Fall 2022 ME/CS/ECE759 Final Project Report
University of Wisconsin-Madison

Analysis of image processing algorithms (implemented using HPC frameworks) & their comparison to OpenCV functions in terms of quality & performance.

Tanmay Anand

December 14, 2022

Abstract

Digital image processing is the use of a digital computer to process digital images through an algorithm. As a subcategory or field of digital signal processing, digital image processing has many advantages over analog image processing. It allows a much wider range of algorithms to be applied to the input data and can avoid problems such as the build-up of noise and distortion during processing. Over the years for use in Computer Vision, AI and Computer Graphics developers have developed super optimized image processing libraries to achieve best possible performance and quality for digital image processing. In this project, I have developed my own set of image processing algorithms using High Performance Computing (HPC) frameworks such as CUDA and OpenMP and compared their performance with simple single threaded image processing workload and OpenCV's implementations.

The link to the project repo is mentioned below, the Project files and data is in the same repo as the CS 759 course (i.e repo 759), directory name as: "Project". Prof. Negrut and the TAs already have access to the repo and should be able to access the Project directory within it. My whole code works perfectly on Euler (except for the OpenCV sections which have been implemented keeping in mind a personal Mac machine. However, if you want you can still compile & run the code for OpenCV sections in my repo on Euler using the CMake scripts I have provided in each directory of my code**). I have also added all the run logs in my git repo for proof that all my implementations run and have provided details to replicate my results in the report below.**

Link to Final Project git repo: **https://git.doit.wisc.edu/TANMAYANAND/repo759/-/tree/main/Project**

Contents

General information

Home department: **Computer Sciences**
Current status: **MS**

"I release the ME759 Final Project code as open source and under a BSD3 license for unfettered use of it by any interested party."

1. Problem statement

In this project, I have implemented multiple image processing algorithms using HPC frameworks like CUDA & OpenMP and evaluated their quality & performance for different image workloads as well as compared their performance to OpenCV's implementation in terms of quality & performance. The image processing algorithm's performance analysis also includes sequential implementations (in simple C++).

Algorithms proposed:

• Image smoothening:

- Gaussian Blur (3x3, 5x5)
- Normalized/Box Blur (3x3, 5x5)
- Median Blur (3x3, 5x5)

• Edge Detection:

- Sobel Filter (3x3)

• Morphological Operations (Square shaped – 3x3):

- Dilation
- Erosion

Implemented using:

1. CUDA ------- Run on Euler
2. OpenMP ------- Run on Euler (num_cpus = 20)
3. OpenCV functions (used the library directly) --- Run on Apple Mac M2 8-core CPU with 8-core GPU
4. Sequential implementation (using only C++) -------- Run on Euler (num_cpus = 1)

1.1 Motivation:

A. Libraries like OpenCV have been very popular with Image processing/Computer Vision enthusiasts over the years. These libraries such as OpenCV have been implemented in optimized C & include ninja level optimization approaches at the backend in order to accelerate quality & performance on both CPU & GPU. Assessing my own optimized implementation using HPC frameworks such as OpenMP & CUDA with OpenCV functions in terms of output quality & performance will provide significant insights into the extent to which these implementations have been optimized.
B. I decided to run my OpenCV implementation for the above-mentioned algorithms on my personal computing machine. For the other implementations I have used Euler. The reason for this choice is: 1. Libraries such as OpenCV are highly popular within personal computing environments (such as video

editing, image editing, special effects etc used by a single user on their laptop machine). This project will be an interesting opportunity to compare performance of workloads within a personal computing environment which is designed for multi-tasking using limited resources (in case of OpenCV) with a HPC environment using CUDA & OpenMP (on Euler**). It will bring in a new perspective of how highly software optimized computing on a personal computer performs against highly hardware optimized computing on Euler (basically how software optimization does against a high-performance dedicated hardware on Euler which uses latest NVIDIA GPUs and high-performance CPU cores)** 2. Euler may not provide me support for using OpenCV (basically I will not be able to install it on Euler as it is CAE managed).

C. I am also doing research on GPU computing under Prof. Matt Sinclair in Dept of CS (https://pages.cs.wisc.edu/~sinclair/). There we are implementing a multi-chip GPU scheduler for enhanced performance. CUDA implementations of this project will help me assess performance of our research work on large image processing workloads (save my work for implementing more compute intensive benchmarks separately).

1.2 Background:

2.2.1 Gaussian Blur

Gaussian blur (also known as **Gaussian smoothing**) is the result of blurring an image using a Gaussian kernel. It is widely used typically to reduce image noise in graphics software.

$$G(x,y) = \frac{1}{2\pi\sigma^2} e^{-\frac{x^2+y^2}{2\sigma^2}}$$

where x is the distance from the origin in the horizontal axis, y is the distance from the origin in the vertical axis, and σ is the standard deviation of the Gaussian distribution. It is important to note that the origin on these axes is at the center (0, 0). When applied in two dimensions, this formula produces a surface whose contours are concentric circles with a Gaussian distribution from the center point.

