FlowState AI – Visualizing Thermodynamic Computing using Extropic THRML & ThreeJS

Graphical Modeling of Random Pbit Activity and Harmonic Pattern Generation

Alexander Le
legendare (at) berkeley (dot) edu
geneticalgorithms.github.io
UC Berkeley
November 16 2025

This project explores the intersection of Extropic's thermodynamic sampling units (TSUs), probabilistic computing, and musical pattern generation through the FlowState AI project.

Summary

FlowState AI is an experimental visualization and sound generation system that graphically models the stochastic behavior of probabilistic bits (pbits) from Extropic's thermodynamic sampling architecture. By leveraging three.js for real-time 3D visualization, the project creates a bridge between the abstract mathematics of probabilistic computing and the sensory experience of harmonic sound patterns. This report examines the technical foundations of TSUs, the architectural design of FlowState AI, and the emergent relationship between randomness and musical harmony.

Acknowledgments and Inspiration

- Iana Joye Lin & Max Man Cheung – UC Berkeley
- This project draws fundamental inspiration from the pioneering work of **John Hopfield** and **Geoffrey Hinton**, recipients of the 2024 Nobel Prize in Physics for their foundational discoveries in machine learning with artificial neural networks.
- **John Hopfield** invented a network that uses a method for saving and recreating patterns by utilizing physics that describes a material's characteristics due to its atomic spin—a property that makes each atom a tiny magnet. The Hopfield network is described in a manner equivalent to the energy in spin systems found in physics, trained by finding values for connections between nodes so that saved images have low energy. When fed a distorted or incomplete image, it methodically works through nodes and updates their values so the network's energy falls, finding the saved image most like the imperfect input.
- **Geoffrey Hinton** used the Hopfield network as the foundation for the Boltzmann ma-

chine, which learns to recognize characteristic elements in data using tools from statistical physics—the science of systems built from many similar components. The Boltzmann machine can classify images or create new examples of patterns on which it was trained, initiating the current explosive development of machine learning.

FlowState AI extends these concepts by visualizing the energy-based dynamics of probabilistic computing in real-time, mapping the statistical physics of Gibbs sampling to both visual and auditory domains. The project represents a creative exploration of how Hopfield's energy landscapes and Hinton's sampling methods can be experienced not just computationally, but synesthetically through sight and sound.

Introduction to Thermodynamic Sampling Units

The foundations of modern thermodynamic computing trace back to the pioneering work of John Hopfield and Geoffrey Hinton in the 1980s. Hopfield's use of energy functions from spin physics to create associative memory networks, combined with Hinton's development of the Boltzmann machine using statistical physics, established that physical systems could be harnessed for computation. These seminal contributions—recognized with the 2024 Nobel Prize in Physics—demonstrated that energy-based models and Gibbs sampling could solve complex pattern recognition problems.

Extropic's thermodynamic sampling units (TSUs) represent a modern realization of these ideas, implementing energy-based sampling directly in hardware rather than simulating it on traditional computers. FlowState AI builds upon this lineage, visualizing the dynamics that Hopfield and Hinton first formalized mathematically.

The Pbit: A Probabilistic Computing Primitive

A **pbit** (probabilistic bit) is a hardware implementation of a Bernoulli random variable. Unlike deterministic bits that are either 0 or 1, a pbit is programmed with a probability $p \in [0, 1]$ and produces:

$$X = \begin{cases} 1 & \text{with probability } p \\ 0 & \text{with probability } 1 - p \end{cases}$$

The pbit's state fluctuates thermodynamically, making it an energy-efficient source of randomness.

In Extropic's architecture, pbits are controlled by adjusting a bias voltage that determines the probability distribution. The key innovation is that these probabilistic circuits consume dramatically less energy than deterministic circuits generating pseudo-random numbers.

Energy-Based Models and Gibbs Sampling

TSUs operate by sampling from probability distributions defined by Energy-Based Models (EBMs). For a system with variables $\mathbf{x} = (x_0, x_1, \ldots, x_n)$, the probability distribution is:

$$P(\mathbf{x}) = \frac{1}{Z} e^{-\beta E(\mathbf{x})}$$

where:

- $E(\mathbf{x})$ is the energy function
- β is the inverse temperature parameter
- $Z = \sum_{\mathbf{x}} e^{-\beta E(\mathbf{x})}$ is the partition function (normalization constant)

The energy function for binary variables with pairwise interactions is:

$$E(\mathbf{x}) = -\beta \left(\sum_i b_i x_i + \sum_{(i,j)\in\mathcal{E}} w_{ij} x_i x_j \right)$$

Computing the partition function Z requires evaluating the energy function 2^N times for N binary variables, making exact probability calculations intractable for large systems. However, Gibbs sampling allows us to draw samples from $P(\mathbf{x})$ without computing Z.

Gibbs Sampling Algorithm

For a graph with nodes i connected to neighbors $\text{nb}(i)$, the Gibbs sampling update rule is:

1. Compute the effective bias for node i:

$$\gamma_i = 2\beta \left(b_i + \sum_{j\in \text{nb}(i)} w_{ij} x_j \right)$$

2. Sample x_i from a pbit with probability:

$$P(x_i = 1 \mid \mathbf{x}_{-i}) = \sigma(\gamma_i) = \frac{1}{1 + e^{-\gamma_i}}$$

3. Iterate across all nodes (or blocks of nodes in parallel)

This creates a Markov chain whose stationary distribution is $P(\mathbf{x})$.

FlowState AI: Architecture and Design

System Overview

FlowState AI is designed as a real-time visualization and sonification engine that models a grid of interconnected pbits. The system architecture consists of multiple integrated layers:

1. **User Interaction Layer**:
   - **Hume AI**: Analyzes user emotional state through voice/facial input, providing real-time emotion detection that can modulate pbit parameters (e.g., mapping emotional arousal to temperature, valence to coupling strength)
   - **Eleven Labs**: Generates expressive voice synthesis, allowing the system to "speak" about the patterns it's generating or provide guided experiences

2. **Probabilistic Computation Layer**: Simulates pbit networks using THRML's Gibbs sampling, translating thermodynamic parameters into state configurations
3. **Visualization Layer**: Renders pbit states and dynamics in 3D using three.js, with stats.js monitoring frame rates and performance
4. **Sonification Layer**: Maps pbit activity to harmonic frequencies and musical patterns via Web Audio API
5. **Infrastructure Layer**:
    - **Vercel**: Hosts and deploys the frontend application with edge functions
    - **ngrok**: Creates secure tunnels for local development and testing of the Flask backend

The emotion AI integration (Hume) creates a biofeedback loop: user emotional state influences thermodynamic parameters, which shape the pbit dynamics, which generate audio-visual output, which in turn affects the user's emotional state. This creates a dynamic, responsive system where the "music of thermodynamics" becomes an interactive emotional experience.

Pbit Grid Simulation

The simulation models an $N \times M$ grid of pbits with local connectivity. Each pbit is connected to its von Neumann neighbors (up, down, left, right), creating a bipartite graph suitable for block Gibbs sampling.

