# Tensor-Accelerated Eager Multi-Resolution Grids for Evolving Large-Scale Substrates

Anonymous Author(s)

## Abstract

ES-HyperNEAT (ES-HN) discovers neural network topology by querying a CPPN at candidate positions and placing nodes where the CPPN output shows high variation. However, ES-HN builds a different quadtree for each network in the population, forcing sequential execution. Each depth has 4× more positions, so computation grows exponentially, and deep substrates become infeasible.

We present Eager Multi-Resolution HyperNEAT (EMR, EMR-HyperNEAT), which matches or exceeds ES-HN results while scaling across parallel cores. Instead of building quadtrees adaptively, we precompute position grids for each depth level, evaluate all positions simultaneously, then filter using the same variance threshold. All networks share these grids, enabling evaluation of the entire population in parallel.

**Main result**: EMR enables scalability to arbitrarily large substrates through parallelization. While ES-HN runs in $O(4^D)$ time sequentially, EMR achieves $O(4^D/P)$ by dividing work across $P$ cores, yielding linear speedup. Comparing ES-HN on CPU against EMR on GPU: on XOR (pop=1000), EMR overtakes at depth 4, 34× per-generation speedup at depth 7 (~100× over 30 gens).

**Additional capabilities**: EMR's scalability enables: (1) recurrent networks through hidden-to-hidden connections; (2) evolution at extreme resolutions via memory streaming (validated on depth 13: 358M positions).

To our knowledge, this is the first massively parallel implementation of adaptive substrate discovery.

## CCS Concepts

• **Computing methodologies** → **Parallel algorithms**; **Neural networks**; **Genetic algorithms**; **Vector / streaming algorithms**; *Bio-inspired approaches*; *Generative and developmental approaches*; • **Computer systems organization** → *Single instruction, multiple data.*

## Keywords

neuroevolution, HyperNEAT, adaptive substrates, GPU parallelization, indirect encoding, recurrent networks

**ACM Reference Format:**

Anonymous Author(s). 2026. Tensor-Accelerated Eager Multi-Resolution Grids for Evolving Large-Scale Substrates. In *Proceedings of Genetic and Evolutionary Computation Conference (GECCO '26).* ACM, New York, NY, USA, 9 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

## 1 Introduction

Neuroevolution, the application of evolutionary algorithms to neural network design, has discovered novel architectures that challenge manually designed network topologies [18, 19]. In *indirect encoding*, a compact genome generates connection patterns rather than specifying each connection explicitly. A *substrate* is the spatial arrangement of nodes that form the neural network; *substrate resolution* refers to how finely this space is divided into potential node positions. Higher resolution enables more complex networks but exponentially increases computational cost, creating a fundamental barrier to practical applications. Modern hardware accelerators (GPUs, TPUs) provide massive parallelism for regular, vectorizable operations, yet many substrate adaptive neuroevolution algorithms contain inherently sequential components that prevent effective utilization of this parallelism.

Evolvable-Substrate HyperNEAT (ES-HyperNEAT) [15] automatically discovers where to place hidden nodes by examining Compositional Pattern-Producing Network (CPPN) [16] output patterns. A CPPN is a neural network that takes spatial coordinates as input and outputs connection weights, enabling compact genomes to generate large-scale connectivity patterns. The algorithm divides space using a *quadtree*: starting from a single region covering the substrate, it recursively splits regions into four quadrants where the CPPN outputs vary significantly. High variance indicates pattern complexity that requires finer resolution; low-variance regions are ignored. This adaptive approach matches network complexity to problem requirements without manual specification.

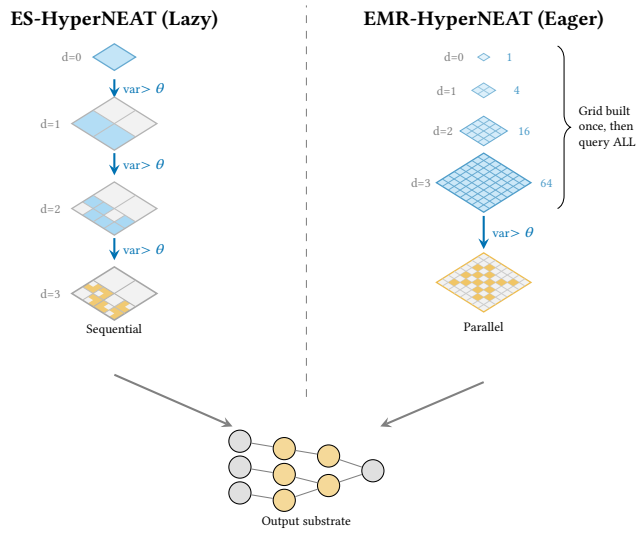


**Figure 1: Lazy vs eager substrate discovery. Left: ES-HyperNEAT subdivides sequentially; each depth must complete before children are evaluated. Right: EMR-HyperNEAT evaluates all depths in parallel, then filters by variance. Both discover sparse substrates from CPPN variance.**

*GECCO '26, Malaga, Spain*
2026. ACM ISBN 978-x-xxxx-xxxx-x/YYYY/MM
https://doi.org/10.1145/nnnnnnn.nnnnnnn

### 1.1 The GPU Parallelization Bottleneck

The quadtree traversal in ES-HyperNEAT creates sequential dependencies that prevent GPU parallelization. ES-HyperNEAT operates in two phases: Phase 1 discovers $H$ hidden nodes by subdividing from each input; Phase 2 discovers feedforward connections between hidden nodes by performing a quadtree traversal from each of the $H$ positions. Standard ES-HyperNEAT constrains Phase 2 to feedforward connections ($y_1 < y_2$), but recurrent substrates (requiring backward, lateral, or self connections) amplify the cost. Phase 2 requires $H$ sequential quadtree traversals, each exploring up to $O(4^D)$ positions to discover connections. At depth 5 with 5,460 positions, Phase 2 dominates runtime.

### 1.2 Lazy to Eager Reformulation

ES-HyperNEAT's quadtree uses *lazy evaluation*: it only examines a region if its parent showed high variance. This creates sequential dependencies that prevent parallelization.

Our key insight: we can *eagerly evaluate* all positions at all resolutions, then filter using the same variance criterion. This performs more CPPN queries than strictly necessary, but all queries become independent and parallelizable. We trade redundant computation for GPU utilization.

