UNIVERSITÀ DEGLI STUDI FIRENZE

# Mathematical Morphology

## Parallel Computing Mid-Term

Angelo D'Amante
Fabian Greavu

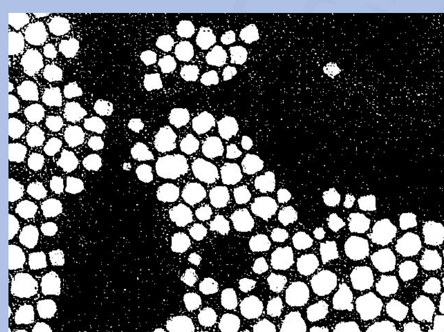
# Introduction

# Introduction: MM Theory

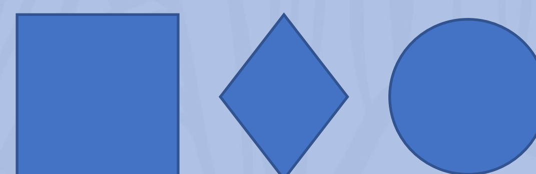
**Mathematical Morphology (MM) is a theory and technique for the analysis and processing of geometrical structures.**

**MM is also the foundation of morphological image processing, which consists of a set of operators that transform images according to any characterizations.**

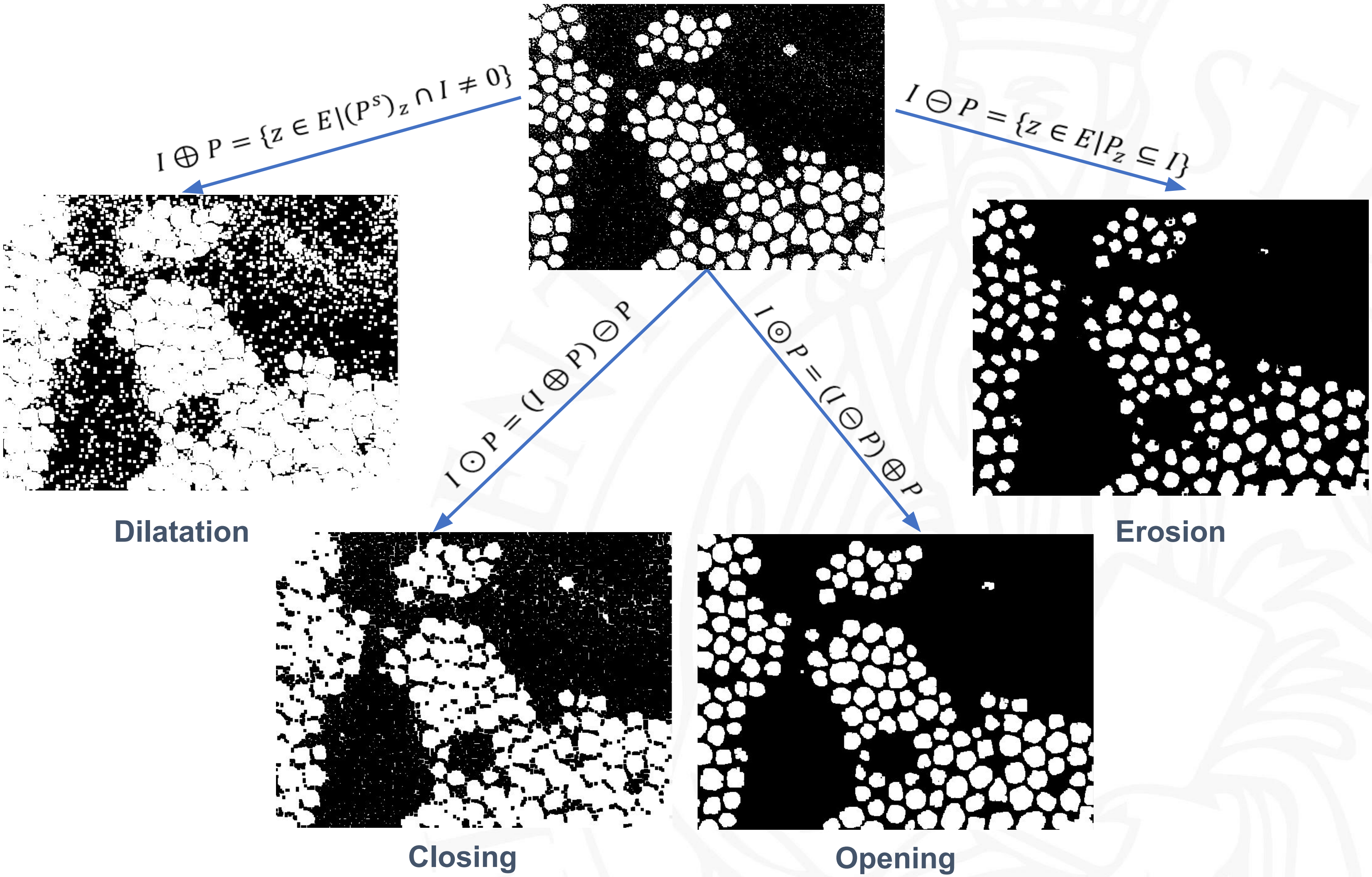
$Image\ (I)$:



$Probe\ Elements\ (\mathrm{P})$:



# Introduction: Morphological Operators



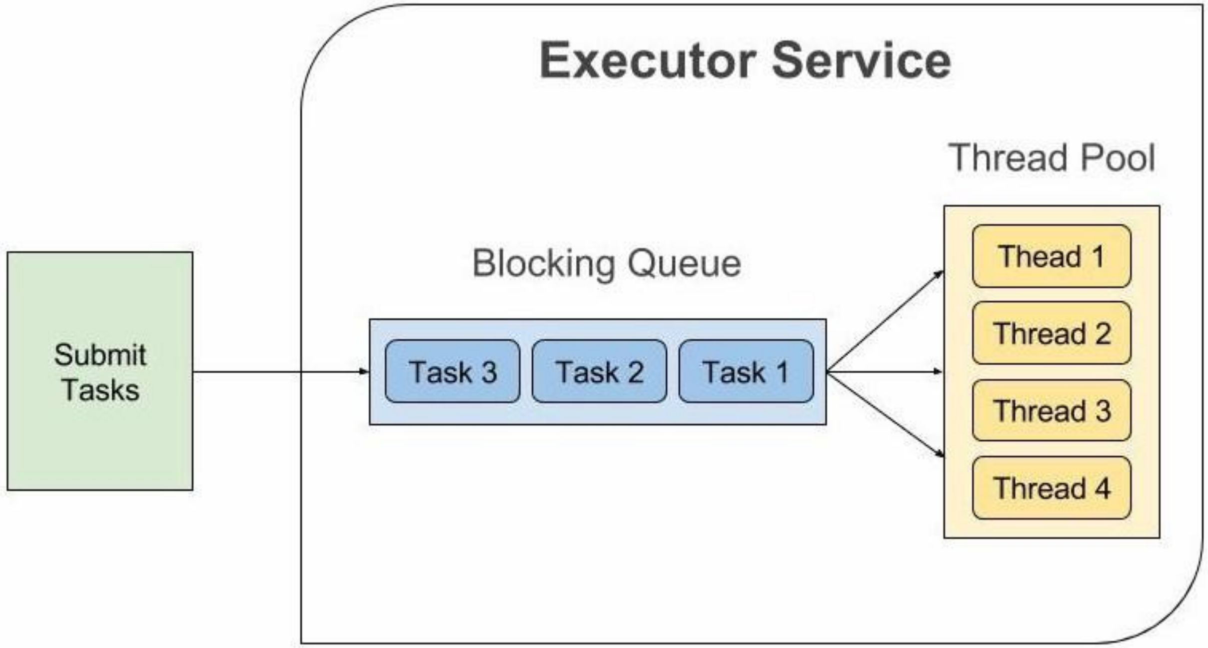
# Sequential Version

## Pseudo-Code Algorithm:

```
for rowImg = 0,1,…,imgH do
    for colImg = 0,1,…,imgW do
        for rowPrb = 0,1,…,probeH do
            for colPrb = 0,1,…,probeW do
                update Neighborhood[ ]
            end for
        end for
        update Img[rowImg*imgW+colImg] with
             max or min of Neighborhood
    end for
end for
```

# JAVA Implementation

# JAVA



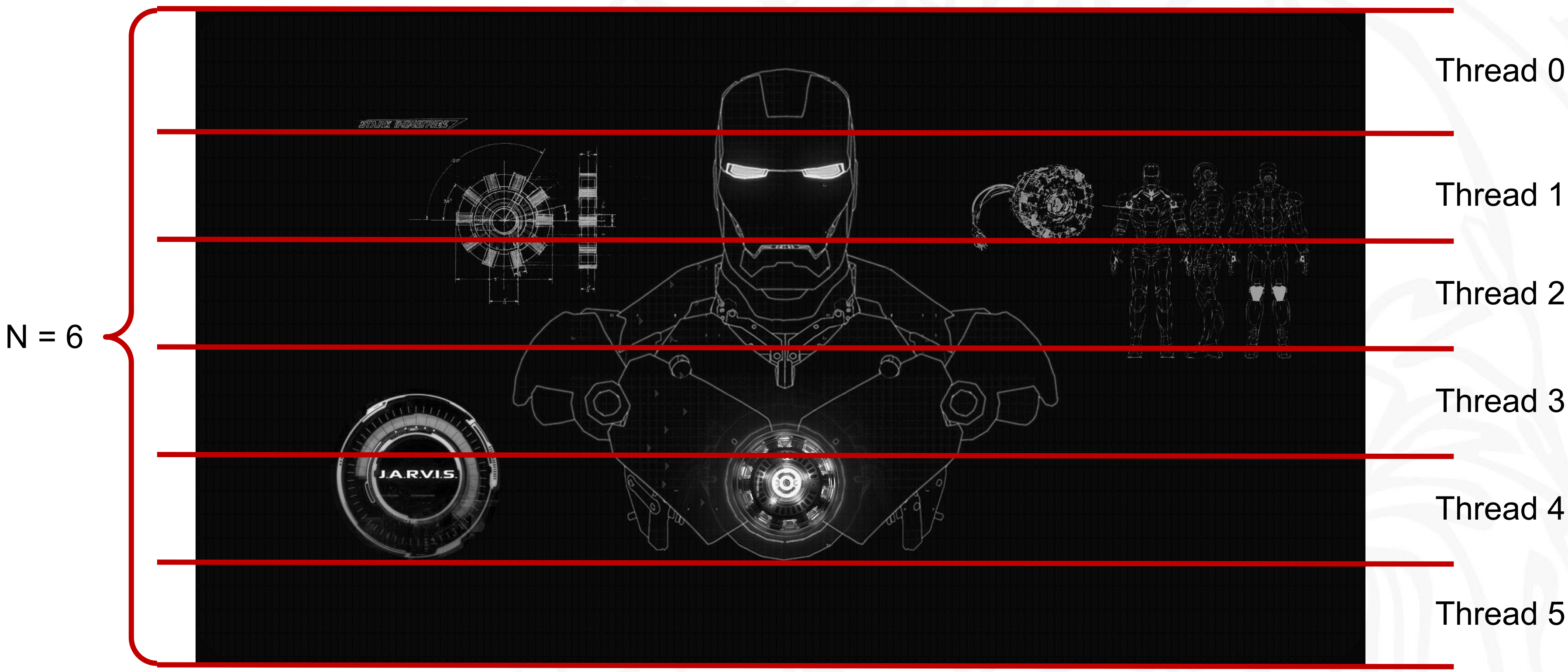

- **ExecutorService** can be used as a **Fork-Join** pattern
- Easy to instantiate and execute a job
- Easy to create a **barrier** with **waitExecutors** method

# JAVA: parallel idea

Since computation of each pixel is **independent** from the others (apart from getting neighbours value) the idea is:

- Given a **Thread** number **N**
- Given a grayscale image with **size W, H**

Split the image in **N chunks of rows** and let each thread process his chunk of data. Finally keep a **barrier** and **wait** for all to complete.



# JAVA: implementation

(Dilation method)

```
int rowsOfTh = Math.floor(H/N); // max rows per thread
ExecutorService tasks_executor = Executors.newFixedThreadPool(N);
for (int r = 0; r < H; r += rowsOfTh){
    MaskApplier t = new MaskApplier(...);
    tasks_executor.execute(t)
}
waitExecutors(tasks_executor); //barrier method
```

(MaskApplier run method)

```
int m = MASK_SIZE;
for (y = y_min; y < y_max; y++){  // loop on Thread's rows
    for (x = 0; x < W; x++) {         // loop on all x values
        p = img[x,y];
        for (ty = y-m/2; ty<=y+m/2; ty++)
            for (tx = x-m/2; tx<=x+m/2; tx++)
                if (p > img[tx, ty])
                    p = img[tx, ty] // find max on neighbours (mask)
        output_img[x, y] = p      // set output pixel
    }
}
```

# JAVA: experiments

- **Sequential Timings** are taken by:

  CPU Intel(R) Core(TM) i7-9700K CPU @ 3.6GHz

- **Parallel Timings** are taken by:

  CPU Intel(R) Core(TM) i7-9700K CPU @ 3.6GHz

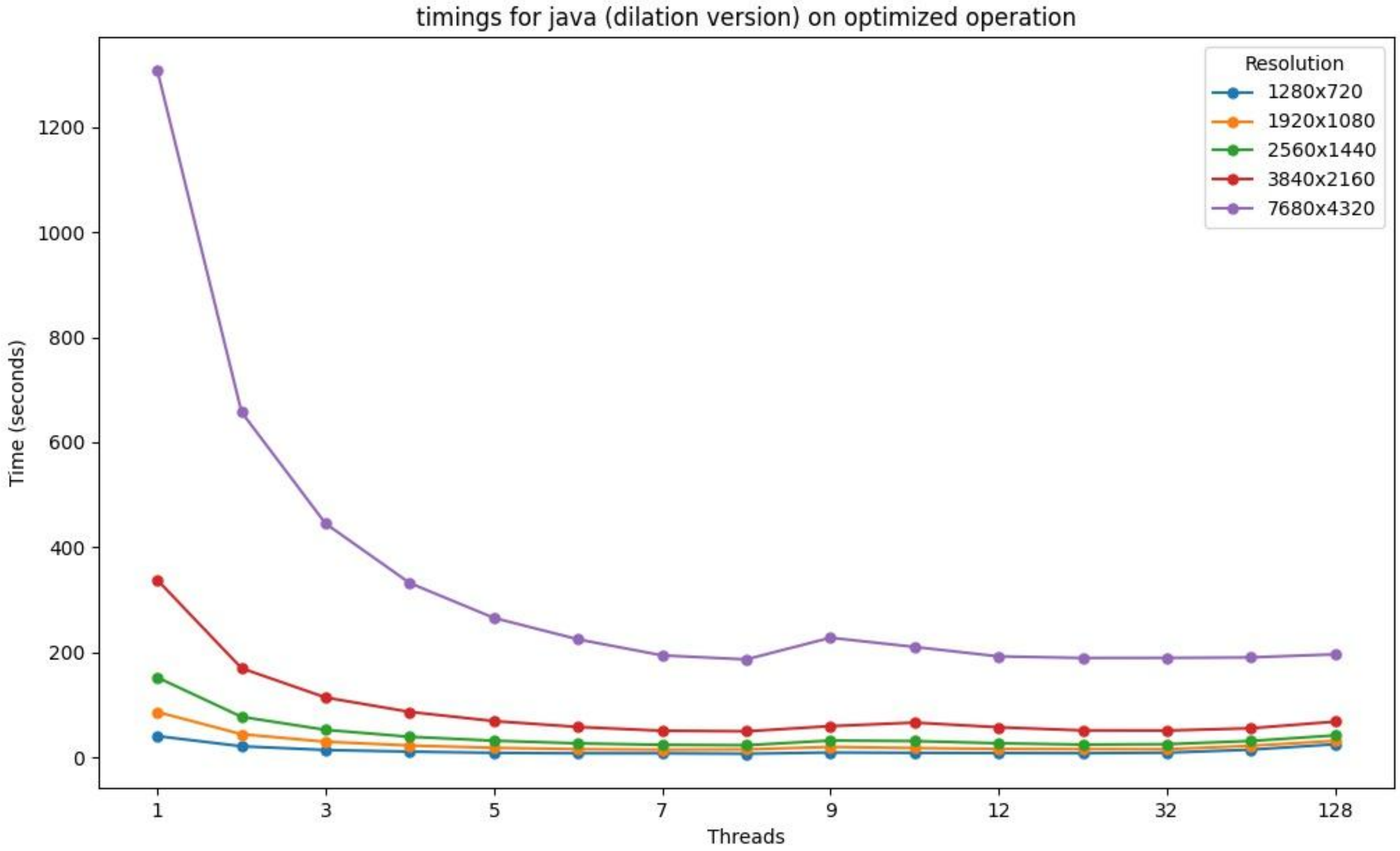
  Each core boosts up to 4.6 Ghz OOTB

  No HyperThreading

- **Experiments** with:
  - Dataset = {1280x720, 1920x1080, 2560x1440, 3840x2160, 7680x4320}
  - Threads = {2, 3, 4, 6, 8, … , 128}

