Dependency-Aware Parallel Decomposition of the Coincidence Algorithm

Warin Wattanapornprom

***Abstract*—Machine learning is usually framed as prediction. Optimization is harder: it must generate a decision that does not yet exist and improve it from feedback, which makes an estimation-of-distribution algorithm (EDA) a form of generative learning applied to search. The Coincidence Algorithm (COIN) makes this explicit through a directed generator matrix H, learning positively from good permutations and negatively from poor ones. This paper studies how a parallel Coincidence Algorithm should be built by following COIN's own dependency structure: population-wise sampling and evaluation are independent, within-permutation position sampling is sequential, histogram accumulation is privatizable with a reduction, and the generator-matrix update is row-separable behind a generation barrier. We implement and measure horizontal, vertical, and two-dimensional CPU decompositions, a compilation ladder from plain Python to multiprocessing, and an evaluator-only OpenCL offload pilot on an integrated and a discrete GPU. On one workstation, horizontal population decomposition is the strongest CPU strategy tested, and the four-worker full pipeline reaches roughly 2.1–$2.8\times$ speedup depending on population size; finer decompositions lose performance to synchronization and reduction cost, and GPU offload is competitive only at the largest population tested and only on one of the two devices measured. All reported numbers are pilot measurements from a single workstation at 5–10 repetitions, reported for their pattern rather than as a confirmatory benchmark.**

***Index Terms*—estimation of distribution algorithms, parallel computing, Coincidence Algorithm, permutation optimization, dependency-aware decomposition, deterministic parallel random number generation, OpenCL, GPU computing**

I. Introduction

Machine learning is usually framed as prediction: given an input, infer a label, a score, or a continuation that already exists, in some sense, latent in the data. Optimization asks for something harder. It has to generate a decision that does not yet exist, test it, and improve it from whatever feedback that test provides. An Estimation-of-Distribution Algorithm (EDA) makes this generative side of learning explicit: instead of predicting an outcome, it learns a probabilistic model of what a good solution looks like and samples new candidates from that model, generation after generation. Read this way, a parallel estimation-of-distribution algorithm is generative machine learning for optimization, not a prediction task with extra steps, and the Coincidence Algorithm (COIN) is a clean instance of it. COIN represents its model as a directed generator matrix H over a permutation's alphabet, samples a new permutation by walking H under an already-visited-symbol constraint, and updates H through explicit reward and punishment: good permutations push mass toward the edges they used (accumulated in a *reward histogram*), poor ones push mass away from the edges they used (accumulated in a *punishment histogram*), and – in its *conservative* update mode – every such adjustment preserves the total outgoing mass of the row it touches. Positive and negative learning acting on the same matrix, rather than a single select-and-rebuild step, is what distinguishes COIN from a conventional positive-only histogram EDA, and it is also what makes its model update a genuinely row-structured operation rather than an easily shared table (Section III) – which turns out to matter a great deal once the algorithm has to run on more than one core.

Model-based evolutionary algorithms do not have one kind of parallelism; they have several, at different structural levels, and these do not compose for free. A generation is sampling, evaluation, selection, and a model update, and each stage carries a different dependency shape. Sampling and evaluation are ordinarily independent across individuals. Selection and the model update are per-generation barriers: every individual's result has to be in before the next generation's model can be trusted. And when the model itself samples sequentially – as a Markov-chain edge-transition model does – the positions *within* one individual's permutation depend on each other and cannot be parallelized the way individuals can. Handing an algorithm more hardware threads without respecting this structure does not make it faster by default; it can just as easily buy more synchronization, more reduction traffic, or more host-device transfer than the sequential version ever paid for.

Two predecessor implementations already built a parallel Coincidence Algorithm, on different hardware. [1] distributed COIN's population generation, evaluation, and matrix update across CPU cores with Intel Threading Building Blocks, using row-wise decomposition for the matrix update, and reported speedups on four Traveling Salesman Problem instances. [2] moved the same three stages onto GPU blocks and threads, keeping host-device transfers to a minimum, and reported larger speedups at larger population sizes on a discrete NVIDIA GPU. We are not claiming to reproduce either paper's numbers here: the hardware, the software stack, the random-number generation, the problem instances, and the timing boundaries all differ, in some cases by more than a decade of process-node and compiler improvement. We treat both as architectural predecessors – evidence that row-wise matrix decomposition and stage-level parallelism were the right places to look, well before this paper existed – and Section VII returns to a qualitative comparison of design

choices against them, not a cross-machine speedup ranking.

