Integrating Project Final Documentation

CEPARCO S11 G6
Juan Diego Javier, John Zechariah Kho, Shawne Michael Tumalad

Implementing CUDA to the LMS Daltonization Algorithm.

Introduction

In this project, we explored the implementation of the LMS Daltonization algorithm using PyCUDA, utilizing the parallel computing capabilities of NVIDIA GPUs. Our goal was to simulate color vision deficiencies and compensate for them in real-time. This method is especially useful for colorblind individuals, enabling them to distinguish colors more effectively. The first step of the LMS Daltonization Algorithm is to convert the image from an RGB color space(Red Green Blue) into LMS (Long Medium Short) color space by performing matrix multiplication. It then simulates color blindness of a human with a specific type of colorblindness (e.g., protanopia, deuteranopia, or tritanopia) by modifying the LMS values through a Daltonization matrix.The values were acquired from a research study by Doliotis, P., Tsekouras, G., Anagnostopoulos, C. N., & Athitsos, V. (2009). Then the values are compensated for better color visibility after which the values are converted back into RGB.

Python Algorithm

```
def daltonize(img, intensity, deficiency):

    # Image RGB
    RGB = np.asarray(img, dtype=float)

    # Deficiency matrices
    daltonization_matrices = {
        'd': np.array([ [1, 0, 0],
                        [0.4942, 0, 1.2483],
                        [0, 0, 1]]),
        'p': np.array([ [0, 2.0234, -2.5258],
                        [0, 1, 0],
```

```
                        [0, 0, 1]]),
        't': np.array([ [1,0,0],
                        [0,1,0],
                        [-0.395913,0.801109,0]])
    }

    daltonization_matrix = daltonization_matrices.get(deficiency, None)
    if daltonization_matrix is None:
        raise ValueError("Invalid deficiency type. Supported types are:
Deuteranomaly(d), Protanomaly(p), Tritanomaly(t)")

    # RGB to LMS
    LMS = np.tensordot(RGB[...,:3], RGB_to_LMS.T, axes=1)

    # LMS to simul LMS
    _LMS = np.tensordot(LMS[...,:3], daltonization_matrix.T, axes=1)

    # simul LMS to compensated LMS
    error = (LMS - _LMS) * intensity
    cLMS = np.tensordot(error[...,:3], shift.T, axes=1)

    # compensated LMS to compensated RGB
    _RGB = np.tensordot(cLMS[...,:3], LMS_to_RGB.T, axes=1)
    cRGB = _RGB + RGB

    # Clip the values to be within [0, 255]
    result = np.clip(cRGB, 0, 255).astype('uint8')

    return result
```

Code Snippet 1: Python Daltonize function

This snippet shows the full python function of the LMS Daltonization algorithm. The program uses the OpenCV library for images and videos and uses the NumPy library to handle matrix operations. This function takes an Image or Video as input via the ‘RGB’ variable and determines the type of colorblindness correction to perform from the ‘deficiency’ function parameter.

The following is the main process of the algorithm:

```
# RGB to LMS
    LMS = np.tensordot(RGB[...,:3], RGB_to_LMS.T, axes=1)

    # LMS to simul LMS
    _LMS = np.tensordot(LMS[...,:3], daltonization_matrix.T, axes=1)

    # simul LMS to compensated LMS
    error = (LMS - _LMS) * intensity
    cLMS = np.tensordot(error[...,:3], shift.T, axes=1)

    # compensated LMS to compensated RGB
    _RGB = np.tensordot(cLMS[...,:3], LMS_to_RGB.T, axes=1)
    cRGB = _RGB + RGB

    # Clip the values to be within [0, 255]
    result = np.clip(cRGB, 0, 255).astype('uint8')
```

Code Snippet 2: Python main algorithm process

The process primarily uses the numpy tensordot function which performs a dot product on each row of a given 3D array like the ‘RGB_to_LMS’ variable. The algorithm starts by converting the RGB image to its LMS color space. The Daltonization matrix is then applied to the LMS values to simulate the desired color blindness. The compensated LMS is then calculated by subtracting the simulated LMS to the LMS color space and performing dot product with the shift matrix. It is then converted back to RGB to be added to the RGB color space to apply the correction.

