```
# Jupyter Notebook for Part III: Code Project in Google Colab with GPU

# Install CuPy for GPU-accelerated operations
!pip install cupy-cuda11x

# Import necessary libraries
import hashlib
import os
from collections import deque
import cupy as cp  # GPU-accelerated NumPy-like library

# Utility function to create text files for testing
def create_test_files(num_files, prefix="test"):
    for i in range(1, num_files + 1):
        with open(f"{prefix}{i}.txt", "w") as f:
            f.write(f"Content of {prefix}{i}")

# Node class for Merkle Tree
class Node:
    def __init__(self, data):
        self.data = data
        self.left = None
        self.right = None

    def isLeaf(self):
        return (self.left is None) and (self.right is None)

# Merkle Tree class with GPU-accelerated hashing
class MerkleTree:
    def __init__(self, arr, hash_func=None):
        self.root = None
        self._merkleRoot = ''
        self.hash_func = hash_func if hash_func else self.__default_sha256
        self.makeTreeFromArray(arr)
        self.calculateMerkleRoot()

    def __default_sha256(self, x):
        # Fallback to CPU-based hashlib.sha256
        return hashlib.sha256(x.encode()).hexdigest()

    def __returnHash(self, x):
        return self.hash_func(x)

    def makeTreeFromArray(self, arr):
        arr = arr.copy()
        def __noOfNodesReqd(arr):
            return 2 * len(arr) - 1

        def __buildTree(arr, root, i, n):
            if i < n:
                temp = Node(str(arr[i]))
                root = temp
                root.left = __buildTree(arr, root.left, 2 * i + 1, n)
                root.right = __buildTree(arr, root.right, 2 * i + 2, n)
            return root

        def __addLeafData(arr, node):
            if not node:
                return
            __addLeafData(arr, node.left)
            if node.isLeaf():
                node.data = self.__returnHash(arr.pop(0))
            else:
                node.data = ''
            __addLeafData(arr, node.right)

        nodesReqd = __noOfNodesReqd(arr)
        nodeArr = [num for num in range(1, nodesReqd + 1)]
        self.root = __buildTree(nodeArr, None, 0, nodesReqd)
        __addLeafData(arr, self.root)

    def calculateMerkleRoot(self):
        def __merkleHash(node):
            if node.isLeaf():
                return node
            left = __merkleHash(node.left).data
            right = __merkleHash(node.right).data
            node.data = self.__returnHash(left + right)
            return node

        merkleRoot = __merkleHash(self.root)
        self._merkleRoot = merkleRoot.data
        return self._merkleRoot

    def getMerkleRoot(self):
        return self._merkleRoot

    def printTree(self):
        if not self.root:
            print("Empty tree.")
            return
        q = deque()
        q.append(self.root)
        level = 0
        while q:
            level_size = len(q)
            print(f"Level {level}: ", end='')
            for _ in range(level_size):
                node = q.popleft()
                print(f"{node.data[:10]} ", end='')
                if node.left:
                    q.append(node.left)
                if node.right:
                    q.append(node.right)
            print()
            level += 1

# Function to read file contents
def read_files(file_list):
    contents = []
    for file in file_list:
        with open(file, 'r') as f:
            contents.append(f.read())
    return contents

# Verify GPU availability
print("GPU Available:", cp.cuda.is_available())
if cp.cuda.is_available():
    print("GPU Device:", cp.cuda.runtime.getDeviceProperties(0)['name'])

# Part 1: Merkle Tree Implementation
print("=== Part 1: Merkle Tree Implementation ===")

# Test with 4 leaf nodes
print("\nTest with 4 Leaf Nodes:")
create_test_files(4)
files_4 = [f"test{i}.txt" for i in range(1, 5)]
contents_4 = read_files(files_4)
tree_4 = MerkleTree(contents_4)
tree_4.printTree()

# Test with 6 leaf nodes
print("\nTest with 6 Leaf Nodes:")
create_test_files(6)
files_6 = [f"test{i}.txt" for i in range(1, 7)]
contents_6 = read_files(files_6)
tree_6 = MerkleTree(contents_6)
tree_6.printTree()

# Part 2: Root Hash Observation
print("\n=== Part 2: Root Hash Observation ===")
original_root = tree_4.getMerkleRoot()
print(f"Original Root Hash (4 leaves): {original_root[:10]}")

# Modify one file
with open("test1.txt", "w") as f:
    f.write("Modified content of test1")
modified_contents_4 = read_files(files_4)
modified_tree_4 = MerkleTree(modified_contents_4)
modified_root = modified_tree_4.getMerkleRoot()
print(f"Modified Root Hash (4 leaves): {modified_root[:10]}")
print("Observation: Changing one file alters the root hash, but unaffected sibling nodes retain their hashes.")

