HPC assigment presentation

Carlo Mattioli: s349351
Fabio Calabrese: s343467
MIchele Merla: s343500

Master degree : Quantum Engineering

July 2025, Torino, Italy

Politecnico
di Torino

Systolic array architecture with MPI implementation

General overview:

1. Introduction
2. Methodology
3. Metrics
4. Results analysis
5. Conclusions

Introduction

- Design the **multiplication** between two **matrices** with the systolic array architecture
- Implement **MPI** to obtain structures
- Performance evaluation
- Results discussion

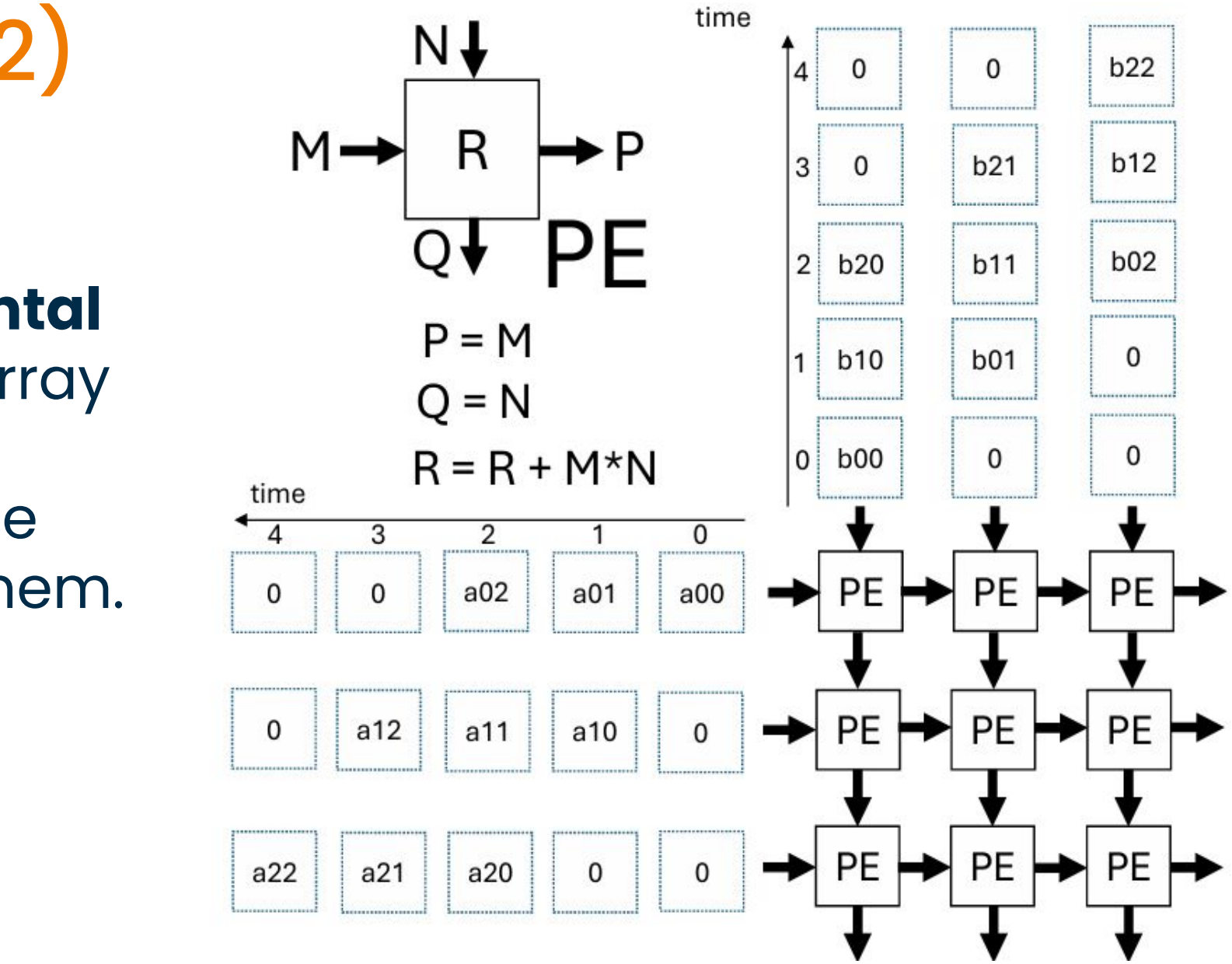
Methodology (1)

Software implementation:

1. Matrices division between **PEs**
2. **Cannon's algorithm** initialization for correct distribution of the blocks among the PEs
3. Cannon's algorithm computation
4. Gather of all PEs result in output matrix

Methodology (2)

PEs are the **fundamental units** of the systolic array architecture. They compute and pass the information among them.



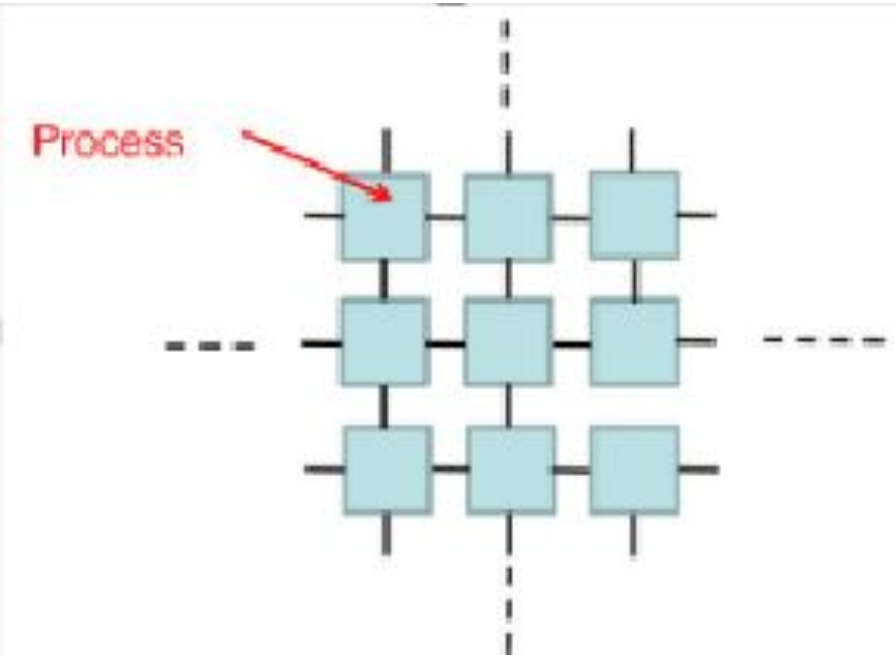
Methodology (3)

Each PE is associated to one **process**

Size of PEs given by **n = N/p**

The processes are distributed in a **2D periodic grid**.

The communication between processes is implemented through MPI.



Methodology (4)

The input matrices are divided between PEs by:

```
MPI_Scatterv(A, counts, displs, blocktype2, A_block, block_size * block_size, MPI_DOUBLE, 0, MPI_COMM_WORLD);
MPI_Scatterv(B, counts, displs, blocktype2, B_block, block_size * block_size, MPI_DOUBLE, 0, MPI_COMM_WORLD);
```

The PEs are **initialized** as requested by the Cannon's algorithm.