The central contribution of this paper is not that COIN can be made to run in parallel. Two groups already showed that. The finding worth reporting is which kind of parallelism actually pays for itself, and why: useful parallelism follows the dependency structure of the algorithm, not the number of cores available to throw at it. Population-wise sampling and evaluation are independent and parallelize cleanly. Sampling positions within one permutation stay sequential no matter how many workers exist. Histogram accumulation tolerates private per-worker statistics with a reduction at the end. The generator-matrix update is row-separable, but it sits behind a generation barrier every worker has to respect. Decompositions that expose more nominal parallelism than this structure actually supports do not merely fail to help – they can lose performance outright to synchronization and reduction cost, and this paper measures that failure mode directly rather than assuming it away.

This paper's pilot targets a single modern workstation with a many-core CPU and two GPUs of different vendor and class (an integrated and a discrete part, both reachable through OpenCL), which lets a *single* deterministic implementation be compared, unchanged, across the CPU and both GPU backends under one explicit dependency taxonomy. The question is simple: does a decomposition that exposes more parallelism – vertical position stripes, two-dimensional tiles, OpenCL offload – actually run faster than the simplest decomposition that matches the algorithm's own independence structure (horizontal, population-wise)? Our answer, on this machine, is no, and we report that as a result worth stating rather than a negative finding to bury. What this paper adds relative to the two 2012 predecessors is not parallel COIN itself, row-wise decomposition, or transfer-conscious GPU design – all three already exist – but a shared-implementation, cross-vendor comparison built on an explicit dependency taxonomy, together with a schedule-independent determinism guarantee neither predecessor reports.

Contributions:

1) A dependency taxonomy for permutation-EDA computation – population-level independence, sequential within-permutation position dependence, a per-generation synchronization barrier, and an associative reward/punishment-histogram reduction – from which horizontal, vertical, and tiled decompositions are derived rather than tried ad hoc (Section III).
2) A race-free parallel implementation of COIN's learning step, using per-worker private reward/punishment histograms with an explicit reduction and row-exclusive ownership of the generator-matrix update, verified by direct equality against the sequential path under a shared deterministic logical random stream rather than by statistical similarity (Sections IV–V).
3) An empirical, equal-hardware comparison of an implementation-strategy baseline (Python, NumPy, and Numba variants), horizontal/vertical/ two-dimensional CPU decompositions, and an evaluator-only cross-vendor OpenCL offload pilot on the Knight's Tour benchmark, reported under one evidentiary standard defined in Section VI, with an explicit account of which further designs remain proposed and unimplemented (Sections VI–VII).

We state the pilot status of every number in this paper once, here: all reported measurements come from a single workstation, at 5–10 repetitions per configuration, with deterministic parity checked before every timing but no archived raw-output file, fixed CPU affinity, or thermal control. Section VIII discusses this limitation in full; it is not repeated as a disclaimer in every subsequent paragraph. We locate this paper's novelty in the dependency-aware decomposition argument, the deterministic and schedule-independent verification method, and the comparative empirical analysis across CPU and GPU acceleration – not in the use of `prange`, OpenCL, or multiprocessing themselves, which are standard tools we apply rather than invent.

II. Background and Related Work

A. Parallel evolutionary algorithms

Coarse-grained parallel evolutionary algorithms are usually classified as master-slave (a single population, parallel evaluation only), island / coarse-grained (multiple semi-isolated populations with periodic migration), or fine-grained / cellular (spatially structured, local mating) [3]; Markov-chain analyses of these models [4] and broader surveys of evolutionary-computation parallelism [5] establish that the right granularity depends on the algorithm's own dependency structure, not on how many independent work items can nominally be identified. COIN's decomposition is closest in spirit to a master-slave model at the population level (Section IV), but its position-wise sequential sampling and row-owned model update have no direct analogue in a conventional genetic algorithm's crossover and mutation operators, which act on already-materialized individuals rather than sampling one position at a time from a shared global model.

