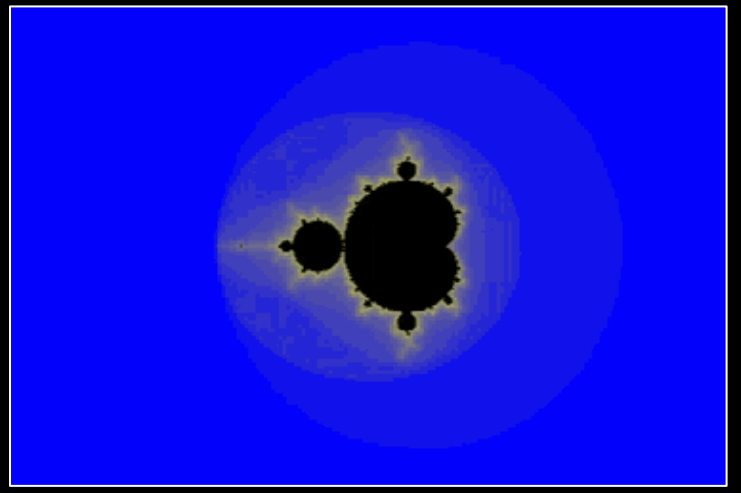
Quickly Computing the Mandelbrot Numbers

MA553 – Dr. Khanal

Fall 2022 Final Project

Tyler Procko

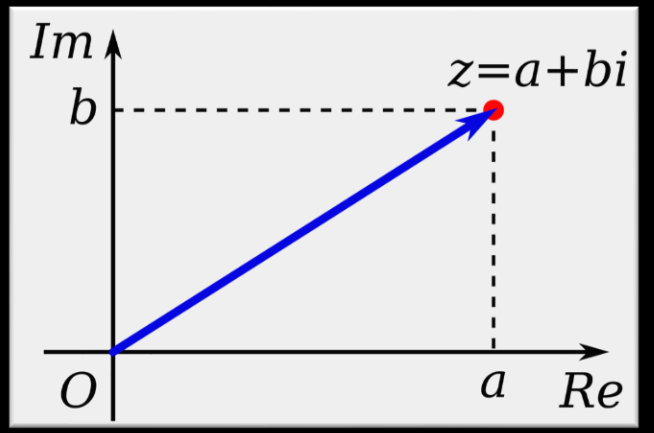
The Mandelbrot Set



- A set of complex numbers that forms a repeating fractal when plotted on the complex plane
  - Complex numbers are represented as a pair of numbers (a, b), of the form $c = a + bi$, where i is the imaginary number
- To know if a complex number, c, is part of the Mandelbrot set, perform the following calculation:

$$\begin{cases} z_0 = 0 \\ z_{n+1} = z_n^2 + c \end{cases}$$

- The complex number, c, is part of the Mandelbrot set if, for all $n > 0$, the absolute value of z does not diverge to infinity
- The number of iterations it takes for z to diverge is called the *escape count*
- Interesting to note:
  - On the complex plane, the Mandelbrot set fits within a circle of *with a radius of 2*, (the radius of divergence) which can be used as a hard limit for stopping computation



Input Values

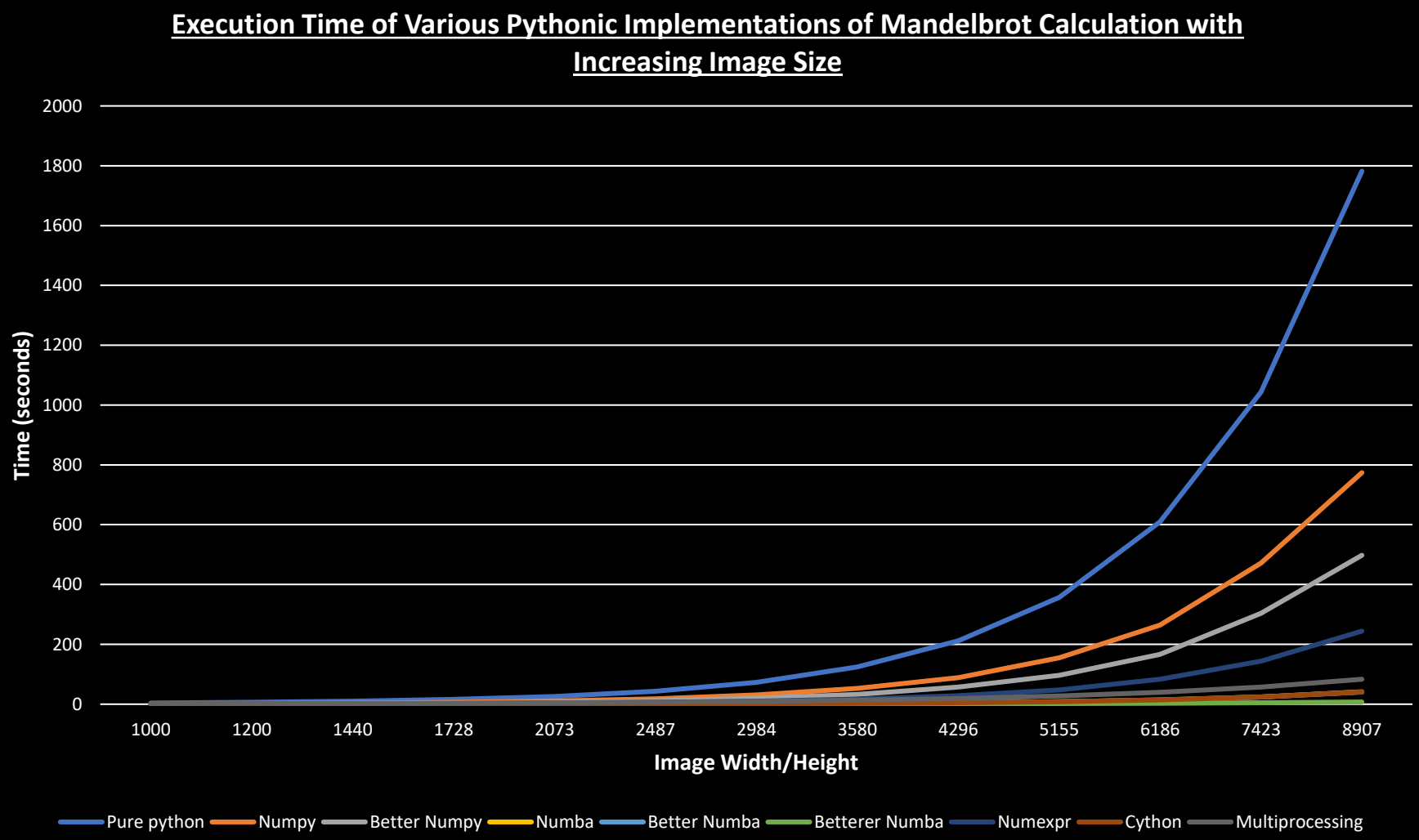
- The Mandelbrot set is not very interesting unless we plot it, so we need a large amount of values

