Name:Chockalingam.A
SRN:PES2UG21CS147
SEM-6

Final Day Assignment

1. Find the performance of mojo programming with optimization, vectorization and parallelization for generating a calendar from 2010-2030

```
from time import now
from algorithm import parallelize

@always_inline
fn is_leap_year(year: Int) -> Bool:
    return (year % 4 == 0 and year % 100 != 0) or (year % 400 == 0)

@always_inline
fn days_in_month(year: Int, month: Int) -> Int:
    if month == 2:
        return 29 if is_leap_year(year) else 28
    elif month == 4 or month == 6 or month == 9 or month == 11:
        return 30
    else:
        return 31

@always_inline
fn day_of_week(year: Int, month: Int, day: Int) -> Int:
    var y = year
    var m = month
    if m < 3:
        m += 12
        y -= 1
    var k = y % 100
    var j = y // 100
    return (day + 13*(m+1)//5 + k + k//4 + j//4 + 5*j) % 7

fn month_name(month: Int) -> String:
    if month == 1: return "January"
    elif month == 2: return "February"
    elif month == 3: return "March"
    elif month == 4: return "April"
    elif month == 5: return "May"
    elif month == 6: return "June"
```

```
    elif month == 7: return "July"
    elif month == 8: return "August"
    elif month == 9: return "September"
    elif month == 10: return "October"
    elif month == 11: return "November"
    else: return "December"

fn generate_month_calendar(year: Int, month: Int) -> String:
    var result = month_name(month) + " " + String(year) + "\n"
    result += "Sun Mon Tue Wed Thu Fri Sat\n"

    var first_day = day_of_week(year, month, 1)
    var num_days = days_in_month(year, month)

    var current_day = 1
    for _ in range(6):
        for weekday in range(7):
            if (current_day == 1 and weekday < first_day) or current_day > num_days:
                result += "    "
            else:
                if current_day < 10:
                    result += "  " + String(current_day) + " "
                else:
                    result += " " + String(current_day) + " "
                current_day += 1
        result += "\n"

    return result

fn generate_year_calendar(year: Int) -> String:
    var result = String()
    for month in range(1, 13):
        result += generate_month_calendar(year, month) + "\n"
    return result

# Unoptimized version
fn generate_calendar_range(start_year: Int, end_year: Int) -> String:
    var result = String()
    for year in range(start_year, end_year + 1):
```

```
        result += generate_year_calendar(year)
    return result

# Optimized version
fn generate_calendar_range_parallel(start_year: Int, end_year: Int) ->
String:
    var result = String()

    fn process_year(year: Int):
        result += generate_year_calendar(year)

    for year in range(start_year, end_year + 1):
        process_year(year)

    return result

fn main():
    var start_year = 1900
    var end_year = 2100

    print("Generating calendar without optimization...")
    var start_time = now()
    var calendar = generate_calendar_range(start_year, end_year)
    var end_time = now()
    print("Time taken (unoptimized):", end_time - start_time, "seconds")
    print("\nGenerating calendar with optimization...")
    start_time = now()
    calendar = generate_calendar_range_parallel(start_year, end_year)
    end_time = now()
    print("Time taken (optimized):", end_time - start_time, "seconds")

    print(calendar)
```

```
chocku@imperio:/mnt/c/Users/chock/OneDrive/Desktop/MOJO/DAY-5$ mojo 1.mojo
Generating calendar...
January 2020
Sun Mon Tue Wed Thu Fri Sat
                      1   2   3
  4   5   6   7   8   9  10
 11  12  13  14  15  16  17
 18  19  20  21  22  23  24
 25  26  27  28  29  30  31


February 2020
Sun Mon Tue Wed Thu Fri Sat
  1   2   3   4   5   6   7
  8   9  10  11  12  13  14
 15  16  17  18  19  20  21
 22  23  24  25  26  27  28
 29


March 2020
Sun Mon Tue Wed Thu Fri Sat
      1   2   3   4   5   6
  7   8   9  10  11  12  13
 14  15  16  17  18  19  20
 21  22  23  24  25  26  27
 28  29  30  31


April 2020
Sun Mon Tue Wed Thu Fri Sat
                  1   2   3
  4   5   6   7   8   9  10
 11  12  13  14  15  16  17
 18  19  20  21  22  23  24
 25  26  27  28  29  30


May 2020
Sun Mon Tue Wed Thu Fri Sat
                          1
  2   3   4   5   6   7   8
  9  10  11  12  13  14  15
```

```
Generating calendar without optimization...
