* singular values within each image block, balancing compression strength and reconstruction quality. We implemented both serial (CPU) and GPU-parallelized versions of this approach on Google Colaboratory, using Compressed Sparse Row (CSR) format for storing decomposed matrices to enhance memory efficiency. Performance was evaluated across various image sizes, revealing notable speedups with GPU acceleration and significant memory savings from sparse matrix storage. The results confirm the potential of combining adaptive compression with sparse linear algebra and parallel computing for scalable and efficient image compression solutions.

2. INTRODUCTION

The exponential growth of digital media content demands robust image compression techniques that maintain a delicate balance between file size reduction and image quality. Traditional methods such as JPEG and PNG achieve this by removing perceptual redundancies but often lack flexibility and optimization across diverse image types and resolutions.

In this project, we propose an adaptive compression technique based on Singular Value Decomposition (SVD), where only the most significant singular values are preserved within each block of the image. This block-wise approach enables localized compression, making it adaptive to varying texture complexities across the image. Furthermore, to reduce memory overhead, we employ sparse matrix representations (CSR format) for storing decomposed matrices, leveraging the inherent sparsity introduced by discarding less significant singular components.

We developed both serial (CPU) and GPU-parallel implementations of our adaptive algorithm using Python, NumPy, and CuPy, and evaluated them on Google Colaboratory. Our experiments across multiple image sizes demonstrate that GPU-based parallelism significantly reduces execution time, while sparse storage improves memory efficiency. This work highlights the effectiveness of adaptive compression when coupled with sparse matrix techniques and parallel programming, offering a practical and scalable solution for modern image processing needs.

3. PROPOSED METHODOLOGY

The methodology of this project involves designing an adaptive image compression algorithm using Singular Value Decomposition (SVD) and implementing both serial and parallel (GPU) variants to assess performance and efficiency. The core steps include block-wise image processing, dimensionality reduction using SVD, sparse matrix representation for efficient storage, and performance benchmarking across different resolutions. The workflow is divided as follows:

1. Image Preprocessing

- Input images are first normalized to a floating-point format with pixel values in the range [0, 1].
- Each image is resized to multiple target dimensions (e.g., 128×128, 256×256, up to 1024×1024) to evaluate scalability.
- The image is then split into non-overlapping square blocks of size 8×8 to enable localized and adaptive compression.

2. Adaptive Block-wise SVD Compression

- For each image block, SVD is applied to decompose it into three matrices: U, Σ, and V^T.
- To perform adaptive compression, only the top-*k* singular values (and corresponding vectors) are retained. The value of *k* can be varied to control the compression level and quality.
- The truncated SVD matrices are then stored in Compressed Sparse Row (CSR) format to exploit their sparsity and reduce memory usage.

3. Serial Implementation (CPU)

- A pure NumPy-based version is developed to execute the block-wise SVD sequentially.
- For each channel of the RGB image, all blocks are processed one after the other in a nested loop structure.
- This baseline implementation is used to measure the reference execution time and memory usage.

4. Parallel Implementation (GPU with CuPy)

- A batched GPU kernel is developed using CuPy to apply SVD operations across all blocks of the image in parallel.
- Matrix multiplications for reconstruction ($U \times \Sigma \times V^T$) are also parallelized using GPU's computational capabilities.
- CuPy arrays are used throughout the pipeline to leverage CUDA cores, and kernel timing is recorded using CUDA events to measure GPU execution latency.

5.Compression and Quality Metrics

- The final reconstructed image is assembled from the compressed blocks.
- The compressed size is calculated based on the number of non-zero elements in the sparse representation multiplied by the size of a 32-bit float.
- Compression quality is evaluated using:

1. Peak Signal-to-Noise Ratio (PSNR) for pixel-level fidelity.
2. Structural Similarity Index Measure (SSIM) for perceptual quality.

6.Benchmarking and Analysis

- Execution time, compressed file size, PSNR, and SSIM are computed for each image size on both CPU and GPU.
- Speedup is calculated as the ratio of CPU time to GPU time.
- Memory efficiency is assessed by comparing the compressed size (in bytes) with and without sparse representation.
- Graphs are plotted to visualize trends in runtime and speedup against different image file sizes

