Group 5

DSCI 560 – Data Science Professional Practicum

Lab-3: CUDA Program and Custom Python Library

Team Details

Team Name: ILR

Member 1 – Lance Vijil Dsilva USC ID: 3824765644
Member 2 - Rafayel Mirijanyan– USC ID: 3487192016
Member 3 - Isabella Yoo – USC ID: 1305966908

GitHub Repository

https://github.com/bellayoo/DSCI_560_Lab/tree/main/Lab_03

Objective

The objective of this lab was to:

1. Implement matrix multiplication on the CPU.
2. Port the implementation to CUDA (naïve and optimized versions).
3. Compare performance with NVIDIA's cuBLAS library.
4. Build a CUDA shared library callable from Python.
5. Implement image convolution on CPU and GPU and compare correctness and performance

Method / Implementation:

A) Matrix Multiplication

1) CPU matrix multiplication (`matrix_cpu.c`)

- Implemented a baseline CPU GEMM that allocates host matrices `A, B, C`, fills `A/B` with values, then computes `C` using nested loops.
- Timed using a host timer (same workflow used later for GPU baselines).
  This file is listed as the CPU implementation in the project overview

2) Naïve CUDA matrix multiplication (`matrix_gpu.c`)

- Implemented a CUDA kernel where **one thread computes one output element `C[row, col]`**.

- In host code: allocated `d_A, d_B, d_C` with `cudaMalloc`, copied host→device, launched kernel with a 2D grid, synchronized, copied device→host, freed device memory.
- This file is listed as the naïve CUDA implementation in the project overview.

3) Optimized CUDA (tiled shared memory) (`matrix_gpu_optimized.c`)

- Implemented a tiled GEMM using `__shared__` tiles `ds_A` and `ds_B`, and iterated over tiles with `m`:
    - Each block loads one tile of A and one tile of B into shared memory (with boundary checks).
    - `__syncthreads()` is called after loads to ensure all threads can safely read shared data.
      Each thread accumulates partial sums over k across the tile.
      Another `__syncthreads()` ensures shared memory isn't overwritten before all threads finish using it.
- Kernel structure and indexing match the lab's recommended tiled template.

4) cuBLAS matrix multiplication (`matrix_gpu_cublas.c`)

- Implemented GEMM using NVIDIA's cuBLAS API (`cublasSgemm`) instead of a custom kernel.
- This file is listed as the cuBLAS implementation in the project overview.
  report
- The assignment explicitly requires using `cublasSgemm(...)` for matrix multiplication.

```
ubuntu@147-224-39-216:~$ gcc matrix_cpu.c -o matrix_cpu -O2
ubuntu@147-224-39-216:~$ ./matrix_cpu 512
CPU execution time (N=512): 0.151671 seconds
ubuntu@147-224-39-216:~$
```

```
ubuntu@147-224-39-216:~$ nvcc -x cu matrix_gpu.c -o matrix_gpu
nvcc warning : Support for offline compilation for architectures prior to '<compute/sm/lto>_75' will be removed in a future release (Use -Wno-deprecated-gpu-targ
ets to suppress warning).
ubuntu@147-224-39-216:~$ ./matrix_gpu 512
GPU execution time (N=512): 0.000278 seconds
ubuntu@147-224-39-216:~$
```

```
ubuntu@147-224-39-216:~$ nvcc -x cu matrix_gpu_optimized.c -o matrix_gpu_optimized
nvcc warning : Support for offline compilation for architectures prior to '<compute/sm/lto>_75' will be removed in a future release (Use -Wno-deprecated-gpu-targ
ets to suppress warning).
ubuntu@147-224-39-216:~$ ./matrix_gpu_optimized 512
GPU execution time (N=512): 0.000246 seconds
ubuntu@147-224-39-216:~$
```

```
ubuntu@147-224-39-216:~$ nvcc -x cu matrix_gpu_cublas.c -o matrix_gpu_cublas -lcublas
nvcc warning : Support for offline compilation for architectures prior to '<compute/sm/lto>_75' will be removed in a future release (Use -Wno-deprecated-gpu-targ
ets to suppress warning).
ubuntu@147-224-39-216:~$ ./matrix_gpu_cublas 512
cuBLAS execution time (N=512): 0.000040 seconds
ubuntu@147-224-39-216:~$
```

