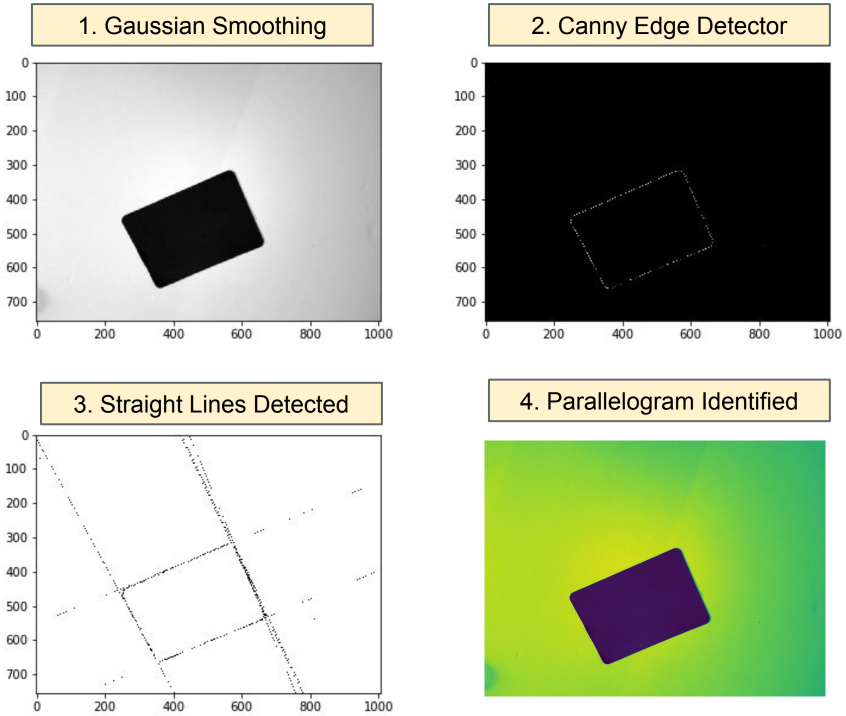
# Image Processing Optimization

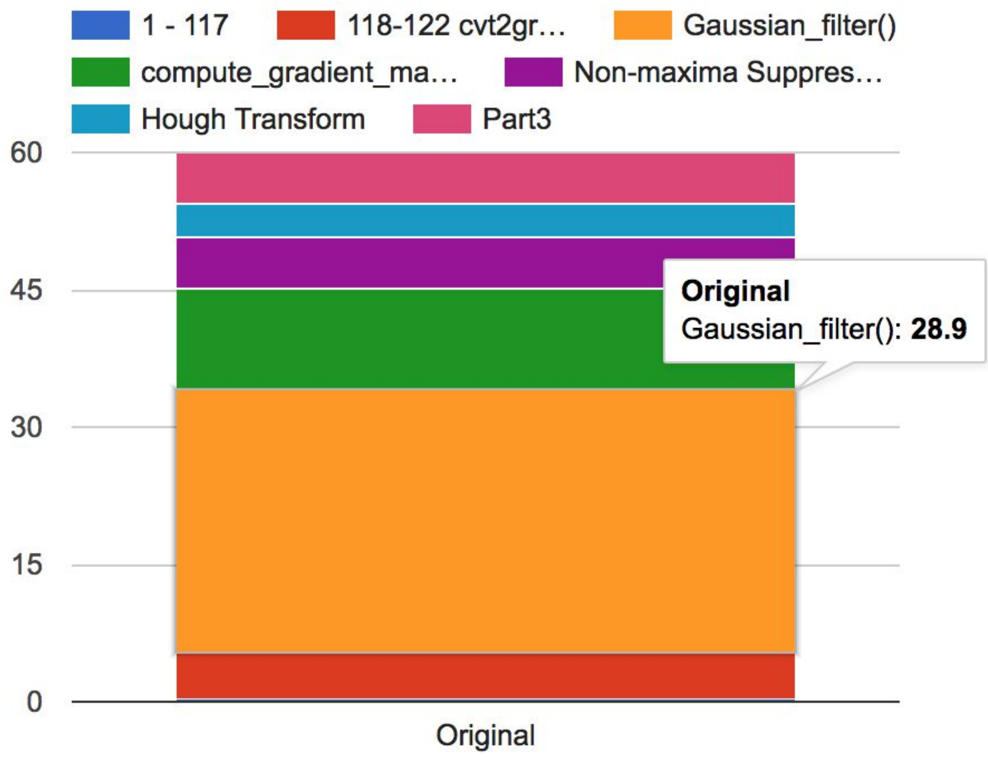
ks4841 ss8955 xg626

# Problem Statement

- Parallelogram detection in images
- Involves multiple pixel by pixel processing of the image
- Implemented “from scratch”
- Runs ~90s for one image



