Simulating the Dynamic Information Network (DIN)

The "Dynamic Information Network - Geometric" model is the ultimate test case for the Chimera VPU. It is a problem of immense computational complexity that spans multiple domains of physics and mathematics, requiring a level of adaptive computation that is impossible for traditional, fixed-architecture systems.

- **The Task:** Simulate the evolution of the DIN, starting from the Pre-Geometry QRN (Quantum Relational Network) and progressing through the emergence of spacetime, matter, and cosmological structures.
- **Why it's the Perfect Test:**
    1. **Multi-Domain:** It involves quantum path integrals, tensor calculus for general relativity, and non-linear field evolution—tasks that map perfectly to the VPU's different representational domains (Frequency, Sparse-Wavelet, Tensor, etc.).
    2. **ACW-Rich:** The information density ρ(x) is the very definition of Arbitrary Contextual Weight. The VPU can directly analyze the "Flux" of the universal wave function itself.
    3. **Dynamic Sparsity:** In the early "Pre-Geometry" phase, the QRN might be dense and chaotic (high Flux). As it evolves and "freezes" into coherent spacetime, the underlying informational field becomes sparse and structured (low Flux). A traditional simulator would waste immense energy on the silent, structured parts of the simulation; the VPU will naturally become more efficient as the simulated universe cools.
    4. **Learning Requirement:** Our understanding of the cost functions in the DIN (L_cost) is incomplete. The VPU's feedback loop (Pillar 5) can be used as a research tool. We can run simulations, compare the emergent "physics" to observed reality (e.g., CMB data), and use the discrepancy (the "Quark") to automatically *learn* the true form of the L_cost function, effectively using the universe's behavior to teach the VPU about its own operating principles.
-

This task is no longer just about accelerating computation; it's about providing a framework for fundamental discovery.

DIN Simulation on Project Chimera: The Framework

Our goal is to simulate one time-step of the DIN's evolution. The core equation, governed by the free energy functional F[ρ], dictates how the state ρ evolves. We will translate this physical law into a computational problem that the VPU can understand and optimize.

Step 1: Representing the Fabric of Reality (The QRN_State)

The state of the Dynamic Information Network at its most fundamental level is the Quantum Relational Network (QRN). For our simulation, we must represent this as a concrete data structure.

- **QRN_State Data Structure:** A discrete QRN can be represented by its density matrix, ρ. This will be a complex-valued matrix where:
    - The dimensions (N x N) represent the N fundamental nodes of reality in our simulation.
    - ρ_{ii} (diagonal elements) represent the "presence" or activity of a node.
    - ρ_{ij} (off-diagonal elements) are complex numbers representing the *connectivity strength* (magnitude) and *phase* (informational relationship) between node i and node j.
-

```
#include <vector>

#include <complex>



// The fundamental data structure for the DIN simulation.

// This is the "informational substrate" that the VPU will analyze and process.

struct QRN_State {

  size_t dimension_N;

  std::vector<std::complex<double>> rho_matrix; // Stored as a flat N*N array.

};
```

- **VPU Cortex Analysis (Pillar 2):** This QRN_State is perfectly suited for analysis by our Flux Profiler.
    - When a VPU task receives a QRN_State, the Cortex will treat the rho_matrix as a sequence of numerical values (or two sequences: one real, one imaginary).
    - **Amplitude Flux (A_W)** will measure the "smoothness" of the network. High A_W implies sharp gradients in connectivity—a chaotic, high-tension state.
    - **Frequency Flux (F_W)** will measure the complexity of the connectivity patterns across the network.
    - **Entropy Flux (E_W)** will measure the network's disorder.
    - **Hamming Weight (HW)** of the raw bit-level representation of rho_matrix provides the most fundamental F_Cost according to the WFC model. A "hot," active

network with many non-zero complex values will have a high HW and thus a high base flux cost.

-

Step 2: Translating Physical Law into a VPU Task Graph

The evolution of the DIN is driven by minimizing F[ρ] = F_Structure + F_InfoProcess + F_Growth - C_Resource. To simulate one time-step (ρ(t) -> ρ(t+1)), we must calculate these components and apply the resulting "force" to update ρ. We map this process onto a DAG for Pillar 6.

