Technical Portfolio: Systems Engineering & Automation

Author: Dhruv Manjrekar

Objective: Demonstration of OS-level interaction and Memory Management logic.

Project 1: Pointer based memory addressing simulation (C Language)

Overview

A low-level demonstration of pointer arithmetic and memory addressing. This module simulates a "low-level memory manipulation in user space" operation where data is manipulated directly via physical memory addresses rather than variable assignment.

Technical Highlights

- **Language:** C (Standard 99)
- **Concepts:** Pointers (`*ptr`), Address-of Operators (`&`), Hexadecimal Memory Tracing.
- **Logic:**
  1. Allocates stack memory for a target integer.
  2. Initializes a pointer to the hardware address.
  3. Modifies the value through pointer dereferencing to demonstrate address-level memory access

Code Snippet

```
// Pointer based memory write
int* ptr = &target_value;
printf("Overwriting memory at address %p...", ptr);
*ptr = 999;
```

Project 2: High-Performance Matrix Operation Kernel (C Language)

Overview

An optimized linear algebra engine designed to perform matrix operations (Multiplication, Transposition) in memory-constrained environments. This project focuses on **dynamic memory management** and algorithm efficiency, bridging the gap between mathematical theory and system performance.

Technical Highlights

- **Language:** C (Standard 99)
- **Concepts:** Dynamic Allocation (`malloc`/`free`), Double Pointers (`**ptr`), Time Complexity (O(n^3)).
- **Logic:**
  1. **Dynamic Allocation:** Runtime generation of N x N matrices (avoiding stack overflow on large datasets).
  2. **Pointer Arithmetic:** Traversal of 2D arrays using pointer offsets rather than simple indexing for deeper hardware understanding.
  3. **Memory Hygiene:** rigorous deallocation routines to prevent memory leaks—critical for embedded systems.

Code Snippet

```
// Dynamic 2D Array Allocation
int** matrix = (int**)malloc(rows * sizeof(int*));
for(int i = 0; i < rows; i++) {
    matrix[i] = (int*)malloc(cols * sizeof(int));
}
```

```
// Logic: Matrix Multiplication Kernel
for (int i = 0; i < r1; i++) {
    for (int j = 0; j < c2; j++) {
        for (int k = 0; k < c1; k++) {
            result[i][j] += matA[i][k] * matB[k][j];
        }
    }
}
```

Project 3: Bare-Metal Neural Inference Engine (Experimental)

Overview

An experimental module built on top of the matrix computation kernel. This program implements a single-layer neural network from scratch in C, introducing non-linear activation functions (`sigmoid`) to the linear algebra engine. It serves as a "proof of concept" that complex AI behaviors can be executed on low-level hardware without heavy external libraries.

Technical Highlights

- **Language:** C (Standard 99)
- **Math:** Sigmoid Activation (1 / (1 + e^-x)), Random Weight Initialization.
- **Architecture:** Manual Forward Propagation (Input Layer -> Hidden Layer).
- **Objective:** To clarify the internal mathematical structure of basic machine learning models through manual implementation.

Code Snippet

```
// The "Neuron" Logic: Weighted Sum + Activation
for (int k = 0; k < inputs; k++) {
    sum += input[i][k] * weights[k][j];
}
// Applying Non-Linearity (Sigmoid)
output[i][j] = 1.0 / (1.0 + exp(-sum));
```

Project 4: Hybrid FFI Accelerator (Python + C Shared Library)

Overview

This project focuses on computational optimization through native code integration. While later projects focus on distributed and networked systems, this project focuses on computational speed. It implements a **Foreign Function Interface (FFI)** bridge, allowing a Python script to offload CPU-intensive tasks to a custom-compiled C shared library (`.dll`).