How CUDA is implemented

Since the LMS Daltonization Algorithm utilizes a lot of matrix multiplication, CUDA is implemented using the process to perform matrix multiplication. Doing this reduces the runtime, thus reducing latency and improving algorithm performance. The simultaneous nature of CUDA allows multiple pixels to be altered at the same time.

```
if (deficiency == 'd') {  // Deuteranomaly
        daltonization_matrices[0][0] = 1; daltonization_matrices[0][1] =
0; daltonization_matrices[0][2] = 0;
        daltonization_matrices[1][0] = 0.4942;
daltonization_matrices[1][1] = 0; daltonization_matrices[1][2] = 1.2483;
        daltonization_matrices[2][0] = 0; daltonization_matrices[2][1] =
0; daltonization_matrices[2][2] = 1;
    } else if (deficiency == 'p') {  // Protanomaly
        daltonization_matrices[0][0] = 0; daltonization_matrices[0][1] =
2.0234; daltonization_matrices[0][2] = -2.5258;
        daltonization_matrices[1][0] = 0; daltonization_matrices[1][1] =
1; daltonization_matrices[1][2] = 0;
        daltonization_matrices[2][0] = 0; daltonization_matrices[2][1] =
0; daltonization_matrices[2][2] = 1;
    } else if (deficiency == 't') {  // Tritanomaly
        daltonization_matrices[0][0] = 1; daltonization_matrices[0][1] =
0; daltonization_matrices[0][2] = 0;
        daltonization_matrices[1][0] = 0; daltonization_matrices[1][1] =
1; daltonization_matrices[1][2] = 0;
        daltonization_matrices[2][0] = -0.395913;
daltonization_matrices[2][1] = 0.801109; daltonization_matrices[2][2] =
0;
    } else {
        printf("Invalid deficiency type. Supported types are:
Deuteranomaly(d), Protanomaly(p), Tritanomaly(t)");
        return;
    }
```

Code Snippet 3: PyCUDA Daltonization Matrices

The LMS Daltonization matrices for Deuteranomaly, Protanomaly, and Tritanomaly are stored in device memory. Depending on the user's selection, a different matrix is used.

```
// RGB to LMS
    float R = img[idx];
    float G = img[idx + 1];
    float B = img[idx + 2];

    float L = RGB_to_LMS[0] * R + RGB_to_LMS[1] * G + RGB_to_LMS[2] * B;
    float M = RGB_to_LMS[3] * R + RGB_to_LMS[4] * G + RGB_to_LMS[5] * B;
    float S = RGB_to_LMS[6] * R + RGB_to_LMS[7] * G + RGB_to_LMS[8] * B;
```

Code Snippet 4: PyCUDA RGB to LMS Image Conversion

The RGB to LMS image conversion is handled by matrix multiplication. This conversion is performed on each pixel in the image to transform the entire image from RGB to LMS color space.

```
// LMS to simul LMS
    float _L = daltonization_matrices[0][0] * L +
daltonization_matrices[0][1] * M + daltonization_matrices[0][2] * S;
    float _M = daltonization_matrices[1][0] * L +
daltonization_matrices[1][1] * M + daltonization_matrices[1][2] * S;
    float _S = daltonization_matrices[2][0] * L +
daltonization_matrices[2][1] * M + daltonization_matrices[2][2] * S;

// Error calculation
    float eL = (L - _L) * intensity;
    float eM = (M - _M) * intensity;
    float eS = (S - _S) * intensity;
```

