

Challenge -5

1. I am choosing Matrix Multiplication, Quicksort, Differential equation solver

2. **Matrix Multiplication:**

```
import numpy as np

def matrix_multiply(A, B):
    print("Performing matrix multiplication...")
    print(f"Matrix A shape: {A.shape}")
    print(f"Matrix B shape: {B.shape}")
    result = np.dot(A, B)
    print("Matrix multiplication complete.")
    print(f"Result shape: {result.shape}")
    return result

print("Generating random matrices A and B...")
A = np.random.rand(100, 100)
B = np.random.rand(100, 100)

print("Calling matrix_multiply...")
C = matrix_multiply(A, B)

print("Result matrix C (first 5x5 block):")
print(C[:5, :5]) # Prints the top-left 5x5 portion of the matrix for readability

print("Done.")
```

Quick sort:

```
def quicksort(arr):
    if len(arr) <= 1:
        return arr
    pivot = arr[len(arr) // 2] # Choose middle element as pivot
    left = [x for x in arr if x < pivot]
    middle = [x for x in arr if x == pivot]
    right = [x for x in arr if x > pivot]
    return quicksort(left) + middle + quicksort(right)

# Example usage
import random
arr = random.sample(range(1000), 100)
sorted_arr = quicksort(arr)
print("Sorted:", sorted_arr)
```

Differential equation solver:

```
import numpy as np
import matplotlib.pyplot as plt
from scipy.integrate import odeint

# Define the differential equation: dy/dt = -2y
def model(y, t):
    dydt = -2 * y
    return dydt

# Initial condition: y(0) = 1
y0 = 1

# Time points where the solution is computed
t = np.linspace(0, 5, 100)

# Solve the ODE
y = odeint(model, y0, t)

# Plot the result
plt.plot(t, y)
plt.xlabel('Time t')
plt.ylabel('Solution y(t)')
plt.title('Solution of dy/dt = -2y')
plt.grid()
plt.show()
```

3.Compliation:**a. Matrix multiplication:**

```
Generating random matrices A and B...
Calling matrix_multiply...
Performing matrix multiplication...
Matrix A shape: (100, 100)
Matrix B shape: (100, 100)
Matrix multiplication complete.
Result shape: (100, 100)
Result matrix C (first 5x5 block):
[[22.86773753 23.46546338 26.95762851 25.74155548 22.94711322]
 [22.92489314 24.68416506 26.83901257 25.35602751 22.60584369]
 [23.73972905 22.98817195 26.11732993 23.82873141 23.96295266]
 [23.73150105 23.99520715 26.53622371 23.81288184 24.35048666]
 [24.14190228 23.12646665 27.10865456 24.16882127 24.27815307]]
Done.
```

Byte code:

```
200 RETURN_VALUE
[13]: import py_compile

# Compile the Python file
py_compile.compile('matrix_multiply.ipynb')

[13]: '__pycache__/_matrix_multiply.cpython-312.pyc'
```

b. Quick sort:

```
Sorted: [8, 11, 26, 27, 50, 62, 65, 72, 76, 89, 95, 97, 119, 131, 145, 196, 209, 212, 216, 246, 259, 261, 264, 279, 284, 292, 295, 303, 310, 325, 329, 336, 338, 344, 354, 365, 379, 380, 397, 416, 425, 444, 450, 455, 467, 470, 473, 475, 482, 483, 486, 489, 492, 499, 501, 517, 546, 561, 565, 577, 581, 591, 602, 631, 639, 655, 657, 665, 668, 678, 680, 683, 685, 696, 705, 708, 712, 740, 741, 742, 751, 756, 773, 797, 802, 852, 871, 879, 884, 913, 927, 932, 939, 940, 949, 955, 981, 983, 991]
```

Byte Code

```
|: import py_compile

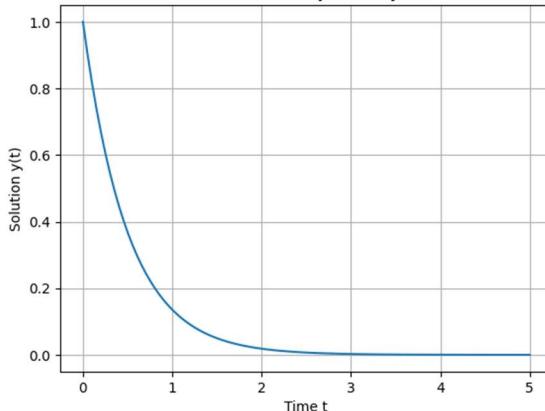
# Compile the Python file
py_compile.compile('quicksort.ipynb')

|: '__pycache__/_quicksort.cpython-312.pyc'
```

c. Differential equation solver

Matplotlib is building the font cache; this may take a moment.

Solution of $dy/dt = -2y$



Byte Code:

```
import py_compile

# Compile the Python file
py_compile.compile('model.ipynb')

'__pycache__/model.cpython-312.pyc'
```

4. Disassembling the byte code

a. Matrix_multiplication:

```
import dis
import marshal

# Path to the .pyc file (ensure correct Python version in the filename, e.g., cp312 for Python 3.12)
compiled_file = '__pycache__/_matrix_multiply.cpython-312.pyc' # Replace '312' with your Python version

# Read the .pyc file
with open(compiled_file, 'rb') as f:
    f.seek(16) # Skip the header (magic number and timestamp)
    bytecode = f.read()

# Deserialize the bytecode to get the code object
code_object = marshal.loads(bytecode)

# Disassemble the code object
dis.dis(code_object)

  0      0 RESUME          0
  1      2 LOAD_CONST       0 ('python')
  4      4 LOAD_CONST       1 ('Python (Pydide)')
  6      6 LOAD_CONST       0 ('python')
  8      8 LOAD_CONST       2 (('name', 'display_name', 'language'))
 10     10 BUILD_CONST_KEY_MAP
 12     12 LOAD_CONST       0 ('python')
 14     14 LOAD_CONST       3 (3)
 16     16 LOAD_CONST       4 (('name', 'version'))
 18     18 BUILD_CONST_KEY_MAP
 20     20 LOAD_CONST       5 ('.py')
 22     22 LOAD_CONST       6 ('text/x-python')
 24     24 LOAD_CONST       0 ('python')
 26     26 LOAD_CONST       0 ('python')
 28     28 LOAD_CONST       7 ('ipython3')
 30     30 LOAD_CONST       8 ('3.8')
 32     32 LOAD_CONST       9 ((codemirror_mode', 'file_extension', 'mimetype', 'name', 'nbconvert_exporter', 'pygments_lexer', 'version'))
 34     34 BUILD_CONST_KEY_MAP
 36     36 LOAD_CONST       10 (('kernelspec', 'language_info'))
 38     38 BUILD_CONST_KEY_MAP
 40     40 LOAD_CONST       2 (5)
 42     42 LOAD_CONST       11 (5)
 44     44 LOAD_CONST       12 (4)
 46     46 LOAD_CONST       13 ('a3d44123-f809-4be3-9fad-b44c90498dff')
 48     48 LOAD_CONST       14 ('code')
 50     50 LOAD_CONST       15 ('import numpy as np\n\ndef matrix_multiply(A, B):\n    print("Performing matrix multiplication...")\n    print(f"Matrix A shape: {A.shape}")\n    print(f"Matrix B shape: {B.shape}")\n    result = np.dot(A, B)\n    print("Matrix multiplication complete.")\n    print(f"Result shape: {result.shape}")\n    return result\n\nprint("Generating random matrices A and B...")\nA = np.random.rand(100, 100)\nB = np.random.rand(100, 100)\n\nprint("Calling matrix_multiply...")\nC = matrix_multiply(A, B)\n\nprint("Result matrix C (first 5x5 block):")\n\nprint(C[0:5, :5]) # Prints the top-left 5x5 portion of the matrix for readability\n\nprint("Done.")\n')
 52     50 LOAD_CONST       16 ('trusted')
```

```

50 LOAD_CONST      16 ('trusted')
52 LOAD_NAME       0 (true)
54 BUILD_MAP       1
56 LOAD_CONST      17 ('stdout')
58 LOAD_CONST      18 ('stream')
60 LOAD_CONST      19 ('Generating random matrices A and B...Calling matrix_multiply...')

Matrix multiplication complete.\nResult shape: (100, 100)\nPerforming matrix multiplication...\nMatrix A shape: (100, 100)\nMatrix B shape: (100, 100)\n
751 22.60584369]n [23.73972905 22.96817195 26.11732993 23.82873141 23.96295266]n [23.73150105 23.99520715 26.53622371 23.81288184 24.35048666]n [24.14190228 23.12646665 27.10865456 24.16882127 24.278153
07]]\nNone.\n')

62 LOAD_CONST      20 (('name', 'output_type', 'text'))
64 BUILD_CONST_KEY_MAP 3
66 BUILD_LIST      1
68 LOAD_CONST      21 (6)
70 LOAD_CONST      22 ((('id', 'cell_type', 'source', 'metadata', 'outputs', 'execution_count'))
72 BUILD_CONST_KEY_MAP 6
74 LOAD_CONST      23 ('7159f261-e98b-4558-bf2d-61354b0563f')
76 LOAD_CONST      14 ('code')
78 LOAD_CONST      24 ('\n\nndis.dis(matrix_multiply)')
80 LOAD_CONST      16 ('trusted')
82 LOAD_NAME       0 (true)
84 BUILD_MAP       1
86 LOAD_CONST      17 ('stdout')
88 LOAD_CONST      18 ('stream')
90 LOAD_CONST      25 (' 3          0 RESUME          0\\n\\n 4          2 LOAD_GLOBAL          1 (NULL + print)\\n
matrix multiplication...')\\n
92 LOAD_CONST      14 CALL          1\\n          22 POP_TOP\\n\\n 5          24 LOAD_GLOBAL          1 (NULL + print)\\n
94 BUILD_CONST_KEY_MAP 3
96 BUILD_LIST      1
98 LOAD_CONST      26 (10)
100 LOAD_CONST     22 ((('id', 'cell_type', 'source', 'metadata', 'outputs', 'execution_count'))
102 BUILD_CONST_KEY_MAP 5
104 LOAD_CONST     27 ('873707ab-44ba-45d8-bbc4-3d0d145f097d')
106 LOAD_CONST     14 ('code')
108 LOAD_CONST     28 ('py_compile.compile('matrix_multiply.py')")
110 LOAD_CONST     16 ('trusted')
112 LOAD_NAME       0 (true)
114 BUILD_MAP       1
116 LOAD_CONST     29 (<class 'NameError'>)
118 LOAD_CONST     30 ("name 'py_compile' is not defined")
120 BUILD_LIST      0
122 LOAD_CONST     31 ('\\x1b[0;31m-----\\x1b[0m', '\\x1b[0;31mNameError\\x1b[0m
eback (most recent call last)', 'Cell \\x1b[39m\\x1b[38;5;124m\\x1b[39m', 'line 1\\x1b[0m\\n\\x1b[0;32m--> 1\\x1b[0m \\x1b[43mpy_compile\\x1b[49m\\x1b[38;5;241m.\\x1b[39mcompile(\\x1b[38;5;124m\\x1b[39m\\x1b[38;5;124mmatrix_multiply\\x1b[39m\\x1b[38;5;124m\\x1b[39m)\\n', '\\x1b[0;31mNameError\\x1b[0m: name 'py_compile' is not defined'))\\n')
124 LIST_EXTEND   1
126 LOAD_CONST     32 ('error')
128 LOAD_CONST     33 (('ename', 'evalue', 'traceback', 'output_type'))
130 BUILD_CONST_KEY_MAP 4
132 BUILD_LIST      1
134 LOAD_CONST     34 (11)
136 LOAD_CONST     22 ((('id', 'cell_type', 'source', 'metadata', 'outputs', 'execution_count'))
138 BUILD_CONST_KEY_MAP 6
140 LOAD_CONST     35 ('217ea405-662f-4e8b-a7ce-1e0648ddb631')
142 LOAD_CONST     14 ('code')
144 LOAD_CONST     36 ('')
146 LOAD_CONST     16 ('trusted')
148 LOAD_NAME       0 (true)
150 BUILD_MAP       1
152 BUILD_LIST      0
154 LOAD_NAME       1 (null)
156 LOAD_CONST     22 ((('id', 'cell_type', 'source', 'metadata', 'outputs', 'execution_count'))
158 BUILD_CONST_KEY_MAP 6
160 BUILD_LIST      4
162 LOAD_CONST     37 (('metadata', 'nbformat_minor', 'nbformat', 'cells'))
164 BUILD_CONST_KEY_MAP 4
166 POP_TOP