After the main computation, the results are **gathered** in the output matrix by:

```
MPI_Gatherv(C_block, block_size * block_size, MPI_DOUBLE, C, counts, displs, blocktype2, 0, MPI_COMM_WORLD);
```

Methodology (5)

Each **iteration** the processes:

1. Compute the local multiplication
2. Send their block to the **neighbors**

```
if ((my_col+my_row) % 2 == 0) {
    MPI_Send(A_block, block_size*block_size, MPI_DOUBLE, left, 0, grid_comm);
    MPI_Recv(tmp_A, block_size*block_size, MPI_DOUBLE, right, 0, grid_comm, &statusA);
    }
else {
    MPI_Recv(tmp_A, block_size*block_size, MPI_DOUBLE, right, 0, grid_comm, &statusA);
    MPI_Send(A_block, block_size*block_size, MPI_DOUBLE, left, 0, grid_comm);
    }
memcpy(A_block, tmp_A, block_size*block_size*sizeof(double));
```

Metrics

- The code was executed in one or two nodes of the **Legion Cluster**

- The number of processes was varied according to the matrix and PEs division criteria

- The time was measured using ***MPI_Wtime()***

Results analysis (1)

For the **500 x 500** matrix:

Sequential time = **3.40 s**

- **1 node**:

Processes	Time (s)	Speed-up
4	0.28	12.14
16	0.15	22.67
25	0.15	22.67

- **2 nodes:**

Processes	Time (s)	Speed-up
4	0.31	10.97
16	0.45	7.56
25	0.63	5.40

Results analysis (2)

For the **1000 x 1000** matrix:

Sequential time = **4.88 s**

- **1 node:**

Processes	Time (s)	Speed-up
4	2.19	2.23
16	0.67	7.28
25	0.65	7.51

- **2 nodes:**

Processes	Time (s)	Speed-up
4	2.10	2.32
16	1.08	4.52
25	1.36	3.59
64	2.33	2.09

Results analysis (3)

For the **2000 x 2000** matrix:

Sequential time = **40.38 s**

- **1 node:**

Processes	Time (s)	Speed-up
4	15.95	2.53
16	3.83	10.54
25	3.42	11.81

- **2 nodes:**

Processes	Time (s)	Speed-up
4	15.35	2.63
16	4.58	8.82
25	4.10	9.85
64	4.24	9.52

Conclusions

- Greater number of processes is not always the best solution

- There is a discrepancy between **single-node** and **multi-node** implementation with the same number of processes

- **Best speed-up** are in the single-mode implementation

Heat Diffusion in a Metal Plate

General Overview

1. Introduction
2. Methodology
3. Data Collection Procedure
4. Results
5. Conclusions

Introduction

- Simulate **heat diffusion** in a metal plate in the case of an ***isotropic*** and an ***anisotropic*** material
- Exploit **openMP** to accelerate performances
- **Performance evaluation**
- Brief **comparison** between **isotropic** and **anisotropic** implementation

Methodology

It applies to both isotropic and anisotropic cases:

1. **Grid initialization** - static schedule
2. **Stencil update** - dynamic/guided schedule, reduction
3. **Copy of updated values** - static schedule

- Isotropic case threshold: **0.01 °C**
- Anisotropic case threshold: **0.015 °C**
- Max iterations: **10’000**

Data Collection Procedure

- Code was runned over one node of the **Legion Cluste**r
- The number of threads was varied from **4** to **128**
- **Execution time** is measured using **omp_get_wtime()**

Results

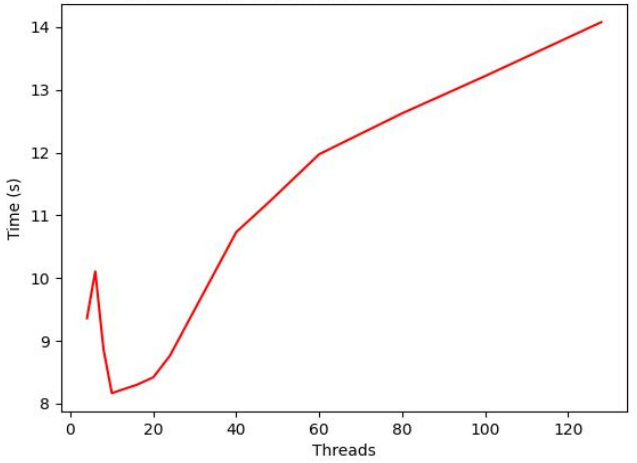
Isotropic case

Sequential time: **32.108 s**

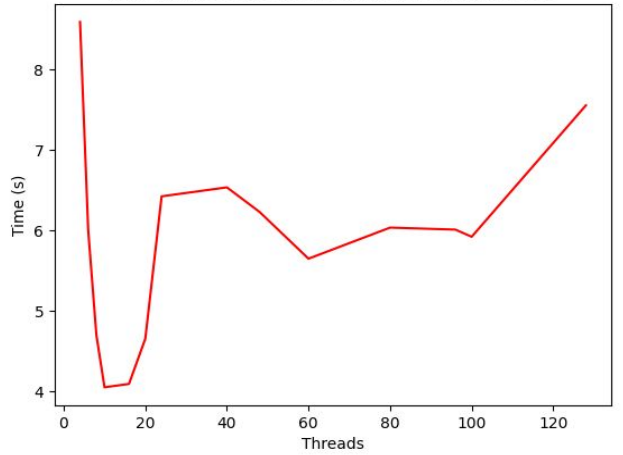
Speed-up:

Cores	Best #Threads	Time [s]	Speed-up
4	10	8.17	3.93
12	10	4.05	7.93
24	24	1.96	16.40

Performance execution of threads with 4 cores



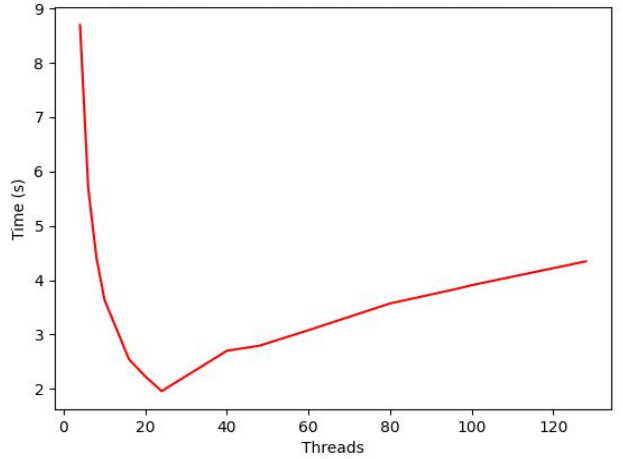
Performance execution of threads with 12 cores

