High Performance Computing

PARALLELIZATION OF SAR IMAGE PROCESSING

Reference link

M.Venkata Sagar - S20220010129
T.Sai Kiran - S20220010225
M.Maneesh Madhav - S20220010131

TEAM 3

INTRODUCTION

- The project focuses on enhancing the performance of SAR (Synthetic Aperture Radar) image processing using parallel computing techniques.
- It aims to compare the efficiency of different implementation strategies: sequential pixel-wise manipulation, OpenCV functions, OpenMP-based parallelization, and CUDA acceleration.
- The objective is to demonstrate significant speedup and scalability using HPC frameworks, making SAR processing faster for real-time applications in remote sensing and defense.

OBJECTIVE OF THE WORK

- The primary objective of this project is to enhance the performance of SAR (Synthetic Aperture Radar) image processing by leveraging parallel computing techniques. SAR image processing involves a series of computationally expensive operations such as grayscale conversion, brightness and contrast adjustment, histogram equalization, and noise filtering.
- The goal is to quantify performance improvements, identify bottlenecks using profiling tools like gprof, and showcase the scalability and speedup achieved through OpenMP and CUDA.

PERFORMANCE METRICS

TEAM 3

1. Grayscale Conversion

- **Objective**: Convert colored images (RGB) into grayscale.
- **Why**: SAR images usually don't have color. They represent intensity or reflectivity. Converting to grayscale simplifies processing by reducing channels from 3 (RGB) to 1.
- **How**: Use a weighted average of R, G, and B values: Gray = 0.299*R + 0.587*G + 0.114*B

2.Rotate and Flip

Objective: Reorient the image.
Why:
Sometimes SAR images are captured at odd angles. Rotation and flipping correct their orientation for consistent analysis or visualization.
How:

- **Rotate:** Clockwise or counterclockwise by 90°, 180°, etc.
- **Flip:** Horizontally (mirror image) or vertically (upside down).

3. HSV Image Conversion

- **Objective:** Convert the image from RGB to HSV color space.
- **Why:**
- HSV (Hue, Saturation, Value) separates color information (H and S) from brightness (V).
- In SAR, manipulating V helps adjust brightness/contrast without affecting "color", aiding analysis of radar return intensities.
- **How:**
- Use standard RGB → HSV conversion formulas.

4.Brightness & Contrast Adjustment

Objective: Modify brightness and contrast to enhance image quality.
Why:
SAR images often have poor lighting or are too dark/light. Adjusting brightness improves visibility; adjusting contrast enhances features like edges.
How:

NewPixel = alpha * Pixel + beta
alpha > 1 increases contrast
beta > 0 increases brightness

5. Image Clipping

- **Objective:** Crop a specific region from the image.
- **Why:**
- Sometimes only a region of interest (ROI) needs to be processed (e.g., a battlefield zone in satellite imagery).
- **How:**
- Select a rectangular subregion:
- clippedImage = image[x1:x2, y1:y2]

6. Histogram Equalization

Objective: Enhance contrast by spreading out pixel intensity values.
Why:
SAR images often have clustered intensity values due to poor lighting or environmental factors. Histogram equalization redistributes these values for a more uniform contrast.
How:

- Compute histogram of pixel intensities.
- Compute CDF (Cumulative Distribution Function).
- Map old values to new values based on the CDF.

7. Wiener Filtering

- **Objective:** Reduce noise while preserving edges.
- **Why:**
- SAR data is usually noisy due to speckle noise. Wiener filter adapts to local image statistics to remove noise without blurring details.
- **How:**
- Based on local mean and variance:
- Output = Mean + (Variance - Noise) / Variance * (Input - Mean)

8. Gaussian Filtering

Objective: Smooth the image by blurring it using a Gaussian kernel.

Why:

Reduces noise and fine details. Useful as a preprocessing step before edge detection or feature extraction.