- Baseline is an image of 1000x1000 (1-million individual) pixels
- In code, each pixel is a complex number that is checked as being part of the Mandelbrot set

- So, 1,000,000 complex numbers are checked up to 80 times each
  - Or until the number escapes (diverges to infinity)

Code / Graphical Setting	Mathematical Equivalent	Value
Minimum X	Smallest real value a of complex $c = (a, b)$	-2.0
Maximum X	Largest real value a of complex $c = (a, b)$	0.5
Minimum Y	Smallest imaginary value b of complex $c = (a, b)$	-1.2
Maximum Y	Largest imaginary value b of complex $c = (a, b)$	-1.2
Image Width	Number of real values	1000
Image Height	Number of imaginary values	1000
Maximum Iterations	Limit for c values that do not diverge to infinity so code stops running	80

Quick Summary



Baseline Pure Python – 5.3 seconds

- Pretty slow…
- Uses Python's built-in *complex* type, which is an object
  - Equivalent to doing *real + imaginary * 1j*
    - Python recognizes *j* after a number as meaning *imaginary*
  - Keep this in mind
- Changing *z = z ** 2 + c* to *z = z * z + c* improved speed around 40%, to 3.6 seconds
  - This is a low-level optimization; Python square exponent calculations are slower than just multiplying the number by itself

```
import numpy as np

def mandelbrot_purepython(c, maxIterations):
    z = 0
    for n in range(maxIterations):
        if abs(z) > 2:
            return n
        z = z ** 2 + c
    return maxIterations

def mandelbrot_set_purepython(xMin, xMax, yMin, yMax, width, height, maxIterations):
    realNums = np.linspace(xMin, xMax, width)
    imagNums = np.linspace(yMin, yMax, height)
    escapeCounts = []
    for r in realNums:
        for i in imagNums:
            escapeCounts.append(mandelbrot_purepython(complex(r, i), maxIterations))
    return (realNums, imagNums, escapeCounts)
```

Profiling

- Baseline acquired; now we profile
- The first function is the actual Mandelbrot code
- The second is the array setup

```
Function: mandelbrot_purepython at line 10

Line #      Hits         Time  Per Hit   % Time  Line Contents
==============================================================
    10                                           def mandelbrot_purepython(c, maxIterations):
    11   1000000    2143867.0      2.1      1.0      z = 0                           # z always starts as 0
    12  25871042   59485330.0      2.3     27.5      for n in range(maxIterations):  # When testing a complex, c, if we hit the max numb
    13  25130132   83202494.0      3.3     38.5          if abs(z) > 2:              # 2 is the radius of the Mandelbrot set circle (div
    14    740910    1849150.0      2.5      0.9              return n                # If we hit this limit, kick out
    15  25130132   68641877.0      2.7     31.8          z = z * z + c               # Mandelbrot (z ** 2 is MUCH slower; about 40%)
    16    259090     598708.0      2.3      0.3      return maxIterations            # Otherwise, return max iterations

Total time: 37.9639 s
File: E:\OneDrive - Embry-Riddle Aeronautical University\college\2022zfall\MA453\project2\mandelbrot1_purepython.py
Function: mandelbrot_set_purepython at line 18

Line #      Hits         Time  Per Hit   % Time  Line Contents
==============================================================
    18                                           def mandelbrot_set_purepython(xMin, xMax, yMin, yMax, width, height, maxIterations):
    19         1        973.0    973.0      0.0      realNums = np.linspace(xMin, xMax, width)
    20         1        435.0    435.0      0.0      imagNums = np.linspace(yMin, yMax, height)
    21         1          2.0      2.0      0.0      escapeCounts = []
    22      1000       4391.0      4.4      0.0      for r in realNums:
    23   1000000    3320744.0      3.3      0.9          for i in imagNums:
    24                                                       # Uses complex(), which is the same as real + imaginary * 1j
    25   1000000  376312281.0    376.3     99.1              escapeCounts.append(mandelbrot_purepython(complex(r, i), maxIterations))
    26         1          5.0      5.0      0.0      return (realNums, imagNums, escapeCounts)
```