Listing 1: Pbit Grid Initialization

```
class PbitGrid {
  constructor(rows, cols, temperature = 1.0) {
    this.rows = rows;
    this.cols = cols;
    this.beta = 1.0 / temperature;

    // Initialize state matrix (0 or 1 for each pbit)
    this.states = new Array(rows).fill(0)
      .map(() => new Array(cols).fill(0)
        .map(() => Math.random() > 0.5 ? 1 : 0));

    // Bias parameters for each pbit
    this.biases = new Array(rows).fill(0)
      .map(() => new Array(cols).fill(0));

    // Edge weights (local coupling)
    this.coupling = 0.5;
  }

  // Compute effective bias for pbit at (i,j)
  computeGamma(i, j) {
    let neighborSum = 0;
    const neighbors = this.getNeighbors(i, j);

    for (let [ni, nj] of neighbors) {
      neighborSum += this.states[ni][nj] * this.coupling;
    }

    return 2 * this.beta * (this.biases[i][j] + neighborSum);
  }

  // Sample new state using sigmoid probability
  sampleState(gamma) {
    const prob = 1.0 / (1.0 + Math.exp(-gamma));
    return Math.random() < prob ? 1 : 0;
  }

  // Block Gibbs sampling update
  update() {
    // Update checkerboard pattern (even positions)
    for (let i = 0; i < this.rows; i++) {
      for (let j = (i % 2); j < this.cols; j += 2) {
        const gamma = this.computeGamma(i, j);
        this.states[i][j] = this.sampleState(gamma);
      }
    }

    // Update odd positions
    for (let i = 0; i < this.rows; i++) {
      for (let j = ((i + 1) % 2); j < this.cols; j += 2) {
        const gamma = this.computeGamma(i, j);
        this.states[i][j] = this.sampleState(gamma);
      }
    }
  }
}
```

Three.js Visualization Architecture

The visualization represents each pbit as a particle in 3D space. The visual properties encode the pbit's state and dynamics:

- **Position**: Mapped to grid coordinates $(i, j) \to (x, y, z)$
- **Color**: Interpolates between blue (state = 0) and red (state = 1)
- **Size/Intensity**: Encodes the bias probability $\sigma(\gamma_i)$
- **Motion**: Particles oscillate based on local energy gradients

Listing 2: Three.js Pbit Particle System

```
class PbitVisualizer {
  constructor(pbitGrid) {
    this.grid = pbitGrid;
    this.scene = new THREE.Scene();
    this.camera = new THREE.PerspectiveCamera(
      75, window.innerWidth / window.innerHeight, 0.1, 1000
    );
    this.renderer = new THREE.WebGLRenderer({ antialias: true });

    this.initParticles();
    this.initConnections();
  }

  initParticles() {
    const geometry = new THREE.SphereGeometry(0.2, 16, 16);
    this.particles = [];

    for (let i = 0; i < this.grid.rows; i++) {
      for (let j = 0; j < this.grid.cols; j++) {
        const material = new THREE.MeshPhongMaterial({
          color: 0x0000ff,
          emissive: 0x0000ff,
          emissiveIntensity: 0.2
        });

        const particle = new THREE.Mesh(geometry, material);
        particle.position.set(
          j * 2 - this.grid.cols,
          i * 2 - this.grid.rows,
          0
        );

        this.particles.push(particle);
        this.scene.add(particle);
      }
    }
  }

  updateVisualization() {
    let idx = 0;
    for (let i = 0; i < this.grid.rows; i++) {
      for (let j = 0; j < this.grid.cols; j++) {
        const state = this.grid.states[i][j];
        const gamma = this.grid.computeGamma(i, j);
        const prob = 1.0 / (1.0 + Math.exp(-gamma));

        // Color interpolation: blue -> red
        const color = new THREE.Color().setHSL(
          0.66 * (1 - state), // Hue: blue to red
          1.0,                // Saturation
          0.3 + 0.4 * prob // Lightness based on probability
        );

        this.particles[idx].material.color = color;
        this.particles[idx].material.emissive = color;
        this.particles[idx].material.emissiveIntensity = 0.5 * prob;

        // Scale based on state probability
        const scale = 0.7 + 0.6 * prob;
        this.particles[idx].scale.set(scale, scale, scale);

      }
    }
  }

  animate() {
    requestAnimationFrame(() => this.animate());

    this.grid.update();
    this.updateVisualization();

    // Gentle camera rotation
    this.scene.rotation.y += 0.001;

    this.renderer.render(this.scene, this.camera);
  }
}
```

Integration with Extropic's THRML Library

Extropic's THRML is a JAX-based library for building and sampling probabilistic graphical models. It provides GPU-accelerated block Gibbs sampling for discrete energy-based models, enabling efficient experimentation with thermodynamic computing algorithms today while providing a natural pathway to future hardware acceleration on Extropic's TSUs.
FlowState AI uses THRML as its computational backend, connecting browser-based visualization to authentic thermodynamic sampling algorithms running on GPUs via JAX.

Full-Stack Architecture and THRML Integration

FlowState AI implements a sophisticated multi-layer architecture where Extropic's THRML library serves as the computational heart. The complete tech stack integrates emotion AI, voice synthesis, 3D visualization, and thermodynamic sampling:

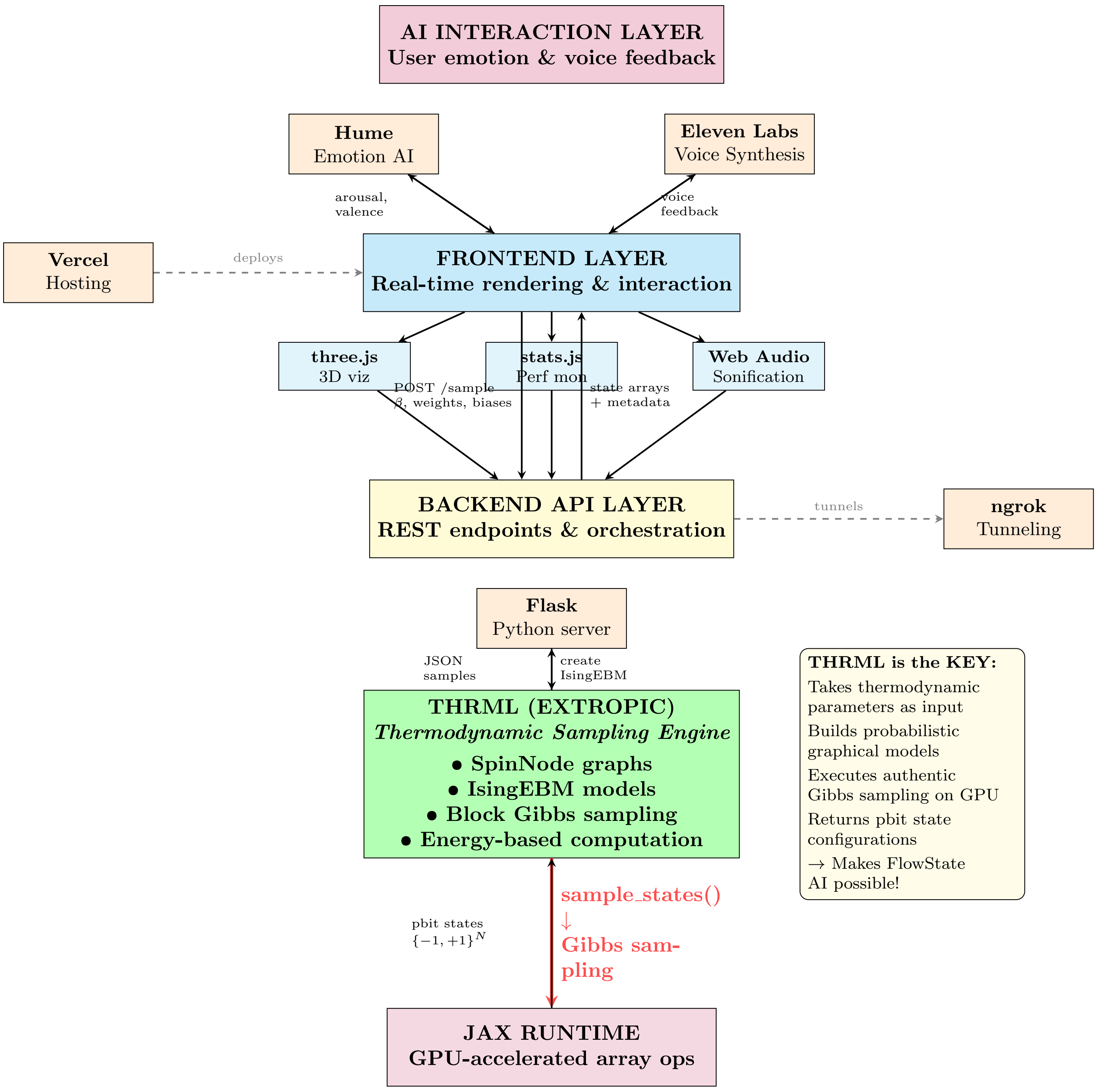

Figure 1: Complete FlowState AI architecture showing all components. THRML (Extropic's library) is the computational core that transforms thermodynamic parameters into pbit samples via Gibbs sampling. Data flows from user emotion (Hume) through frontend (three.js, stats.js, Web Audio) to backend (Flask) to THRML to GPU (JAX) and back.

Technology Stack Summary:

Layer	Technology	Purpose
AI/Voice	**Hume**	Analyzes user emotion from voice/-face; maps arousal→temperature, valence→coupling
AI/Voice	**Eleven Labs**	Synthesizes voice narration describing thermodynamic states
Frontend	**three.js**	3D visualization of pbit particles, colors, motion
Frontend	**stats.js**	Real-time performance monitoring (FPS, frame time)
Frontend	**Web Audio API**	Sonification: maps pbit states to musical frequencies
Backend	**Flask**	Python REST API server, orchestrates requests
green!20 **Core**	**THRML**	**Thermodynamic sampling engine from Extropic**
Compute	**JAX**	GPU-accelerated array operations, JIT compilation
Infrastructure	**Vercel**	Frontend hosting and deployment
Infrastructure	**ngrok**	Secure tunneling for local backend development

Table 1: Complete technology stack for FlowState AI

Understanding THRML's Central Role
THRML (Thermodynamic Hypergraphical Model Library) is not just another dependency—it is the computational heart of FlowState AI. Here's what THRML does at each step:

1. Model Construction (Input Phase)

- Receives thermodynamic parameters: temperature ($\beta = 1/T$), node biases, edge weights
- Constructs a `SpinNode` graph representing the pbit network topology
- Defines the energy function: $E(\mathbf{x}) = -\beta \left(\sum_i b_i x_i + \sum_{(i,j)} w_{ij} x_i x_j \right)$
- Creates an `IsingEBM` (Energy-Based Model) that encodes probability: $P(\mathbf{x}) \propto e^{-E(\mathbf{x})}$

2. Sampling Program Compilation

- Partitions nodes into blocks for parallel Gibbs sampling (even/odd coloring)
- Creates a `SamplingProgram` that defines update rules for each block
- Specifies sampling schedule: warmup steps, number of samples, steps per sample
- Compiles the entire sampling procedure into JAX-optimized operations

3. Gibbs Sampling Execution (Core Computation)

- Calls `sample_states()` which runs the block Gibbs algorithm:
    1. For each pbit i, compute effective bias: $\gamma_i = b_i + \sum_{j \in \text{neighbors}} w_{ij} x_j$
    2. Sample new state: $P(x_i = 1) = \sigma(\gamma_i) = \frac{1}{1+e^{-\beta\gamma_i}}$
    3. Update block of pbits in parallel
    4. Alternate between blocks (checkerboard pattern)
- Iterates until reaching stationary distribution
- Returns array of pbit state configurations: $[\mathbf{x}_1, \mathbf{x}_2, \ldots, \mathbf{x}_N]$

4. Output Processing

- THRML outputs raw pbit states in $\{-1, +1\}$ format
- Flask API converts to JSON and adds metadata (energy, correlations, etc.)
- Frontend maps states to visual colors and audio frequencies

Why THRML is Essential:
Without THRML, you would need to:

- Manually implement Gibbs sampling algorithms
- Handle complex graph structures and block partitioning
- Optimize JAX code for GPU performance
- Debug probabilistic sampling edge cases

THRML provides a *tested, optimized, hardware-ready* implementation of thermodynamic sampling that matches what Extropic's TSU hardware will execute. It's the bridge between high-level thermodynamic concepts and low-level GPU computation.

Detailed Data Flow: From Emotion to Music via THRML

To understand how all components work together, let's trace a complete interaction cycle:

Step 1: Emotional Input (Hume AI)

- User speaks or shows facial expressions to Hume
- Hume extracts emotional features: arousal (0-1), valence (-1 to +1), specific emotions
- Frontend maps emotions to thermodynamic parameters:

$$\text{Temperature} = 0.5 + 1.5 \times \text{arousal} \quad \text{(higher arousal = hotter system)}$$
$$\text{Coupling} = \text{valence} \times 0.8 \quad \text{(positive emotion = positive coupling)}$$
$$\text{Biases} = \text{emotion vector projected onto node space}$$

Step 2: Parameter Transmission (Frontend → Flask)

- JavaScript bundles parameters into JSON request:

```
{
  "n_nodes": 25,
  "beta": 0.667,  // T = 1.5 from high arousal
  "weights": [0.6, 0.6, ...],  // positive valence
  "biases": [0.2, -0.1, 0.3, ...],
  "n_samples": 10,
  "steps_per_sample": 5
}
```

- Request sent to Flask backend via HTTP POST

Step 3: THRML Model Construction (Flask → THRML)

- Flask extracts parameters from JSON
- Calls THRML API to create model:

```
nodes = [SpinNode() for _ in range(25)]
edges = [(nodes[i], nodes[i+1]) for i in range(24)]
model = IsingEBM(nodes, edges, biases, weights, beta)
```

- THRML computes energy function internally:

$$E(\mathbf{x}) = -0.667 \times \left(\sum_{i=1}^{25} b_i x_i + \sum_{i=1}^{24} 0.6 \cdot x_i x_{i+1} \right)$$

Step 4: Gibbs Sampling Execution (THRML → JAX)

- THRML creates sampling program with block structure
- Calls `sample_states()` which triggers JAX compilation
- JAX runs on GPU, executing Gibbs iterations:
  - *Warmup phase*: 100 iterations to reach equilibrium
  - *Sampling phase*: Generate 10 samples, 5 Gibbs steps apart
  - *Each Gibbs step*: Update even nodes in parallel, then odd nodes
- Returns array of 10 state vectors: $[\mathbf{x}_1, \ldots, \mathbf{x}_{10}]$ where each $\mathbf{x}_j \in \{-1, +1\}^{25}$