How:

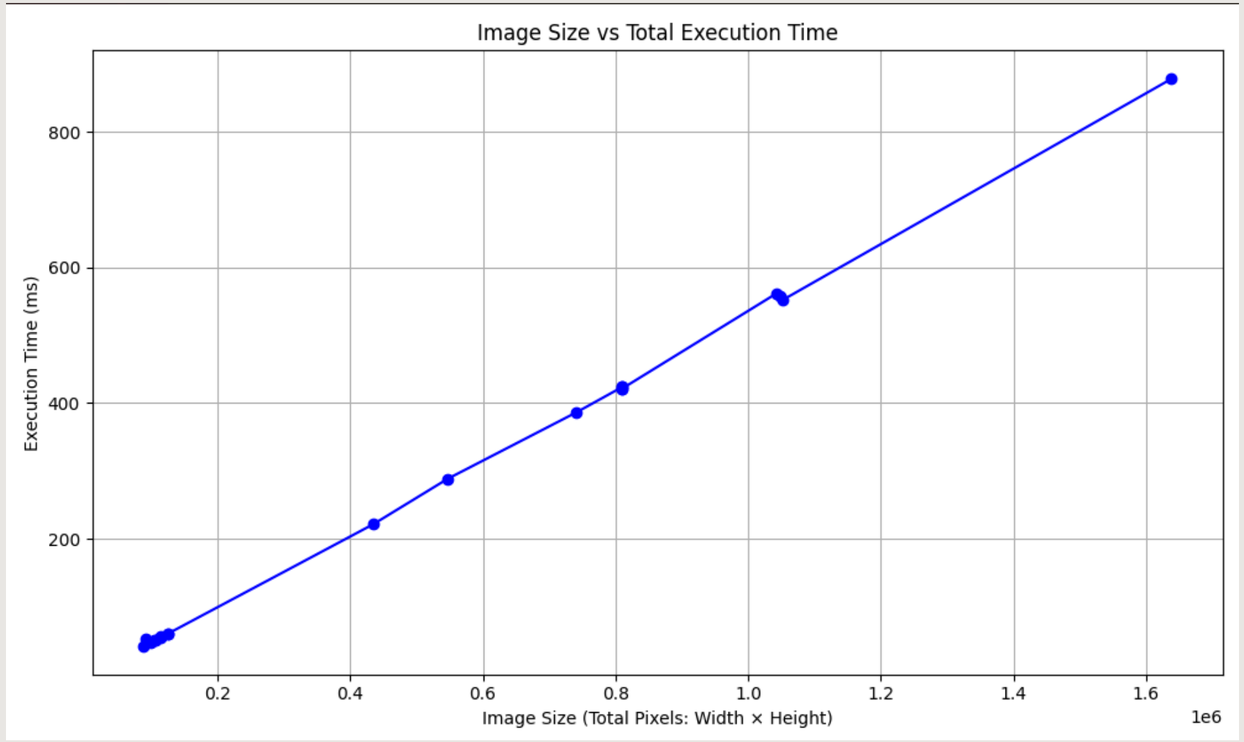
Convolve the image with a 2D Gaussian kernel:

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

RESULTS AND OUTCOMES

STEP 1 - SEQUENTIAL PROCESSING

Image Nar	Width	Height	Total Pixels	Total Execution Time (ms)
sample4.jp	405	226	91530	52.1276
sample2.jp	393	289	113577	56.686
sample18.	328	270	88560	41.9688
sample13.	1242	840	1043280	562.078
sample11.	1280	1280	1638400	878.273
sample16.	1200	675	810000	425.263
sample6.jp	1200	675	810000	421.419
sample9.jp	394	251	98894	47.7472
sample15.	769	711	546759	288.301
sample12.	379	282	106878	51.5736
sample14.	1280	819	1048320	557.167
sample10.	1200	675	810000	423.7
sample17.	435	289	125715	60.3221
sample7.jp	739	589	435271	221.684
sample5.jp	387	270	104490	50.4521
sample3.jp	393	289	113577	54.7305
sample20.	1280	822	1052160	552.165
sample22.	1200	675	810000	421.132
sample8.jp	474	265	125610	60.1483
sample19.	1200	618	741600	386.681