5) Shared CUDA library callable from Python (`matrix_lib.cu to libmatrix.so`)

- Exposed GPU functions through a C-callable interface (so Python can call them via `ctypes`), including:

    - `matrix_multiply(float *h_A, float *h_B, float *h_C, int N)`
    - `convolute_image(unsigned char *h_input, unsigned char *h_output, int width, int height, int N, float *h_kernel)`
- Built the .so using nvcc

```cpp
extern "C" void matrix_multiply(float *h_A, float *h_B, float *h_C, int N) {
    size_t size = N * N * sizeof(float);

    float *d_A, *d_B, *d_C;
    cudaMalloc((void**)&d_A, size);
    cudaMalloc((void**)&d_B, size);
    cudaMalloc((void**)&d_C, size);

    cudaMemcpy(d_A, h_A, size, cudaMemcpyHostToDevice);
    cudaMemcpy(d_B, h_B, size, cudaMemcpyHostToDevice);

    dim3 threadsPerBlock(TILE_WIDTH, TILE_WIDTH);
    dim3 blocksPerGrid((N + TILE_WIDTH - 1) / TILE_WIDTH, (N + TILE_WIDTH - 1) / TILE_WIDTH);

    matrixMultiplyGPU<<<blocksPerGrid, threadsPerBlock>>>(d_A, d_B, d_C, N);

    cudaDeviceSynchronize();

    cudaMemcpy(h_C, d_C, size, cudaMemcpyDeviceToHost);

    cudaFree(d_A);
    cudaFree(d_B);
    cudaFree(d_C);
}

extern "C" void convolute_image(unsigned char *h_input, unsigned char *h_output, int width, int height, int N, float *h_kernel)
    size_t img_size = (size_t)width * height;
    size_t kernel_size = N * N * sizeof(float);

    unsigned char *d_input, *d_output;
    float *d_kernel;
```

```
ubuntu@147-224-39-216:~$ python3 matrix_test.py --task matrix
Testing Matrix Multiplication:
  N=512
    Time: 0.2116 seconds
  N=1024
    Time: 0.0026 seconds
ubuntu@147-224-39-216:~$
```

6) Python drivers(caller.py, matrix_test.py)

- Python scripts were used to:
    - run multiple N sizes and print measured runtimes
    - call the shared library using ctypes and contiguous NumPy buffers

```
ubuntu@147-224-39-216:~$ python3 caller.py
nvcc warning : Support for offline compilation for architectures prior to '<compute/sm/lto>_75' will be removed in a future release (Use -Wno-deprecated-gpu-targ
ets to suppress warning).
nvcc warning : Support for offline compilation for architectures prior to '<compute/sm/lto>_75' will be removed in a future release (Use -Wno-deprecated-gpu-targ
ets to suppress warning).
nvcc warning : Support for offline compilation for architectures prior to '<compute/sm/lto>_75' will be removed in a future release (Use -Wno-deprecated-gpu-targ
ets to suppress warning).
Running matrix_cpu with N=256
CPU execution time (N=256): 0.017927 seconds
Running matrix_gpu with N=256
GPU execution time (N=256): 0.000142 seconds
Running matrix_gpu_optimized with N=256
GPU execution time (N=256): 0.000134 seconds
Running matrix_gpu_cublas with N=256
cuBLAS execution time (N=256): 0.000024 seconds
Running matrix_cpu with N=512
CPU execution time (N=512): 0.151396 seconds
Running matrix_gpu with N=512
GPU execution time (N=512): 0.000273 seconds
Running matrix_gpu_optimized with N=512
GPU execution time (N=512): 0.000243 seconds
Running matrix_gpu_cublas with N=512
cuBLAS execution time (N=512): 0.000041 seconds
Running matrix_cpu with N=1024
CPU execution time (N=1024): 1.247492 seconds
Running matrix_gpu with N=1024
GPU execution time (N=1024): 0.001256 seconds
Running matrix_gpu_optimized with N=1024
GPU execution time (N=1024): 0.000999 seconds
Running matrix_gpu_cublas with N=1024
cuBLAS execution time (N=1024): 0.000175 seconds
Running matrix_cpu with N=2048
CPU execution time (N=2048): 12.609349 seconds
Running matrix_gpu with N=2048
GPU execution time (N=2048): 0.009544 seconds
Running matrix_gpu_optimized with N=2048
GPU execution time (N=2048): 0.007171 seconds
Running matrix_gpu_cublas with N=2048
cuBLAS execution time (N=2048): 0.000954 seconds
ubuntu@147-224-39-216:~$
```