- In the first:
  - We can speed things up using NumPy array elementwise operations instead of a double for loop on a list
- In the second:
  - The if statement Mandelbrot divergence check takes up most of the time, followed by the calculation of the new *z*
    - We can optimize if statements
    - We can optimize the calculation of the new *z*

NumPy – 1.85 seconds

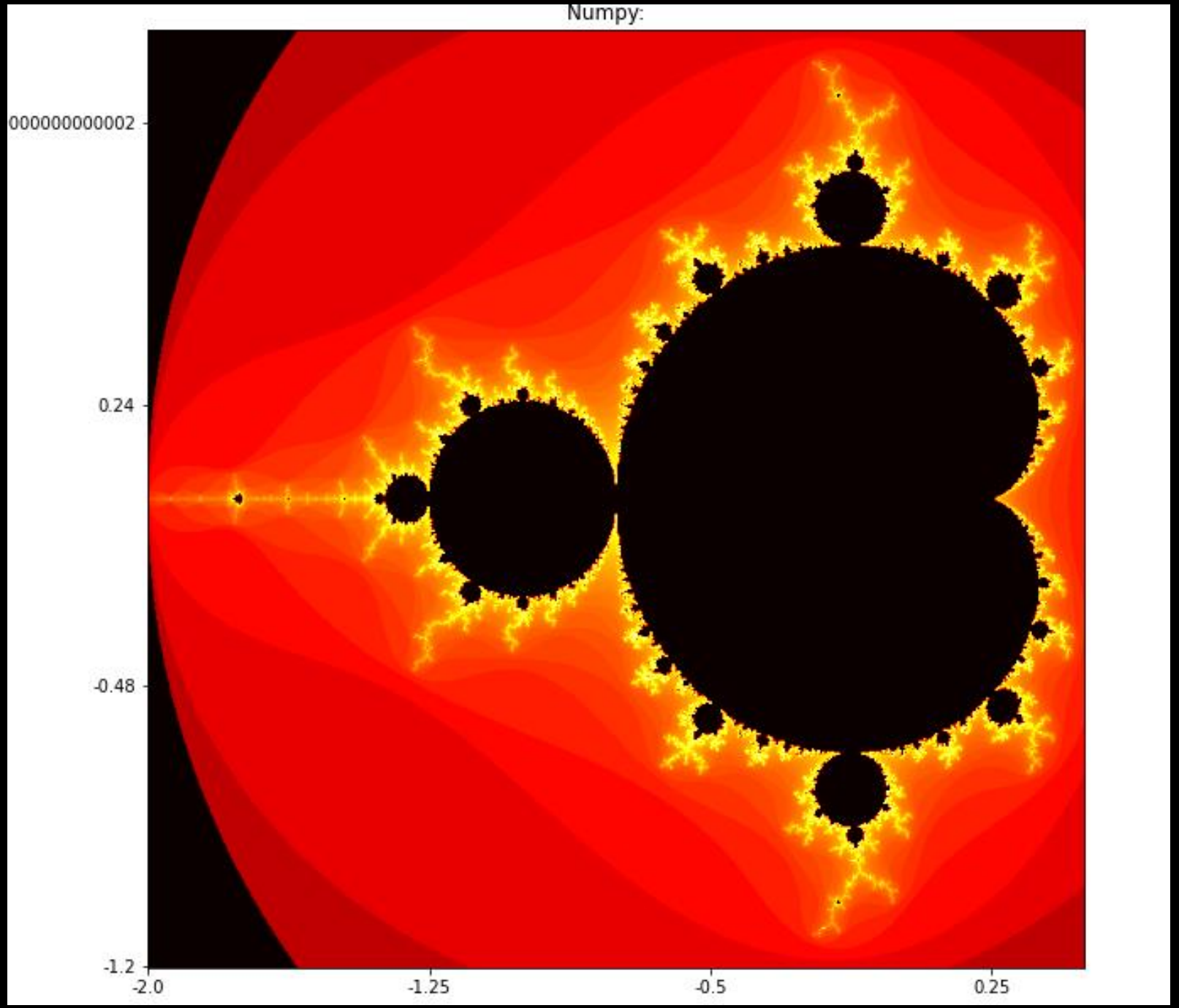
- Uses the *np.where* function, which returns an array given a Boolean condition
  - Faster than if statements
- Does not use Python's *complex* object type, and instead constructs complex numbers manually
  - Less Python class overhead means faster execution
- Uses a NumPy array for the complex numbers instead of a list
  - Now we can call the Mandelbrot function on the entire array
    - Fast!
- Specifying *dtype* declarations of *np.float32* improved speed to 1.23 seconds
  - This is because it was defaulting to *np.float64*

```
realNums = np.linspace(xMin, xMax, width, dtype = np.float32)
imagNums = np.linspace(yMin, yMax, height, dtype = np.float32)
```

```
import numpy as np

def mandelbrot_numpy(c, maxIterations):
  escapeCount = np.resize(np.array(0,), c.shape)
  z = np.zeros(c.shape, np.complex64)

  for iteration in range(maxIterations):
    z = z * z + c
    done    = np.greater(abs(z), 2.0)
    c       = np.where(done, 0 + 0j, c)
    z       = np.where(done, 0 + 0j, z)
    escapeCount = np.where(done, iteration, escapeCount)
  return escapeCount

def mandelbrot_set_numpy(xMin, xMax, yMin, yMax, width, height, maxIterations):
  realNums     = np.linspace(xMin, xMax, width, dtype = np.float32)
  imagNums     = np.linspace(yMin, yMax, height, dtype = np.float32)
  complexNums  = np.ravel(realNums + imagNums[:,None] * 1j)
  escapeCounts = mandelbrot_numpy(complexNums, maxIterations)
  escapeCounts = escapeCounts.reshape((width, height))
  return (realNums, imagNums, escapeCounts.T)
```