Code Snippet 5: PyCUDA LMS to Simulated LMS Image Conversion

This transforms the original LMS values to simulate a color vision deficiency and calculates the error for correction.

```
 // Compensated LMS
    float cL = eL * shift[0] + eM * shift[1] + eS * shift[2];
    float cM = eL * shift[3] + eM * shift[4] + eS * shift[5];
    float cS = eL * shift[6] + eM * shift[7] + eS * shift[8];
```

Code Snippet 6: PyCUDA Simulated LMS to Corrected LMS Image Conversion

Using the values from the shift matrix and error calculations, the image is corrected.

```
// Compensated LMS to compensated RGB
    float _R = LMS_to_RGB[0] * cL + LMS_to_RGB[1] * cM + LMS_to_RGB[2] *
cS;
    float _G = LMS_to_RGB[3] * cL + LMS_to_RGB[4] * cM + LMS_to_RGB[5] *
cS;
    float _B = LMS_to_RGB[6] * cL + LMS_to_RGB[7] * cM + LMS_to_RGB[8] *
cS;

    // Final RGB
    result[idx] = min(max(R + _R, 0.0), 255.0);
    result[idx + 1] = min(max(G + _G, 0.0), 255.0);
    result[idx + 2] = min(max(B + _B, 0.0), 255.0);
```

Code Snippet 7: PyCUDA Corrected LMS to Corrected RGB Image Conversion

The final compensated LMS image is then converted to RGB then checking is done to ensure that the RGB values remain within the valid range [0, 255].

The kernel is executed via the following python function:

```
def daltonize_gpu(img, intensity, deficiency):
    height, width, channels = img.shape
    img = img.astype(np.float32).flatten()
    result = np.zeros_like(img)

    img_gpu = cuda.mem_alloc(img.nbytes)
    result_gpu = cuda.mem_alloc(result.nbytes)

    cuda.memcpy_htod(img_gpu, img)

    block = (16, 16, 1)
    grid = (int(np.ceil(width / block[0])), int(np.ceil(height /
block[1])), 1)

    daltonize_kernel(img_gpu, np.float32(intensity),
np.int8(ord(deficiency)), result_gpu, np.int32(width), np.int32(height),
block=block, grid=grid)
    cuda.Context.synchronize()

    cuda.memcpy_dtoh(result, result_gpu)
```

```
    result = result.reshape((height, width, channels)).astype(np.uint8)

    img_gpu.free()
    result_gpu.free()

    return result
```

Code Snippet 8: Kernel execution function

The function prepares the variables needed before the kernel is executed. It prepares the image/video input by flattening the array. It then allocates memory on the GPU for the input and output and memory copies the function input to the newly allocated input variable in the GPU. The blocks and grid size are then defined, with this snippet initializing 256 threads (16x16x1) and the grid size adjusting to the amount of threads to properly fit the input. The kernel is then executed and is waited on via the synchronize() function to ensure every element is altered before proceeding to move the output back to the CPU. Once the output is back to the host, it is converted back to the input format and is returned.

Python vs. PyCUDA performance

The computer used on collecting the execution times has the following hardware specifications:

- CPU: Intel i7-8750H
- GPU: Nvidia GTX 1050 ti
- RAM: 16GB
- OS: Windows 10

Trying the program on newer hardware may produce better non-cuda program results.
The error correction of the Pycuda results is done by comparing the filtered image of the sequential program to the filtered image of the Pycuda version via vision.

Memory management functions

In our lessons on using CUDA, we learned how to use Unified Memory variables and how to maximize their efficiency with prefetching and memory advising. Pycuda also has Unified memory via the *cuda.managed_empty()* function although it lacks prefetching and memory advising. Since the test machine's OS is windows, prefetching is automatic and not explicitly called, but memory advising is still missing. Because of this, the Unified memory feature in Pycuda, while already significantly faster than sequential, cannot be fully maximized in efficiency.