# Line-by-line Profiler




Over 70%

Top 3 functions:

1. smooth_image_with_Gaussian_filter ( 41.3% )
2. Compute_gradient_magnitude_and_gradient_angle ( 21.9% )
3. Cvt2grayscale ( 6.0% )

# Methodology

## Part 1: Improve by stages:

- Stage 1: Reduce for-loops (itertools, broadcasting), reduce function calls, apply built-in functions, efficient member testing, redundancy control
- Stage 2: Numpy/vectorizing
- Stage 3: Itertools
- Stage 4: Cython
- Stage 5: Numba

## Part 2: Implement with other languages and tools:

julia PySpark

**Each function** in the code iterated through the stages

**Each stage timed and profiled** using prun/cProfile/time -it

**Best outcome for each function** included in the final code

# Itertools to reduce for loops (applied with Numba)

```
for i in range(1, row):
    for j in range(1, col):
        Gx = (image_smoothed[i][j] + image_smoothed[i - 1][j]
              - image_smoothed[i][j - 1] - image_smoothed[i - 1][j - 1])
        Gy = (image_smoothed[i - 1][j - 1] + image_smoothed[i - 1][j]
              - image_smoothed[i][j - 1] - image_smoothed[i][j])
        gradient_magnitude[i][j] = math.sqrt(Gx * Gx + Gy * Gy)
```

11.2s

```
@jit
def compute_gradient_magnitude_and_gradient_angle_Final(image_smoothed):
    indices = product(range(1,row),range(1,col))
    for i in indices:
        Gx = (image_smoothed[i] + image_smoothed[i[0]-1][i[1]]
                - image_smoothed[i[0]][i[1]-1] - image_smoothed[i[0]-1][i[1]-1])
        Gy = (image_smoothed[i[0]-1][i[1]-1] + image_smoothed[i[0]-1][i[1]]
                - image_smoothed[i[0]][i[1]-1] - image_smoothed[i])
        gradient_magnitude[i] = math.sqrt(Gx * Gx + Gy * Gy)
```

3.72s

```
py_bench = @benchmark $python_func()
```

```
BenchmarkTools.Trial:
  memory estimate:  308.19 KiB
  allocs estimate:  12204
  --------------
  minimum time:     4.289 ms (0.00% GC)
  median time:      5.129 ms (0.00% GC)
  mean time:        5.925 ms (9.95% GC)
  maximum time:     379.294 ms (96.57% GC)
  --------------
  samples:          843
  evals/sample:     1
```

```
julia_bench = @benchmark $julia_func()
```

```
BenchmarkTools.Trial:
  memory estimate:  1.31 KiB
  allocs estimate:  12
  --------------
  minimum time:     5.367 μs (0.00% GC)
  median time:      6.950 μs (0.00% GC)
  mean time:        7.408 μs (1.58% GC)
  maximum time:     436.531 μs (96.38% GC)
  --------------
  samples:          10000
  evals/sample:     6
```

## Method:

use **PyCall** in Julia, and declare python function and types with **@pyimport**