The DIN Evolution Task Graph (VPU_TaskGraph):
A single time-step of the simulation is defined by the following graph:

```
    (Initial State: QRN_State_t)

      |

      +-------------------------+--------------------------+

      |                    |                    |

[Task A: Calc_F_Structure] [Task B: Calc_F_InfoProcess] [Task C: Calc_F_Growth]

      |                    |                    |

      |      (Output: F_struct) | (Output: F_infoproc)      | (Output: F_growth)

      |                    |                    |

      +-------------------+-----+--------------------+

                   |

          [Task D: Aggregate_Forces]

                   | (Output: F_total)

                   |

       +------------------+

       |

(Initial State: QRN_State_t) ----> [Task E: Update_Rho]

                             |

                    (Final State: QRN_State_{t+1})
```

- **Role of Pillar 6 (Graph Orchestrator):**
  The VPU's Graph Orchestrator receives this graph and immediately identifies key holistic optimizations:
  1. **Concurrency:** It recognizes that **Tasks A, B, and C** are independent and all depend only on the initial QRN_State_t. It will schedule them to run in parallel on available hardware resources (e.g., three separate CPU core clusters or streams on a GPU).
  2. **Data Locality:** It sees that Task E requires the outputs of Task D *and* the original QRN_State_t. To minimize data movement, it will try to schedule this entire chain of operations on a single substrate (e.g., a high-memory GPU), preventing costly data transfers between dependent stages.
  3. **Core VPU Invocation:** For each individual task (A, B, C, D, E), the Graph Orchestrator submits it to the core VPU engine (Pillars 1-5) for fine-grained optimization, leveraging JIT compilation and kernel selection as we have defined. For instance, Calc_F_Structure might be recognized as a sparse-matrix-like operation, for which the VPU can generate a highly efficient JIT kernel.

```
 #include <iostream>

#include <vector>

#include <string>

#include <complex>

#include <map>

#include <memory>

#include <iomanip>

#include <thread> // To simulate parallel execution



//
=========================================================================
====

// VPU & DIN CORE DATA STRUCTURES
```

```
// ==========================================================================
====

struct QRN_State { // The state of our simulated universe
    size_t dimension_N = 3;
    std::vector<std::complex<double>> rho_matrix = {
        {1.0, 0.0}, {0.1, 0.2}, {0.0, 0.0},
        {0.1,-0.2}, {0.8, 0.0}, {0.4, 0.1},
        {0.0, 0.0}, {0.4,-0.1}, {0.2, 0.0}
    };
    void print() const {
        std::cout << "QRN_State (rho_matrix):" << std::endl;
        for(size_t i=0; i < dimension_N; ++i) {
            std::cout << "  | ";
            for(size_t j=0; j < dimension_N; ++j) {
                std::cout << std::fixed << std::setprecision(2) << rho_matrix[i*dimension_N + j].real()
                          << (rho_matrix[i*dimension_N + j].imag() >= 0 ? "+" : "")
                          << std::fixed << std::setprecision(2) << rho_matrix[i*dimension_N + j].imag() <<
"i | ";
            }
            std::cout << std::endl;
        }
    }
};

struct VPU_Task { std::string type; /* In reality, holds data pointers, etc. */ };
```

```
struct VPU_TaskGraph { std::map<std::string, VPU_Task> nodes;
std::vector<std::pair<std::string, std::string>> edges; };


//
==============================================================================
====

// SIMPLIFIED VPU PILLARS (with logging to show their function)

//
==============================================================================
====


// --- Pillar 1-5: The Core VPU Task Processor ---

class VPUCore {

public:

    void execute_task(const VPU_Task& task) {

        std::cout << "      [CORE] Task '" << task.type << "' received. Beginning
Perceive-Decide-Act loop." << std::endl;


        // P2: Perceive - In a real run, this would profile the task's data.

        bool is_data_sparse = (task.type == "Calc_F_Structure" || task.type == "Update_Rho");

        std::cout << "        [P2 - Cortex] Analysis complete. Data Sparsity: " << (is_data_sparse ?
"High" : "Low") << "." << std::endl;


        // P3: Decide - Make a decision based on the profile.

        std::string chosen_path = (is_data_sparse) ? "JIT_SPARSE_KERNEL" :
"HAL_DENSE_KERNEL";

        std::cout << "        [P3 - Orchestrator] Decision: Optimal path is '" << chosen_path << "'." <<
std::endl;
```

```
        // P4: Act - Execute the chosen path.