The common practice of memory management in Pycuda is explicitly allocating memory on the GPU via *cuda.mem_alloc()* function and explicitly moving memory between the CPU and GPU via *cuda.memcpy_htod()* and *cuda.memcpy_dtoh()* functions. This is shown in the tutorial page of Pycuda's documentation website.

To show the difference in execution times, a simple array squaring function is used as shown:

```
# Array Squaring
def square(n, out, inp):
    for i in range(n):
        out[i] = inp[i] * inp[i]
```

Code Snippet 9: Array Squaring Function

The program uses the Numpy and Timeit libraries for arrays and timing respectively. The following constants are also used to maintain uniformity with the upcoming Pycuda variants:

```
# Constants
d_type = np.float32
ARRAY_SIZE = 1 << 20
ARRAY_BYTES = np.zeros(ARRAY_SIZE).astype(d_type).nbytes
loope = 10
```

Code Snippet 10: Program constants

The input and output variables are then initialized and the function is ran *loope* amount of times:

```
# Init arrays
inp = np.zeros(ARRAY_SIZE).astype(d_type)
out = np.empty_like(inp)
for i in range(ARRAY_SIZE):
    inp[i] = float(i)

# Function
print("Square Function")
print(f"numElements: {ARRAY_SIZE}")
exec_time = 0
for i in range(0,loope):
    start = timer()
    square(ARRAY_SIZE, out, inp)
    end = timer()
    exec_time += end - start
exec_time = exec_time / 10
print(f"Avg Execution Time: {exec_time}")
```

Code Snippet 11: Program execution

In the example, the function is run 10 times and the output would be the average execution time of those 10 runs.

PyCUDA versions

The following is the kernel module used for all Pycuda program versions:

```
module = SourceModule("""
__global__
void square(size_t n, float *out, float *in){
    int index = blockIdx.x * blockDim.x + threadIdx.x;
    int stride = blockDim.x * gridDim.x;
    for (int i = index; i < n; i += stride)
      out[i] = in[i] * in[i];
}
""")
square = module.get_function("square")
```

Code Snippet 12: Kernel module

The following snippets show how each program version allocates memory and initialize arrays:

Unified Memory:

```
# Declare arrays
inp = cuda.managed_empty(ARRAY_SIZE, dtype=d_type,
mem_flags=cuda.mem_attach_flags.GLOBAL)
out = cuda.managed_empty(ARRAY_SIZE, dtype=d_type,
mem_flags=cuda.mem_attach_flags.GLOBAL)
# Host only input array for err check
in1 = np.zeros(ARRAY_SIZE).astype(d_type)

# Init input array
for i in range(ARRAY_SIZE):
    inp[i] = float(i)
    in1[i] = float(i)
```

Code Snippet 13.1: Unified memory allocation and initialization

Prefetching is automatic but Memory advising is unable to be done. The *in1* variable exists to prevent unnecessary counts in Unified Memory Profiling as each use of a managed memory array in the program counts towards Device to Host transfers.

Mixed:

```
# Declare Unified Memory Array for input
inp = cuda.managed_empty(ARRAY_SIZE, dtype=d_type,
mem_flags=cuda.mem_attach_flags.GLOBAL)
# Declare GPU array for output
out_gpu = cuda.mem_alloc(ARRAY_BYTES)
# Host only input array for err check
in1 = np.zeros(ARRAY_SIZE).astype(d_type)
# Host output array
out = np.empty(ARRAY_SIZE).astype(d_type)

# Init input array
for i in range(ARRAY_SIZE):
    inp[i] = float(i)
    in1[i] = float(i)
```

Code Snippet 13.2: Mixed allocation and initialization

Same thing as for the Unified memory profiling, the *in1* variable exists to prevent unnecessary counts in Unified Memory Profiling.