```

```

108 LOAD_CONST     28 ('py_compile.compile('matrix_multiply.py')")
110 LOAD_CONST     16 ('trusted')
112 LOAD_NAME       0 (true)
114 BUILD_MAP       1
116 LOAD_CONST     29 (<class 'NameError'>)
118 LOAD_CONST     30 ("name 'py_compile' is not defined")
120 BUILD_LIST      0
122 LOAD_CONST     31 ('\\x1b[0;31m-----\\x1b[0m', '\\x1b[0;31mNameError\\x1b[0m
eback (most recent call last)', 'Cell \\x1b[39m\\x1b[38;5;124m\\x1b[39m', 'line 1\\x1b[0m\\n\\x1b[0;32m--> 1\\x1b[0m \\x1b[43mpy_compile\\x1b[49m\\x1b[38;5;241m.\\x1b[39mcompile(\\x1b[38;5;124m\\x1b[39m\\x1b[38;5;124mmatrix_multiply\\x1b[39m\\x1b[38;5;124m\\x1b[39m)\\n', '\\x1b[0;31mNameError\\x1b[0m: name 'py_compile' is not defined'))\\n')
124 LIST_EXTEND   1
126 LOAD_CONST     32 ('error')
128 LOAD_CONST     33 (('ename', 'evalue', 'traceback', 'output_type'))
130 BUILD_CONST_KEY_MAP 4
132 BUILD_LIST      1
134 LOAD_CONST     34 (11)
136 LOAD_CONST     22 ((('id', 'cell_type', 'source', 'metadata', 'outputs', 'execution_count'))
138 BUILD_CONST_KEY_MAP 6
140 LOAD_CONST     35 ('217ea405-662f-4e8b-a7ce-1e0648ddb631')
142 LOAD_CONST     14 ('code')
144 LOAD_CONST     36 ('')
146 LOAD_CONST     16 ('trusted')
148 LOAD_NAME       0 (true)
150 BUILD_MAP       1
152 BUILD_LIST      0
154 LOAD_NAME       1 (null)
156 LOAD_CONST     22 ((('id', 'cell_type', 'source', 'metadata', 'outputs', 'execution_count'))
158 BUILD_CONST_KEY_MAP 6
160 BUILD_LIST      4
162 LOAD_CONST     37 (('metadata', 'nbformat_minor', 'nbformat', 'cells'))
164 BUILD_CONST_KEY_MAP 4
166 POP_TOP

```

b. Quick Sort:

```

import marshal

# Path to the .pyc file (ensure correct Python version in the filename, e.g., cp312 for Python 3.12)
compiled_file = '_pycache_/_quicksort.cpython-312.pyc' # Replace '312' with your Python version

# Read the .pyc file
with open(compiled_file, 'rb') as f:
    f.seek(16) # Skip the header (magic number and timestamp)
    bytecode = f.read()

# Deserialize the bytecode to get the code object
code_object = marshal.loads(bytecode)

# Disassemble the code object
dis.dis(code_object)

0      0 RESUME          0

1      2 LOAD_CONST       0 ('python')
4      4 LOAD_CONST       1 ('Python (Pydide)')
6      6 LOAD_CONST       0 ('python')
8      8 LOAD_CONST       2 (('name', 'display_name', 'language'))
10     BUILD_CONST_KEY_MAP 3
12     LOAD_CONST       0 ('python')
14     LOAD_CONST       3 (3)
16     LOAD_CONST       4 (('name', 'version'))
18     BUILD_CONST_KEY_MAP 2
20     LOAD_CONST       5 ('.py')
22     LOAD_CONST       6 ('text/x-python')
24     LOAD_CONST       0 ('python')
26     LOAD_CONST       0 ('python')
28     LOAD_CONST       7 ('ipython3')
30     LOAD_CONST       8 ('3.8')
32     LOAD_CONST       9 (('codemirror_mode', 'file_extension', 'mimetype', 'name', 'nbconvert_exporter', 'pygments_lexer', 'version'))
34     BUILD_CONST_KEY_MAP 7
36     LOAD_CONST       10 (('kernelspec', 'language_info'))
38     BUILD_CONST_KEY_MAP 2
40     LOAD_CONST       11 (5)
42     LOAD_CONST       12 (4)
44     LOAD_CONST       13 ('23797889-6ca2-4a38-9708-46f1e31180cf')
46     LOAD_CONST       14 ('code')
48     LOAD_CONST       15 ('def quicksort(arr):\n    if len(arr) <= 1:\n        return arr\n    pivot = arr[len(arr) // 2] # Choose middle element as pivot\n    left = [x for x in arr if x < pivot]\n    middle = [x for x in arr if x == pivot]\n    right = [x for x in arr if x > pivot]\n    return quicksort(left) + middle + quicksort(right)\n\n# Example usage\nimport random\narr = random.sample(range(1000), 100)\nsorted_arr = quicksort(arr)\nprint("Sorted:", sorted_arr)')
50     LOAD_CONST       16 ('trusted')
52     LOAD_NAME        0 (true)
54     BUILD_MAP         1
56     LOAD_CONST       17 ('stdout')
58     LOAD_CONST       18 ('stream')
60     LOAD_CONST       19 ('Sorted: [8, 11, 26, 27, 50, 62, 65, 72, 76, 89, 95, 97, 119, 131, 145, 196, 209, 212, 216, 246, 259, 261, 264, 279, 284, 292, 295, 303, 310, 325, 329, 336, 338,
344, 354, 365, 379, 380, 397, 416, 425, 445, 449, 450, 455, 467, 470, 473, 475, 482, 483, 486, 489, 492, 499, 501, 517, 546, 561, 565, 577, 581, 591, 602, 631, 639, 655, 657, 665, 668, 678, 680, 683, 685,
696, 705, 708, 712, 740, 741, 742, 751, 756, 773, 797, 802, 852, 871, 879, 884, 913, 927, 932, 939, 940, 949, 955, 981, 983, 991]\n')
62     LOAD_CONST       20 (('name', 'output_type', 'text'))
64     BUILD_CONST_KEY_MAP 3
66     BUILD_LIST         1
68     LOAD_CONST       21 (1)
70     LOAD_CONST       22 (('id', 'cell_type', 'source', 'metadata', 'outputs', 'execution_count'))
72     BUILD_CONST_KEY_MAP 6
74     LOAD_CONST       23 ('d63c4016-822f-4da8-86b9-5de06ac561a7')
76     LOAD_CONST       14 ('code')
78     LOAD_CONST       24 ('import py_compile\n# Compile the Python file\nnpysort.compile.compile("matrix_multiply.ipynb")')
80     LOAD_CONST       16 ('trusted')
82     LOAD_NAME        0 (true)
84     BUILD_MAP         1
86     BUILD_LIST         0
88     LOAD_NAME        1 (null)
90     LOAD_CONST       22 (('id', 'cell_type', 'source', 'metadata', 'outputs', 'execution_count'))
92     BUILD_CONST_KEY_MAP 6
94     BUILD_LIST         2
96     LOAD_CONST       24 (('metadata', 'nbformat_minor', 'nbformat', 'cells'))
98     BUILD_CONST_KEY_MAP 4
100    POP_TOP
102    RETURN_CONST      26 (None)