B. Parallel estimation-of-distribution algorithms

Parallel EDA work has largely targeted Bayesian-network-structured models: [6] parallelized the Bayesian Optimization Algorithm's structure learning and sampling, and [7] generalizes a design methodology around the relative cost of model-building versus model-sampling. GPU-parallel univariate EDAs exist for fixed-length binary strings [8]; GPU island-model mappings exploit the SIMT execution model by running many small, mostly independent sub-populations concurrently [9].

C. Parallel permutation EDAs

Permutation-domain EDAs – edge-histogram, node/position-histogram, and ranking-based models such as Mallows and Plackett–Luce – are surveyed in [10], which documents the representational landscape COIN's generator matrix belongs to, without addressing parallel implementation. COIN's Markov sampling constraint (Section III) makes its

per-individual sampling loop sequential in a way a per-bit compact-GA sampler is not, so the fixed-length binary GPU mapping of [8] does not transfer directly; and COIN's single shared generator matrix has no natural island structure – every individual samples from the same H within a generation – so this paper's GPU opportunity sits at the individual-parallel sampling/evaluation level (Section IV), mirrored by the horizontal-stripe OpenCL kernel we benchmark, rather than at the population-partition level island models exploit.

D. COIN multicore and GPU predecessors

[1] and [2] already established that COIN parallelizes across CPU cores and across a GPU respectively, using row-wise matrix-update decomposition in both cases, and [2] already recognized host-device transfer as a cost worth minimizing by keeping generation, evaluation, and learning GPU-resident. This paper does not claim novelty for parallelizing COIN, for row-wise decomposition, or for minimizing host-device transfer – all three are already present in the 2012 work. What it adds instead is an explicit dependency taxonomy distinguishing individual-, position-, and update-level parallelism, from which horizontal, vertical, and tiled decompositions are derived; a single deterministic implementation shared unchanged across one CPU and two OpenCL GPU backends of different vendors, rather than separate hand-tuned codebases; an explicit separation between evaluator-only and full-pipeline measurement; a schedule-independent determinism guarantee verified by direct output equality; and a negative-result comparison showing when an additional parallel dimension underperforms the simplest matching decomposition. Table V returns to this comparison directly.

E. Deterministic parallel random-number generation

A parallel stochastic algorithm that reseeds its random stream differently under sequential and parallel scheduling cannot be checked for schedule-independent correctness by direct output comparison; it can only be checked statistically, which is a weaker guarantee. Counter-based generators designed specifically for reproducible parallel use, such as Philox and Threefry [11], and splittable generators such as SplitMix64 [12], solve this by making each stream a pure function of an index rather than of generation order. This paper's deterministic logical random stream is SplitMix64-style, keyed by a $(\text{generation}, \text{individual})$ pair, which is what permits the direct equality checks reported in Section V rather than a statistical-similarity argument. The PCG family [13] addresses a related but distinct goal (statistical quality and space efficiency per stream, rather than splittability per se) and is noted as the other major modern non-cryptographic PRNG family, though it is not the generator used here.

F. Histogram/reduction and race avoidance

Building a shared histogram from parallel workers without lock or atomic contention is a well-studied GPU and multicore pattern: give every worker a private copy, accumulate into it with no cross-worker communication, and merge the private copies afterward [14]. This paper applies that pattern to COIN's reward and punishment histograms (Section V), and additionally exploits a COIN-specific structural fact – the generator matrix's rows are mutually independent under both its update rules – to make the *update* step itself race-free by row ownership, a stronger guarantee than generic private-histogram accumulation alone provides. The implementation is built on NumPy's array model [15], Numba's LLVM-based JIT compilation of a Python subset including its `parallel=True`/`prange` construct [16], and PyOpenCL for cross-vendor GPU kernel dispatch [17]. We claim no novelty for these tools individually; the contribution is how COIN's dependency structure is mapped onto them.

III. COIN and Dependency Structure

A. Representation

An individual is a permutation $\pi = (\pi_1, \ldots, \pi_n)$ of an n-element alphabet. COIN's Edge model maintains a non-negative generator matrix $H \in \mathbb{Z}_{\geq 0}^{n \times n}$, interpreted as

$$H_{ij} \;\propto\; P(\pi_{k+1} = j \mid \pi_k = i), \tag{1}$$

i.e., row i is an unnormalized categorical distribution over which symbol follows symbol i.