This reformulation works because CPPN outputs are deterministic functions of coordinates, enabling independent evaluation of all positions. Figure 1 illustrates this transformation.

### 1.3 Contributions

We present EMR-HyperNEAT, enabling scalable adaptive substrate discovery (Section 3). Key contributions:

(1) **GPU Parallelization**: Reformulate ES-HyperNEAT's sequential quadtree subdivision into independent grid operations, reducing complexity from $O(4^D)$ to $O(4^D/P)$ by dividing work across $P$ cores (Section 3).
(2) **Recurrent Networks**: Efficiently compute recurrent and hidden-layer connections within substrates (Section 4).
(3) **Large-Scale Substrates**: Scale to arbitrarily large substrates on consumer hardware through efficient memory management (Section 5).

## 2 Background and Related Work

We provide context on related literature focusing on NEAT, HyperNEAT, ES-HyperNEAT, GPU parallelization primitives, and the limitations that prevent existing ES-HyperNEAT implementations from achieving GPU acceleration.

### 2.1 NEAT and HyperNEAT

NeuroEvolution of Augmenting Topologies (NEAT) [18] evolves neural network topology and weights simultaneously through innovation numbers that track genetic history, enabling meaningful crossover between networks of different structures. While powerful, NEAT directly encodes each connection, limiting scalability to large networks.

HyperNEAT [17] extends NEAT with *indirect encoding* via Compositional Pattern-Producing Networks (CPPNs) [16]. HyperNEAT operates on a *substrate*: a fixed grid of node positions specified before evolution begins. With direct encoding, genome size grows linearly with connection count, making large networks impractical. Indirect encoding solves this: HyperNEAT evolves a compact CPPN that generates connection weights from spatial coordinates. Given source $(x_1, y_1)$ and target $(x_2, y_2)$, the CPPN outputs weight $w = \text{CPPN}(x_1, y_1, x_2, y_2)$; connections with $|w| < \theta_w$ are pruned. The same CPPN can specify millions of connections. CPPNs use activation functions such as sine (for repetition) and Gaussian (for symmetry) to generate spatially-regular patterns.

### 2.2 ES-HyperNEAT

Evolvable-Substrate HyperNEAT [15] addresses HyperNEAT's requirement for predefined substrate topology through adaptive resolution via quadtree subdivision [9]. The key phases creating the GPU bottleneck are:

**Phase 1 (Input→Hidden)**: For each of $I$ input coordinates, perform quadtree subdivision to discover hidden positions where CPPN variance exceeds $\theta_{div}$, yielding $H$ hidden positions:

```
function SUBDIVIDE(node, source, depth)
    var ← Var(CPPN(source, c.center) for c ∈ node.children)
    if var > θ_div and depth < D_max then
        for c ∈ node.children do
            SUBDIVIDE(c, source, depth+1)        ▷ Sequential dependency
        end for
    end if
end function
```

**Phase 2 (Hidden→Hidden)**: For each of $H$ discovered positions, repeat subdivision to find connections to other hidden positions. This requires $H$ sequential quadtree traversals, which can generate $O(H^2)$ potential connections. Output connections are discovered similarly but don't dominate complexity.

**Cold-start behavior**: With initial depth 0, randomly initialized CPPNs produce uniformly low outputs that fail to trigger subdivision, creating no hidden nodes. Prior work avoids this by using initial depth $> 0$ to seed the substrate.

### 2.3 GPU Primitives and Frameworks

We use JAX [3], which provides key primitives and transparently supports CPU, GPU, and TPU backends:

**Vectorization (vmap)**: Transforms $f : \mathbb{R}^n \to \mathbb{R}^m$ into batched $\text{vmap}(f) : \mathbb{R}^{B\times n} \to \mathbb{R}^{B\times m}$, applying $f$ independently to $B$ inputs in parallel.

**Device Parallelism (pmap)**: Distributes computation across multiple GPUs, with each device processing a data shard.

**JIT Compilation**: Compiles Python functions to optimized device code. *Critical constraint*: JIT requires static array shapes known at compile time; dynamic shapes trigger recompilation.

Several JAX-based neuroevolution frameworks use these primitives. EvoJAX [20], evosax [12], and QDax [7] evolve fixed-topology networks only. TensorNEAT [21] is the sole framework supporting topology evolution, with 500× speedup over NEAT-Python [13] by transforming variable-topology networks into uniformly shaped tensors. At the time of writing, TensorNEAT lists ES-HyperNEAT as "future work".

## 2.4 ES-HyperNEAT GPU Limitations

The quadtree traversal in Phase 1 and Phase 2 (Section 2) conflicts with GPU parallelization for three reasons:

(1) **Non-uniform structure**: Different CPPNs produce different subdivision patterns, preventing batching via vmap.
(2) **Sequential dependency**: Children cannot be evaluated until parent variance is computed.
(3) **Dynamic shapes**: Variable leaf counts are incompatible with JAX's static shape requirements for JIT compilation.

Prior work on GPU tree parallelization [5, 10] achieved efficiency for static trees, but ES-HyperNEAT's variance-dependent subdivision differs for each genome, preventing such optimization. Iterated ES-HyperNEAT [14] retains this bottleneck. Prior work demonstrated computational limitations at depths exceeding 5 [1], with subsequent analysis [2] confirming exponential JIT compilation growth with depth. We address this gap via lazy-to-eager reformulation (Section 3).

# 3 EMR-HyperNEAT Algorithm

EMR-HyperNEAT reformulates ES-HyperNEAT's inherently sequential quadtree traversal into a fully parallel pipeline through three key innovations:

**ES-HyperNEAT (Lazy):** `subdivide_if(var > θ)`
**EMR-HyperNEAT (Eager):** `eval_all(); filter(var > θ)`

This eager reformulation (1) evaluates all positions unconditionally then applies variance-based filtering, trading redundant computation for full GPU utilization. It combines with (2) precomputing static multiresolution grids enabling JIT compilation, and (3) decomposing computation into three vectorized stages amenable to vmap. The following subsections detail the hierarchical structure and each processing stage.