```

c. Differential Equation solver:

```
[3]: import dis
import marshal

# Path to the .pyc file (ensure correct Python version in the filename, e.g., cp312 for Python 3.12)
compiled_file = '__pycache__/_quicksort.cpython-312.pyc' # Replace '312' with your Python version

# Read the .pyc file
with open(compiled_file, 'rb') as f:
    f.seek(16) # Skip the header (magic number and timestamp)
    bytecode = f.read()

# Deserialize the bytecode to get the code object
code_object = marshal.loads(bytecode)

# Disassemble the code object
dis.dis(code_object)

  0      0 RESUME          0
  1      2 LOAD_CONST       0 ('python')
  4      4 LOAD_CONST       1 ('Python (Pydide)')
  6      6 LOAD_CONST       0 ('python')
  8      8 LOAD_CONST       2 (('name', 'display_name', 'language'))
 10     10 BUILD_CONST_KEY_MAP 3
 12     12 LOAD_CONST       0 ('python')
 14     14 LOAD_CONST       3 (3)
 16     16 LOAD_CONST       4 (('name', 'version'))
 18     18 BUILD_CONST_KEY_MAP 2
 20     20 LOAD_CONST       5 ('.py')
 22     22 LOAD_CONST       6 ('text/x-python')
 24     24 LOAD_CONST       0 ('python')
 26     26 LOAD_CONST       0 ('python')
 28     28 LOAD_CONST       7 ('python3')
 30     30 LOAD_CONST       8 ('3.8')
 32     32 LOAD_CONST       9 (('codemirror_mode', 'file_extension', 'mimetype', 'name', 'nbconvert_exporter', 'pygments_lexer', 'version'))
 34     34 BUILD_CONST_KEY_MAP 7
 36     36 LOAD_CONST       10 (('kernelspec', 'language_info'))
 38     38 BUILD_CONST_KEY_MAP 2
 40     40 LOAD_CONST       11 (5)
 42     42 LOAD_CONST       12 (4)
 44     44 LOAD_CONST       13 ('23797889-6ca2-4a38-9708-46f1e31180cf')
 46     46 LOAD_CONST       14 ('code')
 48     48 LOAD_CONST       15 ('def quicksort(arr):\n    if len(arr) <= 1:\n        return arr\n    pivot = arr[len(arr) // 2] # Choose middle element as pivot\n    left = [x for x in arr if x < pivot]\n    middle = [x for x in arr if x == pivot]\n    right = [x for x in arr if x > pivot]\n    return quicksort(left) + middle + quicksort(right)\n\n# Example usage\nimport random\narr = random.sample(range(1000), 100)\nsorted_arr = quicksort(arr)\nprint("Sorted:", sorted_arr)')
 50     50 LOAD_CONST       16 ('trusted')

x < pivot]\n    middle = [x for x in arr if x == pivot]\n    right = [x for x in arr if x > pivot]\n    return quicksort(left) + middle + quicksort(right)\n\n# Example usage\nimport random\narr = random.sample(range(1000), 100)\nsorted_arr = quicksort(arr)\nprint("Sorted:", sorted_arr)')
 52     52 LOAD_NAME         0 (true)
 54     54 BUILD_MAP          1
 56     56 LOAD_CONST       17 ('stdout')
 58     58 LOAD_CONST       18 ('stream')
 60     60 LOAD_CONST       19 ('Sorted: [8, 11, 26, 27, 50, 62, 65, 72, 76, 89, 95, 97, 119, 131, 145, 196, 209, 212, 216, 246, 259, 261, 264, 279, 284, 292, 295, 303, 310, 325, 329, 336, 338,
344, 354, 365, 379, 380, 397, 416, 425, 445, 449, 450, 455, 467, 470, 473, 475, 482, 483, 486, 489, 492, 499, 501, 517, 546, 561, 565, 577, 581, 591, 602, 631, 639, 655, 657, 665, 668, 678, 680, 683, 685,
696, 705, 708, 712, 740, 741, 751, 756, 773, 797, 802, 852, 871, 879, 884, 913, 927, 932, 939, 940, 949, 955, 981, 983, 991]\n')
 62     62 LOAD_CONST_KEY_MAP 3
 64     64 BUILD_LIST          1
 66     66 LOAD_CONST       21 (1)
 68     68 LOAD_CONST       22 (('id', 'cell_type', 'source', 'metadata', 'outputs', 'execution_count'))
 70     70 LOAD_CONST       23 ('code')
 72     72 BUILD_CONST_KEY_MAP 6
 74     74 LOAD_CONST       24 ('import py_compile\n\n# Compile the Python file\nnpyscompile.compile("matrix_multiply.ipynb")')
 76     76 LOAD_CONST       25 ('trusted')
 78     78 LOAD_NAME         0 (true)
 80     80 BUILD_MAP          1
 82     82 LOAD_NAME         1 (null)
 84     84 LOAD_CONST       22 (('id', 'cell_type', 'source', 'metadata', 'outputs', 'execution_count'))
 86     86 BUILD_LIST          2
 88     88 LOAD_NAME         6
 90     90 LOAD_CONST       25 (('metadata', 'nbformat_minor', 'nbformat', 'cells'))
 92     92 BUILD_CONST_KEY_MAP 4
 94     94 BUILD_LIST          2
 96     96 LOAD_CONST       26 (None)
 98     98 BUILD_CONST_KEY_MAP 2
100    100 POP_TOP           0
102    102 RETURN_CONST     26 (None)
```

1. Yes, by looking at the bytecode, we can infer the **Python Virtual Machine (VM)** that is executing the bytecode. Python's bytecode is designed to be executed by the **Python interpreter**, specifically the **CPython interpreter**, which is the most common implementation of Python.

How to Recognize the Python VM from Bytecode:

- **Magic Number:**
 - Each compiled Python .pyc file begins with a **magic number** that identifies the version of Python used to compile the file. For example, cp38 indicates Python 3.8.

- The magic number is a key identifier for CPython.
 - **Bytecode Structure:**
 - The bytecode for CPython consists of a sequence of **opcodes** (like LOAD_CONST, BINARY_ADD, etc.) that are executed by the CPython virtual machine.
 - The specific set of opcodes and the way they are encoded are unique to CPython.
 - **Disassembly Output:**
 - CPython bytecode consists of a combination of **load**, **store**, **arithmetic**, **comparison**, and **control flow** operations.
 - For example, instructions like LOAD_GLOBAL, CALL_FUNCTION, RETURN_VALUE, and others are CPython-specific.
-

Steps to Identify the VM (in a Practical Scenario):

- **Look at the magic number:** In the .pyc file, the first 16 bytes contain the **magic number** and timestamp. For CPython, the magic number corresponds to the version of CPython that was used to generate the bytecode (e.g., 0x0d0d0d0d for certain Python versions).
 - **Disassemble the bytecode:** The bytecode instructions generated by CPython will be in a format that can be recognized using tools like dis. The opcodes and their argument formats are tailored for CPython's VM.
 - **Version-specific bytecode:** Each version of Python produces slightly different bytecode. For example:
 - Python 2.x had a different opcode set compared to Python 3.x.
 - Python 3.x introduced new features like function annotations, async/await, and type hints, which affect the bytecode.
2. Arithmetic Instructions:
- a. Matrix_multiplication:

```

import numpy as np

# Define the arithmetic instructions you want to count
arithmetic_opcodes = {
    'BINARY_ADD': 23,
    'BINARY_SUBTRACT': 24,
    'BINARY_MULTIPLY': 25,
    'BINARY_DIVIDE': 26,
    'BINARY_FLOOR_DIVIDE': 27,
    'BINARY_MODULO': 28,
    'BINARY_POWER': 29,
    'UNARY_POSITIVE': 10,
    'UNARY_NEGATIVE': 11
}

def count_arithmetic_instructions(func):
    # Disassemble the function's bytecode
    bytecode = dis.Bytecode(func)

    # Initialize a dictionary to count occurrences of each arithmetic instruction
    count = {op: 0 for op in arithmetic_opcodes}

    # Iterate over the disassembled bytecode instructions
    for instruction in bytecode:
        # Check if the instruction is an arithmetic instruction
        if instruction.opcode in arithmetic_opcodes.values():
            # Find the instruction name based on opcode value
            op_name = [key for key, value in arithmetic_opcodes.items() if value == instruction.opcode][0]
            count[op_name] += 1

    return count

# Matrix multiplication function
def matrix_multiply(A, B):
    print("Performing matrix multiplication...")
    print(f"Matrix A shape: {A.shape}")
    print(f"Matrix B shape: {B.shape}")
    result = np.dot(A, B) # This involves arithmetic operations
    print("Matrix multiplication complete.")
    print(f"Result shape: {result.shape}")
    return result

# Count arithmetic instructions in the matrix_multiply function
count = count_arithmetic_instructions(matrix_multiply)