Kernel Used for blurring: (3x3):

$$\frac{1}{16}\begin{bmatrix} 1 & 2 & 1 \\ 2 & 4 & 2 \\ 1 & 2 & 1 \end{bmatrix}$$

Kernel Used for blurring: (5x5):

$$\frac{1}{256}\begin{bmatrix} 1 & 4 & 6 & 4 & 1 \\ 4 & 16 & 24 & 16 & 4 \\ 6 & 24 & 36 & 24 & 6 \\ 4 & 16 & 24 & 16 & 4 \\ 1 & 4 & 6 & 4 & 1 \end{bmatrix}$$

2.2.2 Normalized Blur

A **box blur** (also known as a box linear filter) is a spatial domain linear filter in which each pixel in the resulting image has a value equal to the average value of its neighboring pixels in the input image. It is a form of low-pass ("blurring") filter. These are used to approximate a gaussian blur (where gaussian blurs are computationally expensive to compute

2.2.3 Median Blur

The Median blur operation is like the other averaging methods. Here, the central element of the image is replaced by the median of all the pixels in the kernel area. This operation processes the edges while removing the noise.

2.2.4 Sobel Filter

The Sobel operator, sometimes called the Sobel–Feldman operator or Sobel filter, is used in image processing and computer vision, particularly within edge detection algorithms where it creates an image emphasizing edges.

The operator uses two 3×3 kernels which are convolved with the original image to calculate approximations of the derivatives – one for horizontal changes, and one for vertical. If we define A as the source image, and Gx and Gy are two images which at each point contain the horizontal and vertical derivative approximations respectively, the computations are as follows:

$$\mathbf{G}_x = \begin{bmatrix} +1 & 0 & -1 \\ +2 & 0 & -2 \\ +1 & 0 & -1 \end{bmatrix} * \mathbf{A} \quad \text{and} \quad \mathbf{G}_y = \begin{bmatrix} +1 & +2 & +1 \\ 0 & 0 & 0 \\ -1 & -2 & -1 \end{bmatrix} * \mathbf{A}$$

The *x*-coordinate is defined here as increasing in the "right"-direction, and the *y*-coordinate is defined as increasing in the "down"-direction. At each point in the image, the resulting gradient approximations can be combined to give the gradient magnitude, using:

$$\mathbf{G} = \sqrt{\mathbf{G}_x^{\,2} + \mathbf{G}_y^{\,2}}$$

2.2.5 Dilation

Dilation expands the image pixels using a kernel whose basic effect on the image is to gradually enlarge the boundaries of regions of foreground pixels. Thus, areas of foreground pixels grow while holes within those regions become smaller. The value of the output pixel is the maximum value of all the pixels in the kernel neighborhood.

2.2.6 Erosion

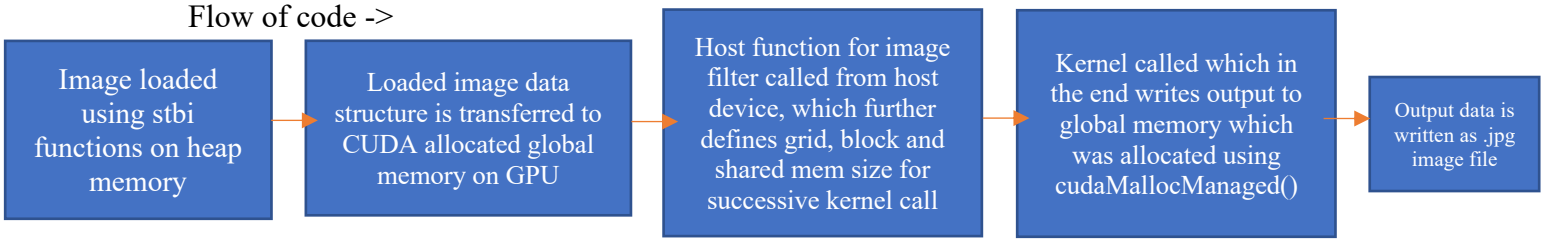
Erosion shrinks the image pixels using a kernel whose basic effect on the image is to gradually contract the boundaries of regions of foreground pixels. The value of the output pixel is the minimum value of all the pixels in the kernel neighborhood.

2. Solution description

The project has been segregated into four sections based upon the implementation framework/methodology.

First let's talks about CUDA implementation which extracts performance using GPU computing.

Flow of code ->



In my CUDA implementations, I make use of shared memory with tiling. Each thread block corresponds or deals with one input image tile.