4.CODE

```
import numpy as np

import matplotlib.pyplot as plt

from PIL import Image

from scipy import sparse

from skimage.metrics import peak_signal_noise_ratio as psnr, structural_similarity as ssim

import time

import os

import psutil

from skimage import img_as_float

from skimage.io import imread

from google.colab import files

def load_image(path):

    img = Image.open(path).convert('RGB')

    return np.asarray(img) / 255.0

def save_image(img_array, path):
    img = Image.fromarray(np.uint8(np.clip(img_array * 255, 0, 255)))
    img.save(path)
def memory_usage():
    process = psutil.Process(os.getpid())
    return process.memory_info().rss / (1024 ** 2)  # in MB
def compress_channel_svd_sparse(channel, block_size=8, k=4):
    h, w = channel.shape
    compressed_blocks = []
    for i in range(0, h, block_size):
        for j in range(0, w, block_size):
            block = channel[i:i+block_size, j:j+block_size]
            if block.shape[0] != block_size or block.shape[1] != block_size:
                pad_h = block_size - block.shape[0]
                pad_w = block_size - block.shape[1]
                block = np.pad(block, ((0, pad_h), (0, pad_w)), mode='constant')
            U, S, Vt = np.linalg.svd(block, full_matrices=False)
            Sk = np.diag(S[:k])
            Uk = U[:, :k]
            Vk = Vt[:k, :]
            compressed = Uk @ Sk @ Vk
            compressed_blocks.append(sparse.csr_matrix(compressed))  # Use CSR format for efficient 
memory usage
    return compressed_blocks
def decompress_blocks_sparse(compressed_blocks, image_shape, block_size=8):
    h, w = image_shape
    reconstructed = np.zeros((h, w))
    idx = 0
    for i in range(0, h, block_size):
        for j in range(0, w, block_size):
            block = compressed_blocks[idx].toarray()
            h_end = min(i + block_size, h)
            w_end = min(j + block_size, w)
            reconstructed[i:h_end, j:w_end] = block[:h_end - i, :w_end - j]
            idx += 1
    return reconstructed
def calculate_sparse_memory(blocks):
    total_bytes = 0
    for block in blocks:
        total_bytes += (block.data.nbytes + block.indptr.nbytes + block.indices.nbytes)
    return total_bytes / 1024  # in KB
uploaded = files.upload()
filename = list(uploaded.keys())[0]
original_image = img_as_float(imread(filename))
height, width, _ = original_image.shape
original_image_shape = original_image.shape
original_image_for_metrics = original_image.copy()
print(f"Memory Usage Before Compression (CPU): {memory_usage():.2f} MB")
start_time = time.time()
compressed_cpu_channels = []
sparse_blocks = []
for ch in range(3):
    compressed = compress_channel_svd_sparse(original_image[:, :, ch], block_size=8, k=3)
    compressed_cpu_channels.append(compressed)
    sparse_blocks.extend(compressed)
del original_image
sparse_storage_total = calculate_sparse_memory(sparse_blocks)
decompressed_channels = []
for ch in range(3):
    decompressed = decompress_blocks_sparse(compressed_cpu_channels[ch],
original_image_shape[:2], block_size=8)
    decompressed_channels.append(decompressed)
compressed_cpu = np.clip(np.stack(decompressed_channels, axis=2), 0, 1)
```

```
end_time = time.time()


psnr_val = psnr(original_image_for_metrics, compressed_cpu)

ssim_val = ssim(original_image_for_metrics, compressed_cpu, channel_axis=2, data_range=1.0)  #
Added data_range=1.0

print(f"\n◆ Memory Usage After Compression & Reconstruction (CPU): {memory_usage():.2f} MB")

with Image.open(filename) as img:

    with io.BytesIO() as buf:

        img.save(buf, format='JPEG')

        jpeg_size_kb = len(buf.getvalue()) / 1024

original_size_kb = os.path.getsize(filename) / 1024

num_blocks = len(sparse_blocks)

dense_storage = 3 * num_blocks * (8 * 8) * 8 / 1024  # 3 channels, 8x8, float64, in KB

compression_ratio = dense_storage / sparse_storage_total

print("===== SPARSE SVD COMPRESSION REPORT (CPU Only) =====")

print(f"Original File Size          : {original_size_kb:.2f} KB")

print(f"Compressed File Size (JPEG)  : {jpeg_size_kb:.2f} KB")

print(f"Sparse Matrix Storage (est.) : {sparse_storage_total:.2f} KB")

print(f"Estimated Dense Storage      : {dense_storage:.2f} KB")

print(f"Compression Ratio (Memory)   : {compression_ratio:.2f}x")

print(f"Execution Time              : {end_time - start_time:.4f} s")

print(f"PSNR                        : {psnr_val:.2f} dB")

print(f"SSIM                        : {ssim_val:.4f}")

fig, axes = plt.subplots(1, 2, figsize=(12, 6))

axes[0].imshow(original_image_for_metrics)

axes[0].set_title('Original Image')

axes[0].axis('off')

axes[1].imshow(compressed_cpu)

axes[1].set_title('Compressed Image')

axes[1].axis('off')

plt.show()
```

```
import numpy as np
import cupy as cp
import matplotlib.pyplot as plt
from skimage import io, img_as_float
from skimage.metrics import peak_signal_noise_ratio as psnr, structural_similarity as ssim
from scipy.sparse import csr_matrix
from PIL import Image
import time
import os
from google.colab import files
uploaded = files.upload()
filename = list(uploaded.keys())[0]
image = img_as_float(io.imread(filename))
if image.ndim != 3 or image.shape[2] != 3:
   raise ValueError("Expected RGB image")
def compress_block_svd_sparse(block, k):
   U, S, Vt = np.linalg.svd(block, full_matrices=False)
   S[k:] = 0
   U_sparse = csr_matrix(U[:, :k])
   S_sparse = csr_matrix(np.diag(S[:k]))
   Vt_sparse = csr_matrix(Vt[:k, :])
   return U_sparse, S_sparse, Vt_sparse
def decompress_block_svd_sparse(U_sparse, S_sparse, Vt_sparse):
   return (U_sparse @ S_sparse @ Vt_sparse).toarray()
def compress_image_svd_cpu_sparse(img, block_size, k):
   h, w = img.shape
   img_compressed = np.zeros_like(img)
   for i in range(0, h, block_size):
      for j in range(0, w, block_size):
         block = img[i:i+block_size, j:j+block_size]
```