## Result:

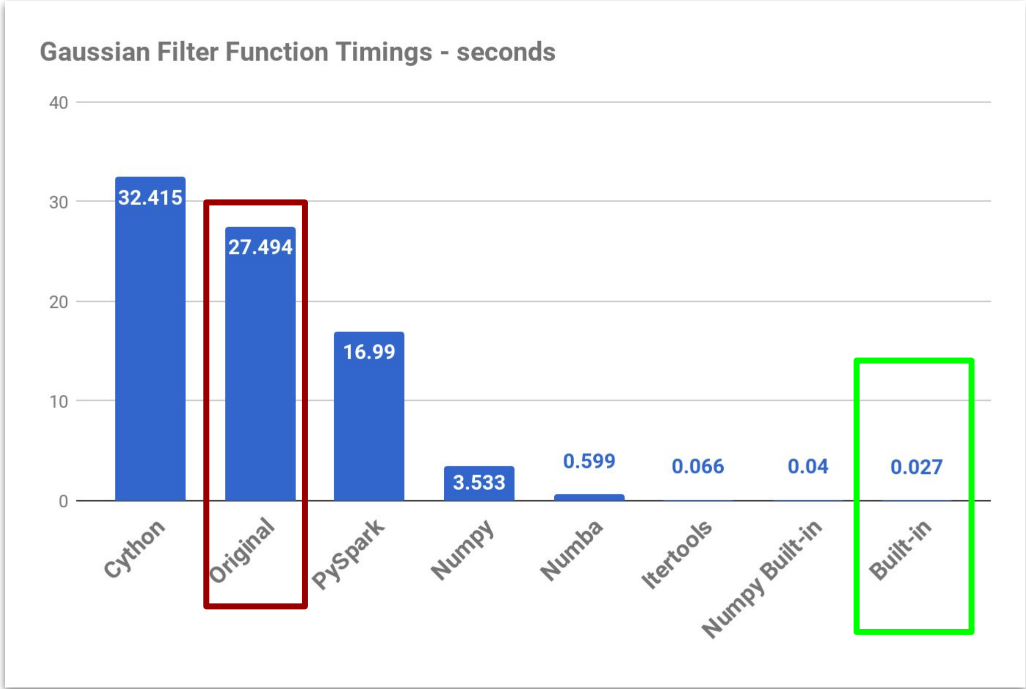
Both functions return the same result, but from the benchmark report, we see huge improvement using **Julia.**

```
d = Dict()
d["Julia"] = minimum(julia_bench.times) / 1e6
d["Python"] = minimum(py_bench.times) / 1e6
d
```

```
Dict{Any,Any} with 2 entries:
  "Julia"  => 0.0244
  "Python" => 4.29684
```

# Numpy & Built-in Functions

- Applying **built-in functions** is the best and easiest way to improve performance
- **Vectorizing with Numpy** vs. for-loops speeds up code significantly
- **Choosing tools wisely**: PySpark helps parallelize processes, but underperforms for a single image