Moreover, kernel block -> 2D structure
Kernel grid -> 2D structure

threads_per_block_dim -> variable defined by user though command line arguments
threads per block along x -> threads_per_block_dim
threads per block along y -> threads_per_block_dim
blocks per grid along x -> ceiling(image width / threads per block along x)
blocks per grid along y -> ceiling(image height / threads per block along y)
Shared Memory Size = (threads per block along x + 1)(threads per block along y +1) * OPERAND SIZE * number of image channels (=3 for RGB images)*

A question might arise: why shared memory size is not = total number of threads in a block * OPERAND SIZE * number of image channels ?

Well, when we convolve a kernel across an image, we also need elements along the edges for our tiled computation as pixels along the edge of a tile block need a few elements outside of tile boundary to be a part of convolution computation for that pixel coordinates.

Consider this:

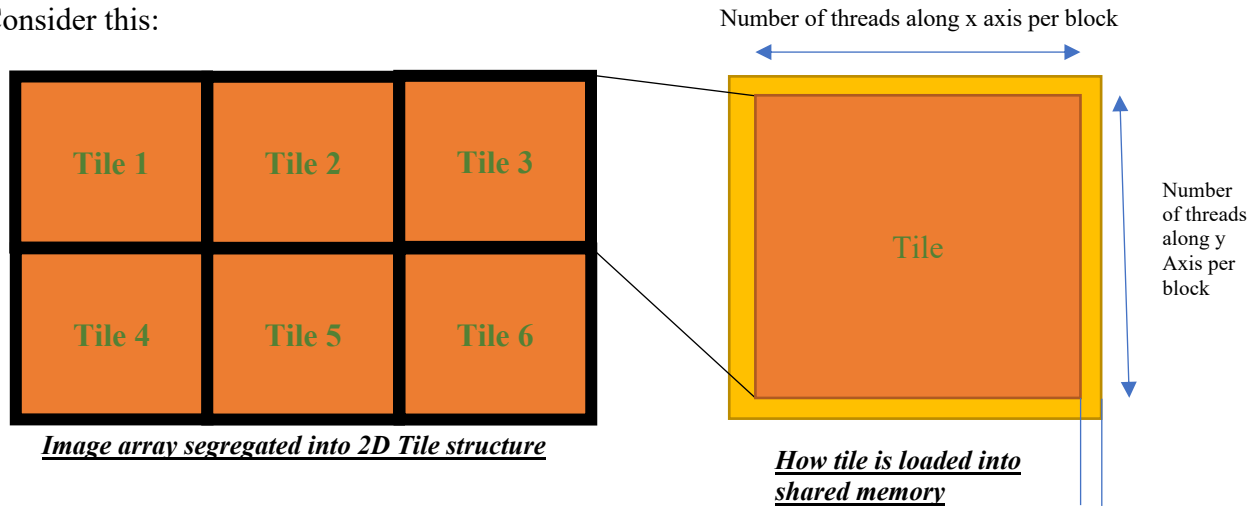


Image array segregated into 2D Tile structure

How tile is loaded into shared memory

How a CUDA kernel is further implemented:

1. Loading of data onto the shared memory takes place

2. Each thread loads RGB values for a single pixel from input image (for the region which corresponds to the current thread block/tile onto the shared memory).

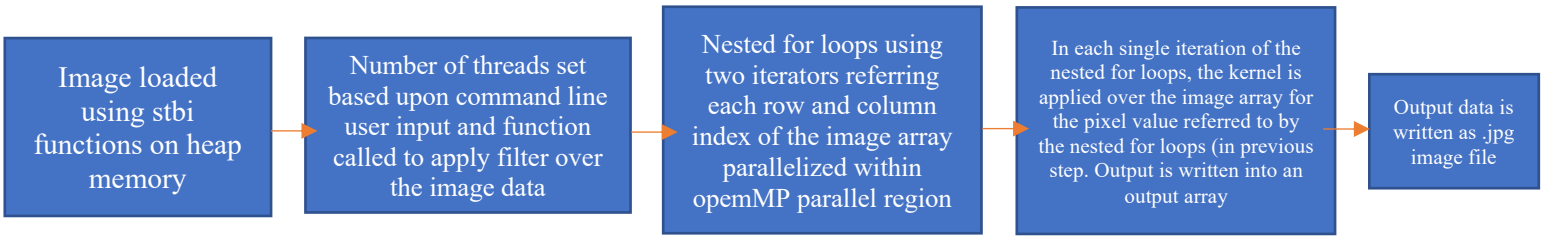
3. However, thread 0 is the only thread which loads all the padded data elements for the input image as well as Pixel values from neighboring tiles for kernel convolution. (Yellow marked region in second figure above)

4. Further computation for final output is done using loop unrolling. As for image processing algorithms the kernel size is mostly only 3x3 or 5x5. Kernel sizes such as 7x7 or above are scarcely used and carry no significant application in real life. Hence as we already know the kernel size we should be working upon; loop unrolling is used to avoid conditional operations within a kernel and reduce addition overhead of maintaining iterators or other secondary iteration logic/variables.