## 3.1 Hierarchical Grid Structure

At initialization, EMR-HyperNEAT precomputes a complete multiresolution grid containing all positions from depth 0 to maximum depth $D$:

Definition 1 (Hierarchical Grid). *For maximum depth $D$, the hierarchical grid $\mathcal{G}_D$ contains all positions:*

$$\mathcal{G}_D = \bigcup_{d=0}^{D} Grid_d,\ Grid_d = \left\{ \left( \tfrac{2i+1}{2^{d+1}} - 1, \tfrac{2j+1}{2^{d+1}} - 1 \right) : i, j \in [0, 2^{d+1}) \right\} \quad (1)$$

The total position count follows the geometric series (Table 1):

$$N_{total} = \sum_{d=0}^{D} 4^{d+1} = \frac{4^{D+2} - 4}{3} \quad (2)$$

The grid stores: (1) position coordinates in $[-1, 1]^2$, (2) depth indices, (3) parent-child relationships as index arrays, and (4) level offsets. All arrays have static shapes, enabling JIT compilation. Compilation time scales with both substrate depth ($O(4^D)$ positions for variance and mask operations) and population size (batched CPPN queries produce tensors of shape pop $\times |S| \times |T|$).

**Table 1: Hierarchical grid position counts. Total includes all levels (not just finest) because variance filtering requires parent-child relationships. GPU memory scales with population size (Section 7).**

| Depth $D$ | Positions per Level ($4^{d+1}$) | Total Positions |
|---|---|---|
| 4 | 4, 16, 64, 256, 1,024 | 1,364 |
| 5 | 4, ..., 4,096 | 5,460 |
| 6 | 4, ..., 16,384 | 21,844 |
| 7 | 4, ..., 65,536 | 87,380 |

## 3.2 Batch CPPN Query

With positions in static arrays, CPPN queries become embarrassingly parallel [11]. For source $s$ and targets $T = \{t_1, ..., t_N\}$, vectorization $\mathbf{W} = \text{vmap}(\text{CPPN})(s, T)$ replaces $N$ sequential queries with wall-clock time $O(N/P)$ for $P$ parallel cores. Population parallelization via $\mathbf{W}_{pop} = \text{vmap}(\text{vmap}(\text{CPPN}))(S_{pop}, T)$ yields tensors of shape (pop_size, $|S|$, $|T|$).

## 3.3 Hierarchical Variance Computation

Variance computation proceeds bottom-up: for each position at depth $d < D$, we compute sample variance from its four children's weights at depth $d+1$. Parent-child relationships are *static* and determinable from grid structure alone. We precompute index mappings $\text{children}[d][i] = \{j : \text{pos}_j \text{ is child of pos}_i\}$. Variance computation becomes a vectorized gather reduce:

**for** $d = 0$ **to** $D - 1$ **do** ▷ Bottom-up, vectorized
  $W_{children} \leftarrow \text{gather}(W, \text{children}[d])$ ▷ Shape: (positions, 4)
  $\text{var}[d] \leftarrow \text{Var}(W_{children}, \text{axis} = 1)$ ▷ Parallel
**end for**

## 3.4 Active Mask Generation

The subdivision decision becomes threshold comparison with top-down propagation:

$$\text{active}[d][i] = \begin{cases} \text{True} & d = 0 \\ \text{active}[d-1][p_i] \wedge (\text{var}[d-1][p_i] > \theta_{div}) & \text{else} \end{cases} \quad (3)$$

where $p_i = \text{parent}(i)$. Figure 2 illustrates this process. The propagation is fully vectorizable:

active[0] ← True ▷ Root always active
**for** $d = 1$ **to** $D$ **do**
  parent_active ← gather(active[$d-1$], parents[$d$])
  parent_var ← gather(var[$d-1$], parents[$d$])
  active[$d$] ← parent_active ∧ (parent_var > $\theta_{div}$)
**end for**

Each parent controls all four children as a block, enabling parallel filtering instead of sequential subdivision.

## 3.5 Complexity Analysis

Theorem 1 (Sequential Complexity). *ES-HyperNEAT's Phase 1 requires $O(4^D)$ sequential CPPN queries in the worst case, and Phase 2 requires $O(H^2)$ sequential queries where $H \leq 4^D$ is the number of discovered hidden positions.*

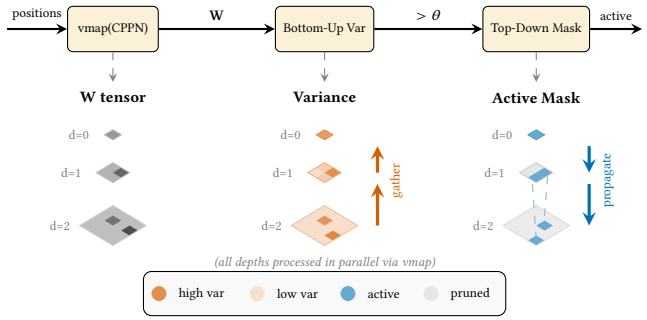


**Figure 2: Hierarchical variance thresholding. Left: CPPN weight tensor W at depths 0–2. Middle: variance computed bottom-up (vermillion arrows); color intensity indicates magnitude. Right: active mask propagated top-down (blue arrows); positions meeting the threshold condition (Eq. 3) are active (blue), others pruned (gray). Dashed lines show parent-child inheritance.**

**Table 2: Computational complexity comparison. $D$ = maximum depth, $H$ = discovered hidden positions, $P$ = parallel execution units. ES-HyperNEAT operations are shown separately to highlight what EMR-HyperNEAT parallelizes; in ES-HyperNEAT, these are interleaved during quadtree traversal rather than distinct passes.**

| Operation | ES-HyperNEAT | EMR-HyperNEAT |
|---|---|---|
| Grid Construction | — | $O(4^D)$ once |
| CPPN Query (Phase 1) | $O(4^D)$ seq. | $O(4^D/P)$ |
| Variance Comp. | $O(4^D)$ seq. | $O(4^D/P)$ |
| Mask Propagation | $O(4^D)$ seq. | $O(4^D/P)$ |
| H→H Query (Phase 2) | $O(H^2)$ seq. | $O(H^2/P)$ |
| **Total** | $O(4^D + H^2)$ | $O((4^D + H^2)/P)$ |

Proof. Phase 1: In the worst case (all positions active), quadtree explores all $\sum_{d=0}^{D} 4^d = O(4^D)$ nodes, each requiring sequential evaluation due to parent dependency. Phase 2: For $H$ hidden positions, all pairs connectivity requires $H^2$ queries, each sequential due to the same dependency structure. □

*Note*: The $O(H^2)$ bound counts logical source-target pair queries. The implementation performs $H$ quadtree traversals, each exploring up to $O(4^D)$ positions; since $H \leq 4^D$, both formulations yield equivalent worst-case bounds.