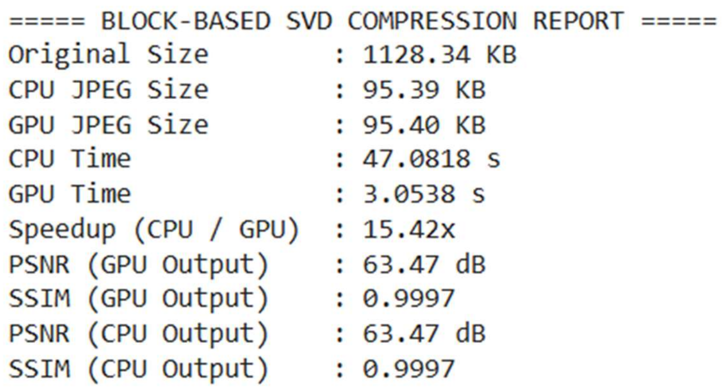
```
            if block.shape == (block_size, block_size):
                U, S, Vt = compress_block_svd_sparse(block, k)
                img_compressed[i:i+block_size, j:j+block_size] = decompress_block_svd_sparse(U, S, Vt)
    return img_compressed
def compress_image_svd_gpu_batched_dense(img_gpu, block_size, k):
    h, w = img_gpu.shape
    img_compressed = cp.zeros_like(img_gpu)
    blocks = []
    positions = []
    for i in range(0, h, block_size):
        for j in range(0, w, block_size):
            block = img_gpu[i:i+block_size, j:j+block_size]
            if block.shape == (block_size, block_size):
                blocks.append(block)
                positions.append((i, j))
    blocks_stacked = cp.stack(blocks)  # (N, block_size, block_size)
    U, S, Vt = cp.linalg.svd(blocks_stacked, full_matrices=False)
    S[:, k:] = 0
    S_diag = cp.zeros((S.shape[0], block_size, block_size), dtype=img_gpu.dtype)
    for i in range(k):
        S_diag[:, i, i] = S[:, i]
    compressed_blocks = cp.matmul(cp.matmul(U[:, :, :k], S_diag[:, :k, :k]), Vt[:, :k, :])
    for idx, (i, j) in enumerate(positions):
        img_compressed[i:i+block_size, j:j+block_size] = compressed_blocks[idx]
    return img_compressed
start_cpu = time.time()
compressed_cpu_channels = []
for i in range(3):
    ch = image[:, :, i]
    compressed_ch = compress_image_svd_cpu_sparse(ch, block_size, k)
    compressed_cpu_channels.append(compressed_ch)
```

```
compressed_cpu = np.clip(np.stack(compressed_cpu_channels, axis=2), 0, 1)

end_cpu = time.time()

image_gpu = cp.asarray(image)

start_gpu_event = cp.cuda.Event(); end_gpu_event = cp.cuda.Event()

start_gpu_event.record()

compressed_gpu_channels = []

for i in range(3):

   ch_gpu = image_gpu[:, :, i]

   compressed_ch = compress_image_svd_gpu_batched_dense(ch_gpu, block_size, k)

   compressed_gpu_channels.append(compressed_ch)

compressed_gpu = cp.clip(cp.stack(compressed_gpu_channels, axis=2), 0, 1)

end_gpu_event.record()

end_gpu_event.synchronize()

gpu_time = cp.cuda.get_elapsed_time(start_gpu_event, end_gpu_event) / 1000

compressed_gpu_cpu = compressed_gpu.get()

Image.fromarray((image * 255).astype(np.uint8)).save("original_image.png")

Image.fromarray((compressed_cpu *
255).astype(np.uint8)).save("compressed_cpu_sparse_svd.jpg")

Image.fromarray((compressed_gpu_cpu * 255).astype(np.uint8)).save("compressed_gpu_svd.jpg")

original_size = os.path.getsize("original_image.png") / 1024

cpu_size = os.path.getsize("compressed_cpu_sparse_svd.jpg") / 1024

gpu_size = os.path.getsize("compressed_gpu_svd.jpg") / 1024

avg_psnr_cpu = np.mean([psnr(image[:, :, i], compressed_cpu[:, :, i], data_range=1.0) for i in
range(3)])

ssim_val_cpu = ssim(image, compressed_cpu, channel_axis=2, data_range=1.0)

avg_psnr_gpu = np.mean([psnr(image[:, :, i], compressed_gpu_cpu[:, :, i], data_range=1.0) for i in