- ‘image_caller.py” specifically orchestrates convolution runs in three modes: CPU executable, GPU executable, and Python→shared-library mode.

```
ubuntu@147-224-39-216:~$ python3 image_caller.py cpu
Compiling CPU version...
Processing clock_256.pgm with cpu
  Running with filter=blur, N=3
CPU convolution time: 0.001706 seconds | image=256x256 | N=3 | filter=blur
  Running with filter=blur, N=5
CPU convolution time: 0.004139 seconds | image=256x256 | N=5 | filter=blur
  Running with filter=blur, N=7
CPU convolution time: 0.007802 seconds | image=256x256 | N=7 | filter=blur
  Running with filter=edge, N=3
CPU convolution time: 0.002344 seconds | image=256x256 | N=3 | filter=edge
  Running with filter=edge, N=5
CPU convolution time: 0.004910 seconds | image=256x256 | N=5 | filter=edge
  Running with filter=edge, N=7
CPU convolution time: 0.007823 seconds | image=256x256 | N=7 | filter=edge
Processing boat_512.pgm with cpu
  Running with filter=blur, N=3
CPU convolution time: 0.007107 seconds | image=512x512 | N=3 | filter=blur
  Running with filter=blur, N=5
CPU convolution time: 0.016732 seconds | image=512x512 | N=5 | filter=blur
  Running with filter=blur, N=7
CPU convolution time: 0.031970 seconds | image=512x512 | N=7 | filter=blur
  Running with filter=edge, N=3
CPU convolution time: 0.009431 seconds | image=512x512 | N=3 | filter=edge
  Running with filter=edge, N=5
CPU convolution time: 0.019733 seconds | image=512x512 | N=5 | filter=edge
  Running with filter=edge, N=7
CPU convolution time: 0.031474 seconds | image=512x512 | N=7 | filter=edge
Processing male_1024.pgm with cpu
  Running with filter=blur, N=3
CPU convolution time: 0.028242 seconds | image=1024x1024 | N=3 | filter=blur
  Running with filter=blur, N=5
CPU convolution time: 0.066390 seconds | image=1024x1024 | N=5 | filter=blur
  Running with filter=blur, N=7
CPU convolution time: 0.126148 seconds | image=1024x1024 | N=7 | filter=blur
  Running with filter=edge, N=3
CPU convolution time: 0.038611 seconds | image=1024x1024 | N=3 | filter=edge
  Running with filter=edge, N=5
CPU convolution time: 0.079833 seconds | image=1024x1024 | N=5 | filter=edge
  Running with filter=edge, N=7
CPU convolution time: 0.127176 seconds | image=1024x1024 | N=7 | filter=edge
All cpu convolutions completed. Outputs in cpu/
```

B) Convolution implementation (what we coded + how we ran it)

1) CPU convolution(image_convolution.c)

- Implements N×N convolution over an M×M grayscale image (PGM input).
- Supports multiple kernel sizes (3×3 / 5×5 / 7×7) and filters (edge + blur).
- Inputs referenced in the work: `clock_256.pgm`, `boat_512.pgm`, `male_1024.pgm`.

```
ubuntu@147-224-39-216:~$ ./image_conv 5 blur boat_512.pgm boat_output_blur.pgm
CPU convolution time: 0.016702 seconds | image=512x512 | N=5 | filter=blur
```

2) GPU convolution (`image_convolution_gpu.c`)

- CUDA kernel maps threads to pixels (one thread computes one output pixel).
- Same N sizes + same filters as CPU version so outputs can be compared directly.

```
ubuntu@147-224-39-216:~$ ./image_conv_gpu 5 blur boat_512.pgm boat_output_blur.pgm
GPU convolution time: 0.000128 seconds | image=512x512 | N=5 | filter=blur | iterations=1
```