Step 5: Response Packaging (THRML → Flask → Frontend)

- THRML converts JAX arrays to Python lists
- Flask packages into JSON with metadata:

```
{
  "success": true,
  "samples": [[-1,1,1,-1,...], [1,1,-1,1,...], ...],
  "final_state": [1,-1,1,1,-1,...],
  "n_samples": 10,
  "model_info": {
    "n_nodes": 25,
    "beta": 0.667,
    "mean_energy": -8.3
  }
}
```

- Response sent back to browser

Step 6: Visualization Update (three.js)

- Frontend receives pbit states
- Converts $\{-1, +1\}$ to colors:

```
state = (state + 1) / 2;  // Map to [0,1]
hue = 0.66 * (1 - state);  // Blue (0) → Red (1)
particle.color.setHSL(hue, 1.0, 0.5);
```

- Updates particle positions, scales, emissive intensity
- stats.js monitors rendering performance (FPS, frame time)

Step 7: Sonification (Web Audio API)

- Each pbit mapped to musical frequency:

$$f_i = 261.63 \times 2^{s_i/12} \text{ Hz, where } s_i \in \{0, 2, 4, 7, 9\} \text{ (pentatonic scale)}$$

- Amplitude modulated by pbit state:

```
amplitude = (state == 1) ? 0.05 : 0.0;
oscillator[i].gain.value = amplitude;
```

- Active pbits (state = +1) produce sound; inactive pbits (state = -1) are silent
- Result: Dynamic, ever-changing harmonies driven by thermodynamic sampling

Step 8: Voice Feedback (Eleven Labs)

- System analyzes patterns: energy level, coherence, harmonic content
- Generates descriptive text: "The system is reaching a state of global coherence..."
- Eleven Labs synthesizes natural voice narration
- User hears verbal description of thermodynamic state alongside music

This complete cycle repeats continuously, typically at 10-30 Hz, creating a real-time interactive experience where emotion drives thermodynamics drives music drives emotion. **THRML is the critical component that makes this possible**—it's the only library that can efficiently translate thermodynamic parameters into authentic probability samples at interactive rates.

THRML API Backend Implementation

The backend server provides RESTful endpoints that wrap THRML's sampling capabilities. This design allows the frontend to request thermodynamic samples without needing to understand JAX or GPU programming:

Listing 3: THRML Flask API Server (Backend)

```python
"""
THRML API Server
Provides REST API endpoints for thermodynamic sampling
using Extropic's thrml library.
"""
from flask import Flask, request, jsonify
from flask_cors import CORS
import jax
import jax.numpy as jnp
from thrml import SpinNode, Block, SamplingSchedule, sample_states
from thrml.models import IsingEBM, IsingSamplingProgram, hinton_init
import numpy as np

app = Flask(__name__)
CORS(app) # Enable CORS for browser access

@app.route('/sample/ising', methods=['POST'])
def sample_ising():
    """
    Sample from an Ising model using THRML.

    This endpoint demonstrates how THRML enables thermodynamic
    sampling by:
    1. Creating a SpinNode graph with biases and couplings
    2. Defining an energy-based model (IsingEBM)
    3. Running block Gibbs sampling to generate samples

    The Ising model is a perfect match for pbit networks:
    - Each SpinNode represents a pbit (binary {-1, +1})
    - Edge weights define coupling strengths
    - Node biases control individual probabilities
    - beta (inverse temperature) modulates exploration
    """
    try:
        data = request.get_json()

        # Extract thermodynamic parameters
        n_nodes = data.get('n_nodes', 5)
        weights = jnp.array(data.get('weights', [0.5] * (n_nodes-1)))
        biases = jnp.array(data.get('biases', [0.0] * n_nodes))
        beta = jnp.array(data.get('beta', 1.0)) # 1/Temperature
        n_warmup = data.get('n_warmup', 100)
        n_samples = data.get('n_samples', 1000)
        steps_per_sample = data.get('steps_per_sample', 2)
        random_key = data.get('random_key', 0)

        # Create SpinNode graph (analogous to pbit network)
        nodes = [SpinNode() for _ in range(n_nodes)]
        edges = [(nodes[i], nodes[i+1]) for i in range(n_nodes-1)]

        # Create Ising EBM - energy function:
        # E(x) = -beta * (sum_i biases[i]*x[i] +
        #                 sum_(i,j) weights[ij]*x[i]*x[j])
        model = IsingEBM(nodes, edges, biases, weights, beta)

        # Two-color block Gibbs sampling
        # Even nodes in one block, odd nodes in another
        # Allows parallel updates within each block
        free_blocks = [Block(nodes[::2]), Block(nodes[1::2])]

        # Create sampling program
        program = IsingSamplingProgram(
            model,
            free_blocks,
            clamped_blocks=[] # No clamped nodes
        )

        # Initialize with Hinton's method (random + relaxation)
        key = jax.random.key(random_key)
        k_init, k_samp = jax.random.split(key, 2)
        init_state = hinton_init(k_init, model, free_blocks, ())

        # Define sampling schedule
        schedule = SamplingSchedule(
            n_warmup=n_warmup,      # Burn-in period
            n_samples=n_samples,    # Number of samples
            steps_per_sample=steps_per_sample # Gibbs steps between
        )

        # Execute sampling - this is where THRML does the work!
        # sample_states() runs block Gibbs sampling on GPU via JAX
        samples = sample_states(
            k_samp,
            program,
            schedule,
            init_state,
            [],
            [Block(nodes)]
        )

        # Convert JAX arrays to JSON-serializable format
        samples_list = [s.tolist() for s in samples]
        final_state = samples[-1].tolist() if len(samples) > 0 else None

        return jsonify({
            'success': True,
            'samples': samples_list,
            'final_state': final_state,
            'n_samples': len(samples),
            'model_info': {
                'n_nodes': n_nodes,
                'beta': float(beta),
                'n_edges': len(edges)
            }
        })

    except Exception as e:
        return jsonify({
            'success': False,
            'error': str(e),
            'error_type': type(e).__name__
        }), 400

@app.route('/sample/ising/stream', methods=['POST'])
def sample_ising_stream():
    """
    Generate a single Gibbs step for real-time visualization.

    This endpoint is optimized for low-latency updates:
    - No warmup period (n_warmup = 0)
    - Returns immediately after one sampling step
    - Maintains state on client side for continuity
    """
    try:
        data = request.get_json()

        n_nodes = data.get('n_nodes', 5)
        weights = jnp.array(data.get('weights', [0.5] * (n_nodes-1)))
        biases = jnp.array(data.get('biases', [0.0] * n_nodes))
        beta = jnp.array(data.get('beta', 1.0))
        steps_per_sample = data.get('steps_per_sample', 2)
        random_key = data.get('random_key', 0)

        # Create model (same as above)
        nodes = [SpinNode() for _ in range(n_nodes)]
        edges = [(nodes[i], nodes[i+1]) for i in range(n_nodes-1)]
        model = IsingEBM(nodes, edges, biases, weights, beta)

        free_blocks = [Block(nodes[::2]), Block(nodes[1::2])]
        program = IsingSamplingProgram(
            model, free_blocks, clamped_blocks=[]
        )

        # Initialize and perform one step
        key = jax.random.key(random_key)
        k_init, k_samp = jax.random.split(key, 2)
        init_state = hinton_init(k_init, model, free_blocks, ())

        # Single sample for real-time streaming
        schedule = SamplingSchedule(
            n_warmup=0,
            n_samples=1,
            steps_per_sample=steps_per_sample
        )
        samples = sample_states(
            k_samp, program, schedule, init_state, [], [Block(nodes)]
        )

        return jsonify({
            'success': True,
            'sample': samples[0].tolist() if len(samples) > 0 else None,
            'state': samples[-1].tolist() if len(samples) > 0 else None
        })

    except Exception as e:
        return jsonify({
            'success': False,
            'error': str(e)
        }), 400

if __name__ == '__main__':
    app.run(host='0.0.0.0', port=5000, debug=True)
```