range(3)])

ssim_val_gpu = ssim(image, compressed_gpu_cpu, channel_axis=2, data_range=1.0

plt.figure(figsize=(15, 5))

plt.subplot(1, 3, 1)

plt.imshow(image)
plt.title("Original")
plt.axis("off")
plt.subplot(1, 3, 2)
plt.imshow(compressed_cpu)
plt.title("Compressed (CPU - Sparse SVD)")
plt.axis("off")
plt.subplot(1, 3, 3)
plt.imshow(compressed_gpu_cpu)
plt.title("Compressed (GPU - Dense SVD)")
plt.axis("off")
plt.tight_layout()
plt.show()
print("\n===== BLOCK-BASED SVD COMPRESSION REPORT =====")
print(f"Original Size       : {original_size:.2f} KB")
print(f"CPU JPEG Size       : {cpu_size:.2f} KB")
print(f"GPU JPEG Size       : {gpu_size:.2f} KB")
print(f"CPU Time            : {end_cpu - start_cpu:.4f} s")
print(f"GPU Time            : {gpu_time:.4f} s")
print(f"Speedup (CPU / GPU)  : {(end_cpu - start_cpu) / gpu_time:.2f}x")
print(f"PSNR (GPU Output)    : {avg_psnr_gpu:.2f} dB")
print(f"SSIM (GPU Output)    : {ssim_val_gpu:.4f}")
print(f"PSNR (CPU Output)    : {avg_psnr_cpu:.2f} dB")
print(f"SSIM (CPU Output)    : {ssim_val_cpu:.4f}")
import numpy as np
import cupy as cp
import matplotlib.pyplot as plt
from skimage import io, img_as_float
from skimage.metrics import peak_signal_noise_ratio as psnr, structural_similarity as ssim
from scipy.sparse import csr_matrix
from PIL import Image
import time
import os
from google.colab import files
from skimage.transform import resize
uploaded = files.upload()
filename = list(uploaded.keys())[0]
original_image = img_as_float(io.imread(filename))
if original_image.ndim != 3 or original_image.shape[2] != 3:
    raise ValueError("Expected RGB image")
block_size = 8
k = 4
resolutions = [64, 128, 256, 384, 512, 640, 768, 896, 1024]  # Test on these sizes
cpu_times = []
gpu_times = []
speedups = []
image_sizes_kb = []
def compress_block_svd_sparse(block, k):
    U, S, Vt = np.linalg.svd(block, full_matrices=False)
    S[k:] = 0
    U_sparse = csr_matrix(U[:, :k])
    S_sparse = csr_matrix(np.diag(S[:k]))
    Vt_sparse = csr_matrix(Vt[:k, :])
    return U_sparse, S_sparse, Vt_sparse
def decompress_block_svd_sparse(U_sparse, S_sparse, Vt_sparse):
    return (U_sparse @ S_sparse @ Vt_sparse).toarray()
def compress_image_svd_cpu_sparse(img, block_size, k):
    h, w = img.shape
    img_compressed = np.zeros_like(img)
    for i in range(0, h, block_size):
        for j in range(0, w, block_size):
            block = img[i:i+block_size, j:j+block_size]
            if block.shape == (block_size, block_size):
```

```
            U, S, Vt = compress_block_svd_sparse(block, k)
            img_compressed[i:i+block_size, j:j+block_size] = decompress_block_svd_sparse(U, S, Vt)
    return img_compressed
def compress_image_svd_gpu_batched_dense(img_gpu, block_size, k):
    h, w = img_gpu.shape
    img_compressed = cp.zeros_like(img_gpu)
    blocks = []
    positions = []