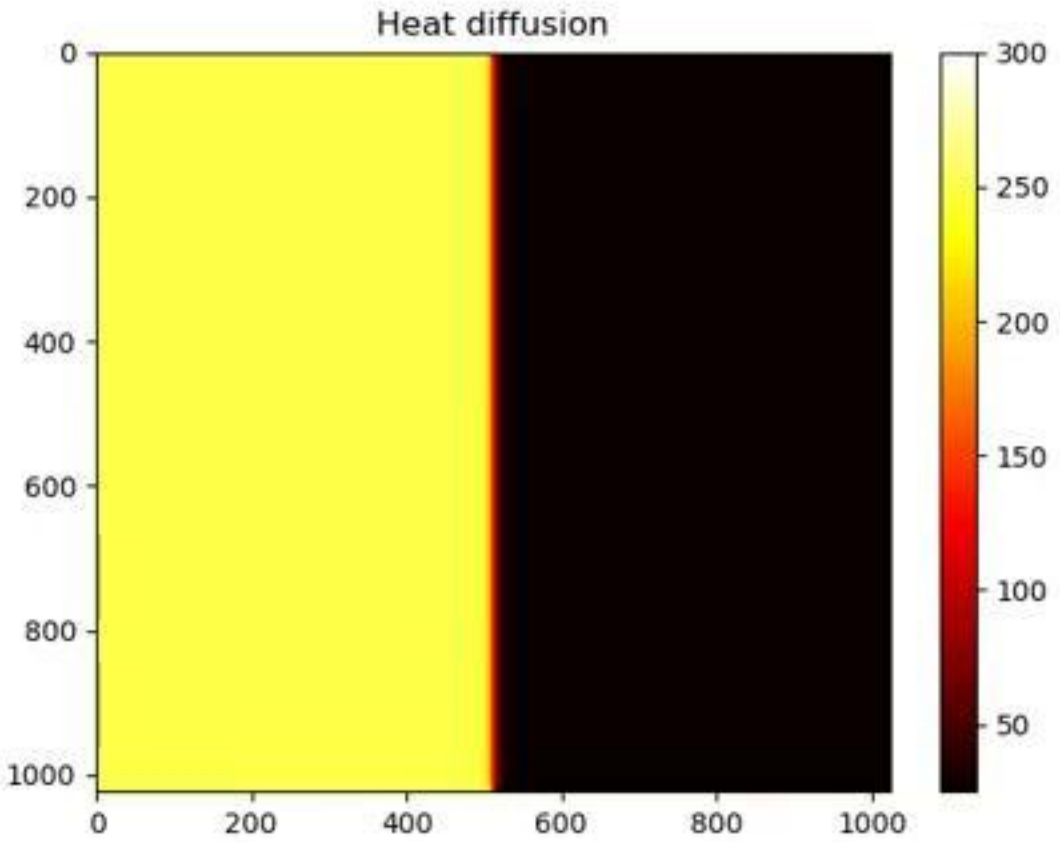


Execution times

Threads	4 cores	12 cores	24 cores
4	9.36	8.59	8.70
6	10.11	6.00	5.70
8	8.87	4.70	4.42
10	8.17	4.05	3.64
16	8.30	4.09	2.54
20	8.42	4.66	2.23
24	8.76	6.43	1.96
40	10.73	6.54	2.70
48	11.22	6.23	2.79
60	11.98	5.65	3.08
80	12.63	6.04	3.57
96	13.10	6.01	3.83
100	13.22	5.92	3.91
128	14.08	7.56	4.35

Performance execution of threads with 24 cores




Heat diffusion
0
200
400
600
800
1000
0
200
400
600
800
1000
300
250
200
150
100
50

Results

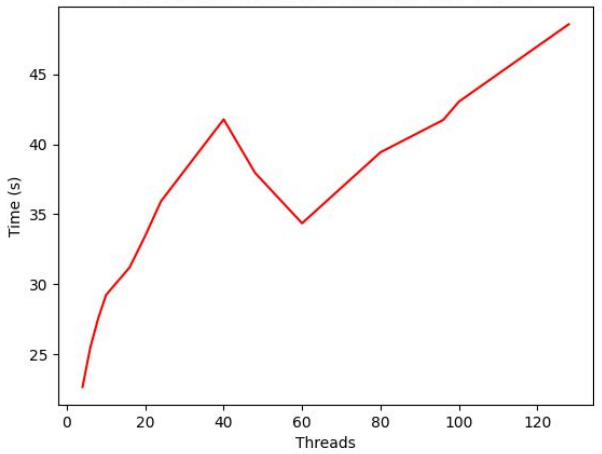
Anisotropic case

Sequential time: **101.48 s**

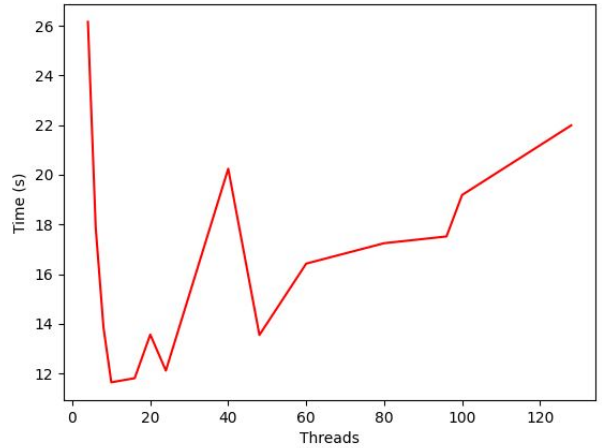
Speed-up:

Cores	Best #Threads	Time [s]	Speed-up
4	4	22.66	4.48
12	10	11.64	8.71
24	24	4.51	22.49

Performance execution of threads with 4 cores



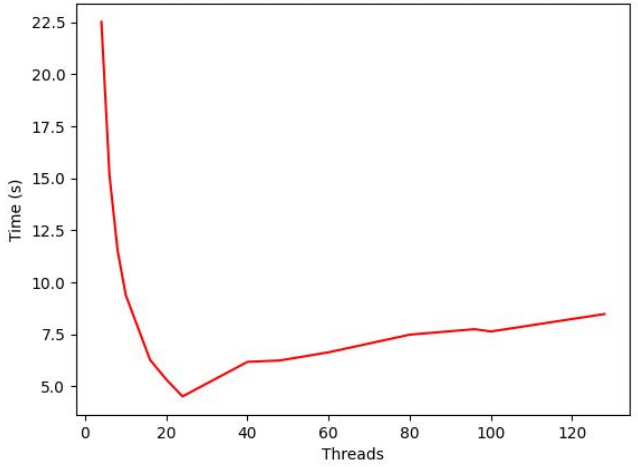
Performance execution of threads with 12 cores

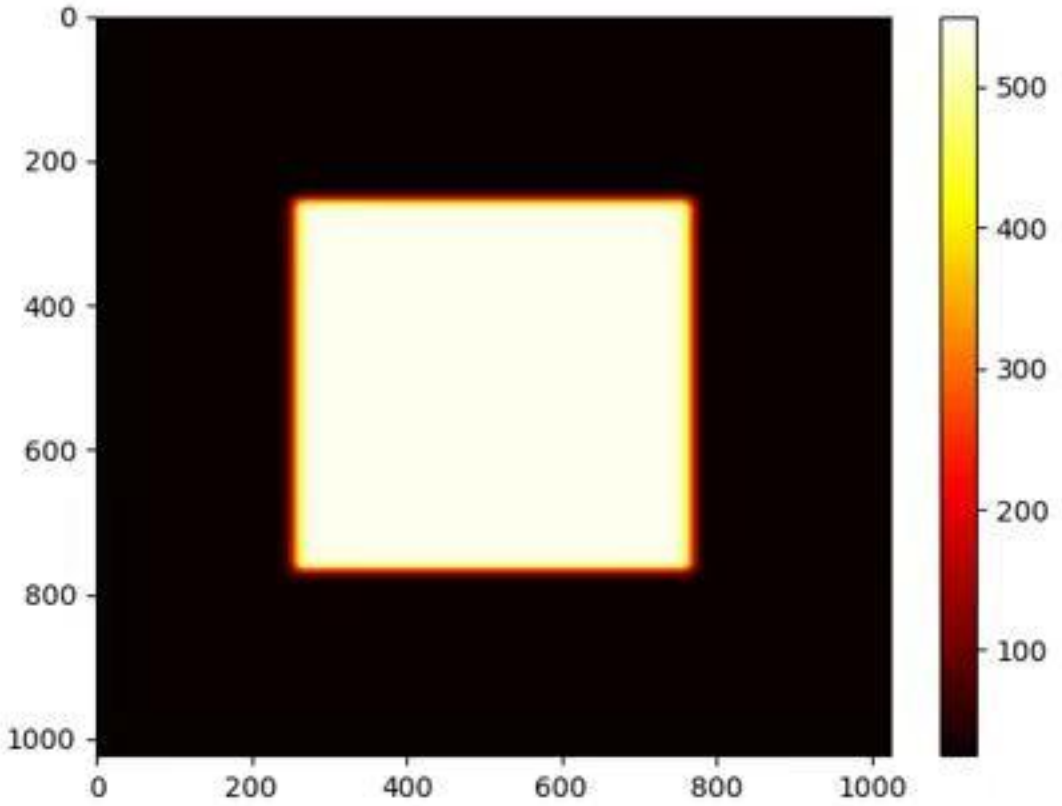


Execution times

Threads	4 cores	12 cores	24 cores
4	22.66	26.16	22.52
6	25.50	17.87	15.13
8	27.56	13.85	11.53
10	29.23	11.64	9.39
16	31.20	11.81	6.27
20	33.47	13.56	5.33
24	35.93	12.11	4.51
40	41.79	20.25	6.17
48	37.97	13.55	6.24
60	34.36	16.42	6.63
80	39.44	17.24	7.48
96	41.75	17.52	7.75
100	43.06	19.19	7.63
128	48.59	21.99	8.47