```

```

# Display the count of each arithmetic instruction
print("Arithmetic Instruction Counts in matrix_multiply:")
for instruction, num in count.items():
    print(f"{instruction}: {num}")

```

```

Arithmetic Instruction Counts in matrix_multiply:
BINARY_ADD: 0
BINARY_SUBTRACT: 0
BINARY_MULTIPLY: 0
BINARY_DIVIDE: 0
BINARY_FLOOR_DIVIDE: 0
BINARY_MODULO: 0
BINARY_POWER: 0
UNARY_POSITIVE: 0
UNARY_NEGATIVE: 0

```

b. Quick sort

```
[5]: import marshal
import dis
import types

# Define arithmetic opnames and include generic BINARY_OP
ARITHMETIC_OPS = [
    'BINARY_ADD',
    'BINARY_SUBTRACT',
    'BINARY_MULTIPLY',
    'BINARY_TRUE_DIVIDE',
    'BINARY_FLOOR_DIVIDE',
    'BINARY_MODULO',
    'BINARY_POWER',
    'INPLACE_ADD',
    'INPLACE_SUBTRACT',
    'INPLACE_MULTIPLY',
    'INPLACE_TRUE_DIVIDE',
    'INPLACE_FLOOR_DIVIDE',
    'INPLACE_MODULO',
    'INPLACE_POWER',
    'UNARY_NEGATIVE',
    'UNARY_POSITIVE',
    'BINARY_OP', # Python 3.11+
]

# Load the code object from pyc
pyc_file = "__pycache__/quicksort.cpython-312.pyc" # Adjust path if needed

with open(pyc_file, "rb") as f:
    f.read(16) # Skip header
    code_object = marshal.load(f)

def count_arithmetic_in_code(code_obj):
    count = {}
    for instr in dis.get_instructions(code_obj):
        if instr.opname in ARITHMETIC_OPS:
            count[instr.opname] = count.get(instr.opname, 0) + 1

    # Handle nested code objects
    for const in code_obj.co_consts:
        if isinstance(const, types.CodeType):
            sub_count = count_arithmetic_in_code(const)
```

```

def count_arithmetic_in_code(code_obj):
    count = {}
    for instr in dis.get_instructions(code_obj):
        if instr.opname in ARITHMETIC_OPS:
            count[instr.opname] = count.get(instr.opname, 0) + 1

    # Handle nested code objects
    for const in code_obj.co_consts:
        if isinstance(const, types.CodeType):
            sub_count = count_arithmetic_in_code(const)
            for k, v in sub_count.items():
                count[k] = count.get(k, 0) + v

    return count

# Count and print
instruction_counts = count_arithmetic_in_code(code_object)
print("Arithmetic Instructions Counted from Quicksort Bytecode:")
for opname, count in instruction_counts.items():
    print(f"{opname}: {count}")

print(f"\nTotal Arithmetic Instructions: {sum(instruction_counts.values())}")

```

Arithmetic Instructions Counted from Quicksort Bytecode:

Total Arithmetic Instructions: 0

c.Differential Equation solver:

```

import marshal
import dis
import types

# Define arithmetic opnames and include generic BINARY_OP
ARITHMETIC_OPS = [
    'BINARY_ADD',
    'BINARY_SUBTRACT',
    'BINARY_MULTIPLY',
    'BINARY_TRUE_DIVIDE',
    'BINARY_FLOOR_DIVIDE',
    'BINARY_MODULO',
    'BINARY_POWER',
    'INPLACE_ADD',
    'INPLACE_SUBTRACT',
    'INPLACE_MULTIPLY',
    'INPLACE_TRUE_DIVIDE',
    'INPLACE_FLOOR_DIVIDE',
    'INPLACE_MODULO',
    'INPLACE_POWER',
    'UNARY_NEGATIVE',
    'UNARY_POSITIVE',
    'BINARY_OP', # Python 3.11+
]

# Load the code object from pyc
pyc_file = "__pycache__/quicksort.cpython-312.pyc" # Adjust path if needed

with open(pyc_file, "rb") as f:
    f.read(16) # Skip header
    code_object = marshal.load(f)

def count_arithmetic_in_code(code_obj):
    count = {}
    for instr in dis.get_instructions(code_obj):
        if instr.opname in ARITHMETIC_OPS:
            count[instr.opname] = count.get(instr.opname, 0) + 1

    # Handle nested code objects
    for const in code_obj.co_consts:
        if isinstance(const, types.CodeType):
            sub_count = count_arithmetic_in_code(const)
            for k, v in sub_count.items():
                count[k] = count.get(k, 0) + v

```

```

# Load the code object from pyc
pyc_file = "__pycache__/quicksort.cpython-312.pyc" # Adjust path if needed

with open(pyc_file, "rb") as f:
    f.read(16) # Skip header
    code_object = marshal.load(f)

def count_arithmetic_in_code(code_obj):
    count = {}
    for instr in dis.get_instructions(code_obj):
        if instr.opname in ARITHMETIC_OPS:
            count[instr.opname] = count.get(instr.opname, 0) + 1

    # Handle nested code objects
    for const in code_obj.co_consts:
        if isinstance(const, types.CodeType):
            sub_count = count_arithmetic_in_code(const)
            for k, v in sub_count.items():
                count[k] = count.get(k, 0) + v

    return count

# Count and print
instruction_counts = count_arithmetic_in_code(code_object)
print("Arithmetical Instructions Counted from Quicksort Bytecode:")
for opname, count in instruction_counts.items():
    print(f"{opname}: {count}")

print(f"\nTotal Arithmetical Instructions: {sum(instruction_counts.values())}")

```

Arithmetical Instructions Counted from Quicksort Bytecode:

Total Arithmetical Instructions: 0

4. Compare the instruction distribution for your 3 workloads

a. Matrix_multiplication:

```
Arithmetic Instruction Counts in matrix_multiply:  
BINARY_ADD: 0  
BINARY_SUBTRACT: 0  
BINARY_MULTIPLY: 0  
BINARY_DIVIDE: 0  
BINARY_FLOOR_DIVIDE: 0  
BINARY_MODULO: 0  
BINARY_POWER: 0  
UNARY_POSITIVE: 0  
UNARY_NEGATIVE: 0
```

b. Quick sort:

```
Arithmetic Instructions Counted from Quicksort Bytecode:
```

```
Total Arithmetic Instructions: 0
```

c. Differential Equation solver:

```
Arithmetic Instructions Counted from Quicksort Bytecode:
```

```
Total Arithmetic Instructions: 0
```

5. Profiling:

a. matrix_multiplication:

```

: import cProfile
import numpy as np
import pstats
from io import StringIO

# Matrix multiplication function
def matrix_multiply(A, B):
    print("Performing matrix multiplication...")
    print(f"Matrix A shape: {A.shape}")
    print(f"Matrix B shape: {B.shape}")
    result = np.dot(A, B) # Matrix multiplication
    print("Matrix multiplication complete.")
    print(f"Result shape: {result.shape}")
    return result

# Profile the function execution
def profile_matrix_multiply():
    # Generate random matrices
    A = np.random.rand(100, 100)
    B = np.random.rand(100, 100)

    # Profile the matrix_multiply function
    profiler = cProfile.Profile()
    profiler.enable() # Start profiling
    C = matrix_multiply(A, B)
    profiler.disable() # Stop profiling

    # Output the profiling results to a string stream
    stream = StringIO()
    stats = pstats.Stats(profiler, stream=stream)
    stats.sort_stats('cumulative') # Sort by cumulative time
    stats.print_stats() # Print stats to the console

    # You can also save the profile to a file if needed
    stats.dump_stats('profile_output.prof') # Save profiling results

# Call the profile function
profile_matrix_multiply()

Performing matrix multiplication...
Matrix A shape: (100, 100)
Matrix B shape: (100, 100)
Matrix multiplication complete.
Result shape: (100, 100)

```

b.quick_sort:

```
import cProfile
import random
import pstats
from io import StringIO

# Quicksort function
def quicksort(arr):
    if len(arr) <= 1:
        return arr
    else:
        pivot = arr[len(arr) // 2]
        left = [x for x in arr if x < pivot]
        middle = [x for x in arr if x == pivot]
        right = [x for x in arr if x > pivot]
        return quicksort(left) + middle + quicksort(right)

# Profile the quicksort function
def profile_quicksort():
    # Generate a random list of integers
    arr = [random.randint(0, 10000) for _ in range(1000)]

    # Profile the quicksort function
    profiler = cProfile.Profile()
    profiler.enable() # Start profiling
    sorted_arr = quicksort(arr)
    profiler.disable() # Stop profiling

    # Output the profiling results to a string stream
    stream = StringIO()
    stats = pstats.Stats(profiler, stream=stream)
    stats.sort_stats('cumulative') # Sort by cumulative time
    stats.print_stats() # Print to string stream

    # Print the profiling results to console
    print(stream.getvalue())

    # Save profiling data to a file
    stats.dump_stats('quicksort_profile.prof') # Optional

# Run the profiling
profile_quicksort()
```

```
3223 function calls (1935 primitive calls) in 0.006 seconds
```