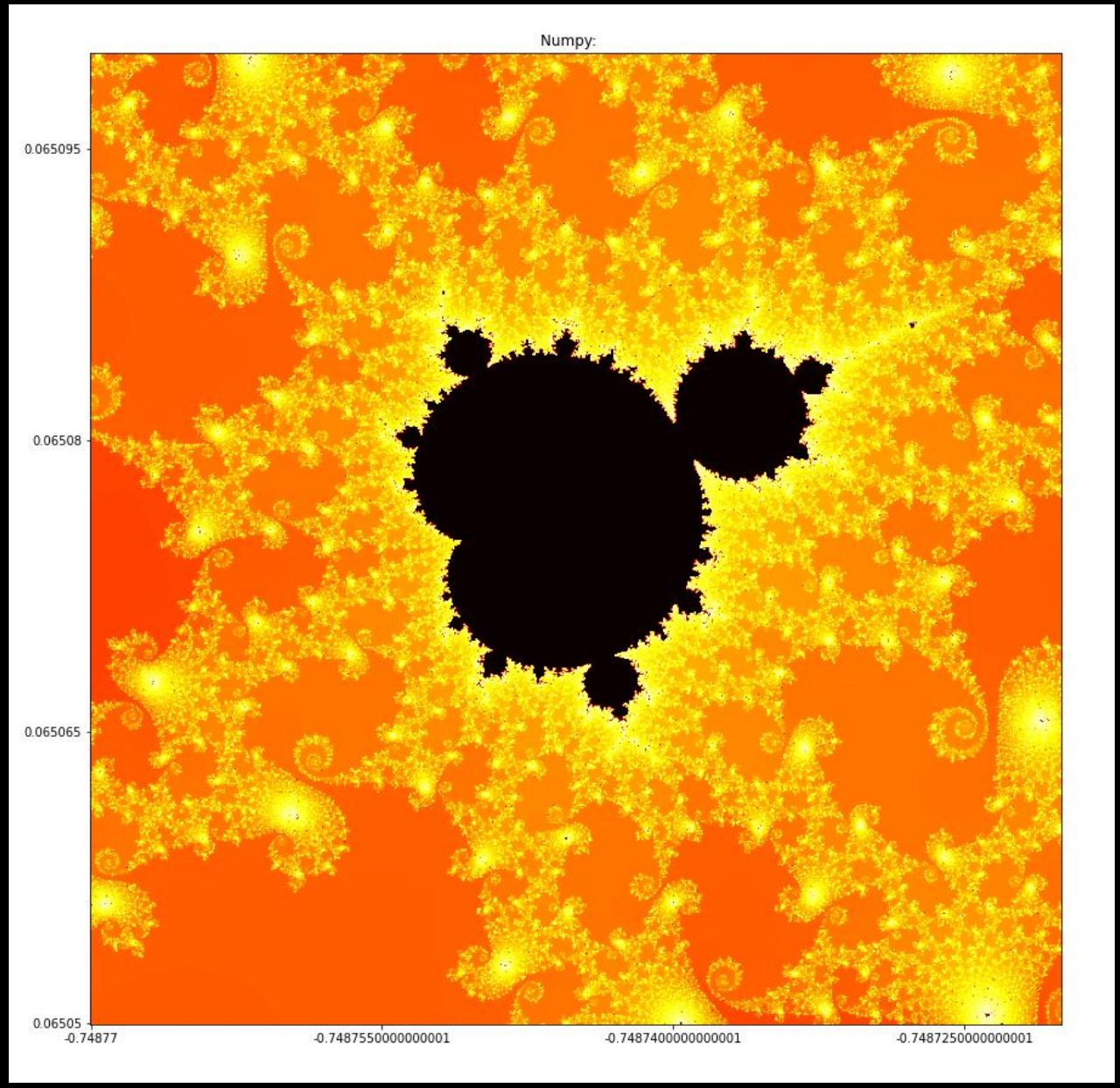
Visualizations

- Validation of the code's accuracy with the default input values
- Yay ☺



Visualizations (Cont.)