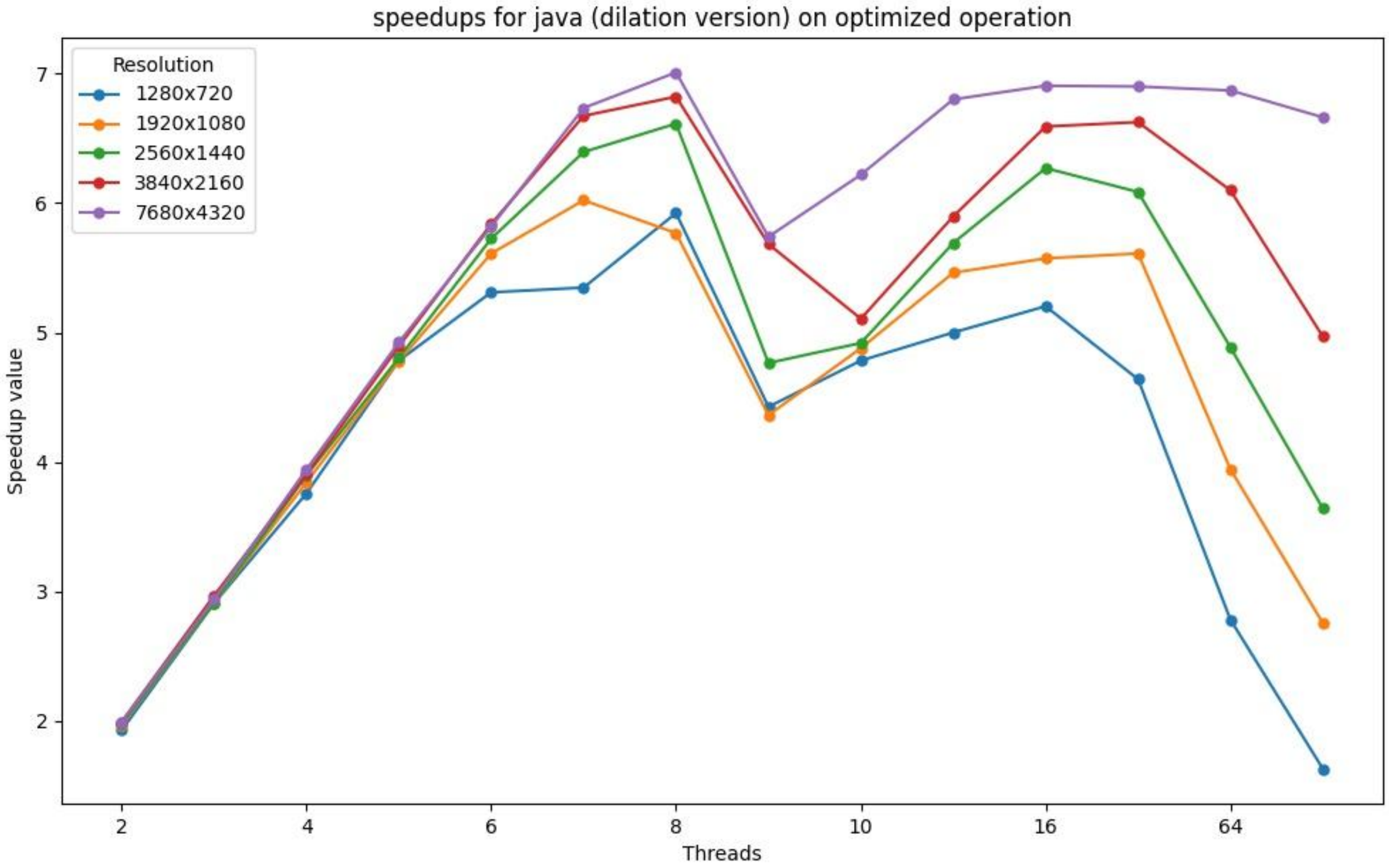
UNIVERSITÀ DEGLI STUDI FIRENZE

# JAVA: results (timing)




Results show a huge timing decrease until 8 Threads. Bigger datasets ⇒ bigger speedup.

# JAVA: results (speedups)

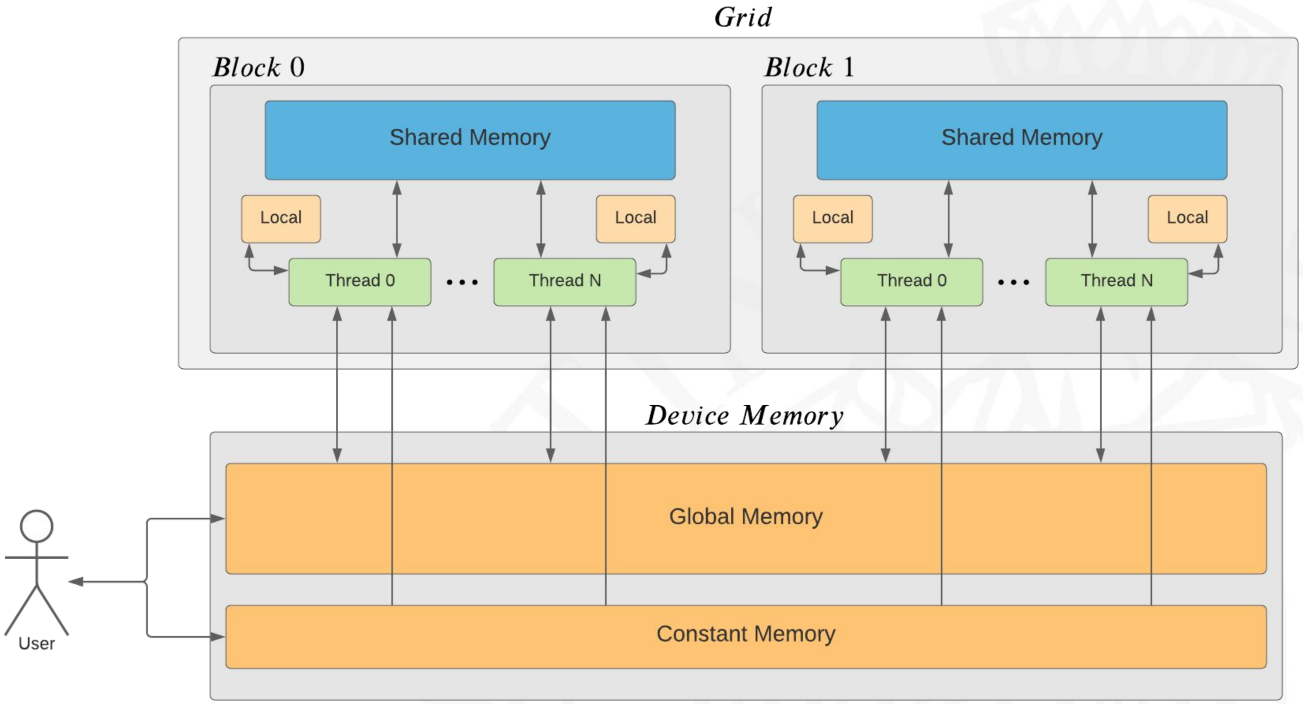



Results show max speedup of 7 at 8 threads. Bigger datasets ⇒ bigger speedup.

UNIVERSITÀ DEGLI STUDI FIRENZE

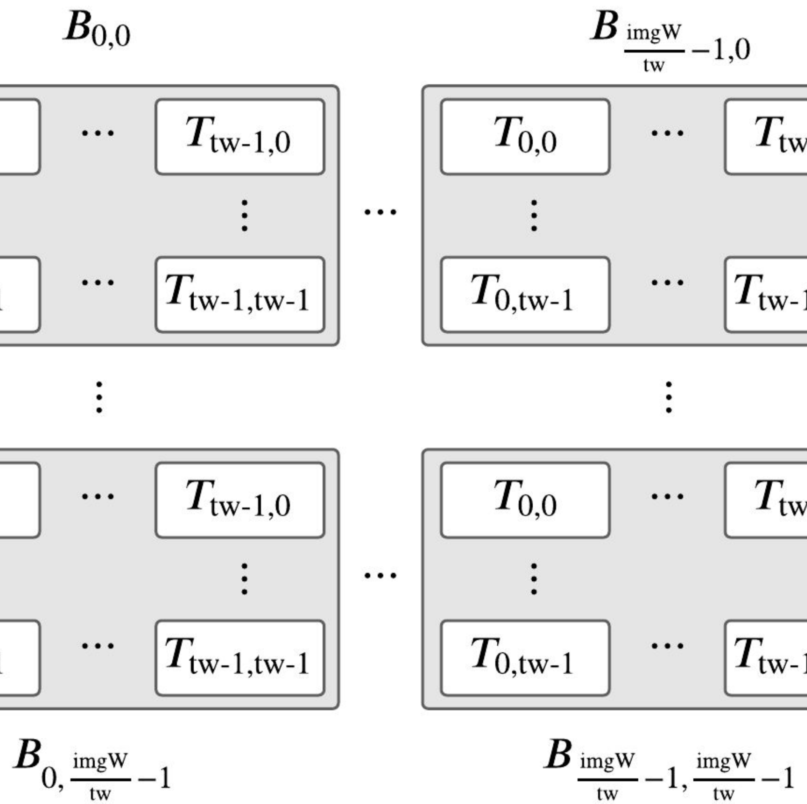
# CUDA Implementation

# CUDA




- **SIMT** (Single Instruction Multiple Thread) model.
- User decides number of threads $T \rightarrow \left\lceil \frac{N}{T} \right\rceil$ blocks.
- Exploit both **global** and **shared** memory access.