Performance execution of threads with 24 cores




0
200
400
600
800
1000
0
200
400
600
800
1000
500
400
300
200
100

Isotropic vs Anisotropic

- Different initial conditions and different conductivities
- Sequential times (101.48 vs 32.108)
- Anisotropic case benefits more from parallelism
- In both cases the optimal number of threads is the one close to the number of physical cores

Conclusions

- Both implementations benefits from the parallelisation
- Performance improves as more threads are used up to the number of physical cores
  - Why?
- Limitations
- Possible improvement

CUDA Image filtering

General Overview

1. Introduction
2. Methodology
3. Metrics
4. Results
5. Conclusions

Introduction

- Apply and analyze a filter on images **with and without noise**
- **Compare** sequential and parallel implementations
- Evaluate performance through:

  1. Speedup
  2. Throughput
  3. Time vs threads

Methodology(1)

Image Processing

- Input images loaded in **RGB** using **OpenCV**
- Channels **split** into Red, Green, Blue (3× 2D arrays)
- Each channel **filtered independently**
- Final output is a **merged RGB image**

CUDA Kernel

- Applies a **3×3 weighted convolution** (excluding borders) to each pixel.

Performance Measurement

- **Parallel execution time** is the **sum of the three kernel times** (R, G, B)

Methodology(2)

sequential implementation:

```
for i in [1, height - 2]:
    row_ref = i × step
    for j in [1, width - 2]:
        sum = 0
        for k in [-1, 0, 1]:
            for l in [-1, 0, 1]:
                col_ref = j + l
                sum += weight[k+1][l+1] * input[row_ref + k × step + col_ref]
        output[i × step + j] = clamp(sum / normalization_factor, 0, 255)
```

parallel implementation:

```
int col = blockIdx.x * blockDim.x + threadIdx.x;
int row = blockIdx.y * blockDim.y + threadIdx.y;
```

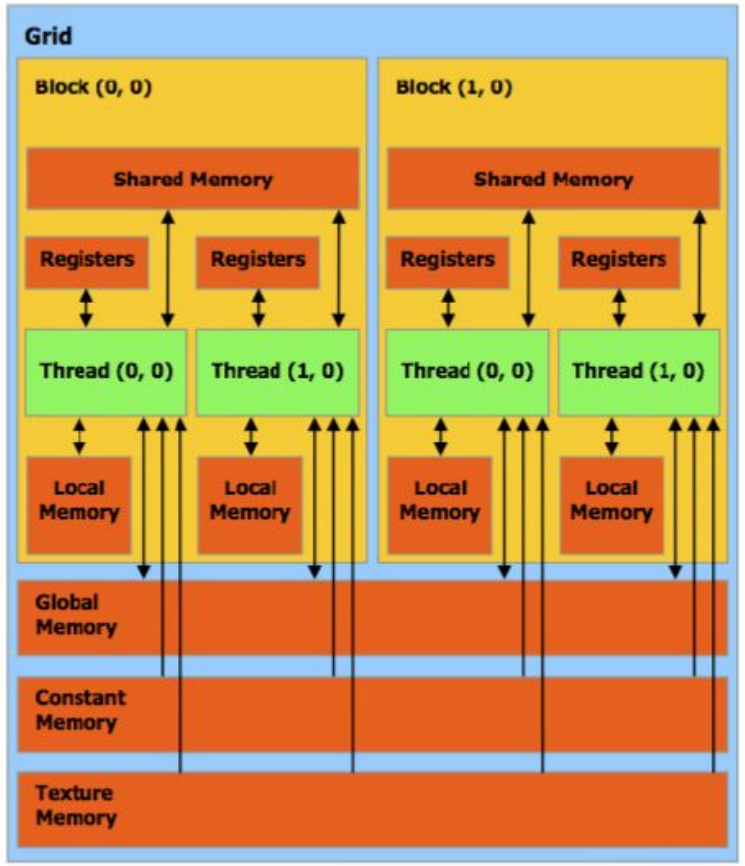
```
for each thread (row, col) in the image
    (excluding borders):
    sum = 0
    for i in [-1, 0, 1]:
        for j in [-1, 0, 1]:
            sum += weight[i+1][j+1] *
                input[row + i, col + j]
    output[row, col] = clamp(sum /
        normalization_factor, 0, 255)
```

Methodology(3)

Block size:

- 16x16
- 32x8
- 32x32

```
dim3 block(16, 16);
dim3 grid((COLS + 15) / 16, (ROWS + 15) / 16);
int totalThreads = grid.x * grid.y * block.x * block.y;
```



Result(preliminary)

Original:

Filtered:

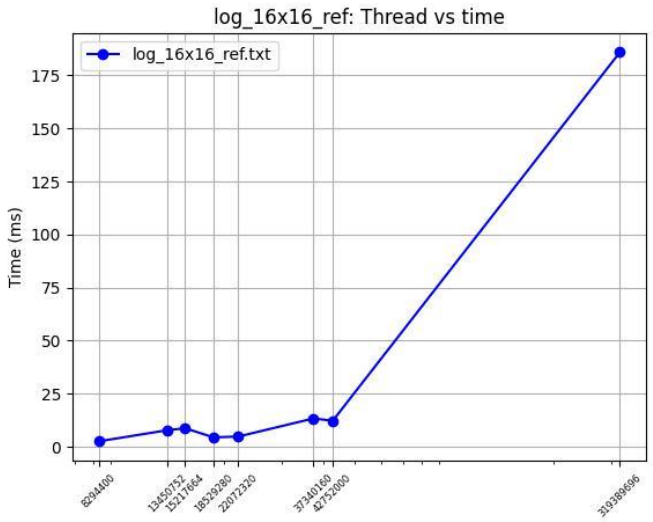
```
output[row, col] = clamp(sum /
    normalization_factor, 0, 255)
```

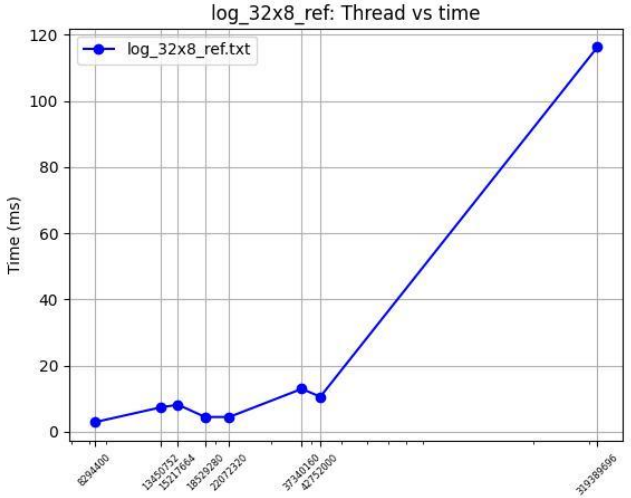
Result(1)

Original:

Filtered:

Result(2)