- You must Google interesting input values to get prettier visualizations
- Just trying random numbers often provides boring visualizations



Better NumPy - .85 seconds

- The biggest optimization here is in redoing the check *if abs(z) > 2*
  - Python's *abs()* function has overhead and is very slow
- Recall, this is the divergence check: when the function is greater than 2 from the origin, it diverges to infinity

- We can replace this check with:

$$\text{if } x^2 + y^2 > 4$$

- This comes from the Pythagorean theorem
- We optimize it further by using NumPy's *less()* function in place of standard > or <

```python
import numpy as np

def mandelbrot_numpy_better(c, maxIterations):
  escapeCount = np.zeros(c.shape)
  z = np.zeros(c.shape, np.complex64)

  for iteration in range(maxIterations):
    notFinished = np.less(z.real * z.real + z.imag * z.imag, 4.0)
    escapeCount[notFinished] = iteration
    z[notFinished] = z[notFinished] * z[notFinished] + c[notFinished]

  escapeCount[escapeCount == maxIterations - 1] = 0
  return escapeCount

def mandelbrot_set_numpy_better(xMin, xMax, yMin, yMax, width, height, maxIterations):
  realNums = np.linspace(xMin, xMax, width, dtype = np.float32)
  imagNums = np.linspace(yMin, yMax, height, dtype = np.float32)
  complexNums = realNums + imagNums[:, None] * 1j
  escapeCounts = mandelbrot_numpy_better(complexNums, maxIterations)
  return (realNums, imagNums, escapeCounts.T)
```

Numba - .24 seconds

- Essentially the same as the pure Python code
- Uses the Pythagorean theorem trick
- Just adding the *@jit* decorator makes the code faster than anything thus far

```
import numpy as np
from numba import jit

@jit
def mandelbrot_numba(c, maxIterations):
    z = 0
    for n in range(maxIterations):
        if z.real * z.real + z.imag * z.imag > 4.0:
            return n
        z = z * z + c
    return 0

@jit
def mandelbrot_set_numba(xMin, xMax, yMin, yMax, width, height, maxIterations):
    realNums = np.linspace(xMin, xMax, width)    # Declaring datatypes here breaks the code :)
    imagNums = np.linspace(yMin, yMax, height)   # ^^^
    escapeCounts = np.empty((width, height))
    for i in range(width):
        for j in range(height):
            escapeCounts[i, j] = mandelbrot_numba(realNums[i] + imagNums[j] * 1j, maxIterations)
    return (realNums, imagNums, escapeCounts)
```

Better Numba - .14 seconds

- Breaks up the complex into its real and imaginary parts
  - Makes computing $z = z * z + c$ faster
- Consider a complex number, $a + bi$
  - Squaring it produces

$$(a^2 - b^2) + (2ab)i$$

  - To calculate the next *z*:

$$(a^2 - b^2) \equiv realNew - imagNew + cReal$$

$$(2ab)i \equiv 2 * real * imag + cImag$$

```python
import numpy as np
from numba import jit

@jit
def mandelbrot_numba_better(cReal, cImag, maxIterations):
    real = cReal
    imag = cImag
    for n in range(maxIterations):
        realNew = real * real
        imagNew = imag * imag
        if realNew + imagNew > 4.0:
            return n
        imag = 2 * real * imag + cImag
        real = realNew - imagNew + cReal
    return 0

@jit
def mandelbrot_set_numba_better(xMin, xMax, yMin, yMax, width, height, maxIterations):
    realNums = np.linspace(xMin, xMax, width)    # Declaring datatypes here breaks the code :)
    imagNums = np.linspace(yMin, yMax, height)   # ^^^
    complexNums = np.empty((width, height))
    for i in range(width):
        for j in range(height):
            complexNums[i, j] = mandelbrot_numba_better(realNums[i], imagNums[j], maxIterations)
    return (realNums, imagNums, complexNums)
```

Guvectorized Numba - .01 seconds

- The *@guvectorize* decorator is Numba's generalized universal function decorator, built specifically for parallelizing multi-dimensional array operations
  - I.e., it's exactly what we want
- Made a wrapper function because *@guvectorize* only takes arrays as input
  - Everything else is the same as the last Numba code
- Recall, the pure Python code took over 5 seconds
  - Numba magic ☺

```python
import numpy as np
from numba import jit, vectorize, guvectorize, complex64, int32

@jit(int32(complex64, int32))
def mandelbrot_numba_betterer3(c, maxIterations):
  realNew = 0
  real  = 0
  imag  = 0
  for n in range(maxIterations):
    realNew = real * real - imag * imag + c.real
    imag = 2 * real * imag + c.imag
    real = realNew
    if real * real + imag * imag > 4.0:
      return n
  return 0

@guvectorize([(complex64[:], int32[:], int32[:])], '(n),()->(n)', target = 'parallel')
def mandelbrot_numba_betterer2(c, maxIterations, output):
  maxIterationsTemp = maxIterations[0]
  for i in range(c.shape[0]):
    output[i] = mandelbrot_numba_betterer3(c[i], maxIterationsTemp)

def mandelbrot_set_numba_betterer(xMin, xMax, yMin, yMax, width, height, maxIterations):
  realNums = np.linspace(xMin, xMax, width, dtype = np.float32)
  imagNums = np.linspace(yMin, yMax, height, dtype = np.float32)
  complexNums = realNums + imagNums[:, None] * 1j
  escapeCounts = mandelbrot_numba_betterer2(complexNums, maxIterations)
  return (realNums, imagNums, escapeCounts.T)
```