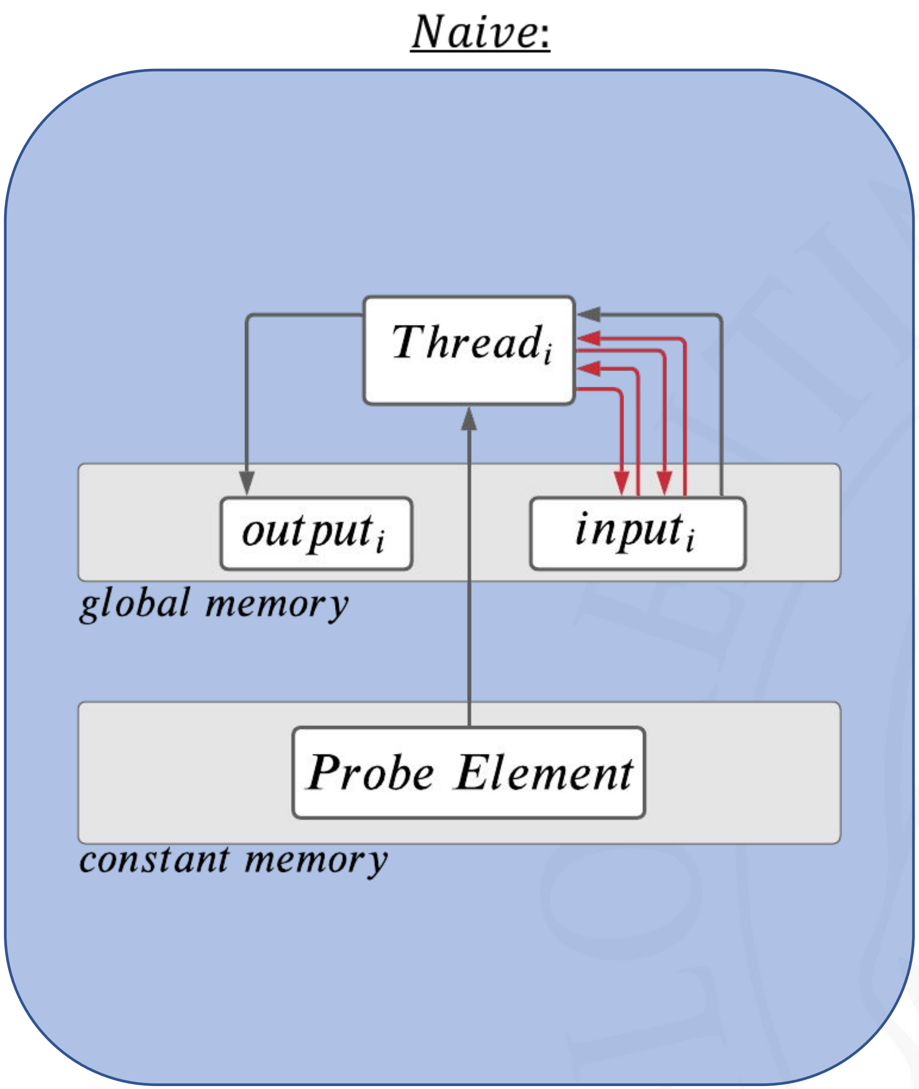
# CUDA: Kernel Calls

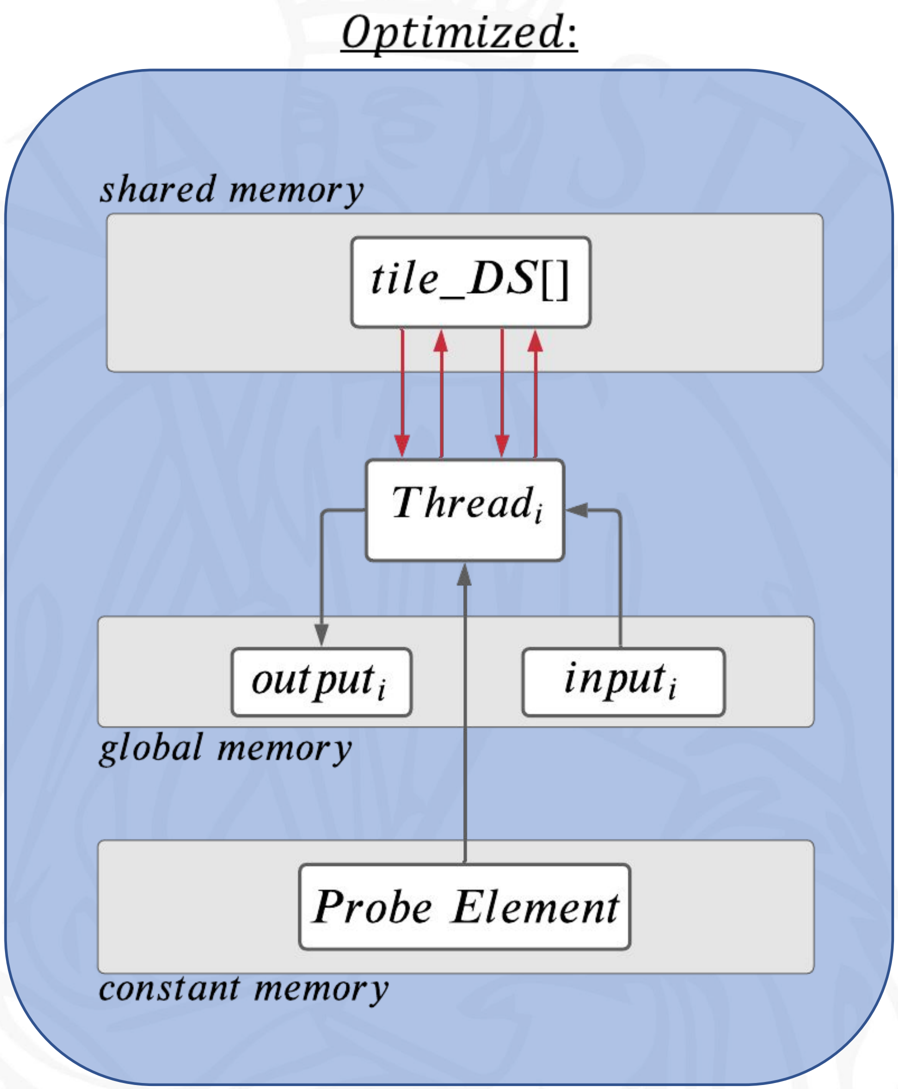



$$Naive \rightarrow process \lll dimGrid, dimBlock \ggg (inImg, probe, outImg)$$

$$Optimized \rightarrow process \lll dimGrid, dimBlock, shared\_amount \ggg (inImg, probe, outImg)$$