Technical Highlights

- **Technology:** `ctypes` (Python), GCC Shared Object Compilation (C).
- **Concept:** Application Binary Interface (ABI) bridging.
- **Result:** Achieved a **~60x performance increase** over pure Python by bypassing the Global Interpreter Lock (GIL) for heavy arithmetic.
- **Why it matters:** This follows a design pattern commonly used in high-performance scientific computing libraries. **Performance Benchmark:** Execution time was compared against an equivalent pure Python implementation on identical input sizes.

```
PS C:\Users\HP\Desktop\CS> python ffi_bridge.py
--- PROJECT 6: HYBRID FFI ACCELERATOR ---
Workload: Processing 100000 computational units...

[1] Running Pure Python Kernel...
    -> Time: 0.0121 seconds
[2] Running C Shared Library (FFI)...
    -> Time: 0.0002 seconds

--- RESULTS ---
SPEEDUP FACTOR: 60.80x FASTER
CONCLUSION: HYBRID ARCHITECTURE VALIDATED.
```

Figure 1: alt text

Performance Benchmark

Code Structure

```
# Loading the custom C kernel into Python memory
lib = ctypes.CDLL("./fast_core.dll")

# Defining binary interface (Pointer types)
lib.cpu_intensive_task.argtypes = [ctypes.POINTER(ctypes.c_int), ctypes.c_int]

# Executing C code from Python
lib.cpu_intensive_task(c_array, size)
```

Project 5: Distributed Tensor Processing Node (Python)

Overview

This project integrates previous system components into a networked application. This project wraps the mathematical logic of the Neural Engine (Project 3) into a networked microservice. It establishes a TCP server that accepts serialized vector data from remote clients, performs server-side inference, and returns predictions in real-time. This reflects common architectural patterns used in cloud-based inference services.

Technical Highlights

- **Language:** Python 3.x
- **Libraries:** `socket` (Network Layer), `json` (Data Serialization), `math` (Sigmoid Logic).
- **Architecture:** Client-Server Model (Request-Response Cycle).
- **Key Feature:** Hot-swappable "Weights" allowing the model to be updated without restarting the server.

Code Structure

```
# Server-Side Inference Logic
def handle_client(conn, addr):
    data = conn.recv(1024).decode()
    vector = json.loads(data) # Deserialize JSON input

    # Compute: Inputs * Weights + Bias
    result = perform_sigmoid_inference(vector)

    # Return JSON response
    response = json.dumps({"status": "200", "prediction": result})
    conn.send(response.encode())
```

Project 6: Multi-Threaded Network Port Scanner (Python)

Overview

A network diagnostic utility designed to map open ports on target IP addresses. Unlike sequential scanners, this tool utilizes **multi-threading** to execute concurrent socket connections, significantly reducing scan latency for large address ranges. This demonstrates a practical application of concurrency and TCP/IP handshake protocols.

Technical Highlights

- **Language:** Python 3.x
- **Libraries:** `socket` (Low-level networking), `threading` (Concurrency), `time` (Latency tracking)
- **Key Logic:**
  - **Socket Management:** Direct socket creation with specified timeouts to prevent hanging on filtered ports.
  - **Concurrency:** Spawns worker threads to scan independent port blocks simultaneously.
  - **Resource Safety:** Implements strict socket closure and error exception handling (`try/except`) to maintain stability.

Code Structure

```
def scan_port(ip, port):
    try:
        sock = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
        sock.settimeout(0.5)
        # 0 = Success (Port Open)
        if sock.connect_ex((ip, port)) == 0:
            print(f"[+] Port {port} is OPEN")
        sock.close()
    except:
        pass
```

Conclusion

This portfolio presents a structured progression through core areas of systems engineering, beginning with low-level memory manipulation and advancing toward high-performance and distributed computing architectures. Each project was designed to reinforce a specific layer of the computing stack, emphasizing both theoretical understanding and practical implementation.

At the foundational level, the projects demonstrate direct interaction with memory, pointer arithmetic, and dynamic allocation, establishing a strong grasp of how software interfaces with hardware. This foundation is extended through the development of optimized computational kernels and hybrid architectures, where performance-critical logic is delegated to C-based implementations while maintaining higher-level control in Python.

The portfolio culminates in networked and concurrent systems that integrate computation, communication, and resource management. Together, these works reflect a consistent focus on efficiency, correctness, and system-level reasoning.

This body of work aligns with the requirements of a rigorous engineering curriculum and reflects readiness for advanced study in computer systems, low-level programming, and high-performance computing at a technical university.