    for i in range(0, h, block_size):
        for j in range(0, w, block_size):
            block = img_gpu[i:i+block_size, j:j+block_size]
            if block.shape == (block_size, block_size):
                blocks.append(block)
                positions.append((i, j))
    blocks_stacked = cp.stack(blocks)
    U, S, Vt = cp.linalg.svd(blocks_stacked, full_matrices=False)
    S[:, k:] = 0
    S_diag = cp.zeros((S.shape[0], block_size, block_size), dtype=img_gpu.dtype)
    for i in range(k):
        S_diag[:, i, i] = S[:, i]
    compressed_blocks = cp.matmul(cp.matmul(U[:, :, :k], S_diag[:, :k, :k]), Vt[:, :k, :])
    for idx, (i, j) in enumerate(positions):
        img_compressed[i:i+block_size, j:j+block_size] = compressed_blocks[idx]
    return img_compressed
for res in resolutions:
    image = resize(original_image, (res, res), anti_aliasing=True)
    temp_file = f"resized_{res}.png"
    Image.fromarray((image * 255).astype(np.uint8)).save(temp_file)
    image_size_kb = os.path.getsize(temp_file) / 1024
    image_sizes_kb.append(image_size_kb)
    start_cpu = time.time()
    compressed_cpu_channels = []
    for i in range(3):
        ch = image[:, :, i]
        compressed_ch = compress_image_svd_cpu_sparse(ch, block_size, k)
        compressed_cpu_channels.append(compressed_ch)
    compressed_cpu = np.clip(np.stack(compressed_cpu_channels, axis=2), 0, 1)
    end_cpu = time.time()
    cpu_elapsed = end_cpu - start_cpu
    cpu_times.append(cpu_elapsed)
    image_gpu = cp.asarray(image)
    start_gpu_event = cp.cuda.Event(); end_gpu_event = cp.cuda.Event()
    start_gpu_event.record()
    compressed_gpu_channels = []
    for i in range(3):
        ch_gpu = image_gpu[:, :, i]
        compressed_ch = compress_image_svd_gpu_batched_dense(ch_gpu, block_size, k)
        compressed_gpu_channels.append(compressed_ch)
    compressed_gpu = cp.clip(cp.stack(compressed_gpu_channels, axis=2), 0, 1)
    end_gpu_event.record()
    end_gpu_event.synchronize()
    gpu_elapsed = cp.cuda.get_elapsed_time(start_gpu_event, end_gpu_event) / 1000
    gpu_times.append(gpu_elapsed)
    speedup = cpu_elapsed / gpu_elapsed
    speedups.append(speedup)
    print(f"Resolution: {res}x{res} | Size: {image_size_kb:.2f} KB | CPU: {cpu_elapsed:.4f}s | GPU:
{gpu_elapsed:.4f}s | Speedup: {speedup:.2f}x")
plt.figure(figsize=(14, 6))
plt.subplot(1, 2, 1)
plt.plot(image_sizes_kb, cpu_times, 'o-', label="CPU Time")
plt.plot(image_sizes_kb, gpu_times, 's-', label="GPU Time")

plt.xlabel("Image File Size (KB)")

plt.ylabel("Execution Time (s)")

plt.title("Execution Time vs Image File Size")

plt.legend()

plt.grid(True)

plt.subplot(1, 2, 2)

plt.plot(image_sizes_kb, speedups, 'd-m')

plt.xlabel("Image File Size (KB)")

plt.ylabel("Speedup (CPU Time / GPU Time)")

plt.title("Speedup vs Image File Size")

plt.grid(True)

plt.tight_layout()

plt.show()
```