3) Python calling GPU convolution through shared library

- `Matrix_test.py and image_caller.py python` calls `libmatrix.so` to run GPU convolution and saves outputs to the `python/` folder

```
ubuntu@147-224-39-216:~$ python3 matrix_test.py --task convolution --input boat_512.pgm --output boat_output.pgm
Testing Image Convolution:
  Input: boat_512.pgm
  Output: boat_output.pgm
  N: 3, Filter: blur
  Convolution time: 0.2120 seconds
  Output saved to boat_output.pgm
ubuntu@147-224-39-216:~$
```

- CPU mode (outputs in `cpu/`)
- GPU mode (outputs in `gpu/`)
- Python mode which uses libmatrix.so shared libraries (outputs in `python/`)

Analysis

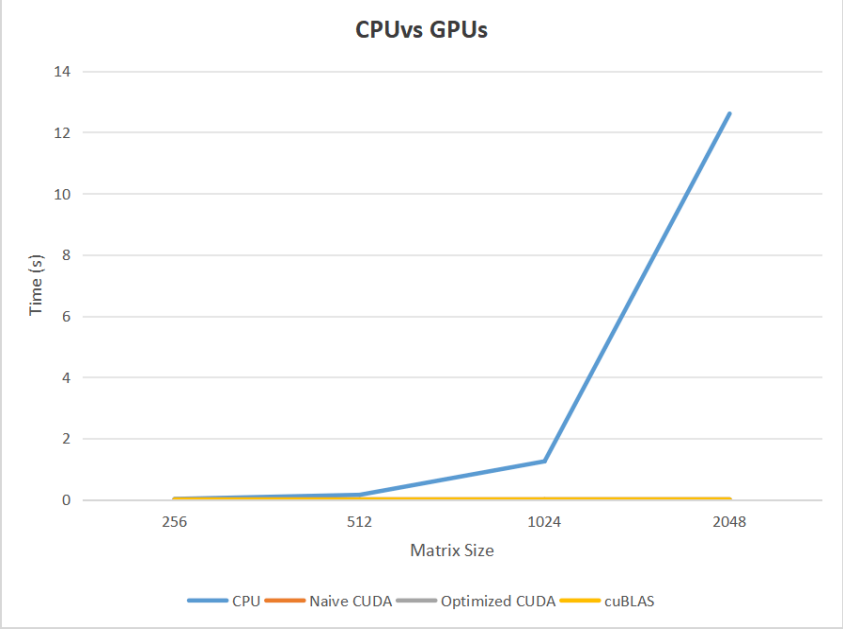
A) Matrix Multiplication

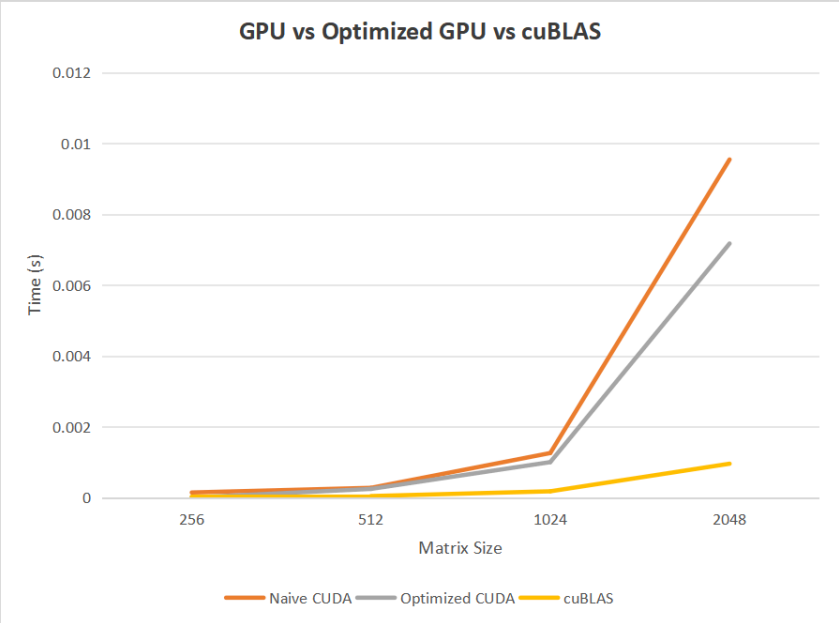
1. How does performance change as matrix size increases?
Across all implementations (CPU, Naïve CUDA, Optimized CUDA, and cuBLAS), matrix multiplication runtime increases as matrix size N increases. However, the GPU implementations scale far more efficiently than the CPU version, highlighting the advantage of parallel execution on the GPU for large matrix workloads.