NOTE: The image processing assumes a padding of value: 0 around the image boundary.

<To understand as an example: check—src/cuda/edge-detection/sobel/sobelfilter.cu>

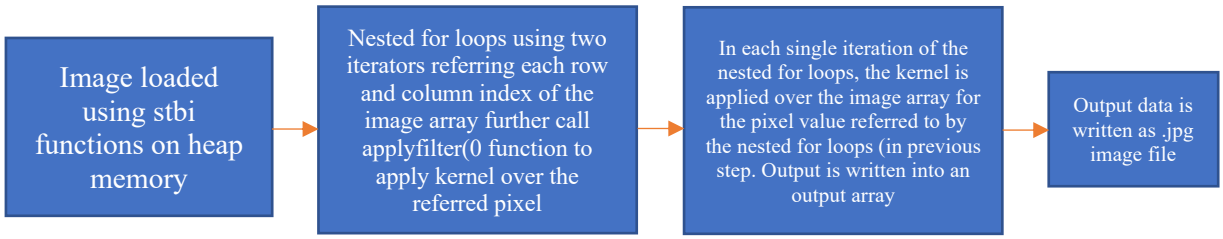
Now we move on to OpenMP implementation:



In Step 4 above:

Further computation for final output is done using loop unrolling. As for image processing algorithms the kernel size is mostly only 3x3 or 5x5. Kernel sizes such as 7x7 or above are scarcely used and carry no significant application in real life. Hence as we already know the kernel size we should be working upon; loop unrolling is used to avoid conditional operations within a kernel and reduce additional overhead of maintaining iterators or other secondary iteration logic/variables. Maximum number of possible openMP threads is 20 in our experimentation. Loop unrolling was an important optimization idea taught in CS759.

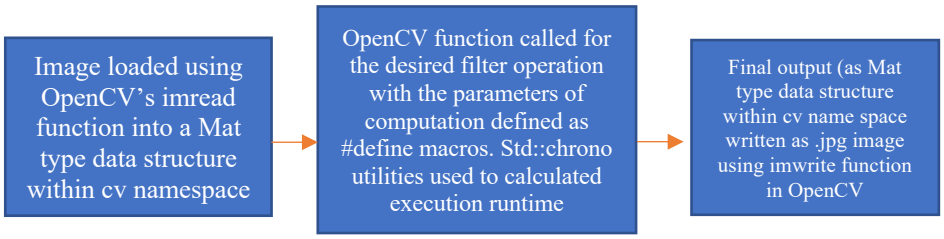
Now we move on to Sequential implementation:



In Step 3 above:

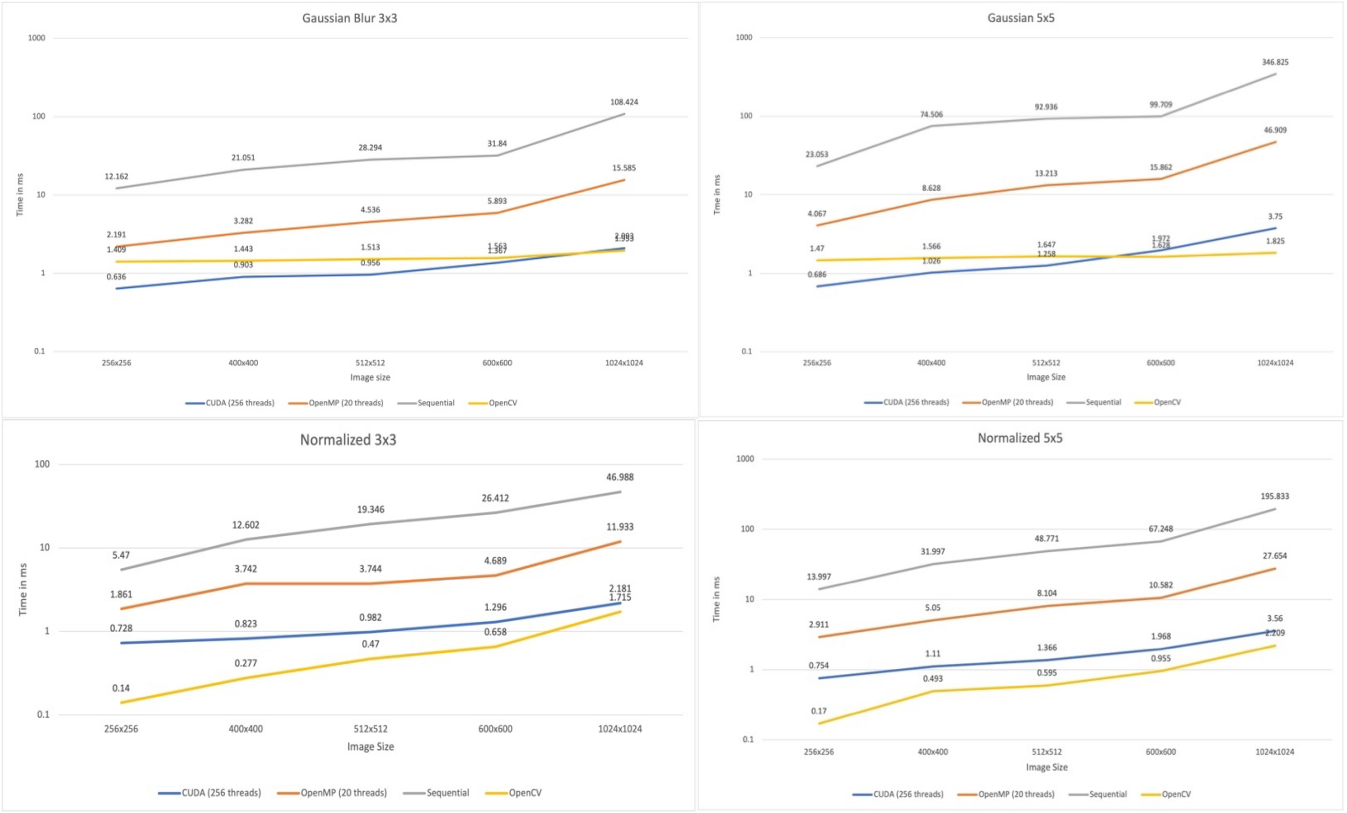
Further computation for final output is done using loop unrolling. As for image processing algorithms the kernel size is mostly only 3x3 or 5x5. Kernel sizes such as 7x7 or above are scarcely used and carry no significant application in real life. Hence as we already know the kernel size we should be working upon; loop unrolling is used to avoid conditional operations within a kernel and reduce addition overhead of maintaining iterators or other secondary iteration logic/variables. . Loop unrolling was an important optimization idea taught in CS759.

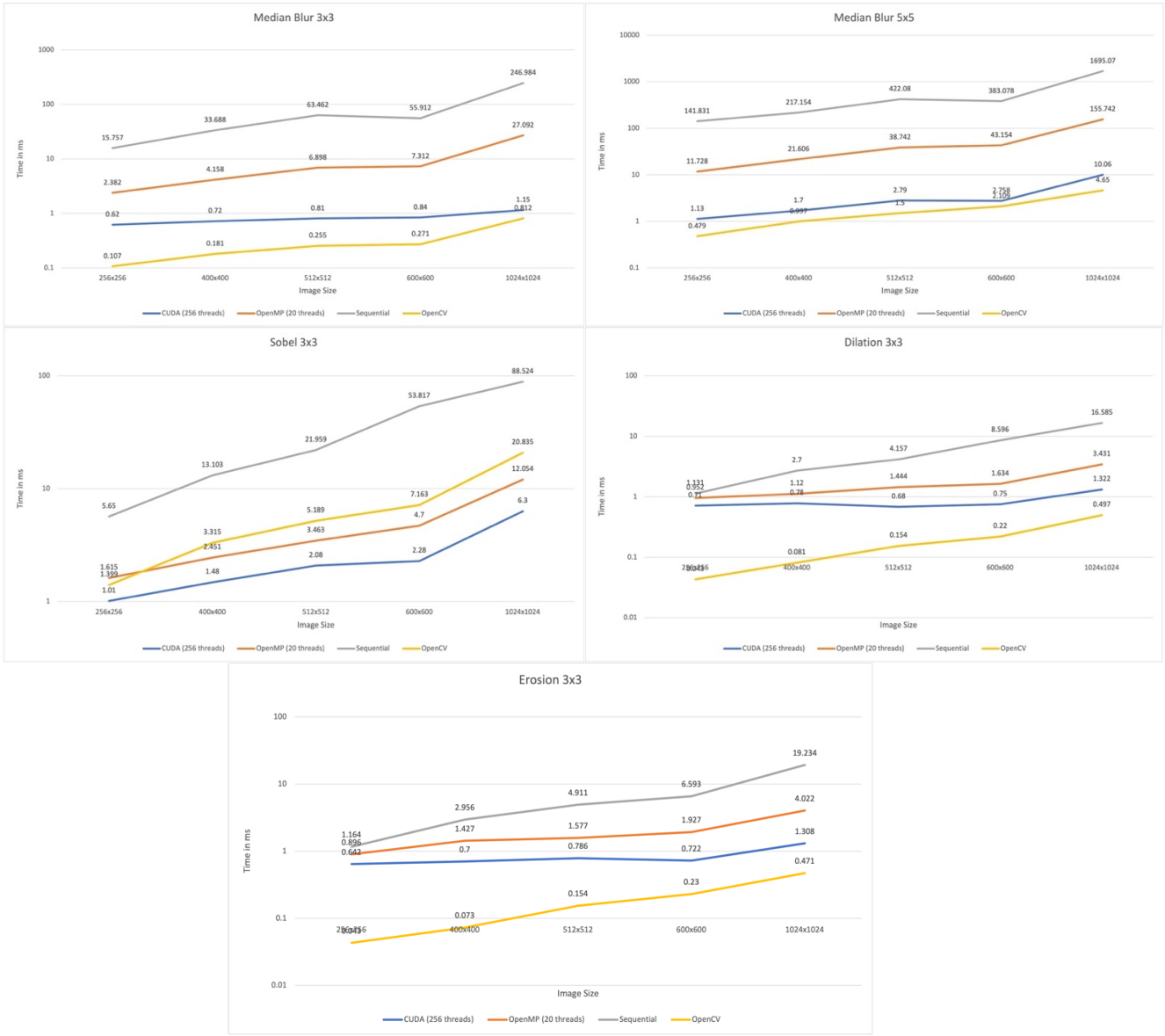
Now moving on to OpenCV implementations:



3. Overview of results. Demonstration of your project

Execution runtime comparison graphs: (NOTE – THESE GRAPHS WITH THE RUNTIME VALUES ARE PRESENT IN THE LOG DIRECTORY ON PROJECT REPO. YOU MAY USE THOSE TO HAVE AN EVEN CLOSER LOOK)





Observation: It is clearly visible that across all the 9 image processing algorithms, CUDA implementation is extremely fast in comparison to OpenMP and Sequential implementations. Moreover, OpenCV based image processing which ran on a personal computing machine (on Apple M2 CPU and 8 core GPU), is the fastest among all the 4 implementations (except for the case of Sobel filter where CUDA implementation which ran on Euler is faster). Among OpenMP and sequential implementations, OpenMP's runtimes are as much as 10x lower than sequential implementations (as in case of median blur above).

Inference: While tremendous acceleration achieved by CUDA implementations over OpenMP and Sequential implementations is expected due to GPU computing being best suited for graphics operations. A nearly close/slightly better execution speed achieved by OpenCV implementations on a personal computing machine is an interesting result. This is partly due to OpenCV implementations being ninja-level optimized as they are used extensively in DNN inferencing and training by data

scientists. My own implementations in CUDA and OpenMP here might have more opportunity for further runtime optimization (which can be further investigated in future works).

Quality of processing:

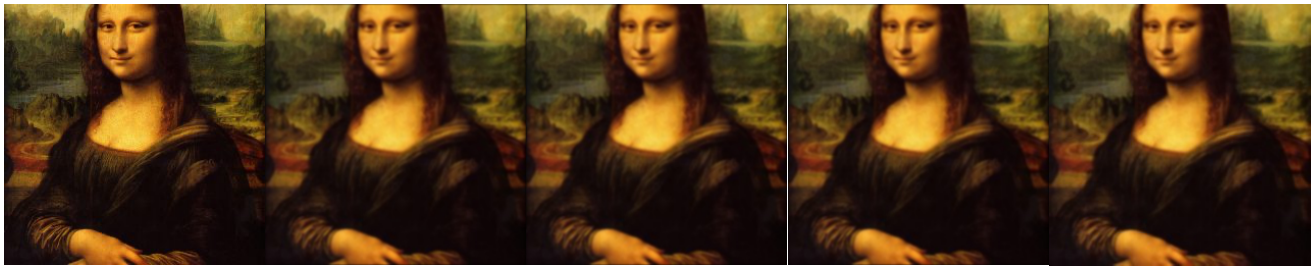
Images below correspond to file: **only one image for each "operation type"** in inputs directory on Project repo. In total there are 5 outputs for 5 inputs for each of the 9 image processing algorithms in the repo.
(NOTE: TO CHECK PROCESSING RESULTS FOR OTHER IMAGES YOU MAY HAVE A LOOK AT THE OUTPUT DIRECTORY IN DETAIL)

PLEASE ZOOM THE PDF FILE TO UNDERSTAND QUALITY DISTINCTIONS OR OPEN IMAGES DIRECTLY FROM OUTPUT DIRECTORY. PS: DISTINCTION IS CLEAR FOR LARGE ZOOM VALUES.