Numexpr - .45 seconds

- Similar to the NumPy code
- Just uses Numexpr
- Decently fast

```python
import numpy as np, numexpr as ne

def mandelbrot_numexpr(c, maxIterations):
  escapeCount = np.zeros(c.shape)
  z = np.zeros(c.shape, np.complex64)

  for iteration in range(maxIterations):
    notFinished = ne.evaluate('z.real * z.real + z.imag * z.imag < 4.0')
    escapeCount[notFinished] = iteration
    z = ne.evaluate('where(notFinished,z ** 2 + c, z)')

  escapeCount[escapeCount == maxIterations - 1] = 0
  return escapeCount

def mandelbrot_set_numexpr(xMin, xMax, yMin, yMax, width, height, maxIterations):
  realNums = np.linspace(xMin, xMax, width, dtype = np.float32)
  imagNums = np.linspace(yMin, yMax, height, dtype = np.float32)
  complexNums  = np.ravel(realNums + imagNums[:,None] * 1j)
  escapeCounts  = mandelbrot_numexpr(complexNums, maxIterations)
  escapeCounts  = escapeCounts.reshape((width, height))
  return (realNums, imagNums, escapeCounts.T)
```

Cython - .06 seconds

- Typical Cython code
- Extremely fast with minimal effort
- Just takes a mapping of the Python code to *cdefs*

- Not parallelized (e.g., with *prange*)
  - Kept getting classic *GIL* errors
- Could definitely be made much faster

```
import cython
import numpy as np

cdef int mandelbrot_cython_func(double cReal, double cImaginary, int maxIterations):
  cdef double real = cReal
  cdef double imaginary = cImaginary
  cdef double real2
  cdef double imaginary2
  cdef int n

  for n in range(maxIterations):
    real2 = real * real
    imaginary2 = imaginary * imaginary
    if real2 + imaginary2 > 4.0:
      return n
    imaginary = 2 * real * imaginary + cImaginary
    real = real2 - imaginary2 + cReal;
  return 0

@cython.cdivision(True)
@cython.nonecheck(False)
@cython.boundscheck(False)
@cython.wraparound(False)
cpdef mandelbrot_set_cython_func(double xMin, double xMax, double yMin, double yMax, int width, int height, int maxIterations):
  cdef double[:] realNums = np.linspace(xMin, xMax, width)
  cdef double[:] imagNums = np.linspace(yMin, yMax, height)
  cdef int[:,:] escapeCounts = np.empty((width, height), np.int)
  cdef int i, j

  for i in range(width):
    for j in range(height):
      escapeCounts[i, j] = mandelbrot_cython_func(realNums[i], imagNums[j], maxIterations)

  return (realNums, imagNums, escapeCounts)
```

Multiprocessing – 1.34 seconds

- All the codes thus far use a double for loop to generate complex numbers to be checked
- These complex numbers can be calculated independently of one another
  - I.e., it is an embarrassingly parallel array population problem
  - And the *parallel map* function can be used to aggregate all the results for every complex number

```
escapeCounts = []
for r in realNums:
  for i in imagNums:
    escapeCounts.append(mandelbrot_purepython(complex(r, i), maxIterations))
```

became...

```
p = Pool()
complexNums = [complex(real, imag) for imag in imagNums for real in realNums]
escapeCounts = p.map(mandelbrot, complexNums)
escapeCounts = np.reshape(escapeCounts, (width, height))
```

```
def mandelbrot(c):
  z = 0
  for n in range(maxIterations):
    if abs(z) > 2:
      return n
    z = z * z + c
  return maxIterations

if __name__ == '__main__':
  time1 = time.time()
  realNums = np.linspace(xMin, xMax, width)
  imagNums = np.linspace(yMin, yMax, height)

  p = Pool()
  complexNums = [complex(real, imag) for imag in imagNums for real in realNums]
  escapeCounts = p.map(mandelbrot, complexNums)
  escapeCounts = np.reshape(escapeCounts, (width, height))
```