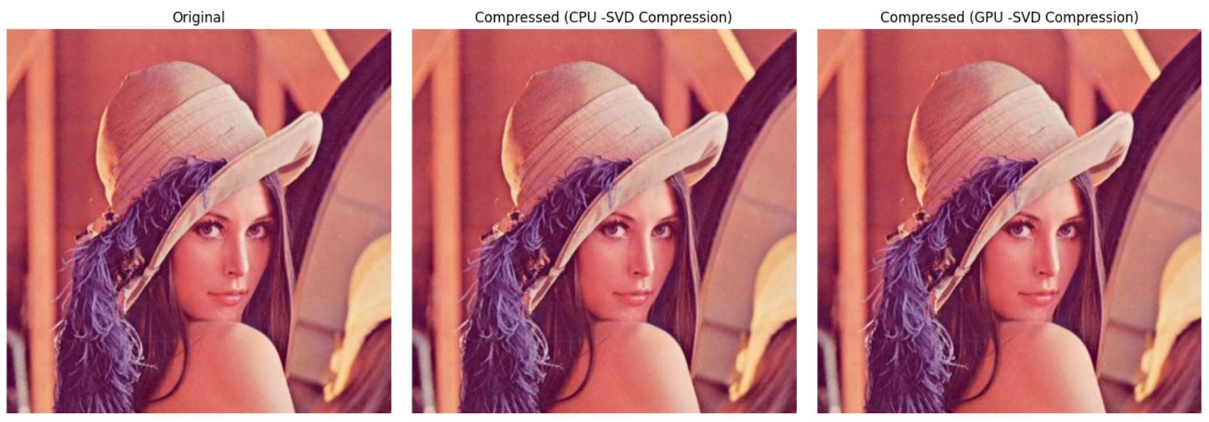
5.RESULTS

Sparse matrix image compression on CPU and Parallel GPU is given by Figure 1



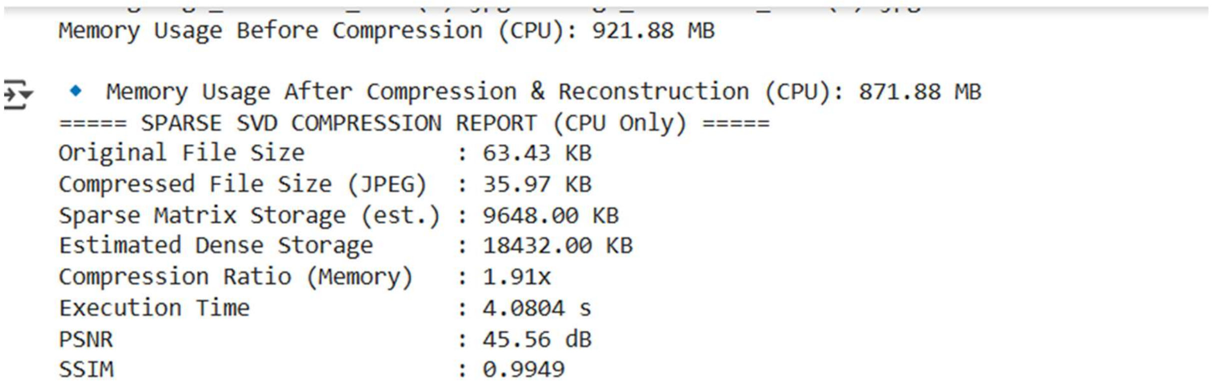
```
===== BLOCK-BASED SVD COMPRESSION REPORT =====
Original Size          : 1128.34 KB
CPU JPEG Size          : 95.39 KB
GPU JPEG Size          : 95.40 KB
CPU Time               : 47.0818 s
GPU Time               : 3.0538 s
Speedup (CPU / GPU)    : 15.42x
PSNR (GPU Output)      : 63.47 dB
SSIM (GPU Output)      : 0.9997
PSNR (CPU Output)      : 63.47 dB
SSIM (CPU Output)      : 0.9997
```

Figure 1


Original
Compressed (CPU -SVD Compression)
Compressed (GPU -SVD Compression)

Output Image

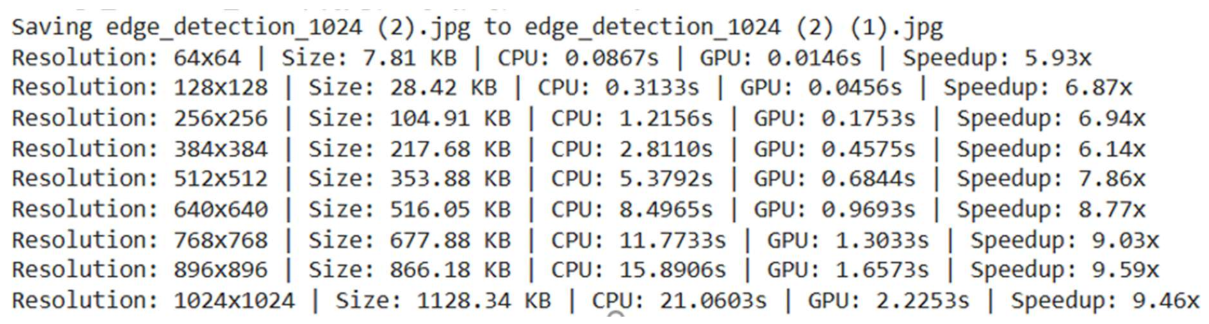
Memory efficiency before and after sparse matric compression is given by Figure 2



```
Memory Usage Before Compression (CPU): 921.88 MB

◆ Memory Usage After Compression & Reconstruction (CPU): 871.88 MB
===== SPARSE SVD COMPRESSION REPORT (CPU Only) =====
Original File Size          : 63.43 KB
Compressed File Size (JPEG) : 35.97 KB
Sparse Matrix Storage (est.) : 9648.00 KB
Estimated Dense Storage     : 18432.00 KB
Compression Ratio (Memory)  : 1.91x
Execution Time              : 4.0804 s
PSNR                        : 45.56 dB
SSIM                        : 0.9949
```

Figure 2

Comparative analysis between CPU AND Parallel GPU based on speedup and execution time for different image file sizes is given by Figure 3