Theorem 2 (Parallel Complexity). *EMR-HyperNEAT achieves $O(4^D/P)$ elapsed time for Phase 1 and $O(H^2/P)$ for Phase 2, where $P$ is the number of parallel execution units.*

Proof. Phase 1: Pre-computed grid has $O(4^D)$ positions. Batch CPPN query via vmap evaluates all positions simultaneously across $P$ cores, achieving $O(4^D/P)$ elapsed time. Variance computation and mask propagation are similarly parallelizable via vectorized operations. Phase 2: Constructing an $H \times H$ query batch and applying vmap yields $O(H^2/P)$ elapsed time. □

**Table 3: Six substrate configurations from four connection primitives (Definition 4.1). F=Forward, B=Backward, L=Lateral, S=Self.**

| Configuration | F | B | L | S | Notes |
|---|---|---|---|---|---|
| Feedforward | ✓ | | | | Fastest; skips Phase 2 |
| Hidden-to-Hidden | ✓ | | | | ES-HyperNEAT default |
| Backward | ✓ | ✓ | | | Feedback loops |
| Lateral | ✓ | | ✓ | | Same-layer links |
| Self | ✓ | | | ✓ | State across iterations |
| Full Recurrent | ✓ | ✓ | ✓ | ✓ | Fixed iteration count |

Theorem 3 (Correctness Equivalence). *Under identical variance thresholds $\theta_{div}$ and band thresholds $\theta_{band}$, when both algorithms successfully discover hidden nodes, EMR-HyperNEAT produces identical connection sets to ES-HyperNEAT.*

Proof. Both algorithms' connection sets are determined by: (1) which positions are active (pass variance threshold from root), and (2) which connections pass the band threshold. EMR-HyperNEAT computes identical variance values (same CPPN, same children per node). The active mask propagation implements identical logic to quadtree subdivision. Band detection applies identical thresholds. Since all intermediate computations match, final connection sets are identical. □

**Note**: This equivalence assumes identical starting conditions. With `initial_depth=0`, ES-HyperNEAT's traversal from root to leaves may fail to discover hidden nodes if root variance is insufficient, while EMR-HyperNEAT's computation from leaves to root evaluates all positions before masking. See Section 6 for practical implications.

## 4 Connection Type Taxonomy

EMR-HyperNEAT generates six substrate types through configurable Phase 2 hidden to hidden connectivity.

In HyperNEAT substrates, nodes occupy a 2D coordinate space where the Y coordinate encodes processing depth: input nodes at $y = -1.0$, output nodes at $y = 1.0$, hidden nodes at intermediate Y values. Comparing source and target Y coordinates classifies connections into four primitive types (Definition 4.1); enabling or disabling these primitives produces six substrate configurations with varying recurrence properties.

Definition 2 (Connection Types). *For source position $s = (x_s, y_s)$ and target position $t = (x_t, y_t)$:*

- ***Forward***: $y_s < y_t$ *(information flows "upward")*
- ***Backward***: $y_s > y_t$ *(feedback connections)*
- ***Lateral***: $y_s = y_t \wedge s \neq t$ *(same layer, different position)*
- ***Self-loop***: $s = t$ *(self connections)*

EMR-HyperNEAT's six connection types are configurations that enable or disable these four primitives (Table 3).

The `iteration_level` parameter controls multihop H→H expansion:

- **Level 0**: Feedforward only (no H→H connections)
- **Level 1**: Direct H→H only (no multihop expansion)

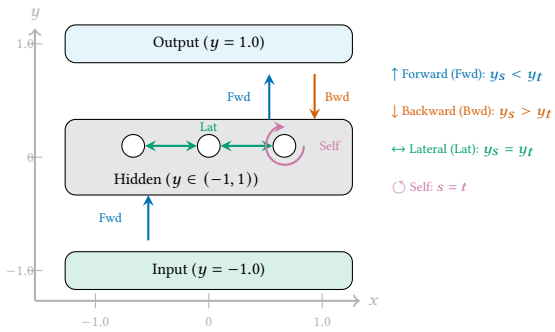


**Figure 3: Connection type classification by Y-coordinate. Substrate layers are organized vertically with connections classified by source/target coordinates (Definition 4.1). The six substrate configurations enable different subsets of these four primitives.**

- **Level** $k$: $k$-hop expansion via $A+\gamma A^2+\gamma^2 A^3+\cdots+\gamma^{k-1}A^k$ [4] where $\gamma = 0.8$ (hop decay factor prevents weight explosion)

**Important distinction**: In EMR-HyperNEAT, `iteration_level` controls signal *propagation* through discovered connections. In ES-HyperNEAT, the same parameter controls position *discovery* iterations. This semantic difference affects experimental comparisons (see Section 6).

## 5 Implementation

EMR-HyperNEAT is implemented in JAX/Python, building on TensorNEAT [21] for NEAT/HyperNEAT primitives. State is maintained as JAX arrays for GPU-resident execution.

**Caching optimizations** accelerate computation at two levels. At the *JAX level*, population batching via `vmap` evaluates all genomes in a single call; JIT compilation caches XLA graphs; persistent compilation caches provide speedup on subsequent runs. At the *algorithm level*, hierarchical grid structures are precomputed once per depth (~683 KB at depth 7), and H→H connections discovered in early generations are cached and reused, reducing subsequent discovery from ~27s to ~3ms. These optimizations achieve **4.6–7.2**× speedup at depths 5–6 (Table 6).

### 5.1 Execution Strategies

EMR-HyperNEAT supports five execution strategies optimized for different hardware and memory constraints.

**Single Device** (Default): Complete evolution loop runs on one device, eliminating transfers between generations. Fastest when memory permits. Runs on any JAX backend (CPU, GPU, TPU).