# CUDA: Solutions

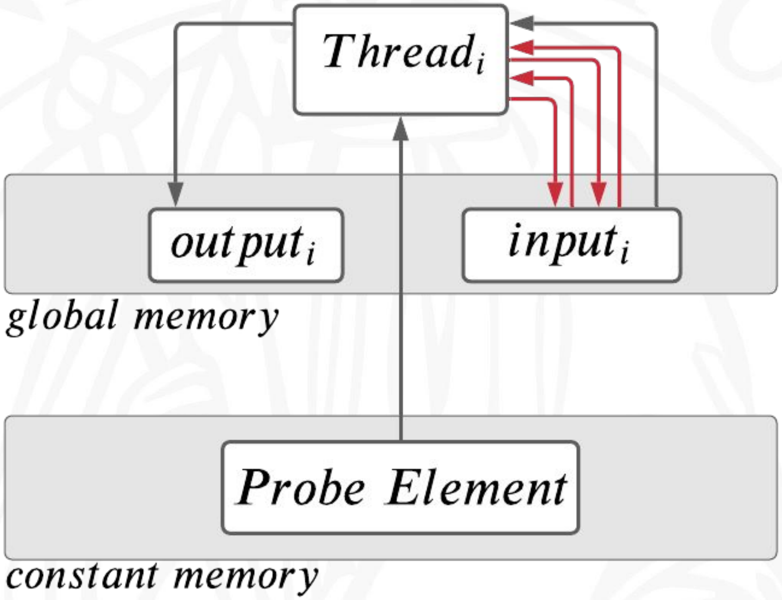


# CUDA: Naive Solution

**Pseudo-Code:**

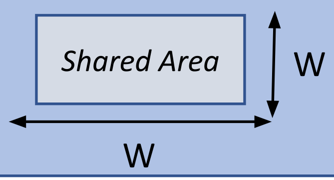
```
tid_x = (blockIdx.x * blockDim.x) + threadIdx.x
tid_y = (blockIdx.y * blockDim.y) + threadIdx.y
max = min = inImg[rowImg * imgW + colImg]
for rowPrb = 0, … , prbH do
    for colPrb = 0, … , prbW do
        update(max)
        update(min)
    end for
end for

If (EROSION) {outImg[rowImg * imgW + colImg] = min}
If (DILATATION) {outImg[rowImg * imgW + colImg] = max}

__syncthreads()
```



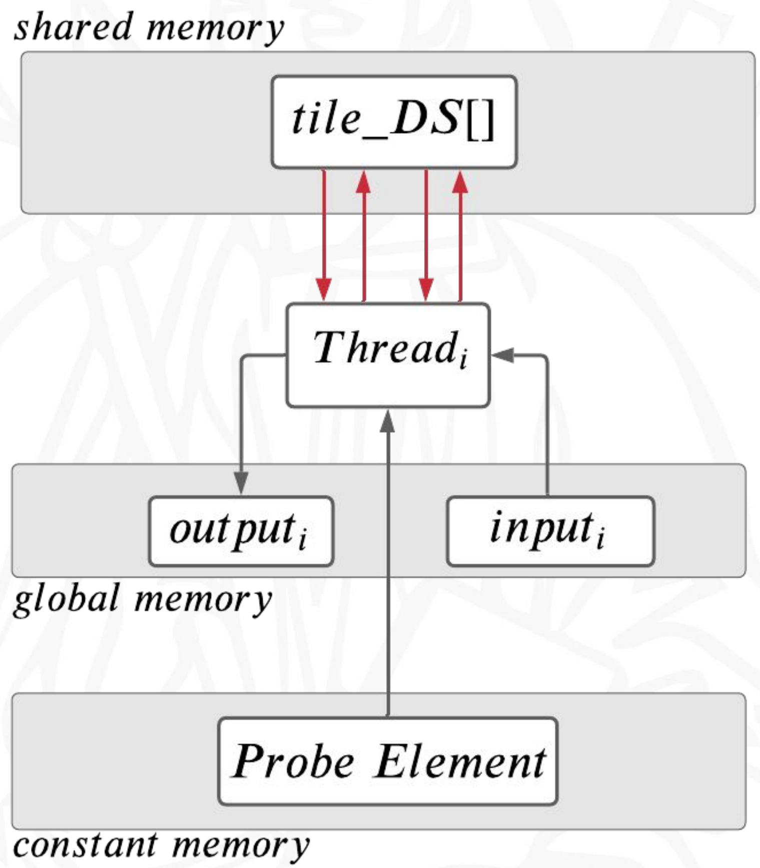
# CUDA: Optimized Solution

## 2-Batch Loading:

- TW = TILE_WIDTH
- Each blocks is made of TW * TW threads.
- W = TW + Probe_Width – 1




```
First Batch Loading:
    dest = threadIdx.y * TW + threadIdx.x
    tile_DS[dest / W][dest % W] = inputImg[src]

Second Batch Loading:
    dest = threadIdx.y * TW + threadIdx.x + TW * TW
    tile_DS[dest / W][dest % W] = inputImg[src]

__syncthreads()

for y = 0, …, probeW do
    for x = 0, …, probeH do
        compute max and min
    end for
end for

__syncthreads()
```



# CUDA: Experiments

- **Sequential Timings** are taken by:

  CPU Intel(R) Core(TM) i7-8750H CPU @ 2.20GHz

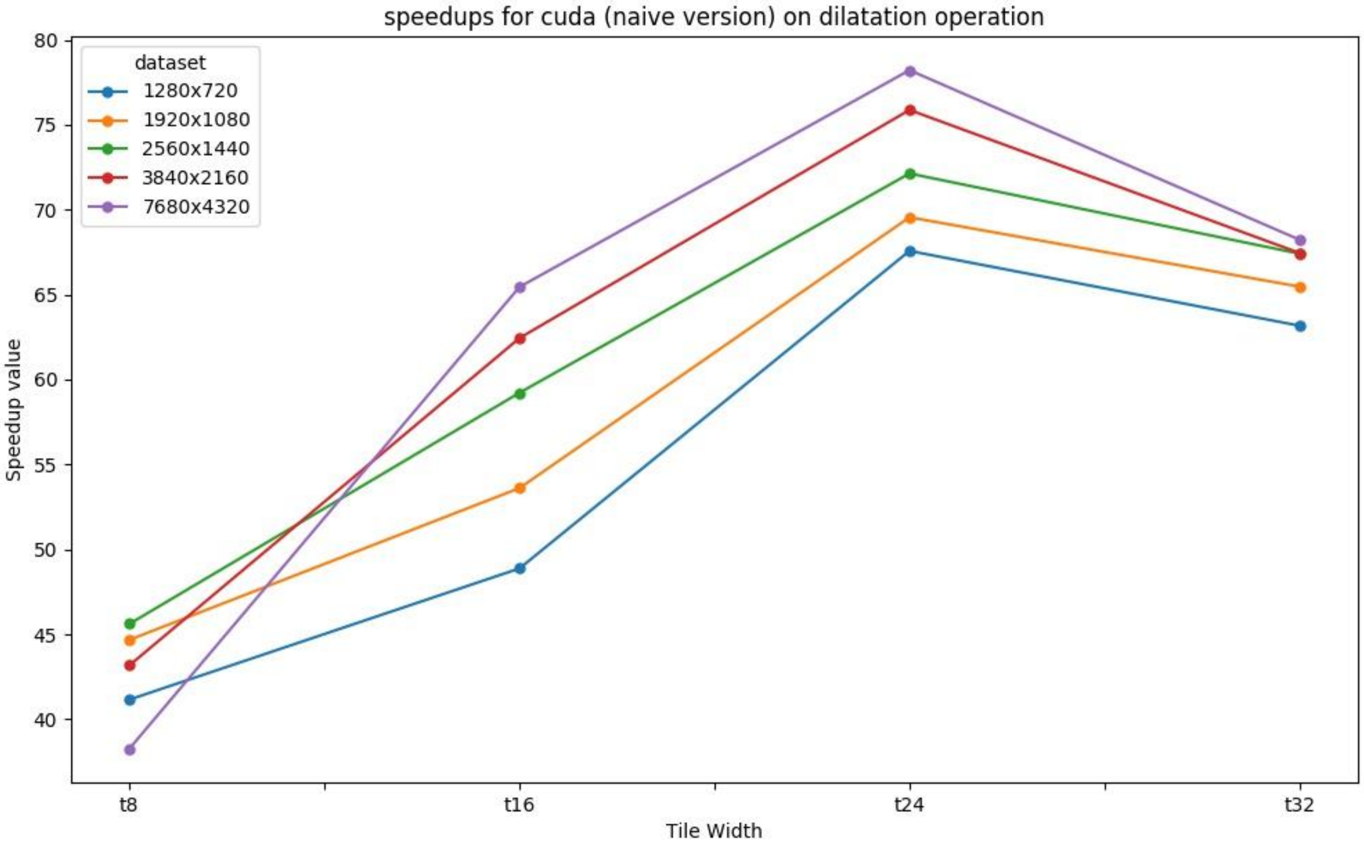
- **Parallel Timings** are taken by:

  GPU NVIDIA GeForce GTX 1050 Ti

  With 4096 MB dedicated and 768 CUDA Cores

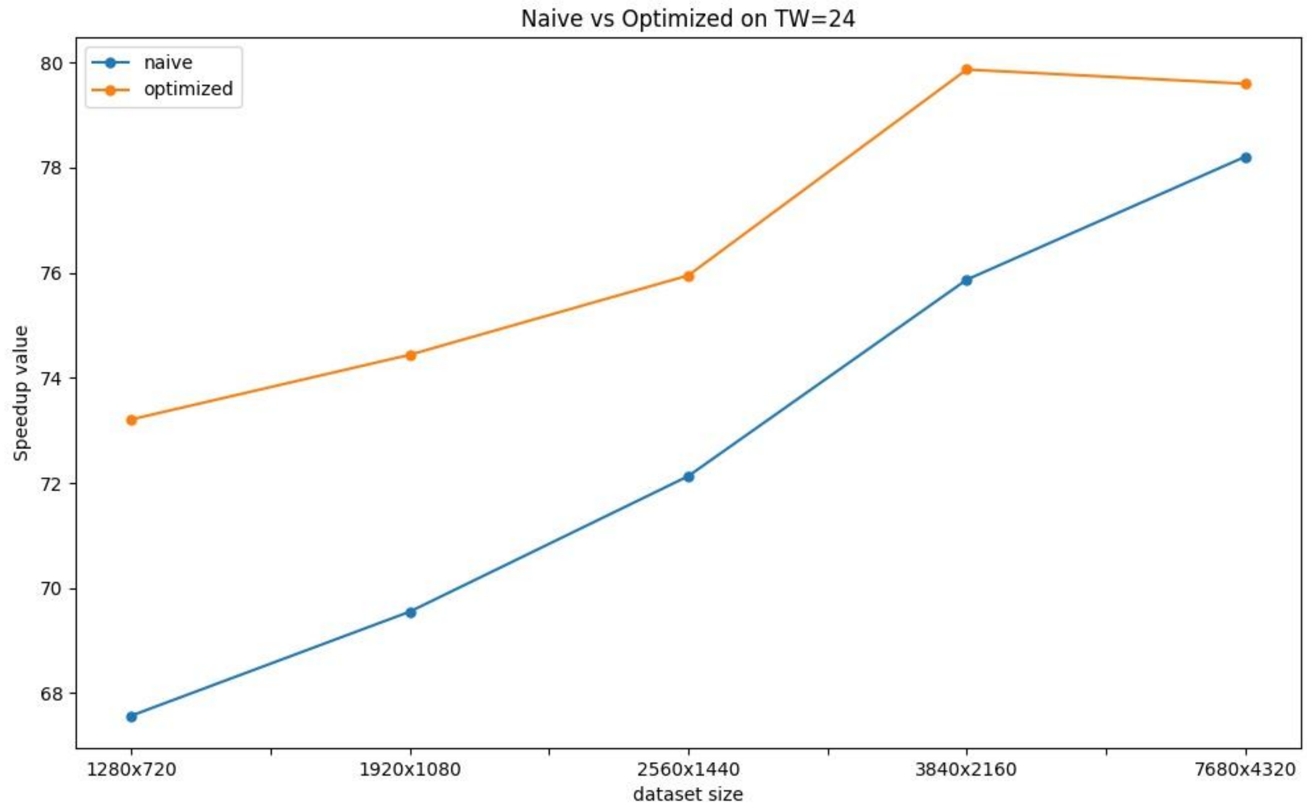
- **Experiments** with:
  - Dataset = {1280x720, 1920x1080, 2560x1440, 3840x2160, 7680x4320}
  - Tiles = {8, 16, 24, 32}