Runtime Comparison (seconds):

Implementation	N=256 (sec)	512	1024	2048
CPU	0.017927	0.151396	1.247492	12.609349
Naive CUDA	0.000142	0.000273	0.001256	0.009544
Optimized CUDA	0.000134	0.000243	0.000999	0.007171
cuBLAS	0.000024	0.000041	0.000175	0.000954
Speed up	746.958333	3692.585366	7128.525714	13217.34696

*Speed up was calculated by CPU/GPU(with cuBLAS)





The results show that GPU execution significantly outperforms CPU execution even at the smallest tested size (N=256). This indicates that the acceleration benefits outweigh overhead costs such as kernel launch overhead and host↔device memory transfer overhead for all tested configurations.

For N=256N the Naïve CUDA implementation achieves an approximate speedup of 126x

This confirms that the GPU's parallelism provides substantial performance gains even at smaller matrix sizes.

2. At what point does the GPU significantly outperform the CPU?
GPU outperformed CPU across all the tested matrix sizes
3. How much speedup is gained by tiling optimization vs. naïve CUDA?

0.000008	0.00003	0.000257	0.002373

Execution time difference increased as the matrix size increased. This trend shows that the optimization became more effective with the larger tasks.

The shared-memory tiling approach improves performance (especially at larger NNN) by increasing data reuse and reducing repeated global memory accesses. As matrix size increases, the benefits of shared memory become more visible since memory access patterns increasingly dominate runtime.