**Memory Streaming**: Enables evolution on *arbitrarily large substrates* through GPU→CPU memory tiering. Weight accumulators are allocated on CPU RAM, while CPPN queries execute on GPU one chunk at a time. After each chunk, weights are immediately offloaded to CPU, keeping GPU peak memory at $O(\max_d |Grid_d|)$ rather than $O(\sum_d |Grid_d|)$. Validated at depth 13 (358M positions). Runs on any JAX backend.

**Multi-GPU Strategies**: Three strategies distribute work across multiple devices, validated with 2× NVIDIA RTX 2080 Ti 11GB (JAX CUDA backend). Multi-GPU provides memory scalability rather than speed due to Python coordination overhead for NEAT fitness aggregation.

**Table 4: Execution strategy comparison (relative to Single Device). Multi-GPU results for 2 GPUs on XOR (see Section 6).**

| Strategy | Speed | Memory | Use Case |
|---|---|---|---|
| Single Device | 1.0× | GPU | Default |
| Memory Streaming | 0.2–1.0× | CPU | OOM prevention |
| Data Parallel | 0.6–0.7× | Dist. | Large data |
| Evaluation Parallel | 0.6–0.7× | Dist. | Recurrent |
| Population Parallel | 0.6–0.7× | Dist. | Large pop |

**Algorithm 1** Memory Streaming: GPU→CPU Memory Tiering

**Require:** Grid $\mathcal{G}_D$, CPPN params, threshold $\theta$, chunk size $S$
**Ensure:** $W_1^{cpu}, W_2^{cpu}$, active mask $M$

```
W1_cpu, W2_cpu ← np.zeros(...)                ▷ CPU numpy arrays
reached[0] ← 1
for d = 0 to D do
    for positions in chunks of S do
        w_gpu ← vmap(CPPN)(chunk)             ▷ GPU query
        W_cpu[...] ← np.asarray(w_gpu)        ▷ Offload to CPU
        del w_gpu                             ▷ Free GPU
    end for
    if d > 0 then
        reached[d] ← (reached[d−1] ∧ var[d−1] > θ)[parents]
    end if
end for
return W1_cpu, W2_cpu, concat(reached)
```

**Data Parallel**: Distributes dataset across GPUs while replicating population. Each GPU evaluates all genomes on its data subset. Use for large feedforward configurations.

**Evaluation Parallel**: Parallelizes only evaluation while keeping h→h caching on GPU 0. Use for recurrent/h→h types.

**Population Parallel**: Splits population across GPUs, with each GPU running the full pipeline on all samples. Use for large populations with expensive h→h.

### 5.2 Forward Pass Mechanisms

The forward pass differs by type:

**Feedforward and Hidden-to-Hidden**: Dense matrix operations compute $h = \sigma(W_1 x)$, $y = \sigma(W_2 h)$. For Hidden-to-Hidden, sparse connections iterate via residual: $h^{(0)} = \sigma(W_1 x)$, $h^{(t+1)} = \sigma(h^{(t)} + W_{hh} h^{(t)})$.

**Full Recurrent types**: Use scatter-add for `activate_time` iterations (default: $2 \times (2^{\text{depth}} + 1)$) to propagate signals through the full depth.

## 6 Experimental Evaluation

We evaluate EMR-HyperNEAT against ES-HyperNEAT to validate three claims: (1) scalability to depths previously infeasible, (2) speedup at high substrate resolutions, and (3) equivalent solution quality. Both algorithms are tested at initial_depth=0, forcing evolution to discover substrate structure from scratch rather than starting with seeded resolution. This is a harder configuration that

**Table 5: Experimental tiers for systematic evaluation of connectivity discovery types. Iter = iteration level (signal propagation depth).**

| Tier | Type | Iter | Purpose |
|---|---|---|---|
| 1 | Sparse (variance) | 1 | Minimal H→H |
| 2 | Dense (all) | 1 | Maximum H→H |
| 3 | Sparse + multihop | 2–3 | Multi-layer propagation |

prior work typically avoids. All speedup comparisons use Hidden-to-Hidden (h→h) connection type for fair comparison; other types are analyzed in Section 4.

## 6.1 Experimental Setup

**Hardware**: Throughout this paper, **CPU** refers to Apple M4 Max (JAX CPU backend) and **GPU** refers to NVIDIA RTX 2080 Ti 11GB (JAX CUDA backend). All timing measurements use these specific configurations for consistency.

**Baseline**: PUREPLES [8], the reference Python implementation of ES-HyperNEAT. Throughout this section, we refer to ES-HyperNEAT as the algorithm and PUREPLES as its implementation; tables and comparisons label the baseline as "ES-HyperNEAT" to emphasize the algorithmic comparison.

**Problem**: XOR benchmark (4 input patterns), standard neuroevolution test case [18].

**Common Parameters**: Both algorithms share: 30 generations max, 8 populations {50, 100, 200, 300, 400, 500, 750, 1000}, depths 1–7 (7 values), initial_depth 0, iteration levels 1–3, 3 replications (seeds 42, 123, 456); replication count is bounded by ES-HyperNEAT's computational cost for fair comparison. Variance threshold 0.03, division threshold 0.5, band threshold 0.3, target fitness 0.99 (XOR solved when ≥99% accuracy). **Max weight**: ES-HyperNEAT uses 5.0 [15]; EMR-HyperNEAT uses 8.0 [21].

**EMR-HyperNEAT**: Each experiment ran in process isolation to force JIT recompilation, ensuring timing measurements include compilation overhead. In practice, JAX's JIT cache persists across runs: compilation happens once and subsequent replications skip directly to evolution. We sandboxed experiments to measure cold start performance; production use benefits from cached compilation. Cache enabled, $C_{max} = 10,000$ sparse connections per genome, 3 connection types (h→h, feedforward, full recurrent). Total: 2,016 runs (7 depths × 8 populations × 3 types × 3 replications × 3 tiers).

**ES-HyperNEAT**: Total: 504 runs (7 depths × 8 populations × 3 replications × 3 tiers).

**Full-Depth Evolution**: The key difference lies in traversal direction: ES-HyperNEAT uses quadtree traversal *from root to leaves* (if root variance $< \theta$, no children explored), while EMR-HyperNEAT uses variance computation *from leaves to root* (evaluate all positions first, then apply masks). EMR-HyperNEAT's upward approach handles initial_depth=0 robustly; ES-HyperNEAT's downward approach can fail when early variance is insufficient.