# CUDA: Results



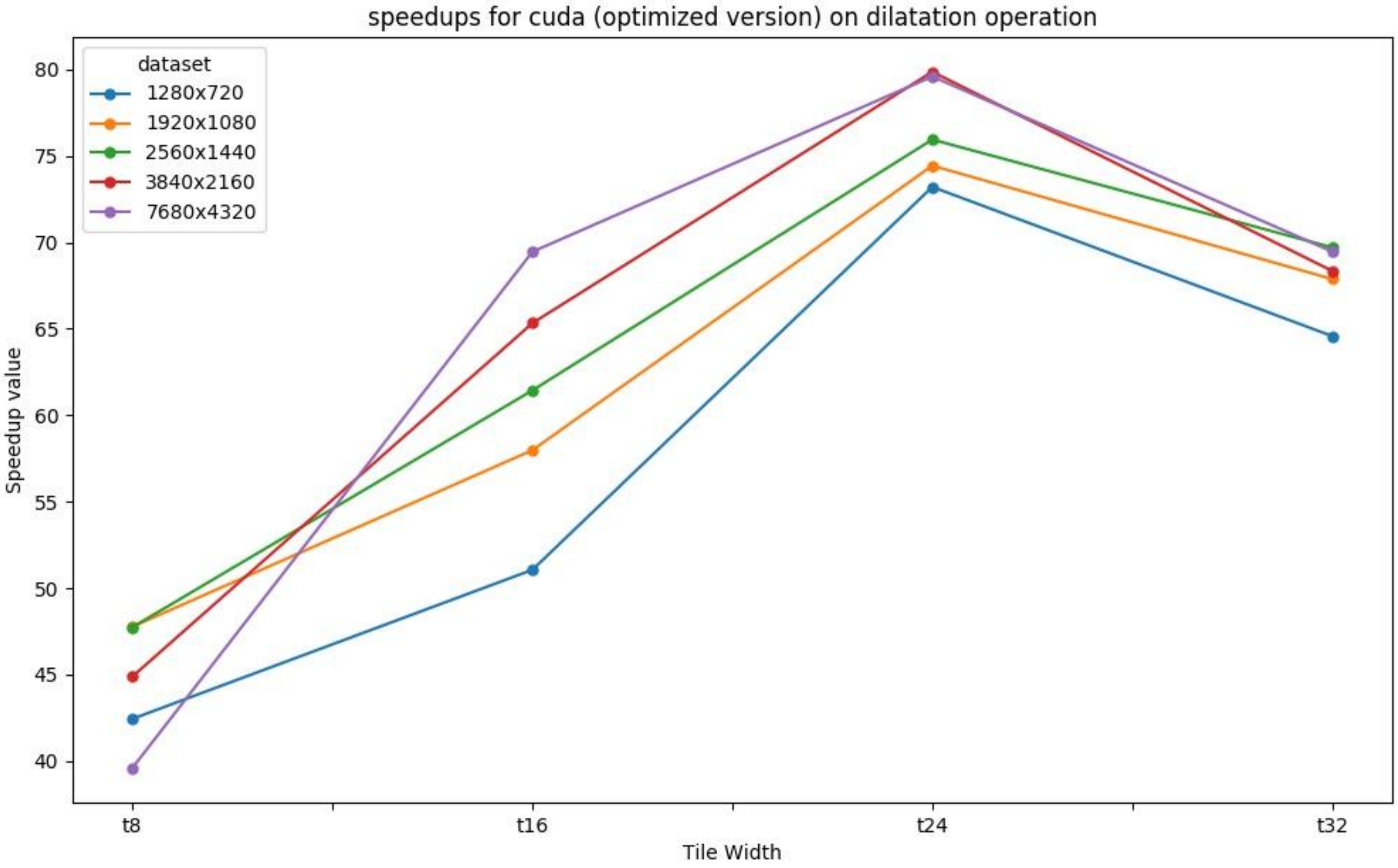

- The best speedup is obtained with Tile Width = 24, thus 576 threads per block.
- Bigger Datasets ⇒ Bigger Speedups

# CUDA: Results



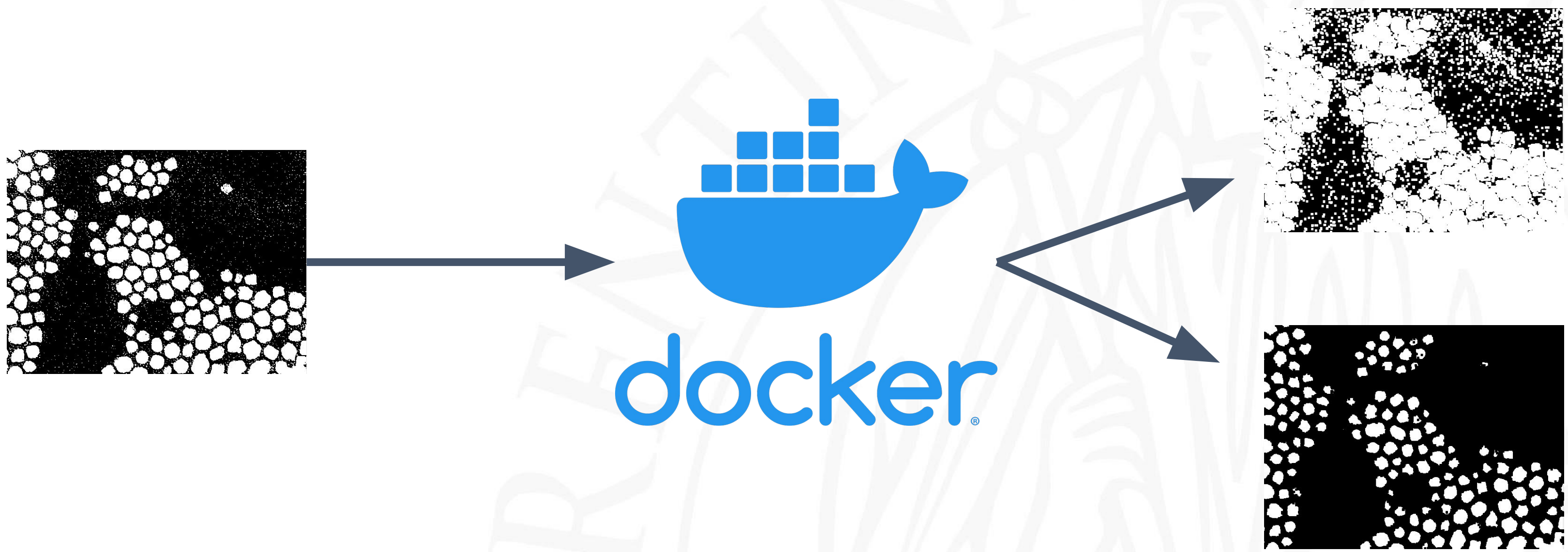

- Max speedup = 91.09 is obtained with Tile Width = 24 and dataset = 3840x2160

# CUDA: Results



# Conclusions:

**Three implementations of MM: one sequential and two parallel.**




https://github.com/AngeloDamante/morphological-image-processing-in-parallel

UNIVERSITÀ DEGLI STUDI FIRENZE

# Thanks for your attention

Angelo D’Amante
Fabian Greavu