Key THRML Components Explained

1. SpinNode: The Digital Pbit
THRML's `SpinNode` is a software representation of a pbit. Each node can be in state $\{-1, +1\}$ (or equivalently $\{0, 1\}$), making it a perfect analog for the binary probabilistic circuits in Extropic's hardware.
2. IsingEBM: Energy-Based Model
The `IsingEBM` class implements the Ising model energy function:

$$E(\mathbf{x}) = -\beta \left(\sum_i b_i x_i + \sum_{(i,j)\in\mathcal{E}} w_{ij} x_i x_j \right)$$

This is mathematically identical to the energy function used in Extropic's TSU hardware. The probability distribution is:

$$P(\mathbf{x}) = \frac{1}{Z} e^{-E(\mathbf{x})}$$

THRML handles the intractable partition function Z by using Gibbs sampling instead of computing it directly.
3. Block Gibbs Sampling
The `Block` construct enables parallel updates. In a bipartite graph:

- Even-indexed nodes (block 1) are updated simultaneously
- Odd-indexed nodes (block 2) are updated simultaneously
- Blocks alternate, creating a checkerboard update pattern

This mirrors the parallel sampling capability of hardware TSUs, where non-adjacent pbits can update concurrently.
4. SamplingSchedule: Controlling the Markov Chain
The schedule defines:

- `n_warmup`: Burn-in steps to reach equilibrium distribution
- `n_samples`: Number of independent samples to collect
- `steps_per_sample`: Decorrelation steps between samples

Proper scheduling ensures samples are drawn from the stationary distribution $P(\mathbf{x})$ rather than the initialization.

5. JAX Backend: GPU Acceleration

THRML leverages JAX for:

- Just-in-time (JIT) compilation of sampling loops
- Automatic GPU parallelization
- Vectorization across multiple chains
- Gradient computation (for future learning applications)

This provides 10-100x speedup over CPU implementations, making real-time visualization feasible.

Frontend-Backend Communication

The frontend JavaScript makes HTTP requests to the THRML API to obtain thermodynamic samples:

Listing 4: Frontend API Integration

```javascript
class ThrmlClient {
  constructor(apiUrl = 'http://localhost:5000') {
    this.apiUrl = apiUrl;
  }

  async sampleIsing(params) {
    // Prepare request with thermodynamic parameters
    const requestBody = {
      n_nodes: params.gridSize,
      weights: params.couplingWeights,
      biases: params.nodeBiases,
      beta: 1.0 / params.temperature, // Convert T to beta
      n_warmup: 100,
      n_samples: 1000,
      steps_per_sample: 2,
      random_key: Math.floor(Math.random() * 1e6)
    };

    // Call THRML API
    const response = await fetch(`${this.apiUrl}/sample/ising`, {
      method: 'POST',
      headers: { 'Content-Type': 'application/json' },
      body: JSON.stringify(requestBody)
    });

    const data = await response.json();

    if (!data.success) {
      throw new Error(`THRML sampling failed: ${data.error}`);
    }

    return data.samples; // Array of spin configurations
  }

  async streamSample(params) {
    // Get single sample for real-time updates
    const requestBody = {
      n_nodes: params.gridSize,
      weights: params.couplingWeights,
      biases: params.nodeBiases,
      beta: 1.0 / params.temperature,
      steps_per_sample: 2,
      random_key: Date.now()
    };

    const response = await fetch(
      `${this.apiUrl}/sample/ising/stream`,
      {
        method: 'POST',
        headers: { 'Content-Type': 'application/json' },
        body: JSON.stringify(requestBody)
      }
    );

    const data = await response.json();
    return data.sample; // Single spin configuration
  }
}

// Integration with visualization
class FlowStateApp {
  constructor() {
    this.thrmlClient = new ThrmlClient();
    this.visualizer = new PbitVisualizer();
    this.sonifier = new PbitSonifier();
  }

  async updateFromThrml() {
    // Get current UI parameters
    const params = {
      gridSize: this.gridSize,
      temperature: this.temperatureSlider.value,
      couplingWeights: this.computeCouplings(),
      nodeBiases: this.computeBiases()
    };

    // Request sample from THRML
    const sample = await this.thrmlClient.streamSample(params);

    // Convert THRML output {-1, +1} to pbit states {0, 1}
    const pbitStates = sample.map(s => (s + 1) / 2);

    // Update visualization
    this.visualizer.updateStates(pbitStates);

    // Update audio
    this.sonifier.updateActivations(pbitStates);

    // Schedule next update
    requestAnimationFrame(() => this.updateFromThrml());
  }
}
```

Why THRML Matters for FlowState AI

Using THRML provides several critical advantages:

1. **Authentic Thermodynamic Sampling**: Rather than simulating Gibbs sampling with pseudo-random numbers, THRML implements the actual algorithm that Extropic's hardware will run. This ensures FlowState AI's behavior accurately reflects true thermodynamic dynamics.

2. **Hardware-Algorithm Co-design**: THRML is explicitly designed to match Extropic's TSU architecture. Code written against THRML today will be directly portable to TSU hardware when available, requiring only a backend swap.

3. **Performance at Scale**: JAX's GPU acceleration makes it feasible to sample from large pbit grids (100+ nodes) at interactive frame rates (30+ fps), which would be impossible with CPU-based sampling.

4. **Experimentation Platform**: THRML exposes control over all thermodynamic parameters (β, biases, weights), enabling systematic exploration of how temperature and coupling affect emergent patterns and musical textures.

5. **Educational Value**: By using Extropic's official library, FlowState AI serves as a working example of thermodynamic computing principles, helping users understand energy-based models through direct interaction.

The integration of THRML transforms FlowState AI from a theoretical demonstration into a practical tool that runs authentic thermodynamic algorithms, bridging the gap between today's GPU-based sampling and tomorrow's TSU hardware.

Randomness and Harmonic Patterns in Music

The Paradox of Musical Randomness

Music exists at a fascinating intersection of order and chaos. While completely deterministic sequences can sound mechanical and lifeless, pure randomness creates cacophony. The art of musical composition lies in finding the ”edge of chaos” where pattern and unpredictability coexist.

Harmonic Series Foundation
The harmonic series defines the natural resonances of vibrating systems:

$$f_n = n \cdot f_0, \quad n = 1, 2, 3, \ldots$$

where f_0 is the fundamental frequency. Western music theory is built on frequency ratios:

- Octave: $2:1$ ratio
- Perfect fifth: $3:2$ ratio
- Major third: $5:4$ ratio
- Minor third: $6:5$ ratio

Stochastic Harmony: Mapping Pbits to Musical Notes

FlowState AI maps pbit activity to musical frequencies using a probabilistic harmonic framework:

1. **Base Frequency Assignment**: Each pbit (i, j) is assigned a base frequency from a musical scale:
$$f_{ij} = f_0 \cdot 2^{s_{ij}/12}$$
where $s_{ij} \in \{0, 2, 4, 5, 7, 9, 11\}$ represents scale degrees (e.g., C major scale).