**Experimental Tiers.** We organize experiments into three tiers to systematically evaluate different discovery types as presented in Table 5

**Table 6: Average Time per generation comparison (XOR, pop=1000). EMR uses h→h type (CPU/GPU). Speedup is per-generation (excludes JIT).**

| Depth | ES-HN | EMR | JIT | Speedup CPU/GPU |
|---|---|---|---|---|
| 1 | 0.30s | 1s / 0.9s | 4s / 7s | 0.3× / 0.3× |
| 2 | 0.38s | 2s / 1.8s | 6s / 9s | 0.19× / 0.2× |
| 3 | 0.96s | 3s / 2.3s | 8s / 17s | 0.32× / 0.4× |
| 4 | 4.27s | 5s / 2.9s | 12s / 18s | 0.85× / **1.5**× |
| 5 | 46.3s | 10s / 3.9s | 15s / 19s | **4.6**× / **12**× |
| 6 | 200.4s | 28s / 8.9s | 30s / 24s | **7.2**× / **23**× |
| 7 | 2,020s | 165s / 60s† | 60s/39s | **12.2**× / **34**×† |

Total time = JIT + (generations × time per generation).
†GPU depth 7 uses streaming mode (50–70s/gen on 11GB VRAM).

**Table 7: Per-generation time and solve rate by population and depth (XOR). EMR uses h→h type with cache (CPU/GPU).**

| Depth | Algorithm | Pop 300 | + JIT | Pop 500 | + JIT |
|---|---|---|---|---|---|
| 4 | ES-HN | 1.5s / 67% | - | 2.5s / 33% | - |
| | EMR (CPU) | 2s / 100% | 11s | 3s / 100% | 10s |
| | EMR (GPU) | 0.9s / 100% | 10s | 1.3s / 100% | 12s |
| 5 | ES-HN | 6.8s / 0% | - | 22.2s / 100% | - |
| | EMR (CPU) | 6s / 100% | 15s | 7s / 100% | 14s |
| | EMR (GPU) | 1.2s / 100% | 10s | 1.9s / 100% | 13s |
| 6 | ES-HN | 147.8s / 33% | - | 220.5s / 0% | - |
| | EMR (CPU) | 12s / 100% | 16s | 40s / 100% | 31s |
| | EMR (GPU) | 2.6s / 100% | 12s | 4.3s / 100% | 15s |
| 7 | ES-HN | 652s / 0% | - | 1,649s / 0% | - |
| | EMR (CPU) | 48s / 100% | 46s | 94s / 100% | 72s |
| | EMR (GPU) | 7.9s / 100% | 17s | 13s / 100% | 23s |

## 6.2 Scaling and Speedup Results

Table 6 presents time per generation comparison. Throughout the experimental tables: **ES-HN** = ES-HyperNEAT via PUREPLES; **EMR** = EMR-HyperNEAT. EMR times are steady state (after JIT); "JIT" columns show single compilation overhead. Speedup = ES-HN time / EMR time.

**Scaling behavior**: ES-HyperNEAT exhibits 4–11× slowdown per depth level at depths 4–6 (depth 4→5: 10.8×; depth 5→6: 4.3×). EMR-HyperNEAT scales more favorably: while position count grows as $4^D$ (exponential in depth), each position requires $O(1)$ evaluation, and GPU parallelization amortizes this across $P$ cores. In practice, EMR exhibits 1.3–2.3× slowdown per depth level at depths 4–6. At depth 7, memory streaming (required for 11GB VRAM) increases this to 6.7×, yet EMR maintains 34× speedup over ES-HyperNEAT. At depth 6, ES-HyperNEAT requires 200s/gen, making evolutionary runs slow but not impractical.

## 6.3 Solution Quality

Table 7 shows performance across population sizes and depths:

Both algorithms achieve identical maximum fitness (0.99+) when successful, validating correctness equivalence (Theorem 3). Total compute time for EMR-HyperNEAT is JIT + (generations × time

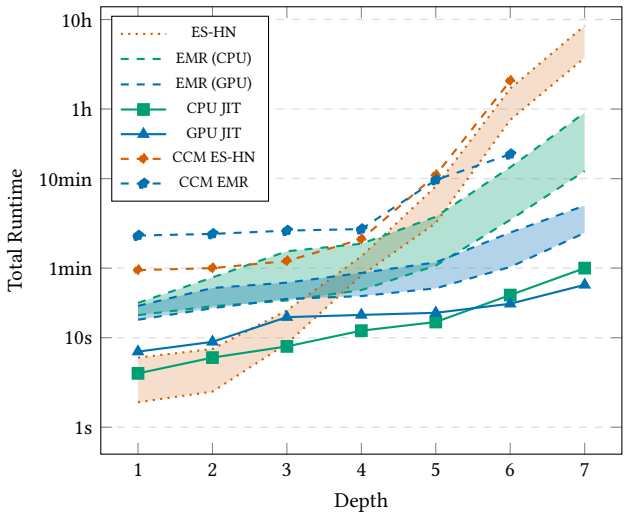


**Figure 4: Total runtime over 30 generations. Bands show IQR (25th–75th percentile) [22], each representing 8 population sizes across 3 replications. Solid lines show one-time JIT compilation overhead. EMR uses h→h connection type. CCM ES-HN and CCM EMR show runtime on CCM financial data [6] (94K samples, pop=100). ES-HN dominates at depths 1–3 where JIT overhead exceeds ES-HN runtime; becomes impractical at depth 6+ (IQR: 45 min–8.5 hours). EMR GPU achieves ∼100× speedup over ES-HN at depth 7 (IQR median ratio), even with Memory Streaming strategy overhead (Section 5.1).**

per generation); JIT compilation occurs once per run. Solve rates differ: ES-HyperNEAT achieves 0–100% (high variance), while EMR-HyperNEAT achieves 100% at all tested configurations. At depth 6, ES-HyperNEAT solves XOR only 33% of runs (at pop=300), while EMR-HyperNEAT achieves 100%. This difference stems from traversal direction (Full-Depth Evolution, Section 6).