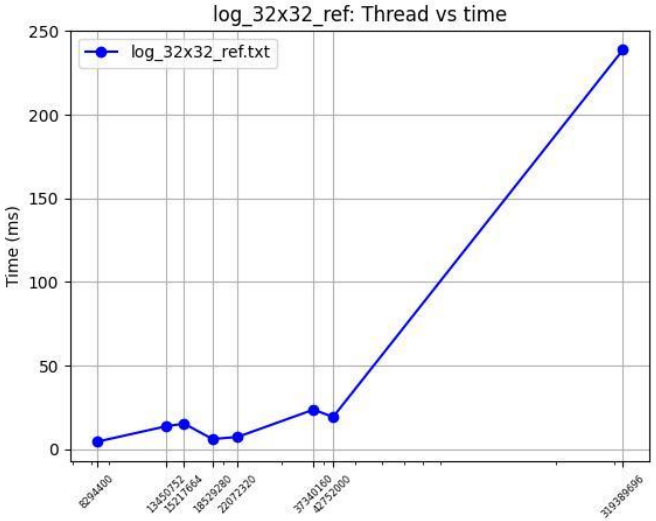


Image	Dimensions	16×16		32×8		32×32	
		S	T	S	T	S	T
climate_change.jpg	4435×3021	198.47	1.71	210.81	1.82	113.21	0.98
forests_cloudy_sky.jpg	3507×5264	480.98	4.15	482.06	4.15	353.04	3.05
harbor_bay_sunset.jpg	4897×3083	200.91	1.73	215.42	1.86	114.81	1.00
heart_shaped_island.jpg	5333×7999	403.91	3.48	468.37	4.04	257.13	2.22
Lake_forest.jpg	5739×3826	526.86	4.56	570.47	4.94	344.13	2.98
pexels_christian_heitz.jpg	21888×14592	200.10	1.72	320.06	2.75	155.76	1.34
Planet_earth.jpg	8154×4570	323.86	2.79	334.10	2.88	182.62	1.57
Winding_road.jpg	3840×2160	373.54	3.23	334.49	2.89	214.77	1.87

$$S = \frac{T_{sequential}}{T_{parallel}}$$

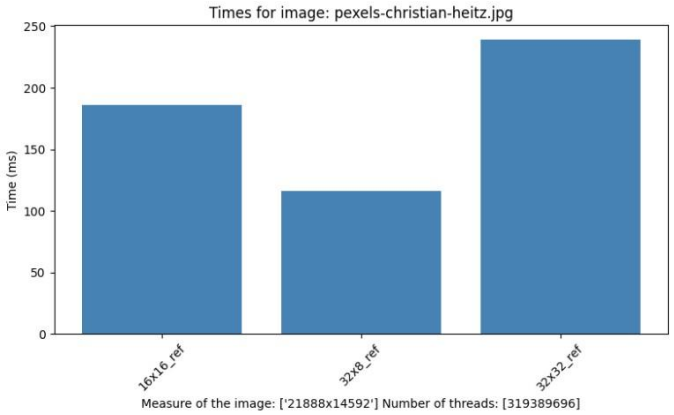
$$TP = \frac{\text{Image Size}}{\text{Execution Time}}$$

Result(3)

pexels christian heitz.jpg

pexels christian heitz.jpg(filtered)

pexels christian heitz.jpg(filtered, low resolution)



- 32x32 block shows the worst performance
- overhead may lead to diminished parallelism

Result(4)

Noise 50%

Image	Dimensions	16×16		32×8		32×32	
		S	T	S	T	S	T
climate_change.jpg	4435×3021	255.63	2.16	213.42	1.81	115.80	0.98
forests_cloudy_sky.jpg	3507×5264	338.58	2.92	355.96	3.07	202.26	1.75
harbor_bay_sunset.jpg	4897×3083	365.15	3.15	390.24	3.37	227.39	1.98
heart_shaped_island.jpg	5333×7999	270.27	2.31	234.19	2.00	119.50	1.02
Lake_forest.jpg	5739×3826	214.23	1.85	286.67	2.47	124.09	1.07
Planet_earth.jpg	8154×4570	216.10	1.86	231.01	1.98	118.72	1.02
Winding_road.jpg	3840×2160	438.36	3.79	446.51	3.86	322.15	2.80

Noise 90%

Image	Dimensions	16×16		32×8		32×32	
		S	T	S	T	S	T
climate_change.jpg	4435×3021	256.04	2.17	214.29	1.81	148.32	1.2
forests_cloudy_sky.jpg	3507×5264	477.78	4.12	477.78	4.12	350.73	3.0
harbor_bay_sunset.jpg	4897×3083	478.26	4.12	214.90	1.86	343.08	2.9
heart_shaped_island.jpg	5333×7999	210.19	1.80	233.72	2.00	119.24	1.0
Lake_forest.jpg	5739×3826	349.73	3.01	335.08	2.89	183.12	1.5
Planet_earth.jpg	8154×4570	216.20	1.86	231.01	1.98	118.69	1.0
Winding_road.jpg	3840×2160	444.95	3.80	451.16	3.86	324.41	2.7

Noise 75%

Image	Dimensions	16×16		32×8		32×32	
		S	T	S	T	S	T
climate_change.jpg	4435×3021	256.87	2.17	214.00	1.81	116.23	0.99
forests_cloudy_sky.jpg	3507×5264	478.84	4.13	476.72	4.11	351.88	3.04
harbor_bay_sunset.jpg	4897×3083	201.37	1.74	215.42	1.86	115.03	1.00
heart_shaped_island.jpg	5333×7999	270.12	2.31	294.64	2.52	152.95	1.31
Lake_forest.jpg	5739×3826	214.05	1.84	286.03	2.47	148.75	1.29
Planet_earth.jpg	8154×4570	275.73	2.37	289.72	2.49	151.91	1.31
Winding_road.jpg	3840×2160	184.41	1.58	306.96	2.62	191.32	1.65

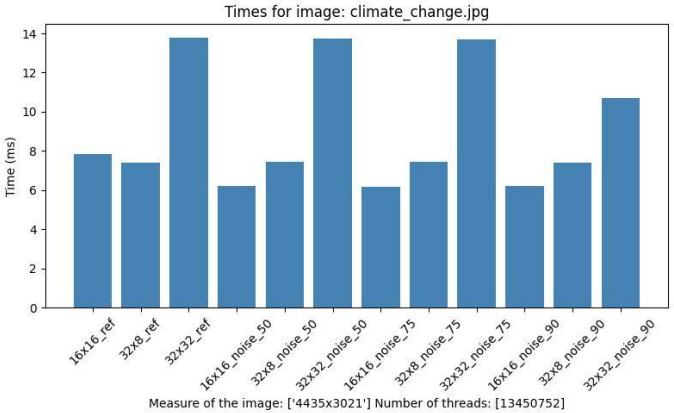
$$S = \frac{T_{sequential}}{T_{parallel}}$$

$$TP = \frac{\text{Image Size}}{\text{Execution Time}}$$

Result(5)

climate_change.jpg(noise = 90%)

climate_change_filtered.jpg(noise = 90%)

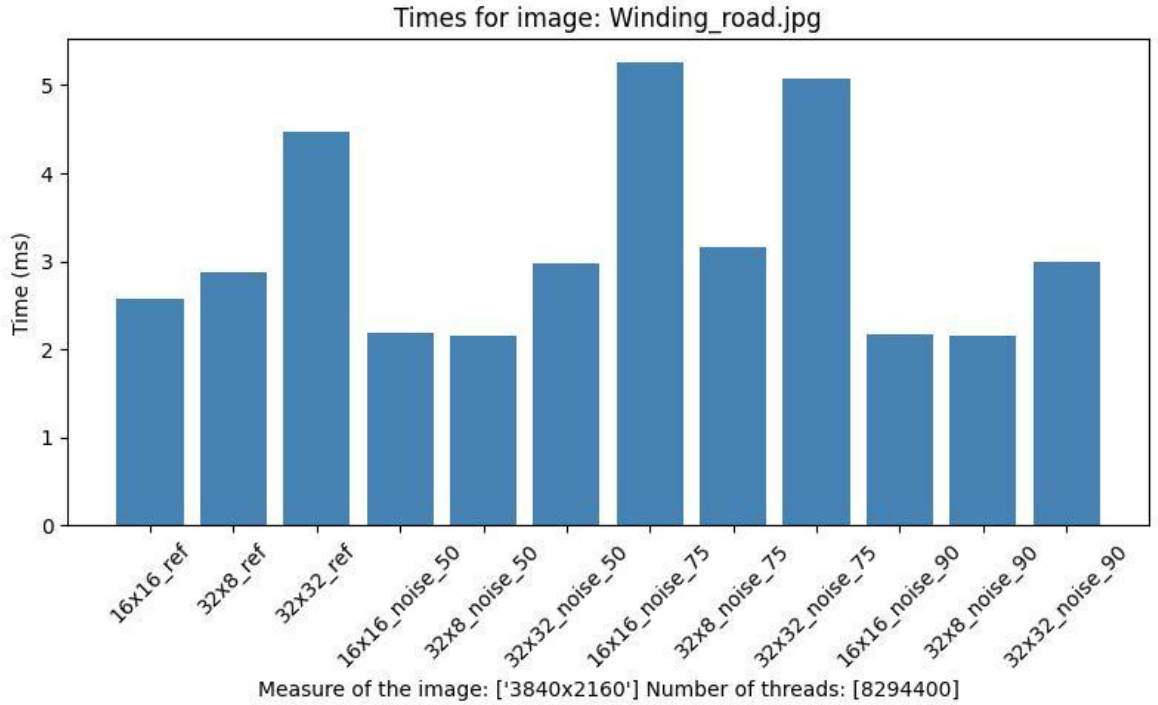


The images constitute a zoom in specific regions.The complete set(original and filtered at 0% 50% 75% 90%) is available on repository