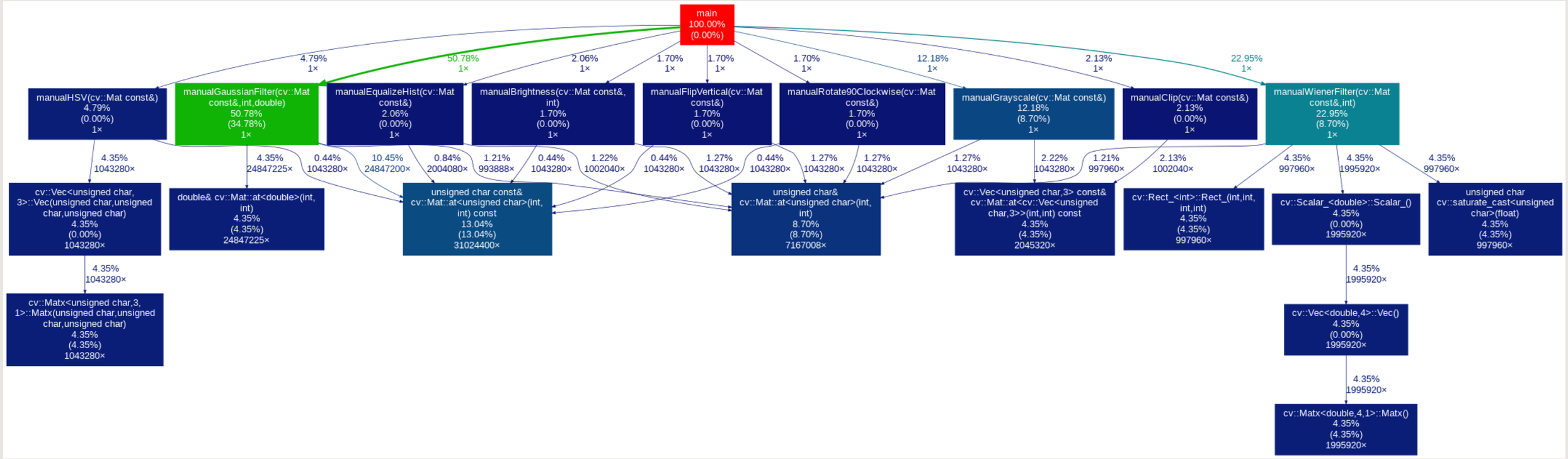


STEP 2 - FINDING HOTSPOTS USING GPROF

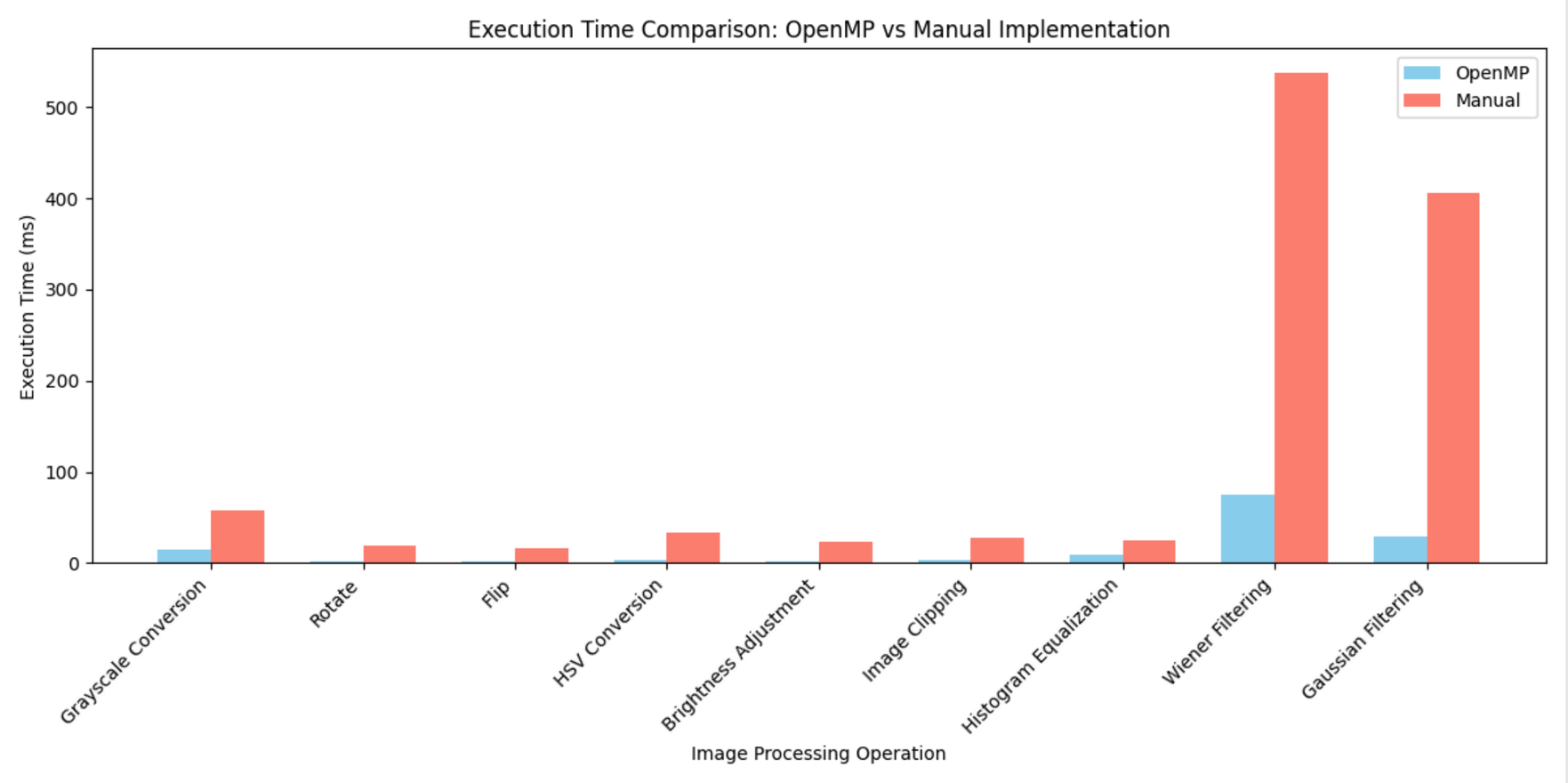
```
Flat profile:

Each sample counts as 0.01 seconds.
  %   cumulative   self              self     total
 time   seconds   seconds    calls  ms/call  ms/call  name
 36.36      0.08     0.08        1    80.00   116.80  manualGaussianFilter(cv::Mat const&, int, double)
 13.64      0.11     0.03 31024400     0.00     0.00  unsigned char const& cv::Mat::at<unsigned char>(int, int) const
  9.09      0.13     0.02  7167008     0.00     0.00  unsigned char& cv::Mat::at<unsigned char>(int, int)
  9.09      0.15     0.02        1    20.00    25.46  manualGrayscale(cv::Mat const&)
  9.09      0.17     0.02        1    20.00    47.78  manualWienerFilter(cv::Mat const&, int)
  4.55      0.18     0.01 24847225     0.00     0.00  double& cv::Mat::at<double>(int, int)
  4.55      0.19     0.01  1995920     0.00     0.00  cv::Matx<double, 4, 1>::Matx()
  4.55      0.20     0.01  1043280     0.00     0.00  cv::Matx<unsigned char, 3, 1>::Matx(unsigned char, unsigned char, unsigned char)
  4.55      0.21     0.01   997960     0.00     0.00  unsigned char cv::saturate_cast<unsigned char>(float)
  2.27      0.21     0.01  2045320     0.00     0.00  cv::Vec<unsigned char, 3> const& cv::Mat::at<cv::Vec<unsigned char, 3> >(int, int) const
  2.27      0.22     0.01   997960     0.00     0.00  cv::Rect_<int>::Rect_(int, int, int, int)
  0.00      0.22     0.00  3991840     0.00     0.00  cv::Vec<double, 4>::operator[](int)
  0.00      0.22     0.00  3129840     0.00     0.00  cv::Vec<unsigned char, 3>::operator[](int)
  0.00      0.22     0.00  2993893     0.00     0.00  cv::_InputArray::init(int, void const*)
  0.00      0.22     0.00  2993891     0.00     0.00  cv::_InputArray::~_InputArray()
  0.00      0.22     0.00  2993891     0.00     0.00  cv::Size_<int>::Size_()
```