Explicit call:

```
# Declare GPU arrays
inp_gpu = cuda.mem_alloc(ARRAY_BYTES)
out_gpu = cuda.mem_alloc(ARRAY_BYTES)
# Declare host arrays
inp = np.empty(ARRAY_SIZE).astype(d_type)
out = np.empty_like(inp)

# Init host input array
inp = np.empty(ARRAY_SIZE).astype(d_type)
for i in range(ARRAY_SIZE):
    inp[i] = float(i)

# Host to Device
cuda.memcpy_htod(inp_gpu, inp)
```

Code Snippet 13.3: Explicit call allocation and initialization

The following would be how the Pycuda programs execute the function

```
# Kernel
numThreads = 1024
numBlocks = (ARRAY_SIZE + numThreads - 1) // numThreads

print("Square Function")
print(f"numElements: {ARRAY_SIZE}")
print(f"numBlocks: {numBlocks}, numThreads: {numThreads}")
for i in range(0,loope):
    square(np.uintp(ARRAY_SIZE), out, inp, block=(numThreads, 1, 1),
grid=(numBlocks, 1))
cuda.Context.synchronize()
```

Code Snippet 14: Kernel execution

The thread count in this example is 1024. The *out* and *inp* parameters would depend on the program version used:

- Unified Memory version: out, inp
- Mixed version: out_gpu, inp
- Explicit call version: out_gpu, inp_gpu

Once the function is finished, the output data is explicitly copied from the GPU to the CPU in the Mixed and Explicit call versions via *.memcpy_dtoh()*

Results

The Pycuda results were obtained by using *nvprof*

Avg time (ms)				
Program	Python	Unified Mem	Mixed	Explicit
	255.5547	0.0966	0.0964	0.0970
Overheads				
Umem HtoD		1.5515	1.5894	
Umem DtoH		42.5002	20.8912	
memcpy HtoD				3.1038
memcpy DtoH			0.8357	0.9159
Total	255.5547	44.1483	23.4127	4.1167

Table 1: Pycuda memory management results

Explicit Call Speedup		
Python	Unified Mem	Mixed
62.0776	10.7242	5.6872

Table 1.1: Explicit call version speedup

The results show that explicit memory handling performed the best among all versions being around 62.1 times faster than the sequential version, around 10.72 times faster than the unified memory version, and around 5.69 times faster than the Mixed version. Despite unified memory having the fastest Host to Device times, its Device to Host transfer time is significantly slower than the explicit memory copy. These results prove that using explicit memory handling is the most efficient method for the test machine.

Video Results

The following code snippet makes up the '*main*' of the python program. It includes the GUI code which has been slightly modified to include an FPS counter for the video output:

```
# Default parameters
correction_level = 100
correction_types = ["d", "p", "t"]
default_correction_type = "d"

# Input video
vid = cv.VideoCapture(0)

# GUI
window_name = 'Daltonization for Color Deficiency Correction'
cv.namedWindow(window_name, cv.WINDOW_AUTOSIZE)
cv.createTrackbar("Correction Level (%)", window_name, correction_level,
500, lambda x: x)
cv.createTrackbar("Correction Type", window_name, 0,
len(correction_types) - 1, lambda x: x)

# FPS variables
fps = 0
frame_count = 0
start_time = timer()
```

```
while True:
    ret, frame = vid.read()

    level = cv.getTrackbarPos("Correction Level (%)", window_name) /
100.0
    correction_type_index = cv.getTrackbarPos("Correction Type",
window_name)
    default_correction_type = correction_types[correction_type_index]
    corrected = daltonize(frame, level, default_correction_type)

    # Calculate FPS
    frame_count += 1
    elapsed_time = timer() - start_time
    if elapsed_time > 1:
        fps = frame_count / elapsed_time
        frame_count = 0
        start_time = timer()
    # Display the FPS on the frame
    cv.putText(corrected, f'FPS: {fps:.2f}', (10, 30),
               cv.FONT_HERSHEY_SIMPLEX, 1, (255, 0, 0), 2, cv.LINE_AA)
    cv.putText(frame, "Original", (10,30),
               cv.FONT_HERSHEY_SIMPLEX, 0.4, (255, 0, 0), 2)
    cv.imshow(window_name, np.hstack([frame, corrected]))

    if cv.waitKey(1) & 0xFF == ord('q'):
        break
vid.release()
cv.destroyAllWindows()
```