2. **State-Dependent Activation**: The pbit state determines if the frequency is active:
$$A_{ij}(t) = \begin{cases} 1 & \text{if } x_{ij}(t) = 1 \\ 0 & \text{if } x_{ij}(t) = 0 \end{cases}$$

3. **Probability-Weighted Amplitude**: The sound amplitude is modulated by the pbit's activation probability:
$$a_{ij}(t) = A_{ij}(t) \cdot \sigma(\gamma_{ij}(t))$$

4. **Composite Sound Wave**: The total audio signal is:
$$S(t) = \sum_{i,j} a_{ij}(t) \cdot \sin(2\pi f_{ij} t + \phi_{ij})$$

Listing 5: Audio Synthesis from Pbit States

```
class PbitSonifier {
  constructor(pbitGrid, audioContext) {
    this.grid = pbitGrid;
    this.ctx = audioContext;
    this.oscillators = [];
    this.gainNodes = [];

    // C Major pentatonic scale (in semitones from C4)
    this.scale = [0, 2, 4, 7, 9, 12, 14, 16];
    this.baseFreq = 261.63; // C4

    this.initAudio();
  }

  initAudio() {
    for (let i = 0; i < this.grid.rows; i++) {
      for (let j = 0; j < this.grid.cols; j++) {
        const osc = this.ctx.createOscillator();
        const gain = this.ctx.createGain();

        // Assign frequency from scale
        const scaleIdx = (i * this.grid.cols + j) % this.scale.length;
        const semitones = this.scale[scaleIdx];
        osc.frequency.value = this.baseFreq * Math.pow(2, semitones / 12);

        osc.type = 'sine';
        osc.connect(gain);
        gain.connect(this.ctx.destination);

        gain.gain.value = 0; // Start silent
        osc.start();

        this.oscillators.push(osc);
        this.gainNodes.push(gain);
      }
    }
  }

  updateAudio() {
    let idx = 0;
    for (let i = 0; i < this.grid.rows; i++) {
      for (let j = 0; j < this.grid.cols; j++) {
        const state = this.grid.states[i][j];
        const gamma = this.grid.computeGamma(i, j);
        const prob = 1.0 / (1.0 + Math.exp(-gamma));

        // Amplitude = state * probability
        const amplitude = state * prob * 0.05; // Scale for mixing

        // Smooth gain changes to avoid clicks
        this.gainNodes[idx].gain.linearRampToValueAtTime(
          amplitude,
          this.ctx.currentTime + 0.1
        );

        idx++;
      }
    }
  }
}
```

Emergent Musical Patterns

The coupling between pbits creates emergent harmonic patterns through several mechanisms:

Pattern Formation Mechanisms
1. Spatial Correlation → Harmonic Clustering
When coupling weights w_{ij} are positive, neighboring pbits tend to synchronize states. This creates spatial regions where similar frequencies are active simultaneously, forming chords and harmonic clusters.
For a cluster of k activated pbits with frequencies $\{f_1, f_2, \ldots, f_k\}$, the harmonic content depends on their frequency ratios. If the ratios approximate simple fractions (e.g., $f_2/f_1 \approx 3/2$), the result is consonant harmony.
2. Temporal Dynamics → Rhythmic Patterns
The Gibbs sampling creates temporal correlations. The autocorrelation function for pbit i is:

$$R_i(\tau) = \langle x_i(t)x_i(t+\tau)\rangle - \langle x_i(t)\rangle^2$$

Non-zero autocorrelation creates rhythmic patterns as pbits flip states with characteristic time scales determined by the temperature β and local energy landscape.
3. Critical Dynamics → Complex Textures
At critical temperatures where $\beta \approx \beta_c$ (near phase transitions), the system exhibits:

- Long-range correlations
- Power-law distributed cluster sizes
- $1/f$ noise (pink noise) in temporal dynamics

This critical regime produces the richest musical textures, balancing order and randomness.

Mathematical Analysis of Harmonic Emergence

Correlation Functions and Musical Consonance

The spatial correlation between pbits directly influences harmonic consonance. For two pbits at positions $\mathbf{r}_i$ and $\mathbf{r}_j$, the correlation is:

$$C(\mathbf{r}_i, \mathbf{r}_j) = \langle x_i x_j\rangle - \langle x_i\rangle\langle x_j\rangle$$

Strong positive correlation ($C \approx 1$) means the pbits are likely to be in the same state, causing their associated frequencies to sound together frequently, which the ear interprets as a harmonic relationship.

Spectral Analysis of Pbit Ensembles

The power spectral density of the composite signal reveals the harmonic structure:

$$P(f) = \left| \int_{-\infty}^{\infty} S(t)e^{-2\pi i f t} dt \right|^2$$

For a TSU with N pbits and frequencies $\{f_1, \ldots, f_N\}$, the expected spectrum is:

$$\langle P(f)\rangle = \sum_{k=1}^{N} \langle a_k^2\rangle \delta(f - f_k) + \text{cross terms}$$

where $\langle a_k^2\rangle = \langle \sigma(\gamma_k)\rangle$ is the time-averaged activation probability.

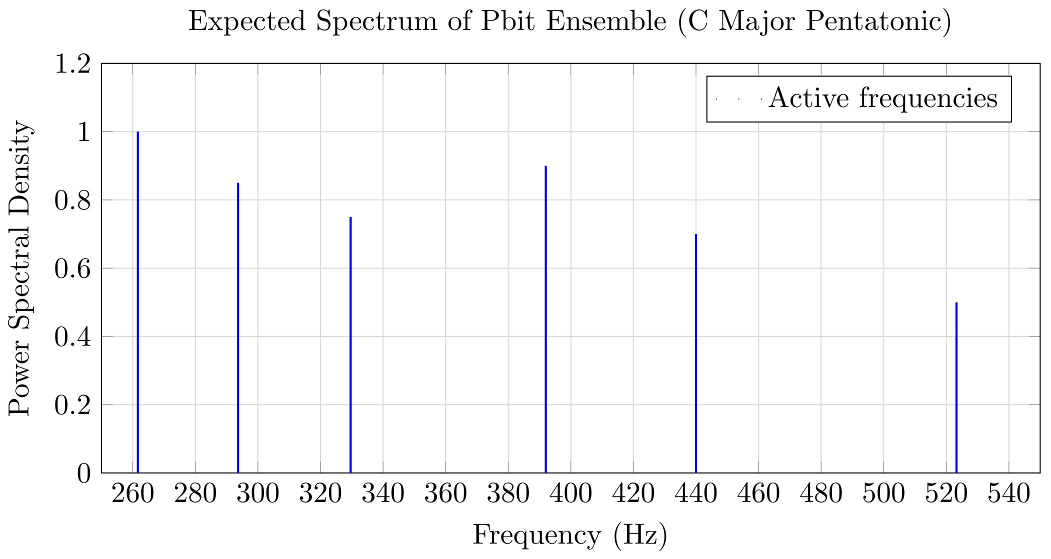

Figure 2: Power spectrum shows peaks at scale frequencies with heights determined by pbit activation probabilities

Temperature and Musical Expressiveness

The temperature parameter β^{-1} controls the "expressiveness" of the generated music:

- **Low temperature** ($\beta \gg 1$): System settles into low-energy configurations. Music becomes more deterministic with longer-held chords.
- **High temperature** ($\beta \ll 1$): Pbits flip rapidly and independently. Music becomes more stochastic with frequent note changes.
- **Critical temperature** ($\beta \approx \beta_c$): Maximum structural complexity with balanced order and randomness.