STEP 2 - FINDING HOTSPOTS USING GPROF

gprof pixel_imp gmon.out | gprof2dot -w | dot -Tpng -o gprof.png

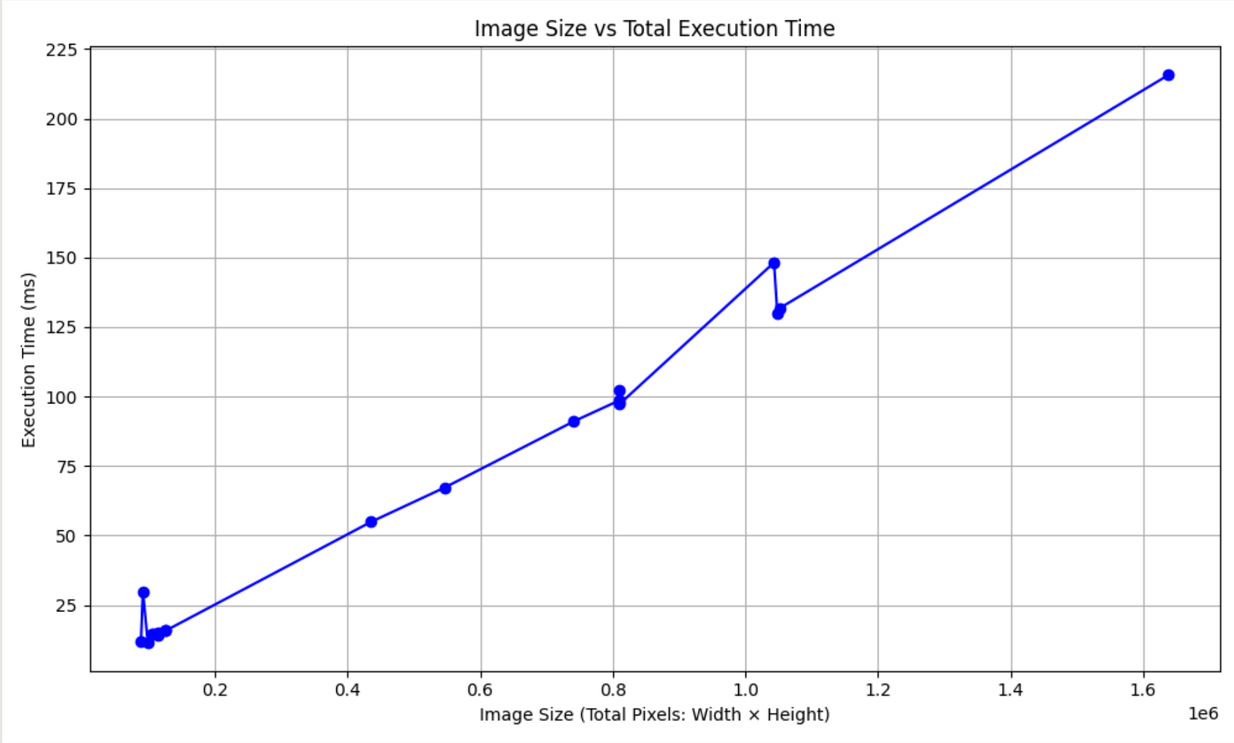


STEP 3 - PARALLELIZATION WITH OPENMP



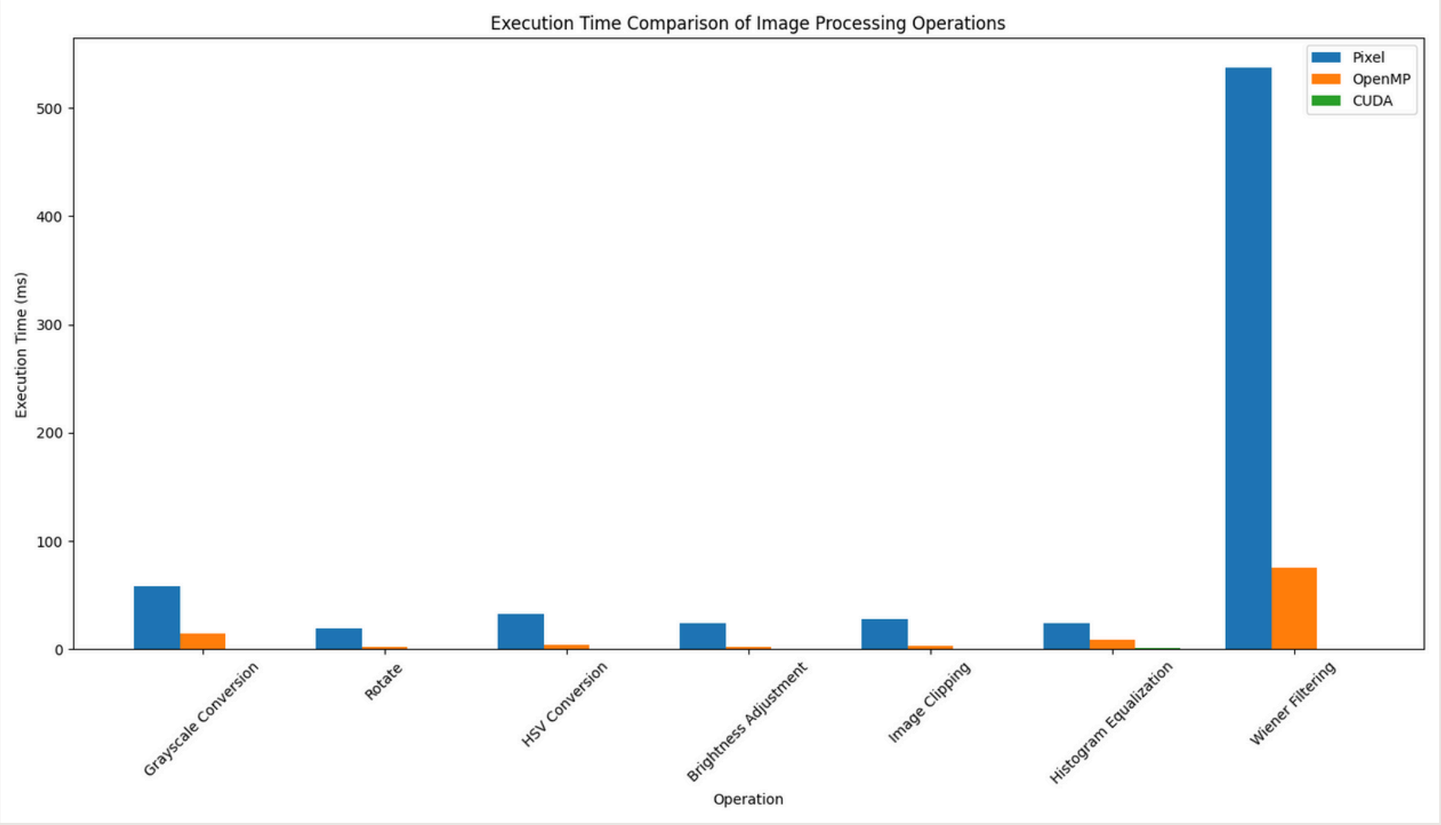
STEP 3 - PARALLELIZATION WITH OPENMP

Image Nar	Width	Height	Total Pixel	Total Execution Time (ms)
sample4.jp	405	226	91530	29.8838
sample2.jp	393	289	113577	15.2119
sample18.	328	270	88560	11.835
sample13.	1242	840	1043280	148.188
sample11.	1280	1280	1638400	215.716
sample16.	1200	675	810000	102.14
sample6.jp	1200	675	810000	97.21
sample9.jp	394	251	98894	11.5439
sample15.	769	711	546759	67.2666
sample12.	379	282	106878	14.391
sample14.	1280	819	1048320	129.954
sample10.	1200	675	810000	98.6047
sample17.	435	289	125715	15.7434
sample7.jp	739	589	435271	54.8689
sample5.jp	387	270	104490	14.5266
sample3.jp	393	289	113577	14.1021
sample20.	1280	822	1052160	131.82
sample22.	1200	675	810000	97.7432
sample8.jp	474	265	125610	15.7273
sample19.	1200	618	741600	91.0933