Result(6)

Winding_road.jpg(noise = 50%)

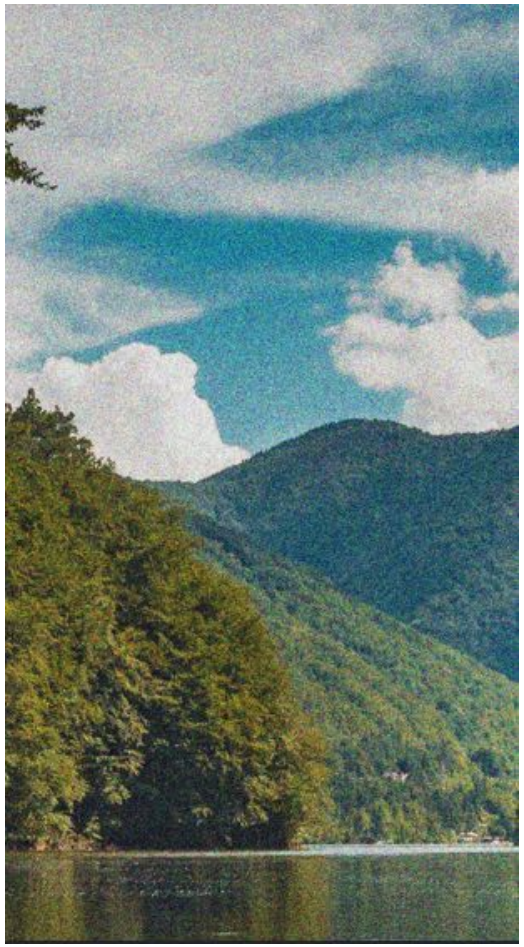
Winding_road.jpg(noise = 50%)



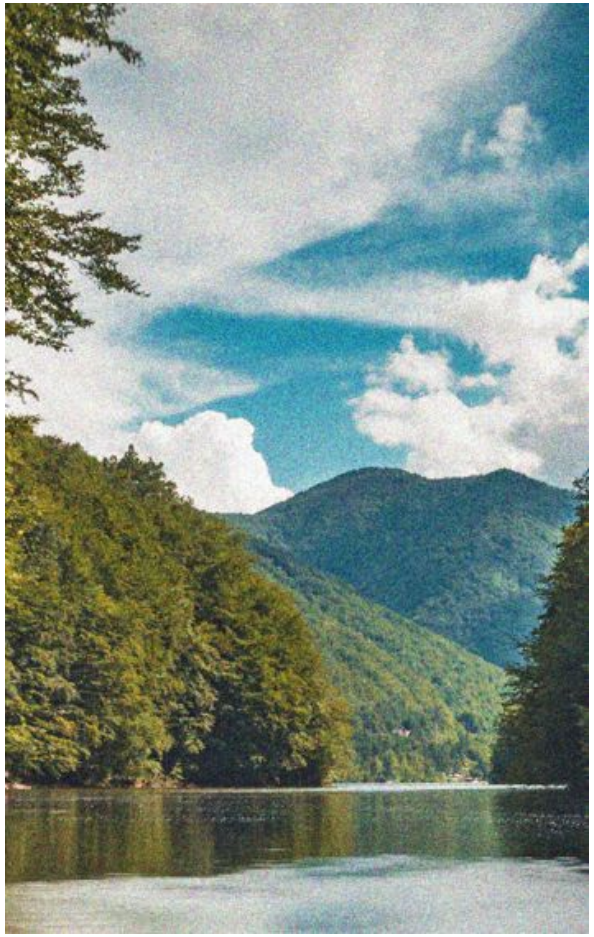
The images constitute a zoom in specific regions.The complete set(original and filtered at 0% 50% 75% 90%) is available on repository

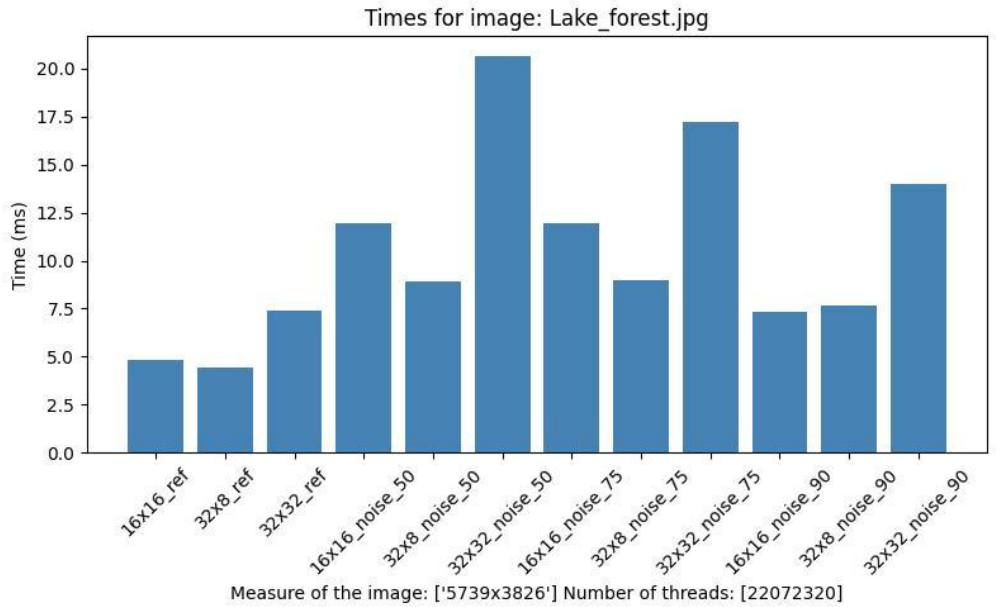
Result(7)

Lake_forest.jpg(noise = 90%)



Lake_forest_filtered.jpg(noise = 90%)