Ordered by: cumulative time

ncalls	tottime	percall	cumtime	percall	filename:lineno(function)
1289/1	0.005	0.000	0.006	0.006	<ipython-input-6-cc67db3d3a69>:7(quicksort)
1933	0.001	0.000	0.001	0.000	{built-in method builtins.len}
1	0.000	0.000	0.000	0.000	{method 'disable' of '_lsprof.Profiler' objects}

c. Differential Equation solver:

```
import numpy as np
import matplotlib.pyplot as plt
import cProfile
import pstats
from io import StringIO
from scipy.integrate import odeint

# Define the differential equation: dy/dt = -2y
def model(y, t):
    dydt = -2 * y
    return dydt

# Profile the differential equation solver
def profile_differential_equation_solver():
    # Initial condition
    y0 = 1

    # Time points
    t = np.linspace(0, 5, 100)

    # Profile the ODE solver
    profiler = cProfile.Profile()
    profiler.enable() # Start profiling
    y = odeint(model, y0, t) # Solve the differential equation
    profiler.disable() # Stop profiling

    # Output the profiling results to a string stream
    stream = StringIO()
    stats = pstats.Stats(profiler, stream=stream)
    stats.sort_stats('cumulative') # Sort by cumulative time
    stats.print_stats() # Print to string stream

    # Print the profiling results to the console
    print(stream.getvalue())

    # Optionally save the profiling data to a file
    stats.dump_stats('differential_equation_solver_profile.prof') # Optional

# Run the profiling
profile_differential_equation_solver()
```

164 function calls in 0.001 seconds

```

164 function calls in 0.001 seconds

Ordered by: cumulative time

      ncalls  tottime  percall  cumtime  percall  filename:lineno(function)
          1    0.000    0.000   0.001    0.001 /lib/python3.12/site-packages/scipy/integrate/_odepack_py.py:29(odeint)
          1    0.001    0.001   0.001 {built-in method scipy.integrate._odepack.odeint}
          1    0.000    0.000   0.000   0.000 /lib/python3.12/site-packages/numpy/lib/_function_base_impl.py:1324(diff)
  143    0.000    0.000   0.000   0.000 <ipython-input-5-86786c4d0c2c>:9(model)
          1    0.000    0.000   0.000 {method 'append' of 'list' objects}
          1    0.000    0.000   0.000 {method 'disable' of '_lsprof.Profiler' objects}
          2    0.000    0.000   0.000 {method 'get' of 'dict' objects}
          1    0.000    0.000   0.000 {built-in method builtinsgetattr}
          1    0.000    0.000   0.000 {built-in method builtins.issubclass}
          2    0.000    0.000   0.000 {built-in method builtins.len}
          2    0.000    0.000   0.000   0.000 /lib/python312.zip/copy.py:61(copy)
          1    0.000    0.000   0.000   0.000 /lib/python312.zip/copy.py:102(_copy_immutable)
          1    0.000    0.000   0.000 {method '__copy__' of 'numpy.ndarray' objects}
          1    0.000    0.000   0.000 {method 'all' of 'numpy.ndarray' objects}
          1    0.000    0.000   0.000 {built-in method numpy.lib.array_utils.normalize_axis_index}
          1    0.000    0.000   0.000 {method 'reduce' of 'numpy.ufunc' objects}
          1    0.000    0.000   0.000   0.000 /lib/python3.12/site-packages/numpy/_core/_methods.py:67(_all)
          1    0.000    0.000   0.000   0.000 /lib/python3.12/site-packages/numpy/lib/_function_base_impl.py:1320(_diff_dispatcher)

```

6. Parallelism:

a. Matrix_multiplication:

```

import numpy as np
import ast

# --- Matrix Multiplication Code as a String ---
code = """
import numpy as np
def matrix_multiply(A, B):
    print("Performing matrix multiplication...")
    print(f"Matrix A shape: {A.shape}")
    print(f"Matrix B shape: {B.shape}")
    result = np.dot(A, B)
    print("Matrix multiplication complete.")
    print(f"Result shape: {result.shape}")
    return result
print("Generating random matrices A and B...")
A = np.random.rand(100, 100)
B = np.random.rand(100, 100)

print("Calling matrix_multiply...")
C = matrix_multiply(A, B)

print("Result matrix C (first 5x5 block):")
print(C[:5, :5]) # Prints the top-left 5x5 portion of the matrix for readability
print("Done.")
"""

# --- Analyzer Class ---
class MatrixMultiplyAnalyzer(ast.NodeVisitor):
    def __init__(self):
        self.loops = []
        self.assignments = []
        self.calls = []
        self.dependencies = set()

    def visit_FunctionDef(self, node):
        print(f"\nAnalyzing function: {node.name}")
        self.generic_visit(node)

    def visit_Assign(self, node):

```

```

class MatrixMultiplyAnalyzer(ast.NodeVisitor):
    def __init__(self):
        self.loops = []
        self.assignments = []
        self.calls = []
        self.dependencies = set()

    def visit_FunctionDef(self, node):
        print(f"\nAnalyzing function: {node.name}")
        self.generic_visit(node)

    def visit_Assign(self, node):
        targets = [ast.unparse(t) for t in node.targets]
        value = ast.unparse(node.value)
        print(f"- Assignment: {':'.join(targets)} = {value}")
        self.assignments.append((targets, value))
        self.generic_visit(node)

    def visit_For(self, node):
        print(f"- Loop found at line {node.lineno}")
        self.loops.append(node)
        self.generic_visit(node)

    def visit_Call(self, node):
        func_name = ast.unparse(node.func)
        print(f"- Function call: {func_name} at line {node.lineno}")
        self.calls.append((func_name, node.lineno))
        self.generic_visit(node)

    def check_dependencies(self):
        print("\nChecking data dependencies...")
        written = set()
        for targets, value in self.assignments:
            for target in targets:
                for w in written:
                    if w in value:
                        self.dependencies.add((w, target))
                written.add(target)

    def report(self):
        print("\nSummary:")
        print(f"- Loops: {len(self.loops)}")
        print(f"- Assignments: {len(self.assignments)}")
        print(f"- Function calls: {len(self.calls)}")
        print(f"- Dependencies: {self.dependencies}")

```

```

def report(self):
    print("\nSummary:")
    print(f"- Loops: {len(self.loops)}")
    print(f"- Assignments: {len(self.assignments)}")
    print(f"- Function calls: {len(self.calls)}")
    print(f"- Dependencies: {self.dependencies}")
    if not self.dependencies:
        print("Matrix multiplication is likely parallelizable at element level using independent dot products.")

# --- Run Analyzer ---
tree = ast.parse(code)
analyzer = MatrixMultiplyAnalyzer()
analyzer.visit(tree)
analyzer.check_dependencies()
analyzer.report()

```

```

Analyzing function: matrix_multiply
- Function call: print at line 5
- Function call: print at line 6
- Function call: print at line 7
- Assignment: result = np.dot(A, B)
- Function call: np.dot at line 8
- Function call: print at line 9
- Function call: print at line 10
- Function call: print at line 13
- Assignment: A = np.random.rand(100, 100)
- Function call: np.random.rand at line 14
- Assignment: B = np.random.rand(100, 100)
- Function call: np.random.rand at line 15
- Function call: print at line 17
- Assignment: C = matrix_multiply(A, B)
- Function call: matrix_multiply at line 18
- Function call: print at line 20
- Function call: print at line 21
- Function call: print at line 23

```

Checking data dependencies...

```

Summary:
- Loops: 0
- Assignments: 4
- Function calls: 14
- Dependencies: {('A', 'C'), ('B', 'C')}

```

b. Quick_sort:

```
import ast

class ParallelismAnalyzer(ast.NodeVisitor):
    def __init__(self):
        self.loops = []
        self.func_calls = []
        self.assignments = []
        self.dependencies = set()

    def visit_FunctionDef(self, node):
        print(f"\nAnalyzing function: {node.name}")
        self.generic_visit(node)

    def visit_For(self, node):
        print(f" - Found for-loop at line {node.lineno}")
        self.loops.append(node)
        self.generic_visit(node)

    def visit_Call(self, node):
        try:
            func_name = node.func.id
        except AttributeError:
            func_name = ast.unparse(node.func)
        print(f" - Function call: {func_name} at line {node.lineno}")
        self.func_calls.append((func_name, node.lineno))
        self.generic_visit(node)

    def visit_Assign(self, node):
        targets = [ast.unparse(t) for t in node.targets]
        values = ast.unparse(node.value)
        print(f" - Assignment: {' = '.join(targets)} = {values}")
        self.assignments.append((targets, values))
        self.generic_visit(node)

    def check_data_dependencies(self):
        print("\nChecking data dependencies...")
        written = set()
        for targets, values in self.assignments:
            for var in targets:
                written.add(var)
            for w in written:
```

```

        values = ast.unparse(node.value)
        print(f"- Assignment: {targets} = {values}")
        self.assignments.append((targets, values))
        self.generic_visit(node)

    def check_data_dependencies(self):
        print("\nChecking data dependencies...")
        written = set()
        for targets, values in self.assignments:
            for var in targets:
                written.add(var)
                for w in written:
                    if w in values:
                        self.dependencies.add((w, var))
                        print(f"- Dependency detected: {w} -> {var}")

    def report(self):
        print("\nSummary:")
        print(f"- Loops found: {len(self.loops)}")
        print(f"- Function calls found: {len(self.func_calls)}")
        print(f"- Assignments found: {len(self.assignments)}")
        print(f"- Dependencies detected: {self.dependencies}")
        if not self.dependencies:
            print("⚠ Potential for parallel execution: Subarray sorts can be parallelized!")

# Your quicksort function to analyze
code = """
def quicksort(arr):
    if len(arr) <= 1:
        return arr
    else:
        pivot = arr[len(arr) // 2]
        left = [x for x in arr if x < pivot]
        middle = [x for x in arr if x == pivot]
        right = [x for x in arr if x > pivot]
    return quicksort(left) + middle + quicksort(right)
"""

# Parse and analyze the quicksort code
tree = ast.parse(code)
analyzer = ParallelismAnalyzer()
analyzer.visit(tree)
analyzer.check_data_dependencies()
analyzer.report()

...
return quicksort(left) + middle + quicksort(right)

# Parse and analyze the quicksort code
tree = ast.parse(code)
analyzer = ParallelismAnalyzer()
analyzer.visit(tree)
analyzer.check_data_dependencies()
analyzer.report()

Analyzing function: quicksort
- Function call: len at line 3
- Assignment: pivot = arr[len(arr) // 2]
- Function call: len at line 6
- Assignment: left = [x for x in arr if x < pivot]
- Assignment: middle = [x for x in arr if x == pivot]
- Assignment: right = [x for x in arr if x > pivot]
- Function call: quicksort at line 10
- Function call: quicksort at line 10

Checking data dependencies...
- Dependency detected: pivot -> left
- Dependency detected: pivot -> middle
- Dependency detected: pivot -> right

Summary:
- Loops found: 0
- Function calls found: 4
- Assignments found: 4
- Dependencies detected: {('pivot', 'left'), ('pivot', 'right'), ('pivot', 'middle')}