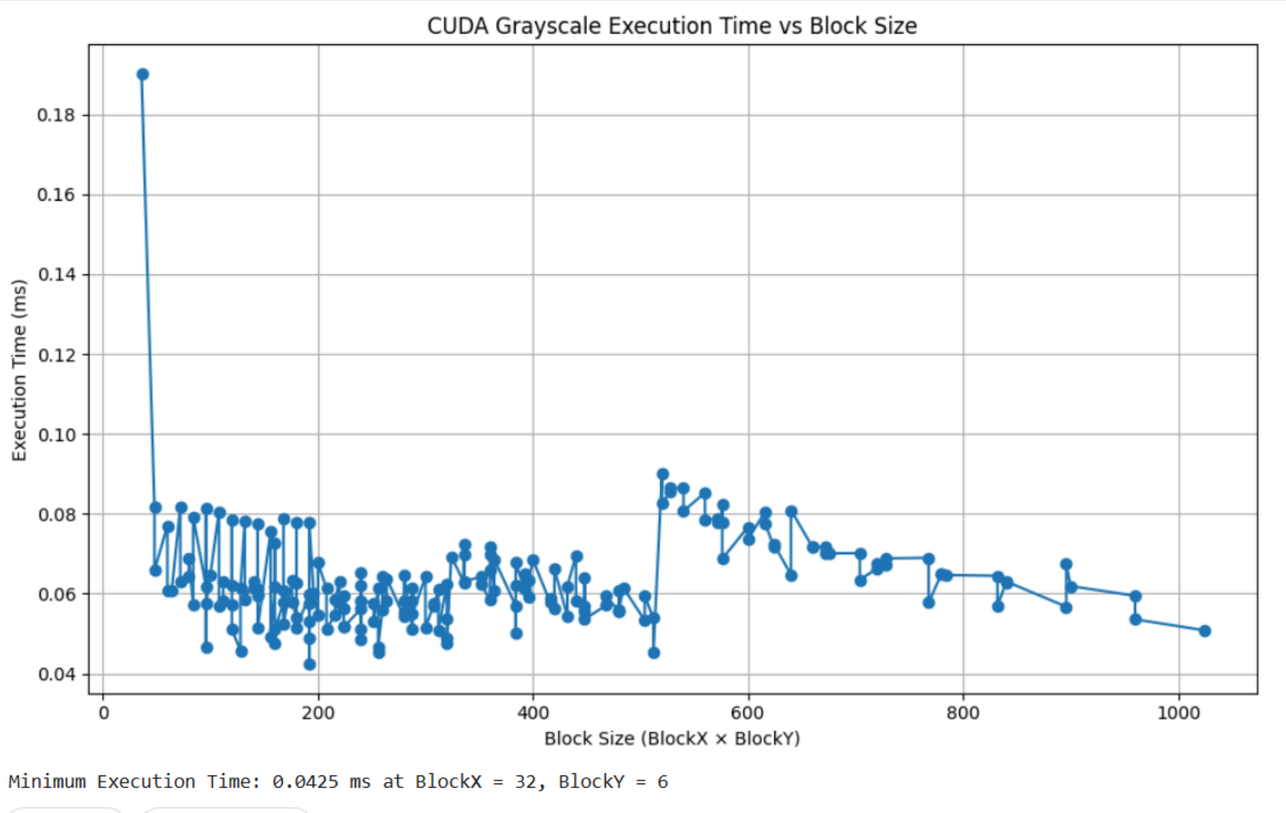
STEP 4 - MASSIVE PARALLELIZATION WITH CUDA

Operation	Execution Time (ms)
Grayscale Conversion	0.422592
Rotate	0.203904
Flip	0.045984
HSV Conversion	0.05552
Brightness Adjustment	0.043008
Image Clipping	0.049056
Histogram Equalization	1.38099
Wiener Filtering	0.26432
Gaussian Filtering	0.268576