4. How close is your optimized kernel to cuBLAS performance?

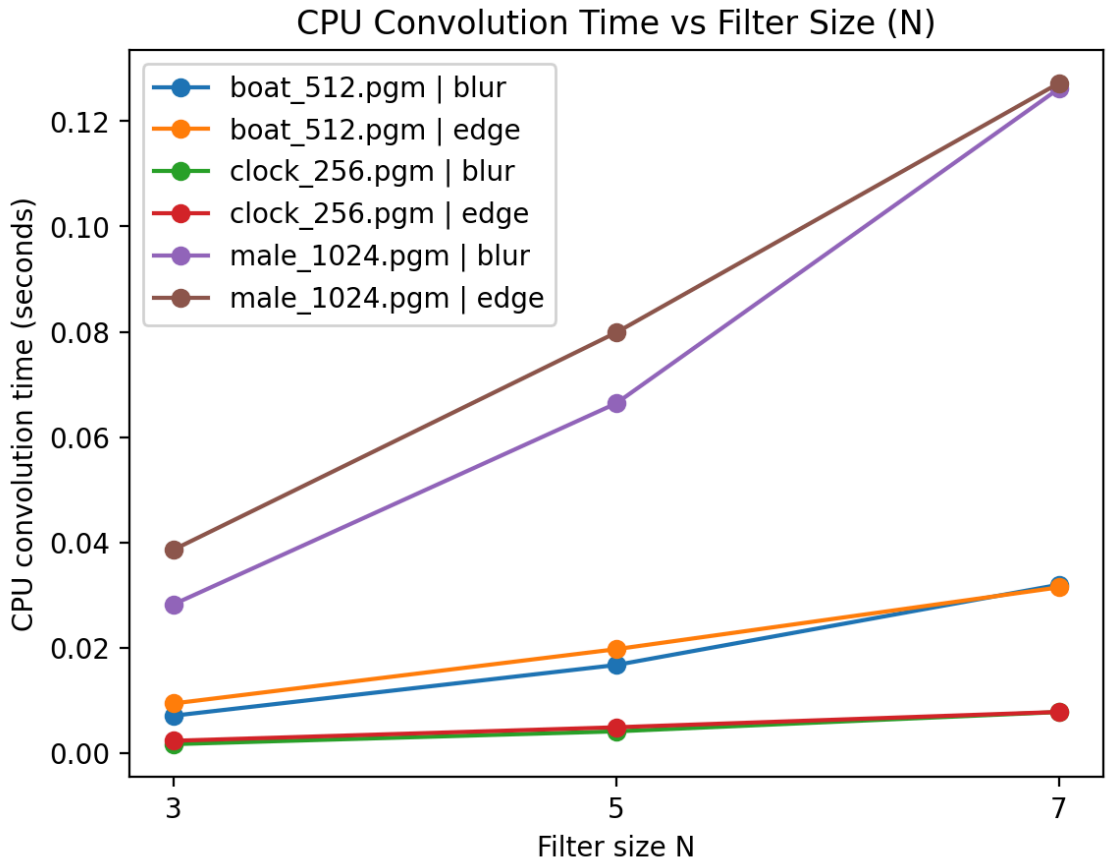
0.00011	0.000202	0.000824	0.006217

5. Why might cuBLAS still outperform hand-written kernels?

Although the optimized tiled CUDA kernel performs well, cuBLAS consistently delivers the best performance across all matrix sizes. This is primarily because cuBLAS:

- Uses highly optimized memory access strategies and heavy register-level optimization.
- Leverages specialized hardware capabilities when available (e.g., Tensor Cores where applicable).
- Is extensively tuned by NVIDIA for different GPU architectures and matrix sizes, making it difficult for custom kernels to match its performance.

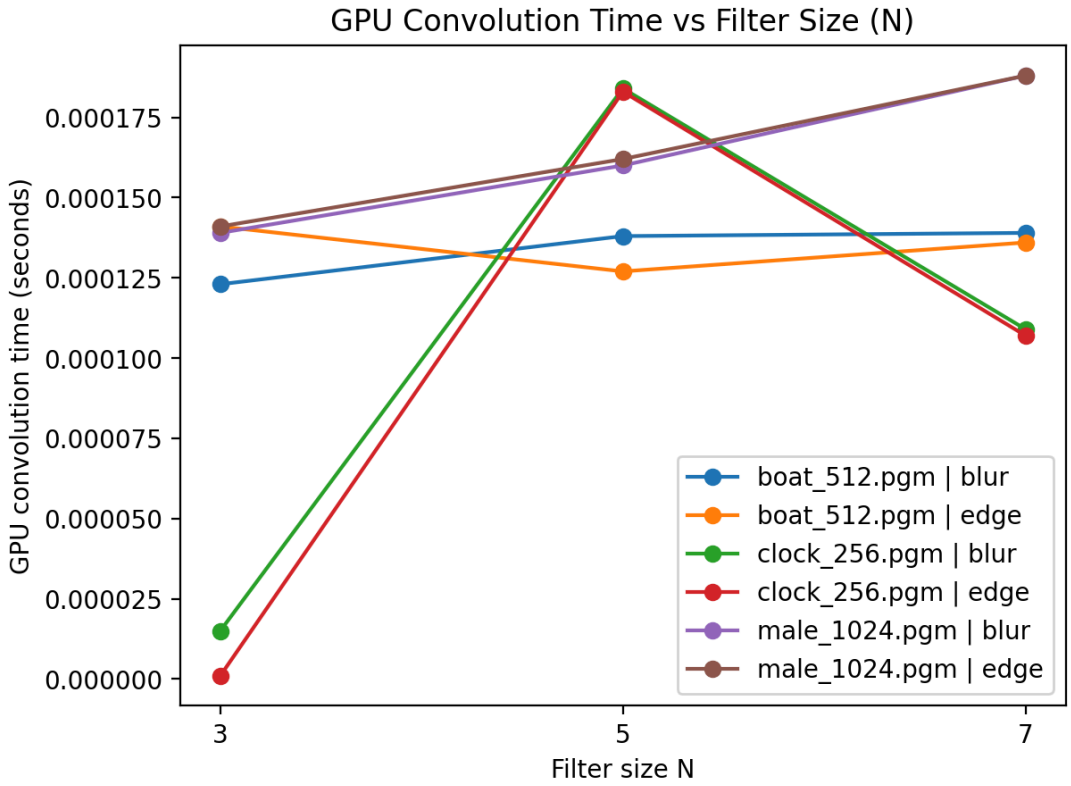
B) Convolution implementation



Interpretation:

CPU convolution time versus filter size (N = 3, 5, 7) for multiple image resolutions and filter types. Convolution time increases significantly as filter size and image resolution grow, with the

1024×1024 image showing the steepest rise. Edge filters are consistently slower than blur filters, reflecting higher computational cost on the CPU. Overall, CPU performance scales poorly compared to the GPU for larger filters and images.