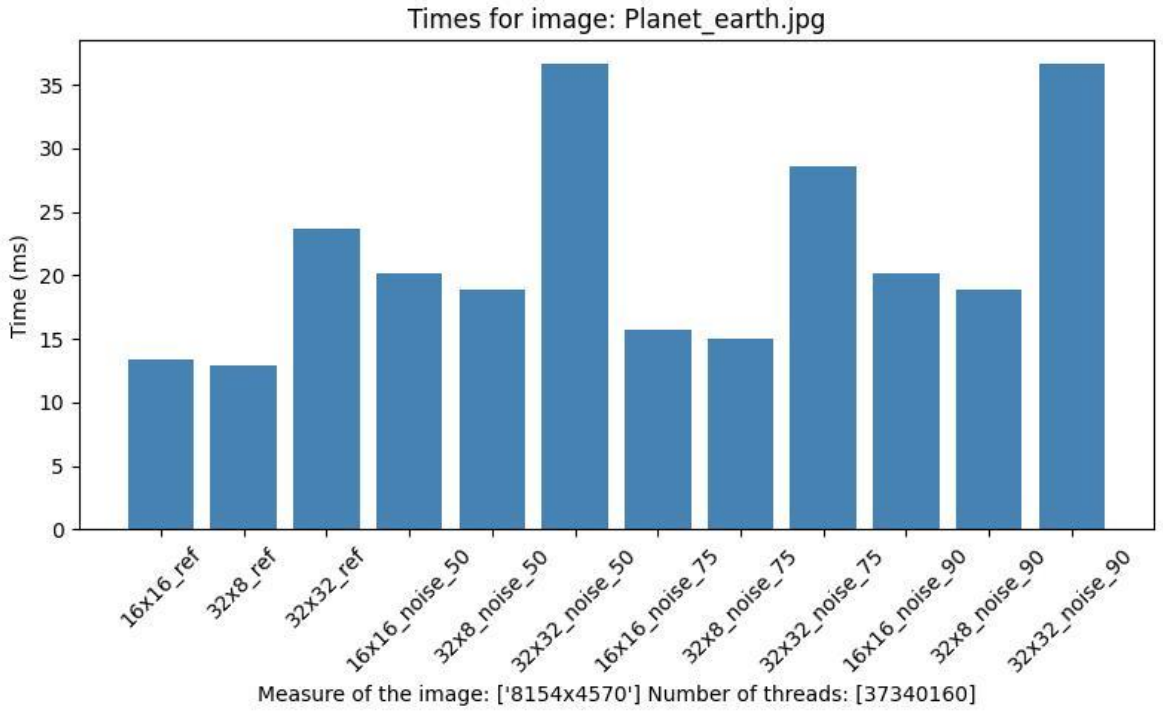


The images constitute a zoom in specific regions.The complete set(original and filtered at 0% 50% 75% 90%) is available on repository

Result(8)

Planet_earth.jpg(noise = 75%)

Planet_earth_filtered.jpg(noise = 75%)

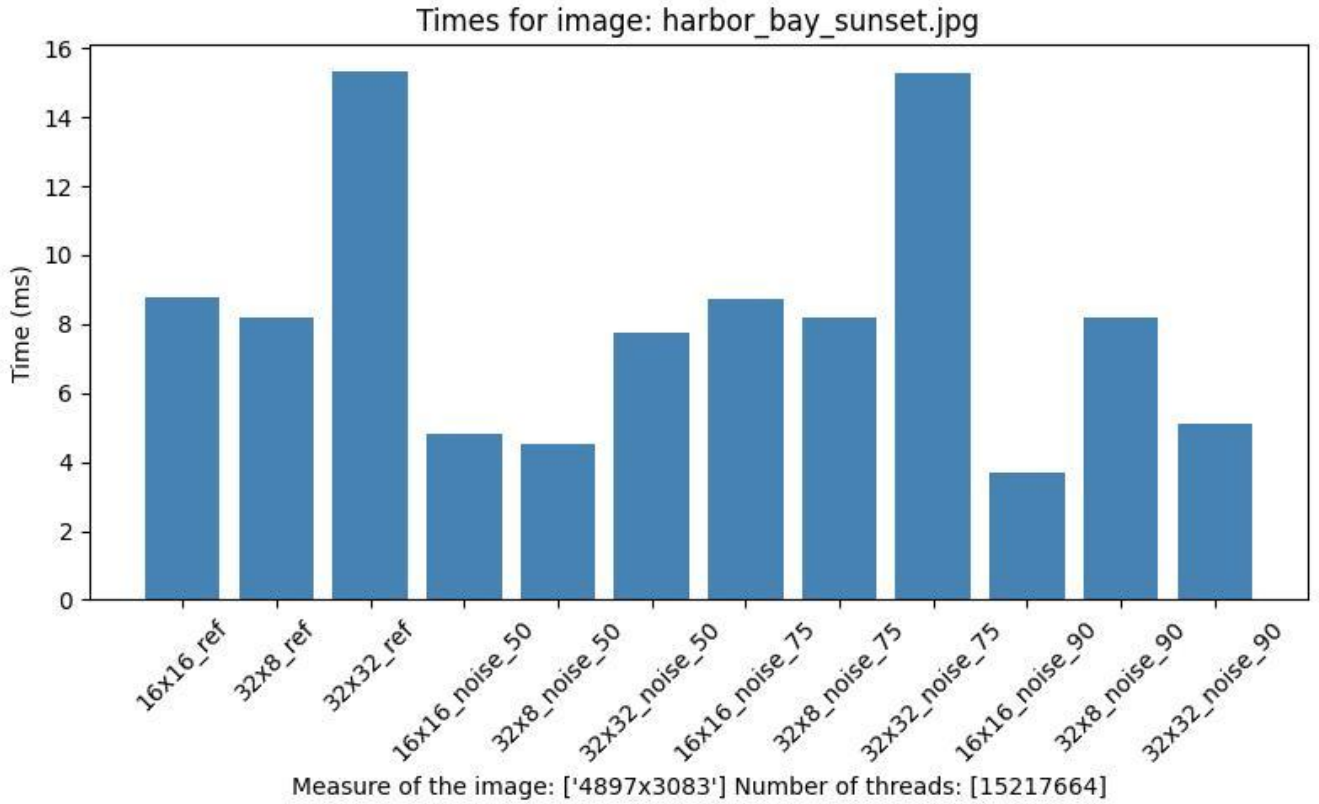


The images constitute a zoom in specific regions.The complete set(original and filtered at 0% 50% 75% 90%) is available on repository

Result(9)

harbor_bay_sunset.jpg(noise = 50%)

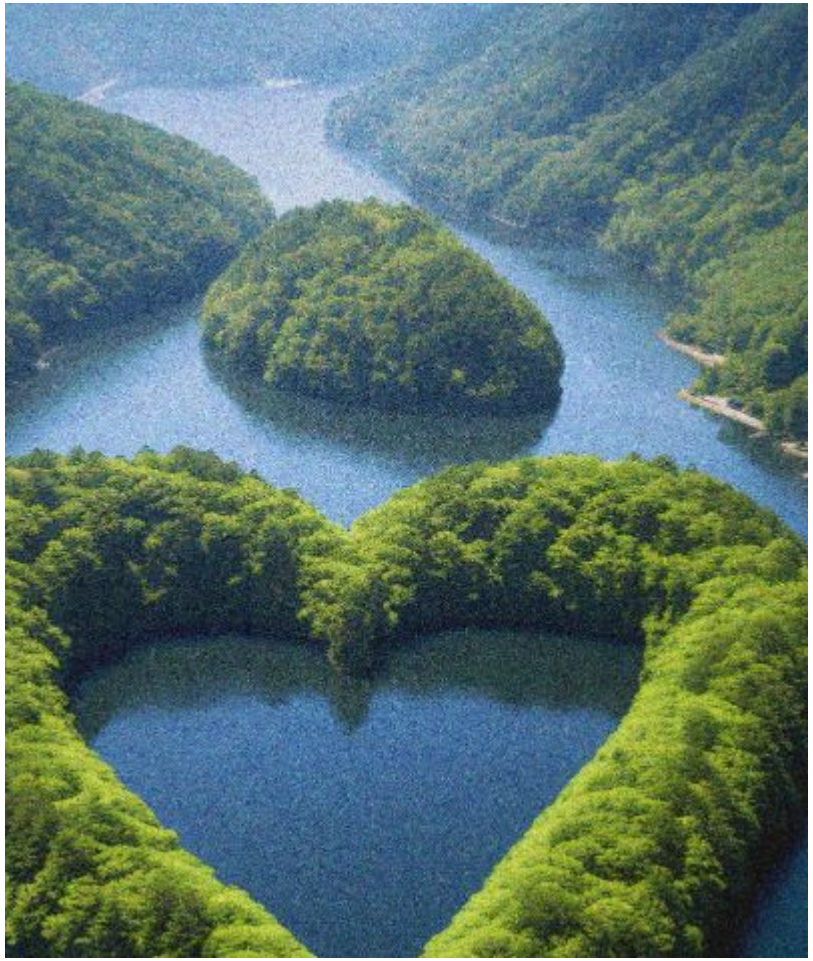
harbor_bay_sunset_filtered.jpg(noise = 50%)