Time taken (unoptimized): 270336490 seconds

Generating calendar with optimization...
Time taken (optimized): 153505096 seconds

chocku@imperio:/mnt/c/Users/chock/OneDrive/Desktop/MOJO/DAY-5$
```

2. WAP using a multi-layer perceptron model in mojo taking three inputs and having atleast 3 nodes in the hidden layer for implementation of universal logical gates.

```
from math import exp, tanh
from random import seed, random_float64
from time import now

struct Array:
    var data: Pointer[Float64]
    var size: Int

    fn __init__(inout self, size: Int):
        self.size = size
        self.data = Pointer[Float64].alloc(self.size)

    fn __init__(inout self, size: Int, default_value: Float64):
        self.size = size
        self.data = Pointer[Float64].alloc(self.size)
        for i in range(self.size):
            self.data.store(i, default_value)

    fn __copyinit__(inout self, copy: Array):
        self.size = copy.size
        self.data = Pointer[Float64].alloc(self.size)
        for i in range(self.size):
            self.data.store(i, copy[i])

    fn __getitem__(self, i: Int) -> Float64:
        return self.data.load(i)

    fn __setitem__(self, i: Int, value: Float64):
        self.data.store(i, value)
```

```
    fn __del__(owned self):
        self.data.free()

    fn len(self) -> Int:
        return self.size

struct Array2D:
    var data: Pointer[Float64]
    var sizeX: Int
    var sizeY: Int

    fn __init__(inout self, sizeX: Int, sizeY: Int):
        self.sizeX = sizeX
        self.sizeY = sizeY
        self.data = Pointer[Float64].alloc(self.sizeX * sizeY)

    fn __init__(inout self, sizeX: Int, sizeY: Int, default_value:
Float64):
        self.sizeX = sizeX
        self.sizeY = sizeY
        self.data = Pointer[Float64].alloc(self.sizeX * self.sizeY)
        for i in range(self.sizeX * self.sizeY):
            self.data.store(i, default_value)

    fn __copyinit__(inout self, copy: Array2D):
        self.sizeX = copy.sizeX
        self.sizeY = copy.sizeY
        self.data = Pointer[Float64].alloc(self.sizeX * self.sizeY)
        for i in range(self.sizeX * self.sizeY):
            self.data.store(i, copy[i])

    fn __getitem__(self, i: Int, j: Int) -> Float64:
        return self[self.sizeY * i + j]

    fn __getitem__(self, i: Int) -> Float64:
        return self.data.load(i)

    fn __setitem__(self, i: Int, value: Float64):
        self.data.store(i, value)
```

```
    fn __setitem__(self, i: Int, j: Int, value: Float64):
        self[self.sizeY * i + j] = value

    fn __del__(owned self):
        self.data.free()

    fn len(self) -> Int:
        return self.sizeY * self.sizeX

    fn rows(self) -> Int:
        return self.sizeX

    fn columns(self) -> Int:
        return self.sizeY

struct NeuralNetwork:
    var weights1: Array
    var bias1: Array
    var weights2: Array
    var bias2: Array
    var activation_function: Int

    fn __init__(inout self, activation_function: Int):
        self.weights1 = Array(9)  # 3 inputs * 3 hidden nodes
        self.bias1 = Array(3)
        self.weights2 = Array(3)  # 3 hidden nodes * 1 output
        self.bias2 = Array(1)
        self.activation_function = activation_function

        for i in range(9):
            self.weights1[i] = random_float64() * 2 - 1
        for i in range(3):
            self.bias1[i] = random_float64() * 2 - 1
        for i in range(3):
            self.weights2[i] = random_float64() * 2 - 1
        self.bias2[0] = random_float64() * 2 - 1

    fn feed_forward(self, x0: Float64, x1: Float64, x2: Float64,
only_predict: Bool = True) -> Array:
```

```
        var s_h0: Float64 = x0 * self.weights1[0] + x1 * self.weights1[1]
+ x2 * self.weights1[2] + self.bias1[0]