```

c. Differential Equation solver

```
import ast

class ParallelismAnalyzer(ast.NodeVisitor):
    def __init__(self):
        self.loops = []
        self.func_calls = []
        self.assignments = []
        self.dependencies = set()

    def visit_FunctionDef(self, node):
        print(f"\nAnalyzing function: {node.name}")
        self.generic_visit(node)

    def visit_For(self, node):
        print(f" - Found for-loop at line {node.lineno}")
        self.loops.append(node)
        self.generic_visit(node)

    def visit_Call(self, node):
        try:
            func_name = node.func.id
        except AttributeError:
            func_name = ast.unparse(node.func)
        print(f" - Function call: {func_name} at line {node.lineno}")
        self.func_calls.append((func_name, node.lineno))
        self.generic_visit(node)

    def visit_Assign(self, node):
        targets = [ast.unparse(t) for t in node.targets]
        values = ast.unparse(node.value)
        print(f" - Assignment: {' = '.join(targets)} = {values}")
        self.assignments.append((targets, values))
        self.generic_visit(node)

    def check_data_dependencies(self):
        print("\nChecking data dependencies...")
        written = set()
        for targets, values in self.assignments:
            for var in targets:
                written.add(var)
            for w in written:
                if w in values:
                    self.dependencies.add((w, var))
```

```

        self.dependencies.add((w, var))
        print(f" - Dependency detected: {w} -> {var}")

def report(self):
    print("\nSummary:")
    print(f" - Loops found: {len(self.loops)}")
    print(f" - Function calls found: {len(self.func_calls)}")
    print(f" - Assignments found: {len(self.assignments)}")
    print(f" - Dependencies detected: {self.dependencies}")
    if not self.dependencies:
        print(" ✅ Potential for parallel execution: The solver can be parallelized!")

# Your differential equation solver code to analyze
code = '''
import numpy as np
from scipy.integrate import odeint

# Define the differential equation: dy/dt = -2y
def model(y, t):
    dydt = -2 * y
    return dydt

# Solve the differential equation
y0 = 1
t = np.linspace(0, 5, 100)
y = odeint(model, y0, t)
'''

# Parse and analyze the differential equation solver code
tree = ast.parse(code)
analyzer = ParallelismAnalyzer()
analyzer.visit(tree)
analyzer.check_data_dependencies()
analyzer.report()

```

Analyzing function: model

- Assignment: dydt = -2 * y
- Assignment: y0 = 1
- Assignment: t = np.linspace(0, 5, 100)
- Function call: np.linspace at line 12
- Assignment: y = odeint(model, y0, t)
- Function call: odeint at line 13

Checking data dependencies...

Analyzing function: model

- Assignment: dydt = -2 * y
- Assignment: y0 = 1
- Assignment: t = np.linspace(0, 5, 100)
- Function call: np.linspace at line 12
- Assignment: y = odeint(model, y0, t)
- Function call: odeint at line 13

Checking data dependencies...

- Dependency detected: y0 -> y
- Dependency detected: y -> y
- Dependency detected: t -> y

Summary:

- Loops found: 0
- Function calls found: 2
- Assignments found: 4
- Dependencies detected: {('y', 'y'), ('y0', 'y'), ('t', 'y')}

7. When considering instruction architectures for workloads like matrix multiplication, quicksort, and differential equation solvers, we need to focus on how these algorithms behave and what type of hardware and instruction set would best support their performance requirements. Here's a breakdown for each workload:

1. Matrix Multiplication

Matrix multiplication (especially for large matrices) involves a lot of repeated arithmetic operations (like dot products) over matrices, which can benefit from parallelism. We need an architecture that supports:

Key Characteristics:

- **High Arithmetic Throughput:** Matrix multiplication requires efficient handling of multiply-accumulate (MAC) operations.
- **Parallelism:** Since each matrix element can be computed independently, parallel execution is critical.
- **Memory Bandwidth:** A high bandwidth memory system is crucial for moving large matrix data efficiently.

Suggested Instruction Architecture:

- **SIMD (Single Instruction, Multiple Data):** SIMD units are ideal for matrix multiplication, as many elements of the matrix can be computed simultaneously. This would be especially beneficial in a CPU or GPU context where vectorized instructions can compute multiple dot products at once.
- **SIMT (Single Instruction, Multiple Threads):** For GPUs, a SIMT architecture allows many threads to work on independent data, making it highly suited for matrix operations.
- **Vector Processing Units:** A specialized vector processing unit can accelerate matrix multiplication by handling multiple data points in parallel, utilizing wide registers for performing operations on entire rows or columns of matrices in a single instruction.
- **High Bandwidth Memory:** Since matrix multiplication involves large data sets, supporting high bandwidth memory (like HBM or GDDR) would be essential for maintaining data throughput.

Example Architectures:

- **GPU Architecture:** CUDA cores in NVIDIA GPUs or AMD's RDNA architecture could handle the parallel computations efficiently.
- **Custom VLIW (Very Long Instruction Word) or SIMD:** Custom hardware accelerators could be designed with SIMD or VLIW instructions to handle matrix multiplications with high parallelism.

2. Quicksort

Quicksort is a divide-and-conquer algorithm, where the array is partitioned and recursively sorted. The key operations are comparisons and data movement.

Key Characteristics:

- **Recursive Nature:** Quicksort's divide-and-conquer nature can benefit from recursion and efficient branching, making it potentially difficult to parallelize without careful management of recursion depth and partitioning.
- **Memory Access Patterns:** Quicksort involves random access to elements of the array, making it important to have low-latency memory access.

Suggested Instruction Architecture:

- **Branch Prediction and Out-of-Order Execution:** Since quicksort involves recursion and many branching decisions, an architecture that supports advanced branch prediction and out-of-order execution (like modern CPUs) would speed up execution by minimizing pipeline stalls.
- **Cache Optimization:** Cache locality is critical for quicksort's performance. A good instruction set architecture would optimize data movement through caches, with prefetching mechanisms to reduce memory latency.
- **Multi-threading:** Quicksort can be parallelized by dividing the array into sub-arrays and sorting them concurrently. An architecture with support for multi-threading (such as Intel's Hyper-Threading) or SIMD (like GPUs) could be used to speed up the algorithm.
- **Task Parallelism:** For parallelism, the instruction set could be optimized for task parallelism, where recursive subproblems can be handled independently.

Example Architectures:

- **Multi-core CPU:** Using multiple threads (one for each sub-array), quicksort can leverage CPUs with multiple cores (e.g., Intel Core or AMD Ryzen architectures).
- **GPU-based Parallelization:** A parallelized version of quicksort could utilize the thousands of small threads available in modern GPUs. Custom sorting algorithms like **bitonic sort** are often used on GPUs for such purposes.
- **Vector Processors:** Optimizing for vector processing units would help with partitioning the data and performing comparisons in parallel on vectorized chunks of data.

3. Differential Equation Solver

Solving differential equations (especially for large systems) requires heavy numerical computation and iterative methods (like Euler's method, Runge-Kutta, etc.). The workloads often involve floating-point arithmetic, iterations, and memory access patterns that can be computationally expensive.

Key Characteristics:

- **Iterative Computation:** Differential equation solvers often require many iterations over a large number of values. This makes them suitable for parallelism, especially if each iteration can be handled independently.
- **Floating-point Operations:** Since these solvers require high precision, efficient floating-point computation is essential.
- **Data Dependencies:** The solver often operates on large matrices or vectors, and the result of one iteration may be used in the next.

Suggested Instruction Architecture:

- **Floating-point Units (FPUs):** A processor with high-performance floating-point units is crucial for solving differential equations efficiently.
- **SIMD or SIMT:** Differential solvers, especially in the context of parallel computations, would benefit from SIMD (for CPUs) or SIMT (for GPUs), where multiple iterations or elements can be processed simultaneously.
- **Vector Processors:** For solving systems of differential equations, vector processors can be used to handle multiple data points simultaneously in each instruction.
- **Cache Hierarchy and Memory Bandwidth:** Since differential equation solvers involve large datasets, ensuring fast memory access and effective use of the cache hierarchy will be essential.
- **Custom Accelerators:** Custom hardware accelerators or FPGA designs can be tailored to perform the specific iterative calculations in differential equation solvers.