## 6.4 GPU Scaling and Memory Management

**Key Finding**: Multi-GPU is 1.5–1.7× *slower* than single-GPU due to a fundamental JAX constraint: single-GPU uses `lax.while_loop` (GPU-resident), while multi-GPU requires Python loops for fitness aggregation. Since CPPN queries dominate runtime (80–85%), parallelizing only evaluation (5–10%) yields minimal benefit. Multi-GPU provides **memory scalability**, not speed. Use Memory Streaming for large substrates.

**Real-World Benchmark**: Table 8 and Figure 4 compare ES-HyperNEAT and EMR-HyperNEAT on CRSP/Compustat Merged (CCM) financial data (94,464 observations) [6], providing a concrete benchmark beyond XOR's 4 patterns. ES-HyperNEAT runs on CPU (PUREPLES is CPU-only); EMR-HyperNEAT runs on GPU. We use h→h type for fair comparison (Section 4). At depths 1–5, ES-HyperNEAT is faster due to EMR's JIT overhead, but at depth 6, EMR wins: **5.5× faster** (45s vs 247.9s) *despite using DepthOffload streaming* (weights on CPU RAM). EMR's hierarchical computation

**Table 8: Per-generation time on CCM financial dataset (94,464 samples, pop=100). EMR uses h→h type (GPU). Speedup is per-generation (excludes JIT).**

| Depth | ES-HN | EMR | EMR JIT | Speedup |
|---|---|---|---|---|
| 1–3 | 1.9–2.4s | 5.5–6.2s | 11–13s | 0.35–0.39× |
| 4 | 4.2s | 6.6s | 13s | 0.64× |
| 5 | 21.9s | 23s | 46s | 0.95× |
| 6 | 247.9s | 45s† | 89s | **5.6×** |

†Uses DepthOffload streaming (weights on CPU RAM).

**Table 9: Connection type comparison on XOR (depth 5, pop 500, 5 replications, GPU).**

| Type | Solve % | Avg Gens | Time/Gen | Connections |
|---|---|---|---|---|
| Feedforward | 60% | 28.2 | 1.6s | 2,730 |
| h→h | 100% | 3.8 | 4.9s | 4,215 |
| Full Recurrent | 100% | 4.6 | 3.0s | 8,847 |

advantage dominates even when memory tiering overhead is incurred. ES-HyperNEAT exhibits exponential scaling (depth 5→6: 11.3× slower from quadtree explosion), while EMR scales linearly. For applications not requiring H→H connections, feedforward type is faster still, since it avoids the $O(H^2)$ hidden connectivity discovery entirely. Both algorithms achieve comparable fitness (∼0.97).

## 6.5 Connection Type Comparison

Table 9 compares connection type performance on XOR:

Feedforward achieves 60% solve rate on XOR because the problem is solvable without H→H connections since hidden nodes provide sufficient intermediate computation, but the constrained search space makes evolution harder. H→H types achieve 100% because additional connectivity paths ease the evolutionary search. All types exhibit ∼20% VRAM overhead versus feedforward; GPU utilization remains low (2.5–3.4%), confirming EMR-HyperNEAT is bounded by memory.

**Cross-Problem Recurrence Analysis**: Using depth 5, population 300, and 10 replications, we evaluated all six recurrence presets across six benchmark problems (Table 10). Four problems show preset differentiation: XOR (nonlinear boolean logic), Visual Discrimination (spatial counting on 5×5 grid), Orientation Selectivity (V1 simple cell tuning curves), and Retina (comparing left versus right geometry). Two additional problem categories achieve 100% convergence across all presets: Boolean gates (and, or, nand, nor) and Neuroscience (sequence prediction and receptive field). On XOR, Backward and Lateral both achieve 100% convergence ($p < 0.001$ vs Feedforward's 20%), while Hidden-to-Hidden underperforms at 50%. On Visual Discrimination, Backward achieves 100% convergence with 90% of seeds reaching perfect 1.0 fitness; Lateral and Self *hurt* performance (40% vs 90% for Feedforward, $p = 0.057$). Orientation Selectivity shows the opposite pattern: Feedforward achieves 100% while Backward and Hidden-to-Hidden drop to 50% ($p = 0.033$) because backward connections interfere with feedforward spatial feature detection. Retina shows no significant differences, with Feedforward, Self, and Full Recurrent all achieving

100%. We recommend Backward for problems requiring feedback, Feedforward for spatial feature detection, and Full Recurrent as a robust default.

**Table 10: Cross-problem recurrence analysis (depth 5, pop 300, 10 replications). Convergence rate (%) by connection type; bold = best or tied. P-values: Fisher's exact test, best vs feedforward. Bottom two problems show no preset differentiation. †Boolean = and, or, nand, nor. ‡Neuroscience = sequence, receptive.**

| Problem | FF | h→h | Back | Lat | Self | Full | $p$ |
|---|---|---|---|---|---|---|---|
| XOR | 20 | 50 | **100** | **100** | 90 | 60 | <0.001 |
| Visual | 90 | 70 | **100** | 40 | 40 | 90 | <0.001 |
| Orientation | **100** | 50 | 50 | 90 | 90 | 90 | 0.033 |
| Retina | **100** | 80 | 90 | 80 | **100** | **100** | n.s. |
| Boolean† | **100** | **100** | **100** | **100** | **100** | **100** | n.s. |
| Neuroscience‡ | **100** | **100** | **100** | **100** | **100** | **100** | n.s. |

## 7 Discussion

The experimental results demonstrate EMR-HyperNEAT's scalability and speedup advantages. This section provides practical guidance for algorithm selection.

EMR-HyperNEAT is advantageous when JIT compilation overhead is amortized: running many generations, multiple configurations or replications, GPU/TPU acceleration is available, or recurrent connectivity types are needed (Section 6). JIT compilation ranges from 4 seconds at depth 1 to over 100 seconds at depth 7 (Table 6); this compiles both the substrate evaluation and CPPN query functions once for a given population size. ES-HyperNEAT remains preferable for few generations at low depth, single exploratory runs, or CPU-only environments for problems solvable with feedforward networks.

Memory scales as $4^D$ with depth: depth 5 requires <620 MB, depth 6 ~2 GB, and depth 7 ~10 GB (population 500). Memory Streaming enables extreme depths beyond ES-HyperNEAT's practical limit (depth 13 with 358M positions is tractable), with one-time JIT compilation (199s) and 350s per generation.