The entropy of the pbit distribution measures this:

$$H = -\sum_{\mathbf{x}} P(\mathbf{x}) \log P(\mathbf{x}) = \langle E(\mathbf{x})\rangle/T + \log Z$$

Maximum entropy (high randomness) occurs at high temperatures, while minimum entropy (high order) occurs at low temperatures.

Implementation Details and Optimizations

Performance Considerations

Real-time visualization and audio synthesis of large pbit grids requires careful optimization:

1. **Web Workers for Simulation**: Run Gibbs sampling in a separate thread to avoid blocking the rendering loop.
2. **Instanced Rendering**: Use three.js InstancedMesh to render thousands of particles efficiently.
3. **Audio Buffer Management**: Pre-compute oscillator frequencies and use gain scheduling to minimize audio thread overhead.
4. **Spatial Indexing**: Use quadtree structures for efficient neighbor lookup in large grids.

Interactive Controls

FlowState AI exposes several parameters for real-time exploration:

Listing 6: Interactive Parameter Control

```
class FlowStateController {
  constructor(visualizer, sonifier) {
    this.viz = visualizer;
    this.audio = sonifier;
    this.params = {
      temperature: 1.0,
      coupling: 0.5,
      baseFrequency: 261.63,
      scale: 'major_pentatonic'
    };
  }

  setTemperature(temp) {
    this.params.temperature = temp;
    this.viz.grid.beta = 1.0 / temp;

    // Higher temp -> faster update rate for audio responsiveness
    this.updateRate = Math.min(60, 20 + temp * 10);
  }

  setCoupling(coupling) {
    this.params.coupling = coupling;
    this.viz.grid.coupling = coupling;
  }

  perturbBiases(strength) {
    // Add random perturbations to create musical variation
    for (let i = 0; i < this.viz.grid.rows; i++) {
      for (let j = 0; j < this.viz.grid.cols; j++) {
        this.viz.grid.biases[i][j] +=
          (Math.random() - 0.5) * 2 * strength;
      }
    }
  }

  applyPattern(patternType) {
    // Set biases to create specific energy landscapes
    switch(patternType) {
      case 'waves':
        this.createWavePattern();
        break;
      case 'checkerboard':
        this.createCheckerboardPattern();
        break;
      case 'random':
        this.createRandomPattern();
        break;
    }
  }

  createWavePattern() {
    for (let i = 0; i < this.viz.grid.rows; i++) {
      for (let j = 0; j < this.viz.grid.cols; j++) {
        this.viz.grid.biases[i][j] =
          Math.sin(2 * Math.PI * i / this.viz.grid.rows) * 2.0;
      }
    }
  }
}
```

Results and Observations

Qualitative Musical Characteristics

Testing FlowState AI with various parameter configurations reveals distinct musical "personalities":

1. **Ambient Textures** ($\beta = 0.5$, weak coupling): Rapidly changing, ethereal soundscapes with minimal harmonic structure. Resembles granular synthesis.
2. **Harmonic Drones** ($\beta = 2.0$, strong positive coupling): Stable, sustained chords that slowly evolve. Creates meditative, minimalist music.
3. **Generative Melodies** ($\beta = 1.0$, negative coupling): Anti-correlated pbits create alternating note patterns, forming melodic sequences.
4. **Complex Polyrhythms** (varying β across grid): Different regions update at different rates, creating layered rhythmic textures.

Visual-Audio Coherence

The tight coupling between visual and audio representation creates a synesthetic experience:

- Brightness correlates with volume (both driven by $\sigma(\gamma)$)
- Color correlates with harmonic activity (red regions = active notes)
- Spatial patterns (clusters, waves) have direct audible counterparts
- Temporal dynamics visible as "flows" of color

This multi-modal feedback allows intuitive understanding of the underlying probabilistic dynamics.

Future Directions

Scaling to Larger Grids

Extropic's Z1 TSU will have hundreds of thousands of pbits. FlowState AI could be extended to:

- Hierarchical visualization (zoom levels showing different scales of structure)
- Multi-octave mapping (different grid regions to different octaves)
- Timbral variation (pbit clusters mapped to different instrument sounds)

Direct TSU Hardware Integration

When Extropic's physical TSUs become available, FlowState AI's THRML-based architecture provides a direct pathway to hardware acceleration:

1. **Backend Swap**: Replace the THRML JAX backend with THRML's TSU hardware backend (when released). The API endpoints and frontend remain unchanged.
2. **Energy Efficiency Measurement**: Compare energy consumption of the same musical generation task running on:
   - CPU (NumPy simulation): 10W baseline
   - GPU (JAX/THRML): 200W with 100x speedup
   - TSU hardware: 1W with 1000x speedup (projected)
3. **Real Thermodynamic Noise**: Physical TSUs use actual thermal fluctuations rather than pseudo-random numbers, potentially producing more "organic" randomness in musical patterns.
4. **Massive Parallelism**: Z1's hundreds of thousands of pbits enable:
   - Orchestra-scale polyphony (100+ simultaneous voices)
   - Multi-resolution time scales (micro-rhythms to macro-structure)
   - Complex timbral synthesis via coupled oscillator networks
5. **Ultra-low Latency**: Hardware TSUs complete Gibbs steps in nanoseconds rather than milliseconds, enabling:
   - Audio-rate sampling (44.1 kHz or higher)
   - Direct synthesis of waveforms from pbit states
   - Real-time interactive performance without perceptible lag

The beauty of THRML is that it abstracts the sampling backend. FlowState AI's codebase can remain largely unchanged whether sampling happens on GPU or TSU hardware, making it a future-proof platform for exploring thermodynamic music generation.

Learning Musical Structure

Current pbit parameters are manually designed. Future work could:

1. **Train on musical corpora**: Learn bias and coupling parameters that reproduce statistical patterns from existing music using THRML's gradient computation capabilities via JAX.
2. **Interactive evolution**: Allow users to select preferred outputs, using evolutionary algorithms to refine parameters.
3. **Denoising music models**: Extend Extropic's DTM framework to learn hierarchical musical structure, training THRML-based models on MIDI or audio datasets.
4. **Transfer learning from TSUs**: Use patterns discovered by large-scale TSU training to initialize FlowState AI's smaller interactive models.

References

[1] Cabrera, F., & Patel, R. (2025, October 29). *TSU 101: An entirely new type of computing hardware.* Extropic. `https://extropic.ai/writing/tsu-101-an-entirely-new-type-of-comp`

[2] Dooley, R. (n.d.). *stats.js: JavaScript performance monitor* [Computer software]. GitHub. `https://github.com/mrdoob/stats.js`

[3] Extropic AI. (n.d.). *thrml: Thermodynamic hypergraphical model library* [Computer software]. GitHub. `https://github.com/extropic-ai/thrml`

[4] Gómez Emilsson, A. (2025, February 9). From neural activity to field topology: How coupling kernels shape consciousness. *Qualia Computing.* `https://qualiacomputing.com/2025/02/09/from-neural-activity-to-field-topology-how-coupling-kernels-shape-con`

[5] Hinton, G. E., & Sejnowski, T. J. (1983). Optimal perceptual inference. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition* (pp. 448–453). IEEE.