Blurring 3x3 (Gaussian):

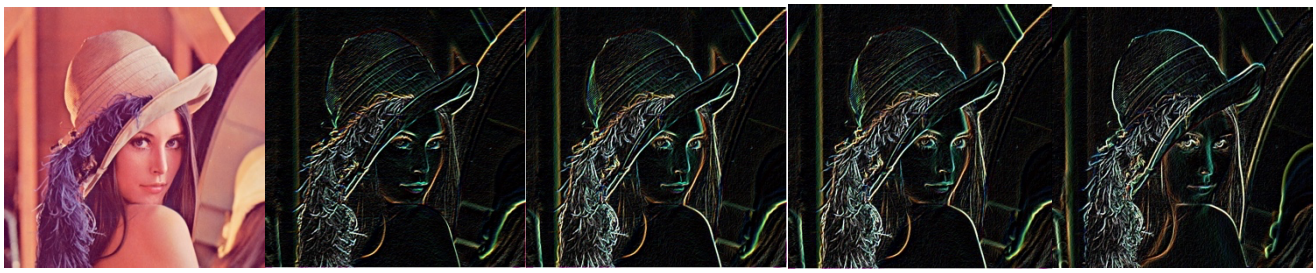
Original **CUDA** **OpenMP** **Sequential** **OpenCV**

Blurring 5x5 (Gaussian):



Original CUDA OpenMP Sequential OpenCV

Edge Detection:



Original CUDA OpenMP Sequential OpenCV

Dilation:

Original CUDA OpenMP Sequential OpenCV

Erosion:

Original CUDA OpenMP Sequential OpenCV

Observation & Inference: The outputs for CUDA, OpenMP, Sequential and OpenCV based implementations are similar in terms of quality for all the operations across all the images, except for edge detection (sobel) filter in which case OpenCV's sobel output seems to be less fine grained in comparison to my own implementation. A reason for that might be the following lines:

```
grad_x.convertTo(grad_x,CV_64F);

grad_y.convertTo(grad_y,CV_64F);
```

which do type conversion to CV_64 for squaring & adding two Mat type data structures under cv namespace. No such conversion was required after calculating Gx and Gy in case of my own implementations in CUDA, OpenMP and Sequential codes (all the computation occurs in float data type space and converted to unsigned int in the end). In my implementations for edge detection, a greater number of edges are visible a bit more precisely.

Thus, it can be inferred that for most applications/algorithms, OpenCV does not implement algorithms within their library any differently in order to get best possible processing quality.

4. Deliverables:

The Project files are within repo759 (the one used for CS 759 homework) under Project directory
The Directory structure is such:
SPECIAL NOTE: TO HELP YOU ALL TRACK MY COMMITS FOR PROJECT I HAVE ADDED PREFIX: [PROJECT] IN EACH OF MY COMMITS WHICH CORRESPOND TO PROJECT SUBMISSION

```
├── inputs -> INPUT IMAGE FILES
│   ├── 1.jpg ->256x256 image
│   ├── 2.jpg ->400x400 image
│   ├── 3.jpg ->512x512 image
│   ├── 4.jpg ->600x600 image
│   └── 5.jpg ->1024x1024 image
├── logs
│   ├── graphs -> Includes graphs and excel log for graphs added in the report
│   ├── out_<implementation_type>-<image_index>.txt -> OUTPUT OF RUN LOGS FOR ALL IMPLEMENTATIONS
├── outputs  -> Includes outputs of all the input images as <image index>.jpg-out.jpg ex: 1.jpg-out.jpg
│   ├── cuda
│   │   ├── edge-detection
│   │   │   └── sobel
│   │   ├── image-smoothening
│   │   │   ├── gaussian-blur
│   │   │   │   ├── 3x3
│   │   │   │   └── 5x5
│   │   │   ├── median-blur
│   │   │   │   ├── 3x3
│   │   │   │   └── 5x5
│   │   │   └── normalized-blur
│   │   │       ├── 3x3
│   │   │       └── 5x5
│   │   └── morphological-operations
│   │       ├── dilation
│   │       └── erosin
│   ├── openmp -> Same directory structure as CUDA outputs
│   ├── opencv -> Same directory structure as CUDA outputs
│   └── sequential -> Same directory structure as CUDA outputs
├── src -> Includes CODE + OUTPUT BINARIES GENERATED for all implementations
│   ├── cuda
│   │   ├── edge-detection
│   │   │   └── sobel
│   │   ├── image-smoothening
│   │   │   ├── gaussian-blur
│   │   │   │   ├── 3x3
│   │   │   │   └── 5x5
│   │   │   ├── median-blur
│   │   │   │   ├── 3x3
│   │   │   │   └── 5x5
│   │   │   └── normalized-blur
│   │   │       ├── 3x3
│   │   │       └── 5x5
│   │   └── morphological-operations
│   │       ├── dilation
│   │       └── erosin
│   ├── image.hpp -> Image Processing library (MIT License) used to open images as array on Euler
```

```
├── image_write.hpp -> Image Processing library (MIT License) used to write image files on Euler
├── opencv -> Includes source code and OUTPUT BINARIES -- .out files
│   ├── edge-detection
│   │   └── sobel
│   │       ├── CMakeCache.txt -> CMake Cache
│   │       ├── CMakeFiles
│   │       ├── CMakelists.txt -> build using cmake .
│   │       ├── Makefile -> Build binaries using make
│   │       ├── sobel.cpp -> source code
│   │       └── sobel.out -> output binary
│   ├── image-smoothening
│   │   ├── gaussian-blur
│   │   │   ├── 3x3
│   │   │   └── 5x5
│   │   ├── median-blur
│   │   │   ├── 3x3
│   │   │   └── 5x5
│   │   └── normalized-blur
│   │       ├── 3x3
│   │       └── 5x5
│   └── morphological-operations
│       ├── dilation
│       └── erosin
├── openmp -> Same directory structure as CUDA source code
└── sequential-> Same directory structure as CUDA source code
├── run_opencv.sh -> USE THIS SCRIPT TO RUN OPENCV BINARIES GENERATED ON APPLE MACHINE FOR ALL IMAGES
├── submit-project-cuda.sh -> SLURM SCRIPTS TO RUN CUDA CODE ON EULER
├── submit-project-openmp.sh -> SLURM SCRIPTS TO RUN OPENMP CODE ON EULER
└── submit-project-sequential.sh -> SLURM SCRIPTS TO RUN SEQUENTIAL CODE ON EULER
```