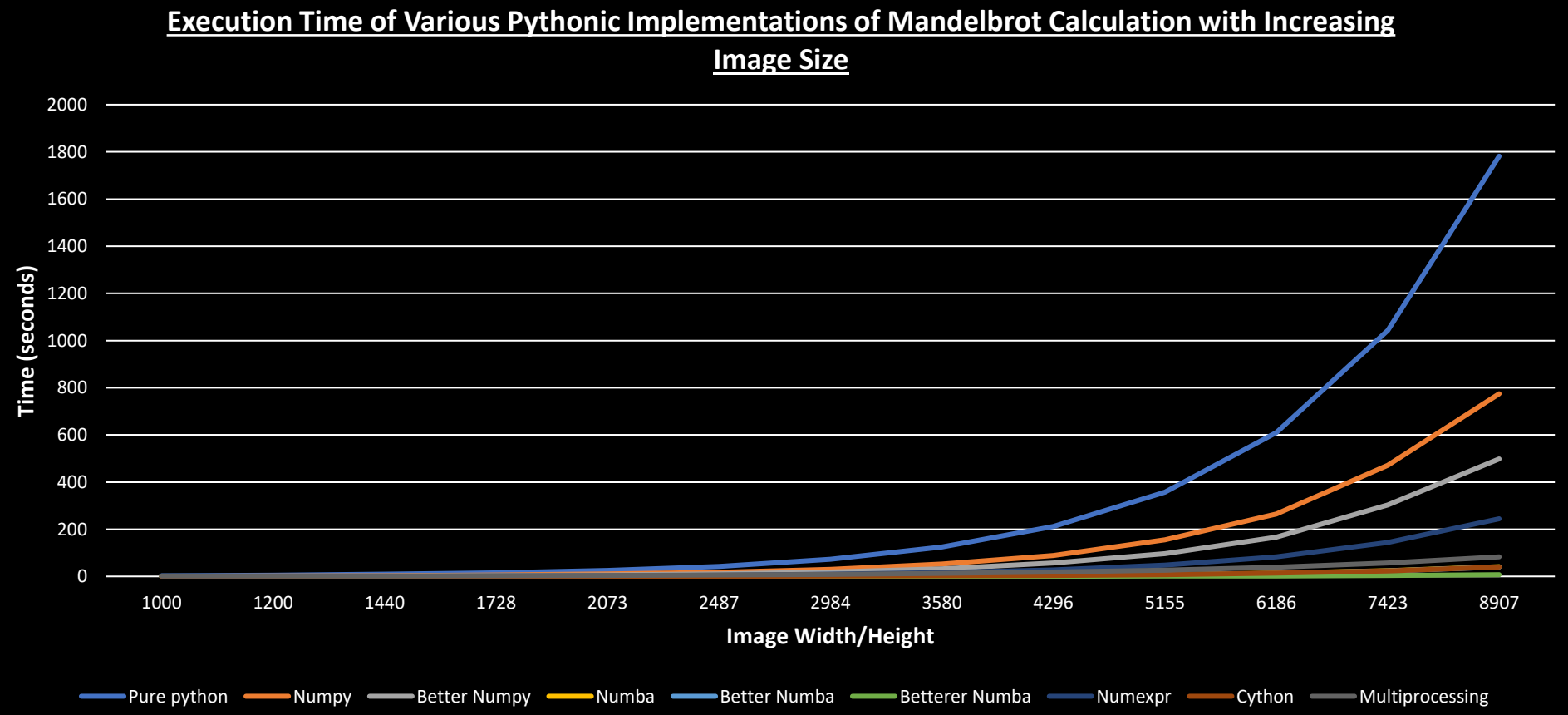
Initial Timings

- Using my test suite, each technique was run 5x to get an average execution time instead of just one, to be more representative
- Guvectorized Numba took .28% of the time as the pure Python code, a speedup of 360x

Technique	Initial Timing (seconds)	% decrease from original	x speedup from original
Pure Python	3.6	-	
Numpy	1.23	65.83%	2.93x
Better Numpy	.85	76.39%	4.24x
Numba	.24	93.33%	15x
Better Numba	.14	96.11%	25.71x
Betterer (guvectorized) Numba	.01	99.72%	360x
Numexpr	.45	87.5%	8x
Cython	.06	98.33%	60x
Multiprocessing	1.34	62.78%	2.69x

Scalability

- Wrote some code to iteratively increase image size to see how the different methods scale
- Took 30 minutes each run so I had to stop eventually ☺



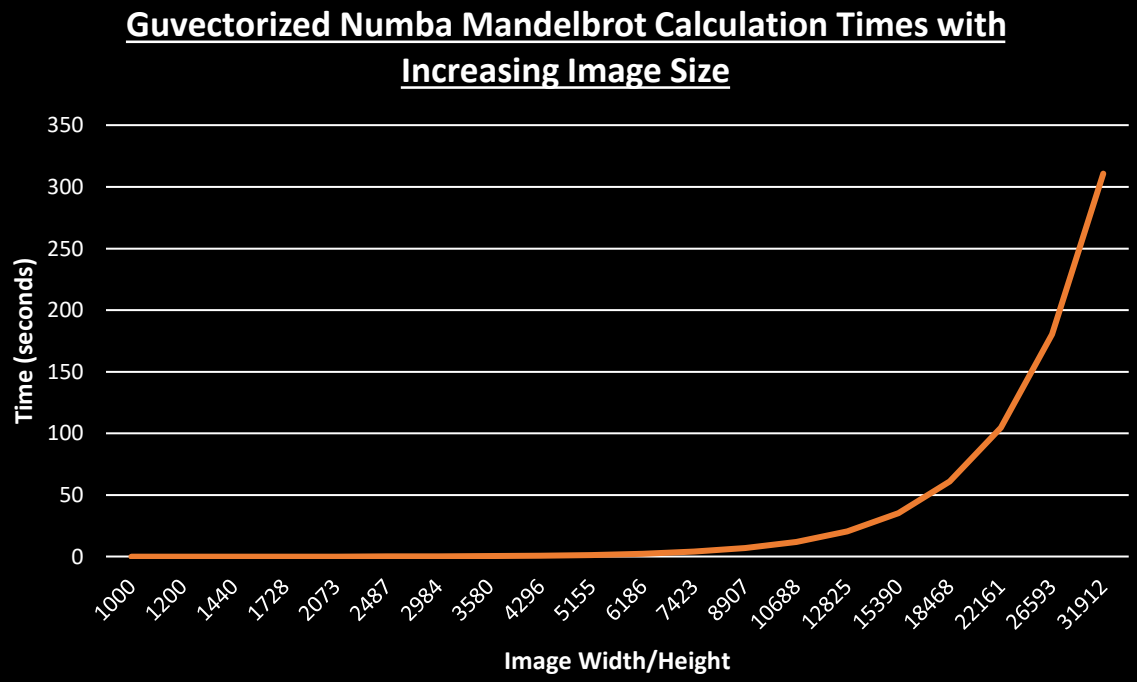
Scalability (Cont.)