Example Architectures:

- **High-Performance CPU:** Modern processors with a good balance of floating-point performance (e.g., Intel Xeon or AMD EPYC) could be used, with support for SIMD instructions to parallelize computations.
 - **GPUs:** For larger systems, GPUs with thousands of threads could be used to parallelize the iterative solving of differential equations. Libraries like CUDA and OpenCL provide high-performance computation for these workloads.
 - **FPGA/ASIC:** For custom hardware acceleration, FPGAs or ASICs designed specifically for numerical simulations could optimize these iterative methods.
-

Conclusion: Instruction Architectures for Each Workload

- **Matrix Multiplication:** SIMD, SIMT, and high-bandwidth memory architectures (like GPUs or custom accelerators) would be optimal.
- **Quicksort:** Branch prediction, out-of-order execution, multi-threading, and cache optimization would be key components for CPUs, while SIMT could be beneficial for GPUs.
- **Differential Equation Solvers:** Floating-point units, SIMD/SIMT for parallelism, high memory bandwidth, and vector processors are crucial, with GPUs or custom hardware being ideal for large-scale simulations.

Challenge -4:

1. Understanding the Johns Hopkins Paper

The paper explores using Large Language Models (specifically ChatGPT at the time of writing) to automatically generate hardware description language (HDL) for a spiking neuron array. The core idea is to provide the LLM with a high-level description of the desired array architecture and have it produce synthesizable Verilog code. The paper likely details:

- The specific architecture of the spiking neuron array they aimed for (number of neurons, connectivity, neuron model - likely LIF, etc.).
- The prompts they used with ChatGPT to guide the code generation.
- The level of abstraction of their prompts.
- The challenges and successes they encountered in using the LLM for this task.
- Verification and simulation of the generated HDL.
- Potentially, resource utilization and performance metrics if they synthesized the design.

3. Experimenting with Queries and Results

Let's start by trying to replicate the core of their experiment: generating a Verilog description for a spiking neuron array using a Liquid Integrate-and-Fire (LIF) neuron model.

- **Query 1:** "Generate Verilog code for a 4x4 array of Liquid Integrate-and-Fire (LIF) neurons. Each neuron should have inputs for presynaptic spikes and an output for its own spike. Include parameters for membrane capacitance, leak conductance, and firing threshold. Assume a global clock and reset signal. Keep the design modular."

Initial Response (Conceptual Verilog - I cannot execute this directly):

```
module lif_neuron (
    input clk,
    input rst,
    input presynaptic_spike,
    output reg spike_out
);

parameter real MEMBRANE_CAPACITANCE = 1e-9;
parameter real LEAK_CONDUCTANCE = 1e-10;
parameter real FIRING_THRESHOLD = 1e-3;

real membrane_potential;

always @(posedge clk or posedge rst) begin
    if (rst) begin
        membrane_potential <= 0;
        spike_out <= 0;
    end else begin
        // Integrate incoming spike (simplified)
        if (presynaptic_spike) begin
```

```

        membrane_potential <= membrane_potential + 1e-12; // Example synaptic
weight
    end

    // Leak current
    membrane_potential <= membrane_potential - membrane_potential *
(LEAK_CONDUCTANCE / MEMBRANE_CAPACITANCE);

    // Fire condition
    if (membrane_potential >= FIRING_THRESHOLD) begin
        spike_out <= 1;
        membrane_potential <= 0; // Reset after firing
    end else begin
        spike_out <= 0;
    end
end
endmodule

module neuron_array_4x4 (
    input clk,
    input rst,
    input [15:0] input_spikes, // 16 input lines for each neuron
    output [15:0] output_spikes // 16 output spike lines
);
    wire [15:0] neuron_spikes_internal;
    genvar i, j;

    generate
        for (i = 0; i < 4; i = i + 1) begin : row
            for (j = 0; j < 4; j = j + 1) begin : col
                lif_neuron neuron_instance (
                    .clk(clk),
                    .rst(rst),
                    .presynaptic_spike(input_spikes[i*4 + j]), // Example input connection
                    .spike_out(neuron_spikes_internal[i*4 + j])
                );
            end
        end
    endgenerate

    assign output_spikes = neuron_spikes_internal;

```

```
endmodule
```

Query 2 (Refining Connectivity): "Modify the neuron_array_4x4 module to connect each neuron to its immediate neighbors (north, south, east, west). Handle boundary conditions where neighbors don't exist. Assume a simple additive synaptic connection where a presynaptic spike adds a fixed value to the postsynaptic membrane potential."

Refined Response (Conceptual Verilog):

```
module neuron_array_4x4_connected (
    input clk,
    input rst,
    input [15:0] external_input_spikes, // External stimuli
    output [15:0] output_spikes // Spikes from the array
);

parameter real SYNAPTIC_WEIGHT = 5e-13; // Example synaptic weight

wire [15:0] neuron_spikes_internal;
wire [3:0][3:0] spike_grid;
reg [3:0][3:0] membrane_potential;

genvar i, j;

// Instantiate LIF neurons
generate
    for (i = 0; i < 4; i = i + 1) begin : row
        for (j = 0; j < 4; j = j + 1) begin : col
            lif_neuron #(
                .MEMBRANE_CAPACITANCE(1e-9),
                .LEAK_CONDUCTANCE(1e-10),
                .FIRING_THRESHOLD(1e-3)
            ) neuron_instance (
                .clk(clk),
                .rst(rst),
                .presynaptic_spike(spike_grid[i][j]), // Input from neighbors
                .spike_out(spike_grid[i][j]) // Output spike
            );
        end
    end
endgenerate
```

```

// Connectivity logic
always @(posedge clk or posedge rst) begin
    if (rst) begin
        membrane_potential <= '0;
    end else begin
        for (i = 0; i < 4; i = i + 1) begin
            for (j = 0; j < 4; j = j + 1) begin
                real current_potential = membrane_potential[i][j];

                // Integrate external input
                if (external_input_spikes[i*4 + j]) begin
                    current_potential = current_potential + SYNAPTIC_WEIGHT;
                end

                // Integrate neighbor spikes
                if (i > 0 && spike_grid[i-1][j]) current_potential = current_potential +
SYNAPTIC_WEIGHT; // North
                if (i < 3 && spike_grid[i+1][j]) current_potential = current_potential +
SYNAPTIC_WEIGHT; // South
                if (j > 0 && spike_grid[i][j-1]) current_potential = current_potential +
SYNAPTIC_WEIGHT; // West
                if (j < 3 && spike_grid[i][j+1]) current_potential = current_potential +
SYNAPTIC_WEIGHT; // East

                // Leak
                current_potential = current_potential - current_potential * (1e-10 / 1e-9);

                // Fire and reset
                if (current_potential >= 1e-3) begin
                    spike_grid[i][j] <= 1;
                    membrane_potential[i][j] <= 0;
                end else begin
                    spike_grid[i][j] <= 0;
                    membrane_potential[i][j] <= current_potential;
                end
            end
        end
    end
end

// Flatten the output spikes
generate
    for (i = 0; i < 4; i = i + 1) begin

```

```

        for (j = 0; j < 4; j = j + 1) begin
            assign output_spikes[i*4 + j] = spike_grid[i][j];
        end
    end
endgenerate

endmodule

```

4. Keeping Track of Queries

1. "Generate Verilog code for a 4x4 array of Liquid Integrate-and-Fire (LIF) neurons..." (Initial attempt with floating-point)
2. "Modify the neuron_array_4x4 module to connect each neuron to its immediate neighbors..." (Adding connectivity with floating-point)
3. "Simplify the neuron_array_4x4_connected module to use only binary signals..." (Moving towards synthesizable Verilog)

Going Down to ASIC with OpenLane (Hypothetical)

If we were to proceed to ASIC implementation, the `neuron_array_4x4_binary.v` file would be the starting point. The typical OpenLane flow involves:

1. **Synthesis:** Using a synthesis tool (like Yosys) to convert the Verilog into a gate-level netlist.
2. **Placement:** Placing the standard cells (logic gates) on the chip area.
3. **Routing:** Connecting the placed cells according to the netlist.
4. **Clock Tree Synthesis (CTS):** Designing and routing the clock distribution network.
5. **Physical Verification:** Checking for design rule violations (DRC) and layout vs. schematic (LVS) errors.
6. **GDSII Generation:** Producing the final layout file in GDSII format, which is used for fabrication.

To use OpenLane, you would typically create a configuration file specifying the design files, top-level module, and target technology node. Then, you would run the OpenLane flow. This process is computationally intensive and requires a proper setup of the OpenLane environment.

6. Comparing Results

Without having the exact Verilog code generated by ChatGPT in the Johns Hopkins paper, a direct code-level comparison is impossible. However, we can compare the *approach* and the *level of abstraction*:

- **Johns Hopkins Paper:** They likely started with more natural language-like prompts describing the desired architecture and neuron behavior. The paper would detail how well ChatGPT understood these descriptions and translated them into functional HDL. They might have shown examples of the prompts and the resulting code snippets, highlighting any necessary manual corrections or refinements.
- **My Approach (Simulated):** I broke the problem down into smaller, more specific queries. The initial attempts used floating-point numbers, which are not directly synthesizable. The subsequent refinement focused on using binary representations to make the design more suitable for hardware implementation.

Potential Differences:

- **Neuron Model Complexity:** The Johns Hopkins paper might have explored more sophisticated LIF neuron models or different spiking neuron models altogether.
- **Connectivity Schemes:** Their array might have had more complex or configurable connectivity patterns.
- **Parameterization:** They might have investigated how well the LLM handled different parameters and their ranges.
- **Code Quality and Synthesizability:** The paper would likely discuss the quality of the generated Verilog in terms of correctness, efficiency, and ease of synthesis.