# The power of built-in functions

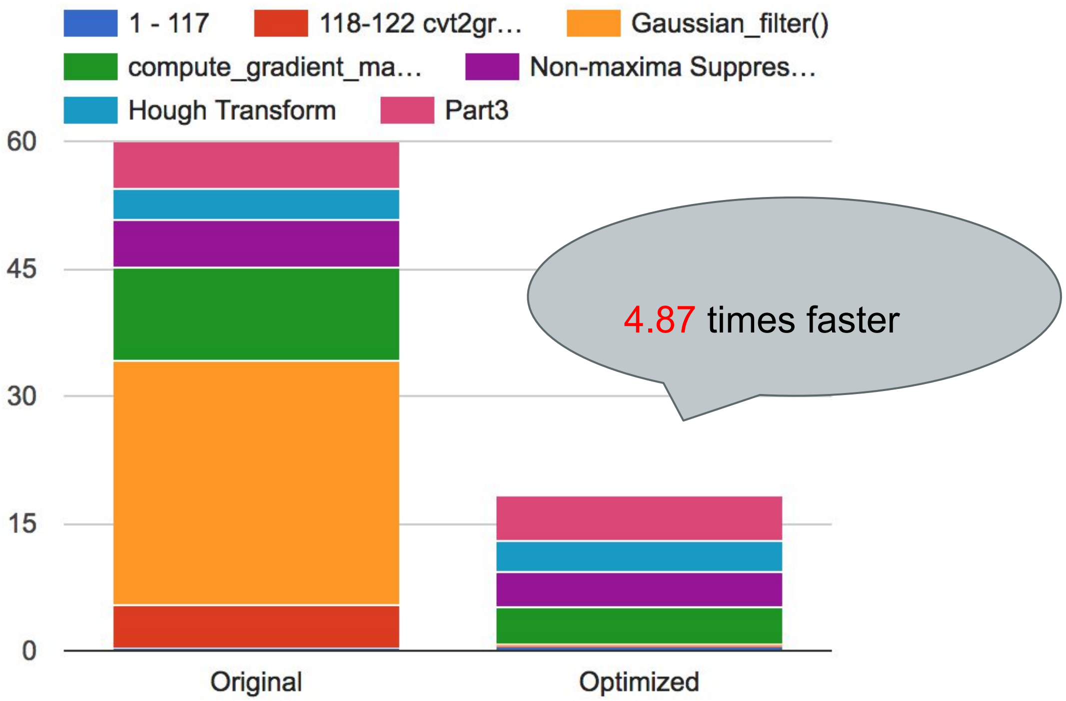
**27 s.**

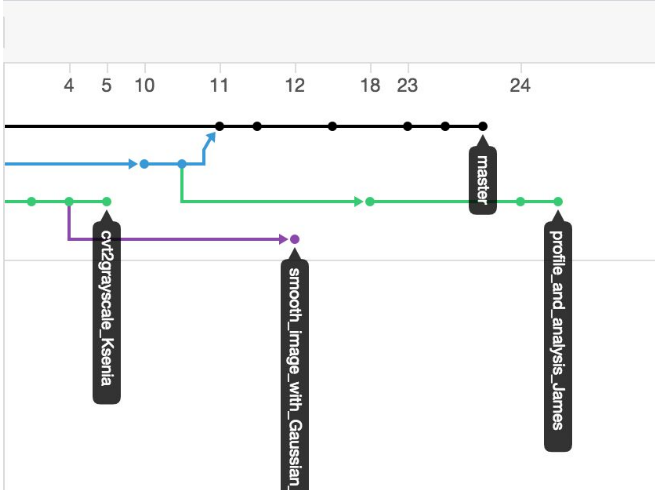
```
def smooth_image_with_Gaussian_filter(img):
    kernel = (0.006, 0.061, 0.242, 0.383, 0.242, 0.061, 0.006)
    kernel_size = len(kernel)
    border_offset = (kernel_size - 1) // 2

    img_copy = np.copy(img)
    for i in range(0, row):
        # Keep border values as they are
        for j in range(border_offset, col - border_offset):
            img_copy_ij = 0
            for k in range((-1) * border_offset, border_offset + 1):
                img_copy_ij += img[i][j + k] * kernel[border_offset + k]
            img_copy[i][j] = img_copy_ij

    img_copy_copy = np.copy(img_copy)
    # Keep border values as they are
    for i in range(border_offset, row - border_offset):
        for j in range(0, col):
            img_copy_copy_ij = 0
            for k in range((-1) * border_offset, border_offset + 1):
                img_copy_copy_ij += img_copy[i +
                                    k][j] * kernel[border_offset + k]
            img_copy_copy[i][j] = img_copy_copy_ij

    return img_copy_copy
```

**0.027 s.**

```
def smooth_image_with_Gaussian_filter(img):
        img_copy = np.copy(img)
        img_copy = ndimage.gaussian_filter(img_copy, sigma=0.1)
        return img_copy
```

# Results


1 - 117
118-122 cvt2gr…
Gaussian_filter()
compute_gradient_ma…
Non-maxima Suppres…
Hough Transform
Part3
60
45
30
15
0
Original
Optimized
4.87 times faster


4
5
10
11
12
18
23
24
master
cvt2grayscale_Ksenia
smooth_image_with_Gaussian
profile_and_analysis_James


IT'S HUUUGE!

# One more problem

## The ability to scale

The real bottleneck.

In part 3, we have to validate all options, which will definitely become the bottleneck.

With complexity over $O(n^2)$.

# What we learned

- Loops kill the performance! ( 1k*1k pic sums up to 1m running complexity.)
- How to apply the skills we acquired in class to one of the most common **problems in image processing**
- Understanding **built-in function algorithm**
- Working with **unfamiliar code/area**
    - Understanding the problem and implementation
    - Profiling the code to determine the most time inefficient functions
- **Choosing right tools** for the problem
    - Oftentimes, vectorizing with Numpy was the fastest way to process the data
    - More complex tools, such as PySpark, are useful for working on multiple files, but can actually slow down code for a single file processing

# Questions?