Code Snippet 15: 'main' of python program

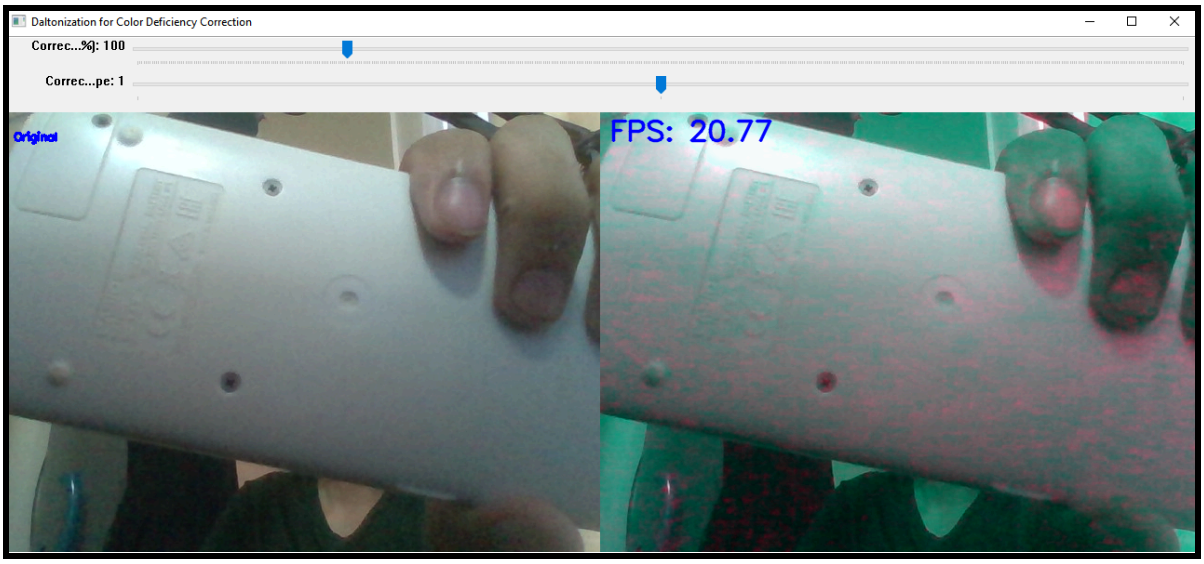

Figure 1: Non-PyCuda program FPS

Using the standard version of the program, the test machine video output is getting around 20 FPS. The FPS is not stable however, sometimes having a frame drop randomly. The FPS also shortly drops to around 15 FPS or less when the program window is significantly moved, though it goes back to its normal frame rate

.Figure 2: PyCuda program FPS

Using the PyCuda version of the program, the test machine has a stable framerate of around 30 FPS. The FPS still drops when the window is moved although it goes back up quickly and normally does not drop to below 20 FPS. The frame rate seems to be capped at the mentioned 30 FPS, possibly due to hardware limitations. Therefore it is possible that the actual performance of this program version is better than it seems.

Image Results

A modified version of the program has been used to calculate the execution times using images. The algorithm and its related functions were not changed but the GUI portion of the code was removed and replaced with the following:

```
# Get image data
path = "Images/img8k.jpg" # Should work, if it doesn't, use full path to
img
frame = cv.imread(path)

for i in range(0,10):
    print(f"[{i}]Run")
    daltonize_gpu(frame, 1, 'd')
```

Code Snippet 16: GUI removed

The code shown runs the Daltonization function 10 times with the same image. The images used in collecting the execution times vary in size, ranging from 1080p up to 8K. When running the PyCuda version of the program, it is executed with the ‘nvprof’ command to also profile the GPU and API call execution times. The average execution time is the sum of the average execution times of every GPU activity. ‘nvprof’ only works on Nvidia GPUs with a Compute version below 8.0. The 1050 ti from the test computer is part of the Pascal architecture which has a Compute version of 6.0 which is why it can be used. The raw results can be found in the “Timing results” folder of this repository.