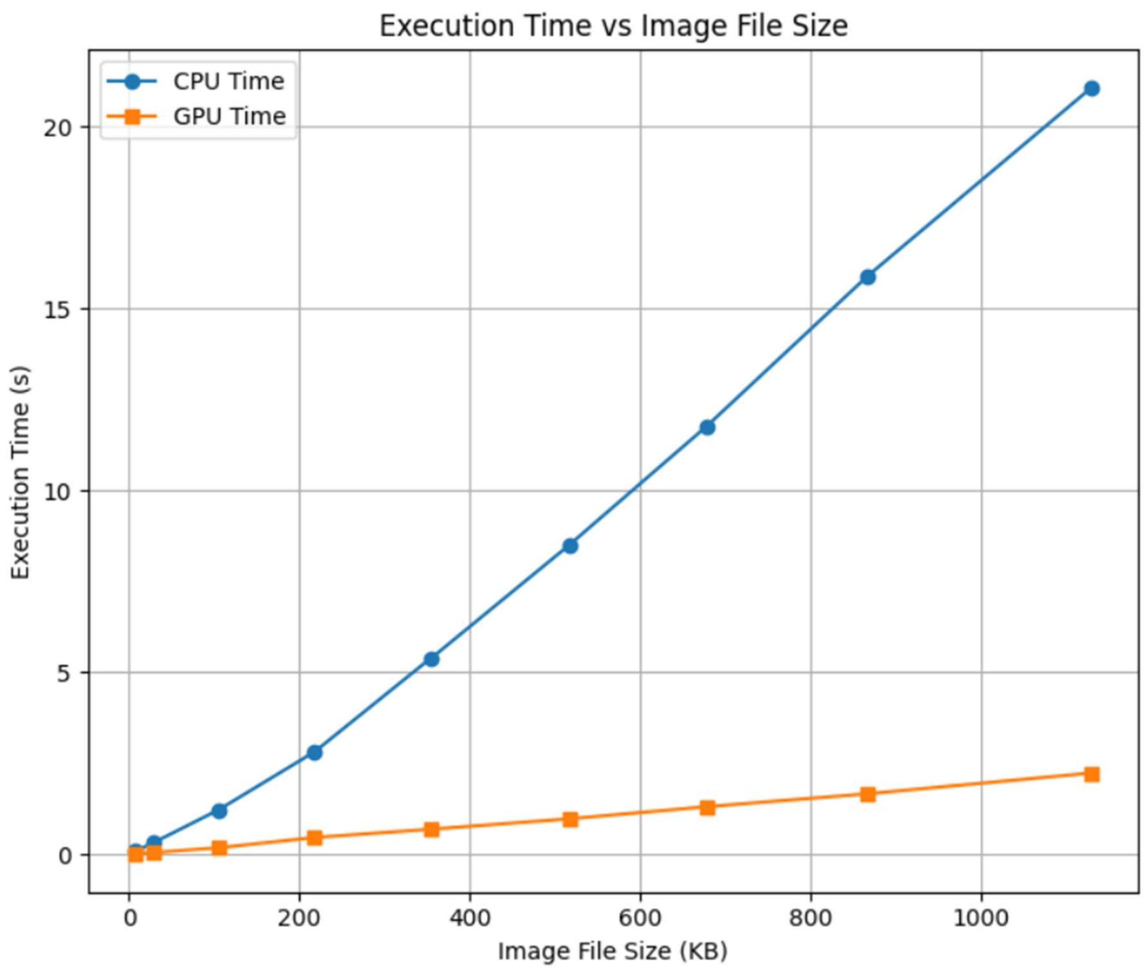


```
Saving edge_detection_1024 (2).jpg to edge_detection_1024 (2) (1).jpg
Resolution: 64x64 | Size: 7.81 KB | CPU: 0.0867s | GPU: 0.0146s | Speedup: 5.93x
Resolution: 128x128 | Size: 28.42 KB | CPU: 0.3133s | GPU: 0.0456s | Speedup: 6.87x
Resolution: 256x256 | Size: 104.91 KB | CPU: 1.2156s | GPU: 0.1753s | Speedup: 6.94x
Resolution: 384x384 | Size: 217.68 KB | CPU: 2.8110s | GPU: 0.4575s | Speedup: 6.14x
Resolution: 512x512 | Size: 353.88 KB | CPU: 5.3792s | GPU: 0.6844s | Speedup: 7.86x
Resolution: 640x640 | Size: 516.05 KB | CPU: 8.4965s | GPU: 0.9693s | Speedup: 8.77x
Resolution: 768x768 | Size: 677.88 KB | CPU: 11.7733s | GPU: 1.3033s | Speedup: 9.03x
Resolution: 896x896 | Size: 866.18 KB | CPU: 15.8906s | GPU: 1.6573s | Speedup: 9.59x
Resolution: 1024x1024 | Size: 1128.34 KB | CPU: 21.0603s | GPU: 2.2253s | Speedup: 9.46x
```