STEP 4 - MASSIVE PARALLELIZATION WITH CUDA

How to decide Grid size and Block Size ?




```
dim3 blockDim (32,6);

dim3 grid ((width + blockDim.x - 1) / blockDim.x,
           (height + blockDim.y - 1) / blockDim.y);
```

High Performance Computing

COMPARISION OF FINAL RESULTS

Waiting task	Serial execution time (ms)	Parallel model	Parallel execution time (ms)
Rotate and Flip	246.734	OpenMP	100.258
Convert HSV image	226.063	OpenMP	71.642
Luminance contrast adjustment	384.000	OpenMP	169.004
Image clipping	208.880	OpenMP	96.052
Histogram equalization	619.304	OpenMP	73.301
Winer filtering	150.009	OpenMP	42.738
Gaussian filtering	154.534	OpenMP	79.989
		CUDA	9.234

PAPER RESULTS

	Sequential	OpenMP	CUDA
Operation	Execution Times (ms)	Execution Times (ms)	Execution Times (ms)
Grayscale Conversion	18.4	5.7	0.42259
Rotate	8.11	1.58	0.203904
Flip	7.01	0.85	0.045984
HSV Conversion	12.56	2.1	0.05552
Brightness Adjustment	9.08	1.04	0.043008
Image Clipping	6.09	1.4	0.049056
Histogram Equalization	10.12	7.81	1.38099
Wiener Filtering	338.15	69.56	0.26432
Gaussian Filtering	144.13	28.23	0.268576

OUR RESULTS

METHODOLOGY

- **Approaches:**
  - Compare OpenCV-based API calls vs manual pixel manipulation
  - Introduce OpenMP directives (#pragma omp) to parallelize loops
  - Use CUDA for offloading compute-intensive parts to GPU.
- **Languages & Tools:**
  - C++ for implementation
  - OpenMP for CPU parallelism
  - CUDA (for GPU acceleration)
  - G++ for compilation
  - gprof, gprof2dot + Graphviz for performance analysis
  - Python & Matlab Plot fro visualizations & graph

LIMITATIONS

- One major limitation is the hardware dependency of CUDA-based GPU acceleration. Although CUDA delivers substantial speedup, it is limited to NVIDIA GPUs, making it unsuitable for systems with AMD or integrated graphics.

- The OpenMP parallelization achieved moderate improvements but is constrained by the number of CPU cores and cache architecture. Also, memory bandwidth becomes a bottleneck for large image sizes, which diminishes the scalability benefits of multithreading. Another limitation is the use of static parameters in brightness, contrast, and filtering operations. These values are fixed for testing purposes and may not adapt well to real-world dynamic conditions or diverse SAR datasets.

FUTURE WORK

- Adopting cross-platform parallel frameworks like OpenCL to support a wider range of GPUs and CPUs.
- Dynamic parameter tuning using adaptive algorithms or machine learning to optimize image enhancement based on content.
- Integration with real-time SAR data streams from satellites or drones to evaluate end-to-end latency and throughput.
- Extending to deep learning-based SAR analysis, where HPC can accelerate both training and inference phases.
- Memory optimization techniques such as shared memory usage and pipelined execution in CUDA for even better performance.

OBSERVATIONS FROM THE STUDY

- The study was designed to systematically evaluate the impact of different computational strategies—sequential processing, OpenCV functions, OpenMP-based CPU parallelization, and CUDA-based GPU acceleration—on the efficiency of SAR image processing. By implementing identical image enhancement tasks across these four methods, the study directly addressed the research question: “How can parallel computing improve the performance of SAR image processing workflows?”

- The results clearly demonstrated that parallelization significantly reduces processing time, with CUDA delivering the most substantial performance gains. OpenMP also offered notable improvements over sequential methods, especially in multi-core environments. These observations validate the hypothesis that High Performance Computing (HPC) can make SAR processing feasible for real-time applications.

- However, certain aspects remain unanswered. While the study highlights execution speed improvements, it does not explore energy efficiency, memory utilization, or accuracy trade-offs in detail. Additionally, real-world SAR datasets and dynamic conditions were not fully integrated, limiting generalizability. The study opens the door for future exploration into adaptive, intelligent processing pipelines and broader hardware support.

THANK YOU