NOTE: My project has delivered all the deliverables as described in the original proposal.

To run CUDA code: --- ON EULER

```
sbatch submit-project-cuda.sh
```

To run OpenMP code: --- ON EULER

```
sbatch submit-project-openmp.sh
```

To run Sequential code: --- ON EULER

```
sbatch submit-project-sequential.sh
```

To run OpenCV code: --- ON MAC MACHINE WITH OPENCV INSTALLED

```
./run_opencv.sh
            To build OpenCV code on a Mac machine (M2 generation)
            Try:  cmake .
            Then: make
            For each opencv source code directory
```

All these will generate output images within the output directory as shown in directory structure above. (PS: You can track paths in the output generated on terminal when you run any binary at the last line with the runtime)

SAMPLE OUTPUT FOR GAUSSIAN BLUR (CUDA) BINARY: --- ON EULER

```
-- GAUSSIAN BLUR -- 3x3
IMAGE FILE: inputs/3.jpg
WIDTH OF IMAGE: 512
HEIGHT OF IMAGE: 512
Runtime: 0.956416
SAVED OUTPUT IMAGE FILE: outputs/cuda/image-smoothening/gaussian-blur/3x3/3.jpg-out.jpg

SAMPLE OUTPUT FOR GAUSSIAN BLUR (OpenMP) BINARY: --- ON EULER

-- GAUSSIAN BLUR -- 3x3
IMAGE FILE: inputs/4.jpg
WIDTH OF IMAGE: 600
HEIGHT OF IMAGE: 600
Runtime: 5.89317
SAVED OUTPUT IMAGE FILE: outputs/openmp/image-smoothening/gaussian-blur/3x3/4.jpg-out.jpg

SAMPLE OUTPUT FOR GAUSSIAN BLUR (OpenCV) BINARY: --- ON APPLE MAC M2

-- GAUSSIAN -- 3x3
IMAGE FILE: inputs/2.jpg
WIDTH OF IMAGE: 400
HEIGHT OF IMAGE: 400
Runtime: 1.54075
SAVED OUTPUT IMAGE FILE: outputs/opencv/image-smoothening/gaussian-blur/3x3/2.jpg-out.jpg
```

5. Conclusions and Future Work

The project shows that OpenCV implementations are still highly optimized and in future deeper optimization techniques can be explored by me to extract better performance from my code.
Moreover, right now my project only implements 9 different cases of image processing (with 6 different algorithms). More such image processing algorithms such as Hough Line transform, Laplace etc can be implemented and their performance and quality of processing can be compared.
The project leverages following aspects of ME 759:

1. CUDA: My work will make use of CUDA implementations of image processing algorithms
2. OpenMP: My work will make use of OpenMP implementations of image processing algorithms
3. Parallel Programming: This project will heavily draw upon exploiting maximum parallelism from High performance computing hardware for image processing.
4. HPC Environment: Euler provides state-of-the-art GPU & CPU cluster for high performance computing. CS759 course homework & lectures have given me good amount of exposure to using Euler & my work draws upon comparison of running workloads on Euler vs running them on personal computing environment too.

References

[1] https://github.com/nothings/stb -- Wonderful open-source (MIT License) image processing library which allowed me to open image files as arrays in my code on Euler when OpenCV was not present.
[2] https://docs.opencv.org/3.4/index.html -- OpenCV Website
[3] https://en.wikipedia.org/wiki/Kernel_(image_processing)