Avg times (ms)	Resolution				
	1k	2k	4k	6k	8k
Python timer	240.5099	472.1305	965.0093	2791.4165	4335.1546
Nvprof GPU total	9.9862	18.6541	44.9642	102.1950	182.5020
nvprof HtoD	3.9952	7.4562	17.5190	40.0880	70.0160
nvprof DtoH	5.0922	9.6020	23.8790	54.1100	99.1610
nvprof func	0.8988	1.5959	3.5662	7.997	13.3250

Table 2: Average execution time (ms) from 10 runs

Figure 3: Execution time line graph

The results shown were performed with 256 threads (16x16x1). As shown, using CUDA significantly decreases the execution time of the Daltonization function. The Nvprof GPU total is the sum of the last 3 values in the table which are the indicated GPU activities in nvprof.

Pycuda speedup					
1k	2k	4k	6k	8k	Avg
24.0842	25.3097	21.4617	27.6963	23.6914	24.4487

Table 3: Pycuda speedup values

As shown in Table 3, the PyCuda version of the program is on average 24.45 times faster than the standard Python-only version which is very significant.

Threads	Resolution				
	1k	2k	4k	6k	8k
256	0.8988	1.5959	3.5662	9.0768	14.1240
512	0.8854	1.5886	3.5218	9.0575	13.8710
1024	0.9146	1.8172	3.9665	8.4423	13.7420

Table 4: GPU Daltonization function average execution time (ms)

This table shows the average execution time of the Daltonization function in the GPU. Comparing the results, the difference is at most only a few hundred microseconds. As shown in the table, using 512 threads shows the best performance up until 4K where using 1024 threads which performed the worst up until this point, now became the fastest with 256 threads now being the slowest. This result indicates that using more threads is the more effective the higher the number of elements are.

Conclusion

Our project demonstrates the substantial benefits of using CUDA for image processing tasks. The PyCUDA implementation of the LMS DAltonization algorithm not only achieves real-time performance but also maintains the accuracy and effectiveness of the sequential Python version. This highlights the potential of GPU computing in enhancing accessibility tools for individuals with color vision deficiencies, showing that there is potential for more responsive and scalable solutions in the field of assistive technology.

A possible future implementation for improving the Daltonization algorithm is to modify its adaptability. The algorithm could adjust based on the image/video contents such as changing the color scheme of specific parts of the image or video. By adding a color detection algorithm to detect specific colors or dominant colors present in the image/video, then color changes would be applied based on the detection results. Techniques such as color histograms, k-means clustering, or other image segmentation methods are possible tools to help implement the improved algorithm.

Expanding the project's platform support is another way of learning more ways to improve and optimize the algorithm. By extending compatibility to macOS and especially turning it into a mobile application, we would be able to gather more feedback on the study. Users will be able to provide feedback, allowing us to learn more about flaws or areas that need improvement.

References

Doliotis, P., Tsekouras, G., Anagnostopoulos, C. N., & Athitsos, V. (2009). Intelligent modification of colors in digitized paintings for enhancing the visual perception of color-blind viewers. In Artificial Intelligence Applications and Innovations III 5 (pp. 293-301). Springer US. Retrieved from https://www.researchgate.net/publication/220828504_Intelligent_Modification_of_Colors_in_Digitized_Paintings_for_Enhancing_the_Visual_Perception_of_Color-blind_Viewers

pycuda 2024.1 documentation. (n.d.). Index of /. Retrieved from

https://documen.tician.de/pycuda/index.html