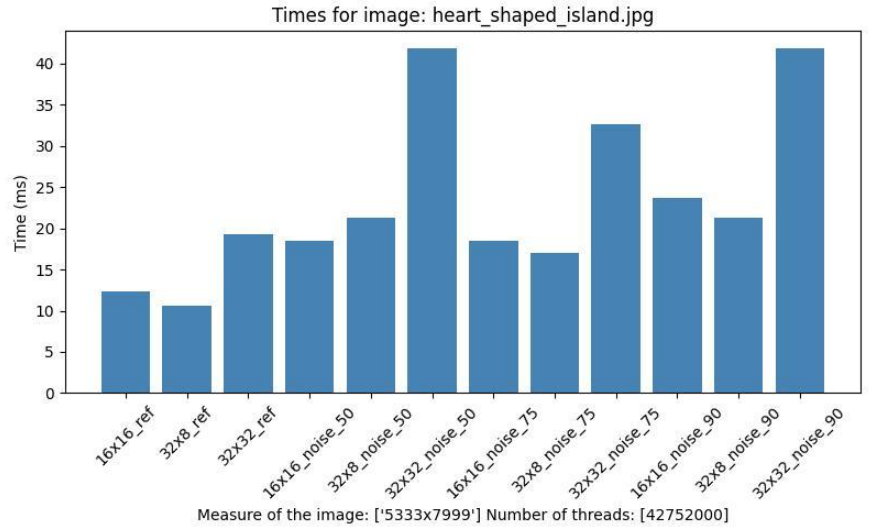
The images constitute a zoom in specific regions.The complete set(original and filtered at 0% 50% 75% 90%) is available on repository

Result(10)

Heart_shaped_island.jpg(noise = 50%)



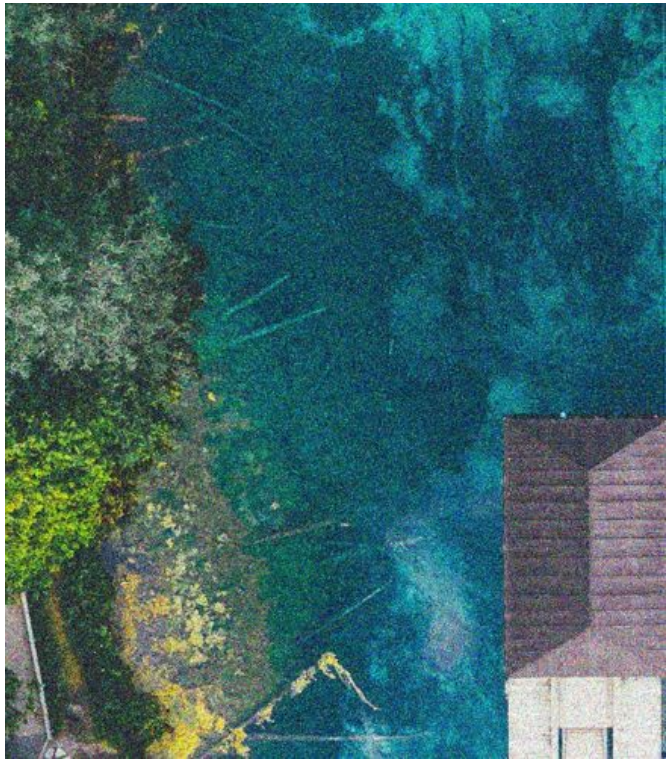
Heart_shaped_island_filtered.jpg(noise = 50%)



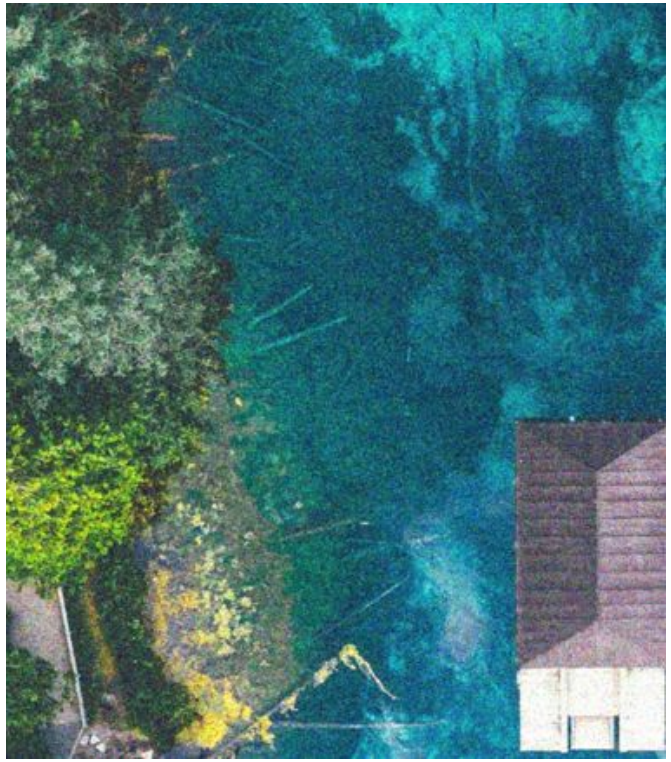
The images constitute a zoom in specific regions.The complete set(original and filtered at 0% 50% 75% 90%) is available on repository

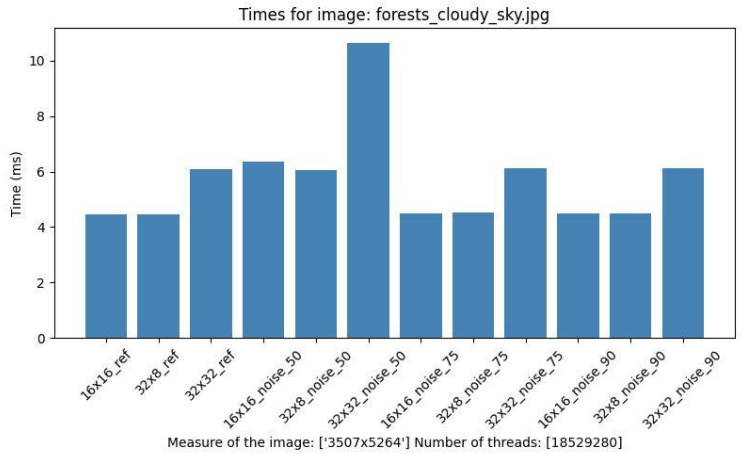
Result(11)

forest_cloudy_sky.jpg(noise = 50%)



forest_cloudy_sky_filtered.jpg(noise = 50%)





The images constitute a zoom in specific regions.The complete set(original and filtered at 0% 50% 75% 90%) is available on repository

Conclusions

- The filter works correctly on both clean and noisy images
- The 16×16 and 32×8 block configurations offer the best performance
- Block size significantly impacts speedup and execution time
- The 32×32 configuration should be avoided for large or noisy images

Thank you for your attention!

Politecnico
di Torino

References

[1] Computational resources were provided by HPC@POLITO (http://hpc.polito.it)

[2] Lu, Hsin-Chen & Su, Liang-Ying & Huang, Shih-Hsu. (2024). Highly Fault-Tolerant Systolic-Array-Based Matrix Multiplication. Electronics.13. 1780. 10.3390/electronics13091780.

[3] HPC documentation: https://www.hpc.polito.it/docs/guide-slurm-it.pdf

[4] E Sanchez & J. E. R. Condia, High Performance Computing (01HEKUU), course lectures, Politecnico di Torino, A.Y. 2024–2025

[5] Github repository: **https://github.com/Merlino2706/HPC_Assignment.git**