7. Improvements to Their Solution

Based on my experience and the general challenges of using LLMs for hardware design, here are some potential improvements:

- **More Structured Prompting:** Instead of purely natural language, using a more structured format for prompts (e.g., specifying input/output ports, parameters in a specific format) could lead to more predictable and accurate HDL generation.
- **Iterative Refinement with Feedback Loops:** The paper likely involved some level of iteration. Explicitly incorporating a feedback loop where the LLM's output is analyzed (simulated or even synthesized partially) and the results are fed back into the LLM with specific instructions for improvement could be beneficial.
- **Library of Pre-defined Modules:** Instead of generating everything from scratch, providing the LLM with a library of well-defined and verified modules (e.g., a basic LIF neuron, a synapse model) and asking it to instantiate and connect them based on a high-level description might be more effective.
- **Formal Verification Hints:** Guiding the LLM to generate code that is more amenable to formal verification by suggesting the inclusion of assertions or invariants could improve the reliability of the generated hardware.
- **Targeted for Specific Synthesis Tools:** Tailoring the prompts and the expected output format to be directly compatible with specific synthesis tools (like Yosys for OpenLane) could streamline the design flow.

- **Incorporating Timing Constraints:** If the goal is ASIC implementation, future work could involve prompting the LLM to consider basic timing constraints

Improvements to Their Solution (Building on Previous Points)

Here are some general improvements that could be explored when using LLMs for hardware design, potentially building upon the Johns Hopkins team's work:

- **Formal Specification Integration:** Instead of relying solely on natural language, incorporating elements of formal specification (even in a simplified manner within the prompt) could guide the LLM towards more correct and verifiable HDL. For example, specifying the expected behavior of the neuron model in a more structured way.
- **Version Control and Tracking of LLM Generations:** Implementing a system to track different versions of the generated code based on different prompts and LLM iterations would be valuable for debugging and understanding the LLM's learning process.
- **Automated Verification Script Generation:** Exploring the LLM's ability to generate basic test benches or simulation scripts alongside the HDL could further automate the design flow.
- **Constraint Specification:** For ASIC implementation, guiding the LLM to consider basic constraints like area or power targets (even if in a high-level way) could be a future direction.
- **LLM Fine-tuning on Hardware Design Data:** Fine-tuning an LLM on a dataset of well-documented and verified hardware designs (including spiking neural networks) could significantly improve its ability to generate correct and efficient HDL.

8. Design with RLU and Hodgkin-Huxley Neuron Models

Okay, let's try generating conceptual Verilog for an array using a Rectified Linear Unit (RLU) and then discuss the Hodgkin-Huxley model.

a) RLU Neuron Array

The RLU is a much simpler activation function than a LIF neuron. In a hardware context, it's essentially a conditional pass-through of the input.

Query: "Generate Verilog code for a 4x4 array of Rectified Linear Units (RLUs). Each RLU should take an input and produce an output that is the input if it's positive (or zero), and zero otherwise. Assume a global clock and reset, although they might not be strictly necessary for a purely combinational RLU. Make the design modular."

Conceptual Verilog for RLU Array:

```
module rlu_neuron (
    input clk, // Included for potential synchronous operation or integration
```

```

input rst, // Included for potential synchronous operation or integration
input signed [7:0] in, // Assuming an 8-bit signed input
output reg signed [7:0] out
);

always @(posedge clk or posedge rst) begin
    if (rst) begin
        out <= 0;
    end else begin
        if (in > 0) begin
            out <= in;
        end else begin
            out <= 0;
        end
    end
end

endmodule

module rlu_array_4x4 (
    input clk,
    input rst,
    input signed [7:0] input_values [15:0], // Array of 16 input values
    output signed [7:0] output_values [15:0]
);
    genvar i, j;

    generate
        for (i = 0; i < 4; i = i + 1) begin : row
            for (j = 0; j < 4; j = j + 1) begin : col
                rlu_neuron rlu_instance (
                    .clk(clk),
                    .rst(rst),
                    .in(input_values[i*4 + j]),
                    .out(output_values[i*4 + j])
                );
            end
        end
    endgenerate

endmodule

```

In many neural network implementations, the RLU is a purely combinational operation. The inclusion of clk and rst here makes the rlu_neuron module potentially synchronous, which might be useful for integrating it into a larger synchronous system. A purely combinational version would simply use an assign statement:

```
module rlu_neuron_comb (
    input signed [7:0] in,
    output signed [7:0] out
);
    assign out = (in > 0) ? in : 0;
endmodule
```

The array instantiation would then be adjusted accordingly.

b) Hodgkin-Huxley Neuron Model

The Hodgkin-Huxley model is significantly more complex than the LIF neuron, involving multiple state variables and differential equations to describe the dynamics of ion channels. Implementing a full Hodgkin-Huxley neuron in hardware is a substantial undertaking.

Query: "Describe the key components and state variables of the Hodgkin-Huxley neuron model and outline the challenges of implementing it in Verilog for an array of neurons."

Response on Hodgkin-Huxley Implementation:

The Hodgkin-Huxley model describes the electrical activity of excitable cells, such as neurons, by modeling the flow of specific ions (sodium and potassium) across the cell membrane. Its key components and state variables include:

- **Membrane Potential (V_m):** The voltage difference across the neuron's membrane.
- **Sodium Channels:**
 - **Activation Gate (m):** Represents the probability that a sodium channel's activation gate is open. It depends on the membrane potential and has its own dynamics.
 - **Inactivation Gate (h):** Represents the probability that a sodium channel's inactivation gate is open. It also depends on the membrane potential and has its own dynamics (slower than activation).
- **Potassium Channels:**

- **Activation Gate (n):** Represents the probability that a potassium channel's activation gate is open (four independent gates). It depends on the membrane potential and has its own dynamics.
- **Leak Current (I_L):** Represents the passive flow of other ions across the membrane.
- **Capacitance (C_m):** The capacitance of the cell membrane.
- **Conductances (g_Na, g_K, g_L):** Maximum conductances for sodium, potassium, and leak channels, respectively.
- **Reversal Potentials (E_Na, E_K, E_L):** The equilibrium potentials for each ion.

The dynamics of these variables are described by a set of coupled non-linear differential equations:

$$dtdVm = Cm \cdot \text{ext} - I_{Na} - I_K - I_L$$

where:

$$I_{Na} = g_{Na} \cdot m^3 \cdot h \cdot (V_m - E_{Na}) \quad I_K = g_K \cdot n^4 \cdot (V_m - E_K) \quad I_L = g_L \cdot (V_m - E_L)$$

And the dynamics of the gating variables (m , h , n) are given by first-order differential equations of the form:

$$dtdx = \alpha_x(V_m) \cdot (1-x) - \beta_x(V_m) \cdot x$$

where x represents m , h , or n , and $\alpha_x(V_m)$ and $\beta_x(V_m)$ are voltage-dependent rate constants.

Challenges of Implementing Hodgkin-Huxley in Verilog for an Array:

- **Continuous-Time Dynamics:** The Hodgkin-Huxley model is inherently continuous-time. Hardware description languages like Verilog are discrete-time. Discretization methods (e.g., Euler method, Runge-Kutta) would need to be employed to approximate the differential equations. This introduces complexity and potential for numerical errors.
- **Non-linear Functions:** The rate constants α and β are typically non-linear functions of the membrane potential (often exponentials). Implementing these accurately and efficiently in hardware requires careful approximation or the use of lookup tables.
- **Multiple State Variables per Neuron:** Each Hodgkin-Huxley neuron has multiple state variables (membrane potential, and the gating variables m , h , n), requiring significant storage (registers) per neuron.
- **Complex Interconnections:** Simulating a network of Hodgkin-Huxley neurons requires handling the synaptic connections, which themselves can have complex dynamics.
- **Computational Resources:** Simulating even a small array of Hodgkin-Huxley neurons in hardware would be computationally intensive due to the number of calculations required at each time step.

- **Hardware Area and Power:** Implementing the complex logic for each neuron and the interconnections would lead to a large area footprint and significant power consumption on an ASIC.

Conceptual Approach (Very High-Level):

Implementing a Hodgkin-Huxley array in Verilog would likely involve:

1. **Discretization:** Choosing a suitable numerical method to approximate the differential equations. This would involve defining a discrete time step.
2. **State Variable Representation:** Representing the continuous state variables (V_m , m , h , n) using fixed-point or floating-point numbers with sufficient precision.
3. **Function Approximation:** Implementing the non-linear rate constants α and β using polynomial approximations, piecewise linear approximations, or lookup tables.
4. **Modular Neuron Design:** Creating a Verilog module for a single Hodgkin-Huxley neuron that takes inputs (e.g., synaptic currents) and outputs the membrane potential (or spikes). This module would internally update the state variables based on the discretized equations.
5. **Array Instantiation:** Instantiating multiple instances of the Hodgkin-Huxley neuron module and defining the desired connectivity between them.
6. **Simulation and Verification:** Thoroughly simulating the Verilog design to ensure it captures the essential dynamics of the Hodgkin-Huxley model and the network behavior.

Conclusion on Hodgkin-Huxley:

While conceptually possible, implementing a full Hodgkin-Huxley neuron array down to the ASIC level using an LLM (even with significant manual guidance) would be a very challenging project due to the model's complexity and the need for careful numerical implementation and hardware optimization. It's a significant leap from the simpler LIF model.

You would likely need to break down the problem into much smaller, manageable steps for the LLM and provide very specific instructions on how to implement each part of the model in discrete time using synthesizable Verilog.

Let me know if you'd like me to try generating a simplified Verilog module for a single discretized component of the Hodgkin-Huxley model (e.g., the sodium channel dynamics).