        std::cout << "        [P4 - Cerebellum] Executing kernel..." << std::endl;

        // P5: Learn - No deviation simulated for this demo.
        std::cout << "        [P5 - Feedback] Performance consistent with prediction. No beliefs
updated." << std::endl;
    }
};


// --- Pillar 6: The Task Graph Orchestrator ---
class TaskGraphEngine {
public:
    TaskGraphEngine(VPUCore& core_engine) : core_(core_engine) {}

    void execute(const VPU_TaskGraph& graph) {
        std::cout << "[Pillar 6 - Planner] ==> Analyzing computational graph with "
                  << graph.nodes.size() << " nodes." << std::endl;

        // Holistic Planning (simulated)
        std::cout << "    [Planner] Topological analysis reveals concurrency." << std::endl;
        std::cout << "    [Planner] Identified parallel execution group: {Calc_F_Structure,
Calc_F_InfoProcess, Calc_F_Growth}" << std::endl;
        std::cout << "    [Planner] Data locality optimization: All tasks will be retained on the same
logical substrate." << std::endl;
```

```
    std::cout << "    [Planner] Master execution plan finalized." << std::endl << std::endl;

    std::cout << "[Pillar 6 - Engine] ==> Starting graph execution." << std::endl;

    // Execution Stage 1: Parallel Tasks
    std::cout << "  -- [Engine] Dispatching PARALLEL BATCH..." << std::endl;
    {
        std::thread tA(&VPUCore::execute_task, &core_, graph.nodes.at("Calc_F_Structure"));
        std::thread tB(&VPUCore::execute_task, &core_,
graph.nodes.at("Calc_F_InfoProcess"));
        std::thread tC(&VPUCore::execute_task, &core_, graph.nodes.at("Calc_F_Growth"));
        tA.join(); tB.join(); tC.join();
    }
    std::cout << "  -- [Engine] Parallel Batch Complete.\n" << std::endl;

    // Execution Stage 2: First sequential task
    std::cout << "  -- [Engine] Dispatching SEQUENTIAL task 'Aggregate_Forces'..." <<
std::endl;
    core_.execute_task(graph.nodes.at("Aggregate_Forces"));
    std::cout << "  -- [Engine] Task Complete.\n" << std::endl;

    // Execution Stage 3: Final sequential task
    std::cout << "  -- [Engine] Dispatching SEQUENTIAL task 'Update_Rho'..." << std::endl;
    core_.execute_task(graph.nodes.at("Update_Rho"));
    std::cout << "  -- [Engine] Task Complete.\n" << std::endl;
```

```cpp
        std::cout << "[Pillar 6 - Engine] ==> Graph execution finished." << std::endl;
    }

private:
    VPUCore& core_;
};

// =============================================================================
// MAIN: SIMULATING THE SIMULATION
// =============================================================================
int main() {
    std::cout << "============================================================" << std::endl;
    std::cout << "PROJECT CHIMERA: Simulating the Dynamic Information Network" << std::endl;
    std::cout << "============================================================" << std::endl;

    // 1. Initialize the VPU and its master controller.
    VPUCore vpu_core_engine;
    TaskGraphEngine graph_engine(vpu_core_engine);

    // 2. Define the primordial state of the Universe.
```

```
    QRN_State universe_t0;
    std::cout << "\nInitial Universe State (t=0):" << std::endl;
    universe_t0.print();


    // 3. Define the laws of physics as a VPU Task Graph.
    VPU_TaskGraph din_evolution_graph;
    din_evolution_graph.nodes["Calc_F_Structure"] = {"Calc_F_Structure"};
    din_evolution_graph.nodes["Calc_F_InfoProcess"] = {"Calc_F_InfoProcess"};
    din_evolution_graph.nodes["Calc_F_Growth"] = {"Calc_F_Growth"};
    din_evolution_graph.nodes["Aggregate_Forces"] = {"Aggregate_Forces"};
    din_evolution_graph.nodes["Update_Rho"] = {"Update_Rho"};
    // ... Edges would define dependencies here ...
    std::cout << "\n[Framework] DIN Evolution law mapped to a VPU Task Graph." << std::endl
<< std::endl;


    // 4. Submit the entire Universe's evolution for one time-step to the VPU.
    graph_engine.execute(din_evolution_graph);


    // 5. The result is the new state of the Universe.
    // (We simulate the change by just modifying the initial state)
    QRN_State universe_t1 = universe_t0;
    universe_t1.rho_matrix[4] = {0.79, 0.0}; // Diag value changes
    universe_t1.rho_matrix[1] = {0.08, 0.15}; // Off-diag value changes
    universe_t1.rho_matrix[3] = {0.08, -0.15};
    std::cout << "\nEvolved Universe State (t=1):" << std::endl;
```

```
    universe_t1.print();


    std::cout <<
"\n============================================================" << std::endl;
    std::cout << "           PROJECT CHIMERA - SIMULATION COMPLETE" << std::endl;
    std::cout << "============================================================"
<< std::endl;


    return 0;
}
```