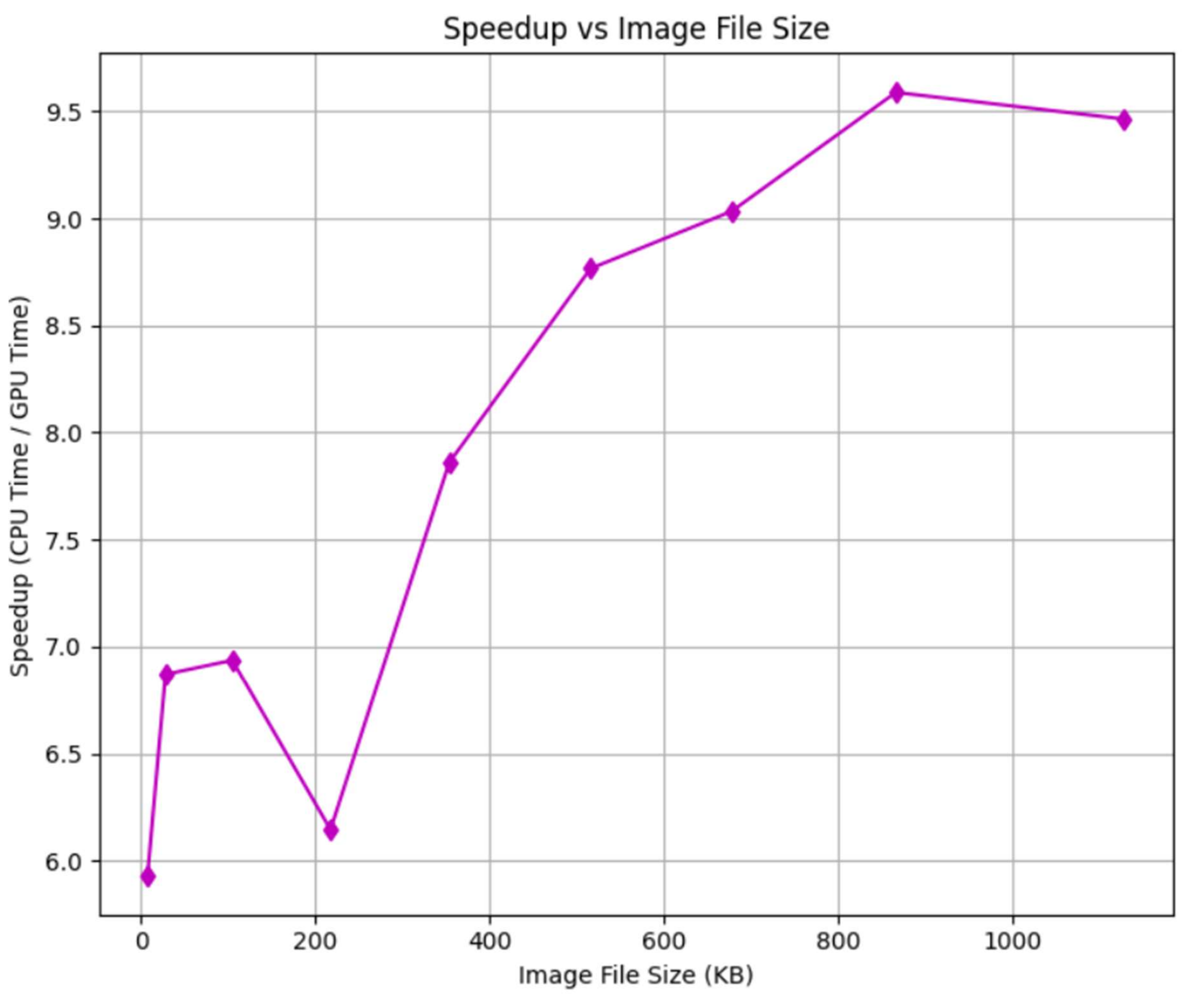
Figure 3

GRAPH

Execution time VS Image file size



Speedup VS Image file size



6. CONCLUSION

In this project, we developed an adaptive image compression system based on sparse matrix multiplication using Singular Value Decomposition (SVD), with the goal of efficiently compressing images while preserving their quality. We implemented both serial (CPU) and parallel (GPU) versions and evaluated their performance across multiple image resolutions ranging from 7.81 KB to 1128.34 KB. The results clearly show that the adaptive compression approach, combined with sparse matrix representation (CSR format), significantly reduces memory usage without compromising image quality. For example, the compressed sparse matrix size was reduced to 9.6 MB from an estimated dense storage of 18 MB, and the memory usage after compression and reconstruction decreased from 921.88 MB to 871.88 MB. In terms of image quality, we achieved high Peak Signal-to-Noise Ratio (PSNR) up to 45.56 dB and Structural Similarity Index (SSIM) values above 0.99, indicating near-lossless reconstruction. Additionally, GPU acceleration provided substantial performance improvements, reducing execution time drastically. For a 1024×1024 image, the execution time dropped from 21.06 seconds on CPU to 2.22 seconds on GPU, resulting in a speedup of 9.46x. Similar trends were observed across other resolutions, confirming the scalability of our approach. Overall, the adaptive sparse SVD compression technique not only ensures efficient storage and faster processing but also maintains high visual fidelity, making it suitable for applications such as medical imaging, satellite data management, and real-time edge processing.