# Part 3: Hash Collision with 4-bit Hash
# Part 3: Hash Collision with 4-bit hash
print("\n=== Part 3: Hash Collision ===")
def short_hash(x):
    full_hash = hashlib.sha256(x.encode()).hexdigest()
    return full_hash[:1]  # First 4 bits (1 hex char)

# Test Merkle Tree with 4-bit hash
tree_4bit = MerkleTree(contents_4, hash_func=short_hash)
print("Merkle Tree with 4-bit Hash:")
tree_4bit.printTree()

# Generate files to find collision (limited to 100 attempts to avoid freezing)
collision_files = []
hash_set = set()
count = 0
max_attempts = 100  # Prevent infinite loop
while len(collision_files) < 2 and count < max_attempts:
    text = f"Text {count}" + " " * (count % 5)
    h = short_hash(text)
    if h in hash_set and text not in [f[0] for f in collision_files]:
        collision_files.append((text, h))
    else:
        hash_set.add(h)
    count += 1
print(f"Collision search stopped after {count} attempts:")
for text, h in collision_files:
    print(f"Text: '{text}', Hash: {h}")
print("Discussion: With 4 bits (16 values), collisions occur quickly (~sqrt(16) = 4 attempts) due to the birthday paradox.")

# Strategies for larger hashes
print("Strategies for 4-bit to 160-bit: Birthday attack; expect ~2^(n/2) attempts. For 160-bit SHA-1, ~2^80 attempts, infeasible

# Part 4: Hash Puzzle
print("\n=== Part 4: Hash Puzzle ===")
def solve_puzzle(prefix_zeros, max_attempts=10000):
    target = "0" * prefix_zeros
    count = 0
    while count < max_attempts:
        text = f"Nonce {count}"
        h = short_hash(text)
        if h.startswith(target):
            print(f"Found hash with {prefix_zeros} leading zeros: {text} -> {h}")
            break
        count += 1
    return count if count < max_attempts else max_attempts

print("Solving for 1 leading zero bit:")
attempts_1 = solve_puzzle(1)
print(f"Attempts: {attempts_1}")

print("Solving for 2 leading zero bits:")
attempts_2 = solve_puzzle(2)
print(f"Attempts: {attempts_2}")

print("Workload for 20-bit prefix: Expected ~2^20 (~1M) attempts. For full SHA256, exponentially harder, computationally infeasib
```

```
Follow these steps to resolve this issue:

  1. For all packages listed above, run the following command to remove all
     existing CuPy installations:

       $ pip uninstall <package_name>

    If you previously installed CuPy via conda, also run the following:
```

```
Test with 4 Leaf Nodes:
Level 0: 9b027fac92
Level 1: 83a87ca911 cc8c3e4b2d
Level 2: e8eaf756ab c7cfc205fd 80ef3b783c b4331f9e17

Test with 6 Leaf Nodes:
Level 0: dd8ebf3e1d
Level 1: 9b027fac92 c31cc3b75d
Level 2: 83a87ca911 cc8c3e4b2d 0acb799075 141430aff5
Level 3: e8eaf756ab c7cfc205fd 80ef3b783c b4331f9e17

=== Part 2: Root Hash Observation ===
Original Root Hash (4 leaves): 9b027fac92
Modified Root Hash (4 leaves): 873babfa3d
Observation: Changing one file alters the root hash, but unaffected sibling nodes retain their hashes.

=== Part 3: Hash Collision ===
Merkle Tree with 4-bit Hash:
Level 0: 8
Level 1: 1 4
Level 2: e c 8 b
Collision search stopped after 10 attempts:
Text: 'Text 1 ', Hash: e
Text: 'Text 9    ', Hash: c
Discussion: With 4 bits (16 values), collisions occur quickly (~sqrt(16) = 4 attempts) due to the birthday paradox.
Strategies for 4-bit to 160-bit: Birthday attack; expect ~2^(n/2) attempts. For 160-bit SHA-1, ~2^80 attempts, infeasible to

=== Part 4: Hash Puzzle ===
Solving for 1 leading zero bit:
Found hash with 1 leading zeros: Nonce 36 -> 0
Attempts: 36
Solving for 2 leading zero bits:
Attempts: 10000
Workload for 20-bit prefix: Expected ~2^20 (~1M) attempts. For full SHA256, exponentially harder, computationally infeasible
```