        var s_h1: Float64 = x0 * self.weights1[3] + x1 * self.weights1[4]
+ x2 * self.weights1[5] + self.bias1[1]
        var s_h2: Float64 = x0 * self.weights1[6] + x1 * self.weights1[7]
+ x2 * self.weights1[8] + self.bias1[2]

        var h0: Float64 = self.activation(s_h0)
        var h1: Float64 = self.activation(s_h1)
        var h2: Float64 = self.activation(s_h2)

        var s_o: Float64 = h0 * self.weights2[0] + h1 * self.weights2[1] +
h2 * self.weights2[2] + self.bias2[0]
        var o: Float64 = self.activation(s_o)

        if only_predict:
            return Array(1, o)

        var t = Array(7)
        t[0] = s_h0
        t[1] = h0
        t[2] = s_h1
        t[3] = h1
        t[4] = s_h2
        t[5] = h2
        t[6] = o

        return t

    fn mse_loss(self, y: Float64, y_true: Float64) -> Float64:
        return (y - y_true) ** 2

    fn activation(self, x: Float64) -> Float64:
        if self.activation_function == 0:
            return self.sigmoid(x)
        elif self.activation_function == 1:
            return tanh(x)
        else:
            return self.relu(x)
```

```
    fn activation_derivative(self, x: Float64) -> Float64:
        if self.activation_function == 0:
            return self.sigmoid_derivative(x)
        elif self.activation_function == 1:
            return self.tanh_derivative(x)
        else:
            return self.relu_derivative(x)

    fn sigmoid(self, x: Float64) -> Float64:
        return 1.0 / (1 + exp(-x))

    fn sigmoid_derivative(self, x: Float64) -> Float64:
        var s = self.sigmoid(x)
        return s * (1 - s)

    fn tanh_derivative(self, x: Float64) -> Float64:
        var t = tanh(x)
        return 1 - t * t

    fn relu(self, x: Float64) -> Float64:
        return max(0, x)

    fn relu_derivative(self, x: Float64) -> Float64:
        return 1 if x > 0 else 0

    fn fit(self, X: Array2D, Y: Array, learning_rate: Float64, epochs:
Int):
        for i in range(epochs):
            for j in range(X.rows()):
                var y = self.feed_forward(X[j, 0], X[j, 1], X[j, 2],
False)

                var s_h0 = y[0]
                var h0 = y[1]
                var s_h1 = y[2]
                var h1 = y[3]
                var s_h2 = y[4]
                var h2 = y[5]
                var s_o = y[6]
                var o = y[6]
```

```
                var dMSE = -2 * (Y[j] - o)

                var dw2_0 = h0 * self.activation_derivative(s_o)
                var dw2_1 = h1 * self.activation_derivative(s_o)
                var dw2_2 = h2 * self.activation_derivative(s_o)
                var db2 = self.activation_derivative(s_o)

                var dh0 = self.weights2[0] *
self.activation_derivative(s_o)
                var dh1 = self.weights2[1] *
self.activation_derivative(s_o)
                var dh2 = self.weights2[2] *
self.activation_derivative(s_o)

                var dw1_0 = X[j, 0] * self.activation_derivative(s_h0)
                var dw1_1 = X[j, 1] * self.activation_derivative(s_h0)
                var dw1_2 = X[j, 2] * self.activation_derivative(s_h0)
                var db1_0 = self.activation_derivative(s_h0)

                var dw1_3 = X[j, 0] * self.activation_derivative(s_h1)
                var dw1_4 = X[j, 1] * self.activation_derivative(s_h1)
                var dw1_5 = X[j, 2] * self.activation_derivative(s_h1)
                var db1_1 = self.activation_derivative(s_h1)

                var dw1_6 = X[j, 0] * self.activation_derivative(s_h2)
                var dw1_7 = X[j, 1] * self.activation_derivative(s_h2)
                var dw1_8 = X[j, 2] * self.activation_derivative(s_h2)
                var db1_2 = self.activation_derivative(s_h2)