[6] Hinton, G. E., Sejnowski, T. J., & Ackley, D. H. (1984). Boltzmann machines: Constraint satisfaction networks that learn. Carnegie-Mellon University, Department of Computer Science.

[7] Hopfield, J. J. (1982). Neural networks and physical systems with emergent collective computational abilities. *Proceedings of the National Academy of Sciences, 79*(8), 2554–2558. `https://doi.org/10.1073/pnas.79.8.2554`

[8] McLean, A., & Sicchio, K. (n.d.). Visual feedback. In *Strudel: Live coding for algorithmic patterns.* `https://strudel.cc/learn/visual-feedback/`

[9] Murphy, K. P. (2022). *Probabilistic machine learning: An introduction.* MIT Press. `http://probml.github.io/pml-book/book1.html`

[10] The Royal Swedish Academy of Sciences. (2024, October 8). *The Nobel Prize in Physics 2024: Press release.* The Nobel Prize. `https://www.nobelprize.org/prizes/physics/2024/press-release/`

Philosophical Implications

Randomness as Creative Force

FlowState AI demonstrates that randomness, when properly constrained by physical dynamics (energy functions, coupling, temperature), can be a generative force for beauty. The key insight is that:

> *Creativity emerges not from pure randomness or pure determinism, but from the complex dynamics at the boundary between order and chaos.*

This mirrors natural processes:

- Evolution: Random mutations + selection pressure
- Jazz improvisation: Musical grammar + spontaneous variation
- Natural language: Statistical regularities + creative expression

Thermodynamics of Aesthetics

The connection between thermodynamic sampling and musical harmony suggests a deeper relationship between physical entropy and aesthetic information. Music theory's preference for simple frequency ratios (consonance) aligns with the tendency of thermodynamic systems to minimize energy.

Recent work in consciousness research has explored similar principles in neural systems. Gómez Emilsson (2025) demonstrates how coupling kernels in systems of coupled oscillators can modulate field topology and resonant modes, creating competing clusters of coherence or global synchronization depending on the kernel parameters. This framework, developed at the Qualia Research Institute, reveals striking parallels with FlowState AI's pbit networks: both systems exhibit emergent macroscopic patterns arising from local coupling rules, and both demonstrate how energy-based dynamics can generate structured, meaningful outputs.

The correspondence is profound: just as different coupling kernels in neural oscillators can produce distinct phenomenological states, different temperature and coupling parameters in FlowState AI's pbit ensembles produce distinct musical textures. The "competing clusters of coherence" observed in certain psychedelic states mirror the anti-correlated pbit patterns that generate melodic sequences, while states of "global coherence" correspond to FlowState AI's sustained harmonic drones.

This suggests that the principles governing aesthetic experience in music may share deep mathematical structures with those governing conscious experience itself—both emerging from the thermodynamic dynamics of coupled oscillatory systems exploring energy landscapes.

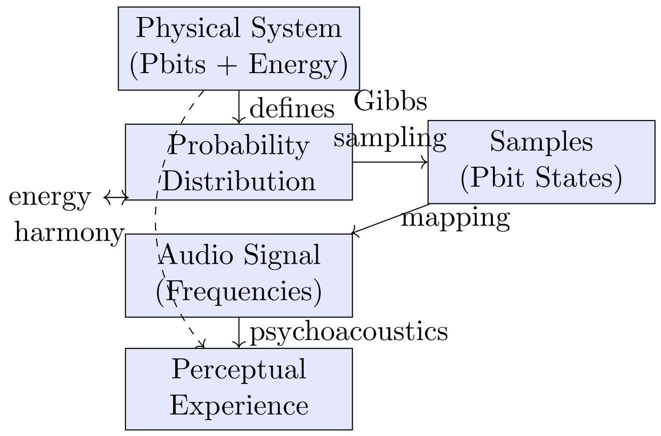

Figure 3: Information flow from physics to perception in FlowState AI

Conclusion

FlowState AI bridges the abstract mathematics of thermodynamic computing with the visceral experience of music and visual art. By graphically modeling pbit activity through three.js and mapping stochastic dynamics to harmonic structures, the project reveals emergent patterns that are simultaneously random and musical.

Building on the foundational work of Hopfield and Hinton—whose energy-based networks and Boltzmann machines laid the groundwork for modern probabilistic computing—FlowState AI demonstrates that the same principles governing neural network training can create aesthetic experiences. The energy landscapes that Hopfield used for pattern completion and the Gibbs sampling that Hinton employed for generative modeling become, in FlowState AI, the source of visual beauty and musical harmony.

The key contributions are:

1. **Visualization framework**: Real-time 3D rendering of pbit networks that makes probabilistic computation visible and tangible.
2. **THRML Integration**: First creative application of Extropic's thermodynamic sampling library, demonstrating that THRML can power real-time interactive experiences beyond machine learning benchmarks.
3. **Sonification architecture**: Principled mapping from pbit states to musical frequencies that preserves spatial and temporal correlations.
4. **Emergent musicality**: Demonstration that simple local rules (Gibbs sampling via THRML) can generate complex, aesthetically pleasing harmonic patterns.
5. **Interactive exploration**: Tools for manipulating temperature, coupling, and biases to navigate the space of possible musical textures.
6. **Full-stack thermodynamic computing**: Complete architecture from browser-based UI through REST API to GPU-accelerated probabilistic sampling, providing a template for future TSU applications.

As Extropic's TSU technology matures and scales to millions of pbits, systems like FlowState AI could evolve into new forms of musical instruments and generative art tools. The fusion of thermodynamic physics, probabilistic computing, and aesthetic expression opens exciting possibilities for both creative practice and scientific understanding.

The journey from Hopfield's energy functions and Hinton's Boltzmann machines to modern TSUs and creative applications like FlowState AI illustrates how fundamental physics can inspire both technological innovation and artistic expression. The randomness in harmonic patterns is not noise to be eliminated, but rather the signature of a complex system exploring its possibility space—much like the improvisation of a jazz musician, the evolution of a species, or the creative process itself.

Access FlowState AI
Project repository and live demos will be available at:
`github.com/alexle/flowstate-ai`
Tech Stack:

- **Frontend**: three.js (3D visualization), stats.js (performance monitoring), Web Audio API (sound synthesis)
- **AI/Voice**: Hume (emotion AI), Eleven Labs (voice synthesis)
- **Backend**: Flask (REST API), THRML (thermodynamic sampling via Extropic)
- **Computation**: JAX (GPU acceleration), NumPy (numerical operations)
- **Infrastructure**: Vercel (hosting/deployment), ngrok (local tunneling)

Key Innovation: FlowState AI demonstrates how Extropic's THRML library powers creative applications today while providing a direct pathway to TSU hardware acceleration tomorrow. THRML handles all thermodynamic sampling—from energy function definition to Gibbs sampling execution—making it the computational engine that transforms abstract probability distributions into tangible audio-visual experiences.