Joshua Caudill

CS6068

Assignment #2

Table of Contents

[Introduction 4](#_Toc176630992)

[Tools 4](#_Toc176630993)

[Code 4](#_Toc176630994)

[Results 6](#_Toc176630995)

[Conclusion 17](#_Toc176630996)

Table of Figures

[Figure 1: Parallel, (24, 1024), 20 Iterations 7](#_Toc176630997)

[Figure 2: Parallel, (1024, 2024), 10 Iterations 8](#_Toc176630998)

[Figure 3: Parallel, (1024, 2024), 20 Iterations 9](#_Toc176630999)

[Figure 4: Parallel, (1024, 2024), 40 Iterations 10](#_Toc176631000)

[Figure 5: Parallel, (2024, 3024), 20 Iterations 11](#_Toc176631001)

[Figure 6: Sequential, (24, 1024), 20 Iterations 12](#_Toc176631002)

[Figure 7: Sequential, (1024, 2024), 10 Iterations 13](#_Toc176631003)

[Figure 8: Sequential, (1024, 2024), 20 Iterations 14](#_Toc176631004)

[Figure 9: Sequential, (1024, 2024), 40 Iterations 15](#_Toc176631005)

[Figure 10: Sequential, (2024, 3024), 20 Iterations 16](#_Toc176631006)

# Introduction

The goal of this assignment was to demonstrate the speedup of a Python script that generated the Mandelbrot Set for an image. The Ohio Supercomputer Center (OSC) was used to create an environment for demonstrating the speedup achieved by using Numba. Numba is an open-source, just-in-time compiler used to speed up Python scripts. Using the @jit decorator, Numba translates Python code into bytecode optimized for the given environment. The Python script used in this assignment was a good candidate for using Numba since it looped through NumPy arrays. Numba is especially good at generating efficient compiled code for scripts that use NumPy.

# Tools

* generate\_mandelbrot\_set.py
* time, numpy, numba.jit, pylab.imshow, pylab.show
* Pitzer Desktop (1 GPU, 48 Cores, 1 Visualization Node)
* Python 3.6.6 :: Anaconda Custom (64-bit)

# Code

"""

Starter code for the fast generation of the Mandelbrot Set.

"""

import time

import numpy as np

from numba import jit

from pylab import imshow, show

@jit(nopython=True)

def mandel(x, y, max\_iters):

    """

    Computes the behavior of '0' under max\_iters iterations for the value c.

    """

    c = complex(

        x, y

    )  # Generate complex number, c, given real and imaginary components.

    z = 0.0j

    for i in range(max\_iters):

        z = z \* z + c  # Compute the function.

        if (

            z.real \* z.real + z.imag \* z.imag

        ) >= 4:  # Return iteration value if z becomes larger than 4.

            return i

    return max\_iters  # Return max\_iters otherwise.

@jit(nopython=True)

def create\_fractal(min\_x, max\_x, min\_y, max\_y, img, iters):

    """

    The Mandelbrot Set is a fractal. Create the fractal.

    """

    height = img.shape[0]

    width = img.shape[1]

    # Calculate pixel sizes.

    pixel\_size\_x = (max\_x - min\_x) / width

    pixel\_size\_y = (max\_y - min\_y) / height

    for x in range(width):

        real = min\_x + x \* pixel\_size\_x

        for y in range(height):

            imag = min\_y + y \* pixel\_size\_y

            color = mandel(real, imag, iters)

            img[y, x] = color  # Assign a color to the image.

if \_\_name\_\_ == "\_\_main\_\_":

    image = np.zeros((1024, 2024), dtype=np.uint8)  # Generate the image.

    start = time.time()  # Get the start time.

    create\_fractal(-2.0, -1.7, -0.1, 0.1, image, 20)

    end = time.time()  # Get the end time.

    print(f"Elapsed = {(end - start)}")  # Print the elapsed time.

    imshow(image)

    show()  # Show the Mandelbrot Set.

# Results

The Python script, generate\_mandelbrot\_set.py, was executed in both sequential and parallel fashion. The Python script was executed in a sequential fashion with the @jit decorators commented, and it was executed in a parallel fashion with the @jit decorators uncommented. The following image shapes were tested with 20 iterations: (24, 1024), (1024, 2024), and (2024, 3024). The following number of iterations were tested with an image shape of (1024, 2024): 10 iterations, 20 iterations, and 40 iterations. The speedups shown below were calculated. The screenshots shown below were taken from OSC.

* + Not an error. Strange behavior for low number of rows.

A screenshot of a computer

Description automatically generated

Figure 1: Parallel, (24, 1024), 20 Iterations

A screenshot of a computer

Description automatically generated

Figure 2: Parallel, (1024, 2024), 10 Iterations

A screenshot of a computer

Description automatically generated

Figure 3: Parallel, (1024, 2024), 20 Iterations

A screenshot of a computer

Description automatically generated

Figure 4: Parallel, (1024, 2024), 40 Iterations

A screenshot of a computer

Description automatically generated

Figure 5: Parallel, (2024, 3024), 20 Iterations

A screenshot of a computer

Description automatically generated

Figure 6: Sequential, (24, 1024), 20 Iterations

A screenshot of a computer screen

Description automatically generated

Figure 7: Sequential, (1024, 2024), 10 Iterations

A screenshot of a computer

Description automatically generated

Figure 8: Sequential, (1024, 2024), 20 Iterations

A screenshot of a computer

Description automatically generated

Figure 9: Sequential, (1024, 2024), 40 Iterations

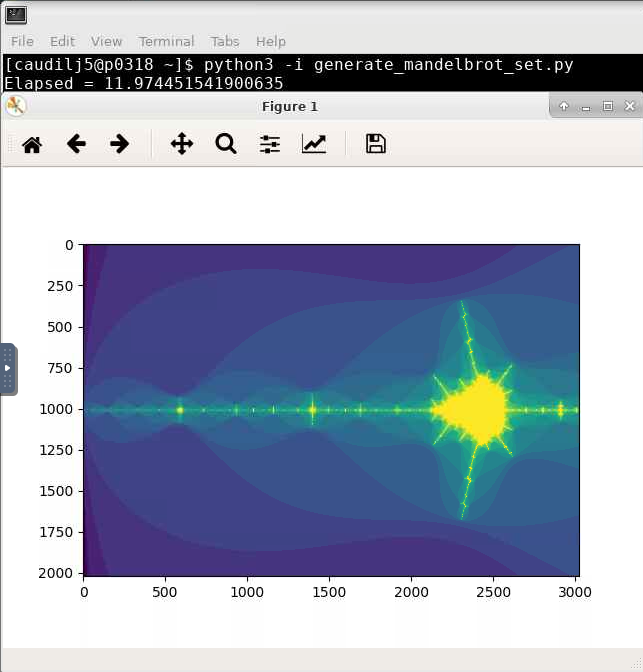


Figure 10: Sequential, (2024, 3024), 20 Iterations

# Conclusion

Numba was used to speed up execution of generate\_mandelbrot\_set.py, which generated the Mandelbrot Set for an image. Using the @jit decorator resulted in a speedup in four out of five tests. In general, the speedup was larger for tests that performed more work (e.g., larger image size).