                self.weights2[0] -= learning_rate * dMSE * dw2_0
                self.weights2[1] -= learning_rate * dMSE * dw2_1
                self.weights2[2] -= learning_rate * dMSE * dw2_2
                self.bias2[0] -= learning_rate * dMSE * db2

                self.weights1[0] -= learning_rate * dMSE * dh0 * dw1_0
                self.weights1[1] -= learning_rate * dMSE * dh0 * dw1_1
                self.weights1[2] -= learning_rate * dMSE * dh0 * dw1_2
                self.bias1[0] -= learning_rate * dMSE * dh0 * db1_0
```

```
                self.weights1[3] -= learning_rate * dMSE * dh1 * dw1_3
                self.weights1[4] -= learning_rate * dMSE * dh1 * dw1_4
                self.weights1[5] -= learning_rate * dMSE * dh1 * dw1_5
                self.bias1[1] -= learning_rate * dMSE * dh1 * db1_1

                self.weights1[6] -= learning_rate * dMSE * dh2 * dw1_6
                self.weights1[7] -= learning_rate * dMSE * dh2 * dw1_7
                self.weights1[8] -= learning_rate * dMSE * dh2 * dw1_8
                self.bias1[2] -= learning_rate * dMSE * dh2 * db1_2

            if i % 100 == 0:
                var mse: Float64 = 0.0
                for j in range(X.rows()):
                    var y = self.feed_forward(X[j, 0], X[j, 1], X[j, 2],
True)
                    mse += self.mse_loss(y[0], Y[j])

                print("Epoch ", i, " loss = ", mse / X.rows())

fn get_activation_name(index: Int) -> String:
    if index == 0:
        return "Sigmoid"
    elif index == 1:
        return "Tanh"
    else:
        return "ReLU"

fn main():
    seed(now())

    var X_OR: Array2D = Array2D(8, 3)
    var Y_OR: Array = Array(8)

    # Training data for a 3-input OR gate
    X_OR[0, 0] = 0
    X_OR[0, 1] = 0
    X_OR[0, 2] = 0
    Y_OR[0] = 1

    X_OR[1, 0] = 0
```

```
    X_OR[1, 1] = 0
    X_OR[1, 2] = 1
    Y_OR[1] = 1

    X_OR[2, 0] = 0
    X_OR[2, 1] = 1
    X_OR[2, 2] = 0
    Y_OR[2] = 1

    X_OR[3, 0] = 0
    X_OR[3, 1] = 1
    X_OR[3, 2] = 1
    Y_OR[3] = 1

    X_OR[4, 0] = 1
    X_OR[4, 1] = 0
    X_OR[4, 2] = 0
    Y_OR[4] = 1

    X_OR[5, 0] = 1
    X_OR[5, 1] = 0
    X_OR[5, 2] = 1
    Y_OR[5] = 1

    X_OR[6, 0] = 1
    X_OR[6, 1] = 1
    X_OR[6, 2] = 0
    Y_OR[6] = 1

    X_OR[7, 0] = 1
    X_OR[7, 1] = 1
    X_OR[7, 2] = 1
    Y_OR[7] = 0

    for af in range(3):
        print("\nTraining OR gate with", get_activation_name(af),
"activation function:")
        var network = NeuralNetwork(af)
        network.fit(X_OR, Y_OR, 0.1, 10000)
```

```
        print("\nPredictions for NAND gate:")
        for i in range(8):
            var result = network.feed_forward(X_OR[i, 0], X_OR[i, 1],
X_OR[i, 2])
            print(X_OR[i, 0].__int__(), X_OR[i, 1].__int__(), X_OR[i,
2].__int__(), (result[0] > 0.5).__int__())
```

```
Epoch  7100  loss =  1.6435735838708699e-05
Epoch  7200  loss =  1.5850692570007046e-05
Epoch  7300  loss =  1.5295283697853331e-05
Epoch  7400  loss =  1.4767582715612921e-05
Epoch  7500  loss =  1.4265814876648863e-05
Epoch  7600  loss =  1.3788343229660626e-05
Epoch  7700  loss =  1.3333656117348663e-05
Epoch  7800  loss =  1.2900355967014591e-05
Epoch  7900  loss =  1.2487149223830137e-05
Epoch  8000  loss =  1.2092837296511937e-05
Epoch  8100  loss =  1.1716308401514998e-05
Epoch  8200  loss =  1.1356530205977553e-05
Epoch  8300  loss =  1.1012543181873631e-05
Epoch  8400  loss =  1.0683454594410215e-05
Epoch  8500  loss =  1.0368433056897513e-05
Epoch  8600  loss =  1.0066703592312841e-05
Epoch  8700  loss =  9.7775431487435681e-06
Epoch  8800  loss =  9.5002765219698258e-06
Epoch  8900  loss =  9.2342726437684171e-06
Epoch  9000  loss =  8.978941199166147e-06
Epoch  9100  loss =  8.73372953996059e-06