- Pure Python is slowest in all cases
- Guvectorized Numba is best in all cases
- Cython did very well in all cases
- Multiprocessing began to do better with larger images
  - Parallelization scales well, it seems

Technique	Timing in seconds (smallest input value)	Technique	Timing in seconds (largest input value)
Pure Python	3.46	Pure Python	1781.76
Multiprocessing	1.34	Numpy	773.96
Numpy	1.24	Better Numpy	497.83
Better Numpy	.77	Numexpr	243.92
Numexpr	.45	Multiprocessing	83.27
Numba	.31	Numba	41.07
Better Numba	.18	Cython	40.20
Cython	.07	Better Numba	39.97
Guvectorized Numba	.01	Guvectorized Numba	6.99

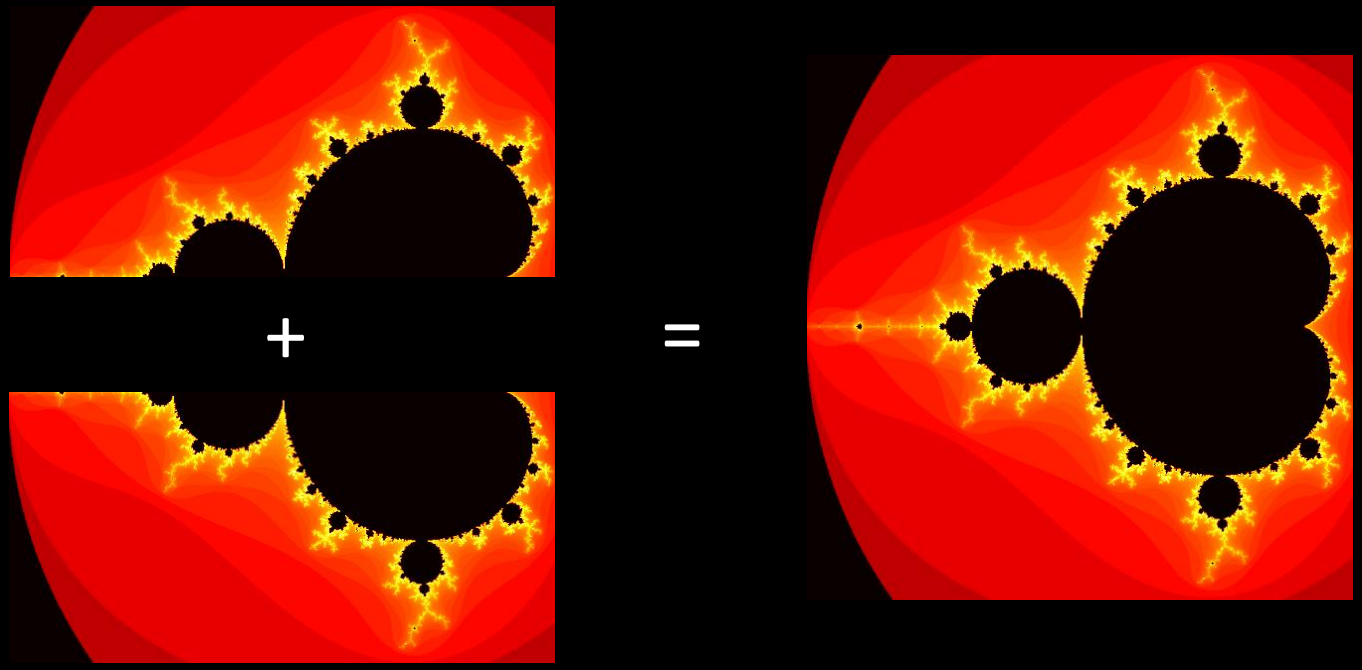
Guvectorized Numba

- With an image of 32,000x32,000 pixels, guvectorized Numba only took 6 minutes
  - This is *1-billion* complex numbers
  - Recall, Pure Python took 6 minutes for a 5000x5000 image
- We get the benefit of an embarrassingly parallel array-heavy problem coupled with a JIT compiler
- Guvectorized Numba is incredibly fast
  - It beat every other optimization technique across every image size



Biggest Future Optimization

- Split horizontally, the Mandelbrot set is *symmetrical*
- I.e., we only need to compute *half* the Mandelbrot numbers, then mirror them, effectively cutting our number of computations in half!
- In code, compute the array of complexes like normal, but stop halfway through, then flip the array and append it to itself



Conclusion

- Computer specs:
  - OS: Windows 10
  - CPU: i9-9900k @5.0GHz
  - GPU: RTX 3080 10GB GDDR6
  - RAM: 32GB DDR4 @3200MHz

- Pretty fun problem with a nice visualization to check accuracy with
- Embarrassingly parallel problem, but with many low-level optimizations that were done
- A lot of potential future work
  - E.g., Pythran/PyPy Interpreter, PyCUDA, PyOpenCL, Tensorflow, etc.

- In the end, parallel Numba was the simplest and easiest
  - Basically, just adding decorators and adhering to function input templates
    - No significant change of code
  - JIT compiler + automatic parallelization on large, independent array operations = blazing speed

- Thank you ☺