Since the hierarchical grid depends only on maximum depth (not on problem data, I/O configuration, or CPPN weights), grids can be *precomputed once and reused across all problems*. A depth-7 grid (87,380 positions, ~683 KB) serves unlimited evolutionary runs regardless of input/output dimensions.

## 8 Conclusion

We presented EMR-HyperNEAT, which, to the best of our knowledge, is the first massively parallel adaptive substrate discovery algorithm. By reformulating ES-HyperNEAT's lazy quadtree as eager grid evaluation with variance filtering, EMR-HyperNEAT enables scalability to substrate resolutions previously infeasible: depth 13 (358M positions) on consumer hardware, where ES-HyperNEAT's exponential memory growth prevents evolution entirely.

This scalability yields speedup as a natural consequence: 12–34× on GPU at depths 5–7 for XOR (h→h type), with 5.6× on large datasets. EMR-HyperNEAT scales linearly with depth while ES-HyperNEAT exhibits 4–11× slowdown per level.

Key contributions include: (1) GPU parallelization that reduces complexity from $O(4^D)$ to $O(4^D/P)$ across $P$ cores; (2) recurrent network support through a connection type taxonomy from feedforward to full recurrent; and (3) large-scale substrate evolution through five execution strategies and efficient memory management.

These capabilities open systematic exploration of substrate configurations that were computationally unreachable, enabling future investigation of complex domains such as visual reasoning, continuous control, and other problems where high-resolution substrates are essential.

## Code Availability

The EMR-HyperNEAT implementation, including benchmarks and configurations, will be made publicly available upon paper acceptance.

## References


[1] Anonymous Author(s). [n. d.]. Details omitted for double-blind review.
[2] Anonymous Author(s). [n. d.]. Details omitted for double-blind review.
[3] James Bradbury, Roy Frostig, Peter Hawkins, Matthew James Johnson, Chris Leary, Dougal Maclaurin, George Necula, Adam Paszke, Jake VanderPlas, Skye Wanderman-Milne, et al. 2018. JAX: composable transformations of Python+NumPy programs. (2018).
[4] Richard A Brualdi, Herbert John Ryser, et al. 1991. *Combinatorial matrix theory*. Vol. 39. Springer.
[5] Martin Burtscher and Keshav Pingali. 2011. An efficient CUDA implementation of the tree-based barnes hut n-body algorithm. In *GPU computing Gems Emerald edition*. Elsevier, 75–92.
[6] Center for Research in Security Prices (CRSP). 2025. *CRSP/Compustat Merged Database (CCM)*. CRSP. https://www.crsp.org/products/research-products/crspcompustat-merged-database Accessed via WRDS.
[7] Felix Chalumeau, Bryan Lim, Raphael Boige, Maxime Allard, Luca Grillotti, Manon Flageat, Valentin Macé, Guillaume Richard, Arthur Flajolet, Thomas Pierrot, et al. 2024. QDax: A library for quality-diversity and population-based algorithms with hardware acceleration. *Journal of Machine Learning Research* 25, 108 (2024), 1–16.
[8] Westh et al. 2017. *Pureples - pure python library for es-hyperneat*.
[9] Raphael A Finkel and Jon Louis Bentley. 1974. Quad trees a data structure for retrieval on composite keys. *Acta informatica* 4, 1 (1974), 1–9.
[10] Tero Karras. 2012. Maximizing parallelism in the construction of BVHs, octrees, and k-d trees. In *Proceedings of the Fourth ACM SIGGRAPH/Eurographics Conference on High-Performance Graphics*. 33–37.
[11] Vipin Kumar, Ananth Grama, Anshul Gupta, and George Karypis. 1994. *Introduction to parallel computing*. Vol. 110. Benjamin/Cummings Redwood City, CA.
[12] Robert Tjarko Lange. 2023. evosax: Jax-based evolution strategies. In *Proceedings of the Companion Conference on Genetic and Evolutionary Computation*. 659–662.
[13] Alan McIntyre, Matt Kallada, Cesar G. Miguel, Carolina Feher de Silva, and Marcio Lobo Netto. [n. d.]. *neat-python*.
[14] Sebastian Risi and Kenneth O Stanley. 2011. Enhancing es-hyperneat to evolve more complex regular neural networks. In *Proceedings of the 13th annual conference on Genetic and evolutionary computation*. 1539–1546.
[15] Sebastian Risi and Kenneth O Stanley. 2012. An enhanced hypercube-based encoding for evolving the placement, density, and connectivity of neurons. *Artificial life* 18, 4 (2012), 331–363.
[16] Kenneth O Stanley. 2007. Compositional pattern producing networks: A novel abstraction of development. *Genetic programming and evolvable machines* 8, 2 (2007), 131–162.
[17] Kenneth O Stanley, David B D'Ambrosio, and Jason Gauci. 2009. A hypercube-based encoding for evolving large-scale neural networks. *Artificial life* 15, 2 (2009), 185–212.
[18] Kenneth O Stanley and Risto Miikkulainen. 2002. Evolving neural networks through augmenting topologies. *Evolutionary computation* 10, 2 (2002), 99–127.
[19] Felipe Petroski Such, Vashisht Madhavan, Edoardo Conti, Joel Lehman, Kenneth O Stanley, and Jeff Clune. 2017. Deep neuroevolution: Genetic algorithms are a competitive alternative for training deep neural networks for reinforcement learning. *arXiv preprint arXiv:1712.06567* (2017).
[20] Yujin Tang, Yingtao Tian, and David Ha. 2022. Evojax: Hardware-accelerated neuroevolution. In *Proceedings of the Genetic and Evolutionary Computation Conference Companion*. 308–311.

[21] Lishuang Wang, Mengfei Zhao, Enyu Liu, Kebin Sun, and Ran Cheng. 2024. Tensorized NeuroEvolution of augmenting topologies for GPU acceleration. In *Proceedings of the Genetic and Evolutionary Computation Conference*. 1156–1164.

[22] Claus O Wilke. 2019. *Fundamentals of Data Visualization: A Primer on Making Informative and Compelling Figures*. O'Reilly Media. https://clauswilke.com/dataviz/