Interpretation:

GPU convolution time as a function of filter size (N = 3, 5, 7) for different image sizes and filter types. Overall, convolution time increases slightly with larger filter sizes, with higher-resolution images (1024×1024) showing the highest runtimes. Blur and edge filters exhibit similar performance trends, though smaller images show more variability at lower filter sizes due to GPU overhead and kernel launch costs.

The final screenshot shows three viewer windows side-by-side:

- **Left:** original
- **Middle:** blurred output (details softened; edges less sharp)
- **Right:** edge output (edges highlighted; background largely suppressed, appearing very bright/washed depending on viewer scaling)

Interpretation:

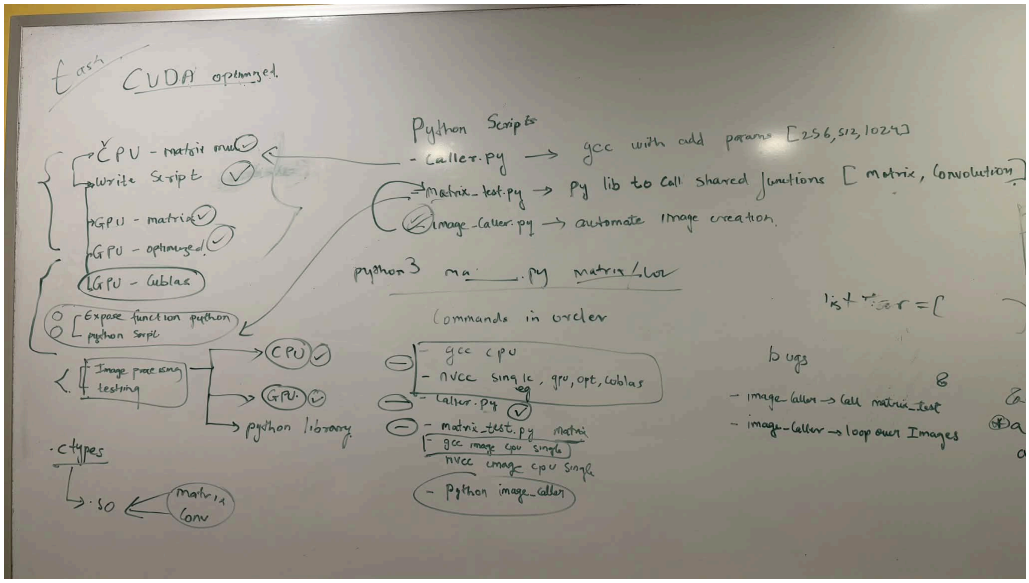
- The blur output clearly smooths high-frequency details (rigging lines + texture look softer).
- The edge output keeps only strong intensity changes (outlines / mast lines), and can look “mostly white” if the viewer auto scales contrast this is still consistent with edge detection output behavior when values clip or display normalization differs.

Shared Library for Python

A CUDA shared library was built to enable GPU-accelerated functions to be called directly from Python via `ctypes`.

Component	Description
Source File	matrix_lib.cu
Shared Library	libmatrix.so (built using nvcc)
Interface	C-callable functions exposed for Python ctypes
Key Functions	matrix_multiply(float *h_A, float *h_B, float *h_C, int N) for GPU matrix multiplication; convolute_image(unsigned char *h_input, unsigned char *h_output, int width, int height, int N, float *h_kernel) for GPU image convolution
Usage	Python driver scripts (caller.py, matrix_test.py, image_caller.py) invoke the shared library using contiguous NumPy buffers

Appendix



This is our pipeline which outlines a CUDA-optimized project comparing CPU and GPU implementations for matrix multiplication and image processing.
The core computation is built as shared `.so` libraries using `gcc` (CPU) and `nvcc` (GPU), then exposed to Python via `ctypes`. Python scripts orchestrate testing, benchmarking, and image generation by calling these shared libraries.