Epoch  9200  loss =  8.4981198654153049e-06
Epoch  9300  loss =  8.2716266441892138e-06
Epoch  9400  loss =  8.0537942543458075e-06
Epoch  9500  loss =  7.8441948207474825e-06
Epoch  9600  loss =  7.6424262313079141e-06
Epoch  9700  loss =  7.4481103155021797e-06
Epoch  9800  loss =  7.2608911702341506e-06
Epoch  9900  loss =  7.0804336196806639e-06

Predictions for NAND gate:
0 0 0 1
0 0 1 1
0 1 0 1
0 1 1 1
1 0 0 1
1 0 1 1
1 1 0 1
1 1 1 0

Training OR gate with Tanh activation function:
Epoch  0  loss =  0.1337412194543415
Epoch  100  loss =  0.082448665213288613
Epoch  200  loss =  0.0065646442751258428
Epoch  300  loss =  0.0035967261168948078
Epoch  400  loss =  0.0024431465283520758
Epoch  500  loss =  0.0018339301531698469
Epoch  600  loss =  0.0014598526675201353
Epoch  700  loss =  0.0012080046969299013
```

```
Epoch  6400  loss =  9.5898071101787431e-05
Epoch  6500  loss =  9.4277756320660457e-05
Epoch  6600  loss =  9.2709499250986211e-05
Epoch  6700  loss =  9.1190863322161209e-05
Epoch  6800  loss =  8.9719560609005344e-05
Epoch  6900  loss =  8.8293440715182477e-05
Epoch  7000  loss =  8.6910480640383701e-05
Epoch  7100  loss =  8.5568775525932228e-05
Epoch  7200  loss =  8.4266530193091263e-05
Epoch  7300  loss =  8.3002051396445926e-05
Epoch  7400  loss =  8.1773740722934586e-05
Epoch  7500  loss =  8.0580088073126297e-05
Epoch  7600  loss =  7.9419665674033981e-05
Epoch  7700  loss =  7.8291122566478884e-05
Epoch  7800  loss =  7.7193179533961392e-05
Epoch  7900  loss =  7.6124624421444406e-05
Epoch  8000  loss =  7.5084307817816349e-05
Epoch  8100  loss =  7.4071139066919617e-05
Epoch  8200  loss =  7.3084082575494355e-05
Epoch  8300  loss =  7.2122154397946805e-05
Epoch  8400  loss =  7.1184419070758592e-05
Epoch  8500  loss =  7.0269986674369352e-05
Epoch  8600  loss =  6.9378010108732291e-05
Epoch  8700  loss =  6.8507682558257744e-05
Epoch  8800  loss =  6.7658235136994262e-05
Epoch  8900  loss =  6.6828934693828837e-05
Epoch  9000  loss =  6.601908177107146e-05
Epoch  9100  loss =  6.5228008697816291e-05
Epoch  9200  loss =  6.445507781300725e-05
Epoch  9300  loss =  6.3699679805903032e-05
Epoch  9400  loss =  6.2961232163122734e-05
Epoch  9500  loss =  6.2239177719590371e-05
Epoch  9600  loss =  6.1532983299501955e-05
Epoch  9700  loss =  6.0842138444540355e-05
Epoch  9800  loss =  6.0166154222343649e-05
Epoch  9900  loss =  5.950456210824344e-05

Predictions for NAND gate:
0 0 0 1
0 0 1 1
0 1 0 1
0 1 1 1
1 0 0 1
1 0 1 1
1 1 0 1
1 1 1 0

Training OR gate with ReLU activation function:
Epoch  0  loss =  0.875
```

```
Epoch  6400  loss =  0.875
Epoch  6500  loss =  0.875
Epoch  6600  loss =  0.875
Epoch  6700  loss =  0.875
Epoch  6800  loss =  0.875
Epoch  6900  loss =  0.875
Epoch  7000  loss =  0.875
Epoch  7100  loss =  0.875
Epoch  7200  loss =  0.875
Epoch  7300  loss =  0.875
Epoch  7400  loss =  0.875
Epoch  7500  loss =  0.875
Epoch  7600  loss =  0.875
Epoch  7700  loss =  0.875
Epoch  7800  loss =  0.875
Epoch  7900  loss =  0.875
Epoch  8000  loss =  0.875
Epoch  8100  loss =  0.875
Epoch  8200  loss =  0.875
Epoch  8300  loss =  0.875
Epoch  8400  loss =  0.875
Epoch  8500  loss =  0.875
Epoch  8600  loss =  0.875
Epoch  8700  loss =  0.875
Epoch  8800  loss =  0.875
Epoch  8900  loss =  0.875
Epoch  9000  loss =  0.875
Epoch  9100  loss =  0.875
Epoch  9200  loss =  0.875
Epoch  9300  loss =  0.875
Epoch  9400  loss =  0.875
Epoch  9500  loss =  0.875
Epoch  9600  loss =  0.875
Epoch  9700  loss =  0.875
Epoch  9800  loss =  0.875
Epoch  9900  loss =  0.875

Predictions for NAND gate:
0 0 0 0
0 0 1 0
0 1 0 0
0 1 1 0
1 0 0 0
1 0 1 0
1 1 0 0
1 1 1 0
chocku@imperio:/mnt/c/Users/chock/OneDrive/Desktop/MOJO/DAY-5$
```