B. Sampling equation

A permutation is sampled by choosing a start symbol, then repeatedly choosing the next symbol by roulette selection over row $H_{\pi_k,\cdot}$ restricted to symbols not yet used:

$$P(\pi_k = j \mid \pi_{k-1} = i, \text{visited}) = \frac{H_{ij}}{\sum_{u \notin \text{visited}} H_{iu}}. \tag{2}$$

Figure 1 illustrates this on a five-symbol toy example: the current symbol's row is masked against already-visited columns before roulette selection runs. This masking is exactly what makes position $k+1$'s distribution depend on the entire prefix $(\pi_1, \ldots, \pi_k)$, not just on π_k: two different prefixes that share the same last symbol can still sample π_{k+1} from different supports, because they have used different symbols already. Positions inside one individual's permutation are therefore *sequentially dependent* and cannot be sampled out of order or independently; only the choice of *which individual* is independent of every other individual's sampling.

C. Reward and punishment equations

After a population of N individuals is sampled and evaluated, they are ranked by fitness and split into a good cohort G (top fraction) and a bad cohort B (bottom fraction, under the algorithm's configured reward/penalty ratios). Treating each permutation as a cycle (position n wraps to position 1, matching the implementation's cyclic adjacency convention), the reward histogram and punishment histogram are

$$C^{+}_{ij} = \sum_{\pi \in G} \mathbb{1}[(i,j) \in \pi], \tag{3}$$

$$C^{-}_{ij} = \sum_{\pi \in B} \mathbb{1}[(i,j) \in \pi], \tag{4}$$

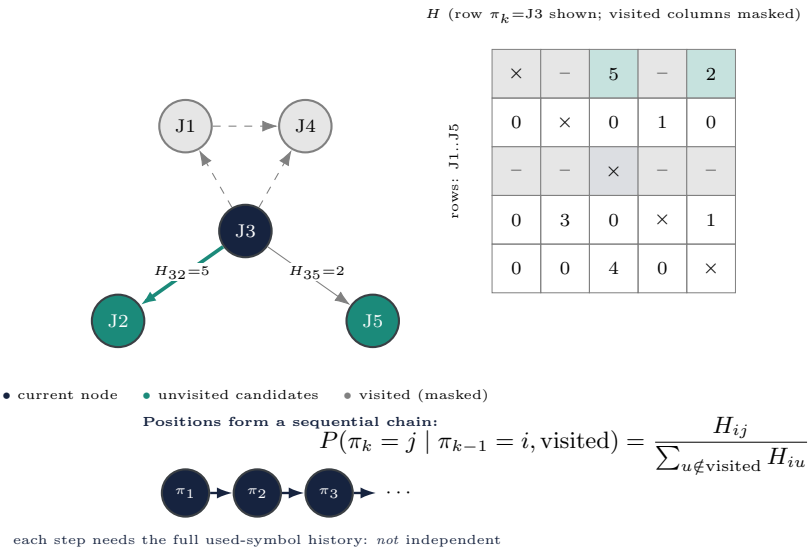

Fig. 1. COIN's edge representation and sampling step. Left: a directed toy graph with the current symbol (navy), unvisited candidates (teal), and visited symbols masked out (gray). Right: the corresponding row of H, with visited columns masked before roulette selection (Equation 2). Bottom: positions inside one individual form a sequential chain – sampling position $k{+}1$ requires the outcome of every earlier position, so positions are not independent the way individuals are.

where $(i, j) \in \pi$ means symbol j immediately follows symbol i in π's cyclic order. Both sums decompose over individuals, so if a population is partitioned into disjoint chunks across W workers, each worker can accumulate a private reward/punishment histogram from its own chunk, and the histograms recombine by ordinary summation,

$$C^{+} = \sum_{w=1}^{W} C^{+(w)}, \qquad C^{-} = \sum_{w=1}^{W} C^{-(w)}, \tag{5}$$

with no cross-worker communication required until the final sum. This is the formal statement of why reward/punishment accumulation is *associative and reducible*.

D. Conservative update

COIN's *reconstruction* learning mode discards H's previous state each generation and rebuilds it directly from the reward histogram,

$$H_{ij} \leftarrow 10\, C^{+}_{ij} + 1 \quad (i \neq j), \tag{6}$$

which behaves like a conventional positive-only rebuilt histogram EDA and carries no memory of any generation before the current one. COIN's *conservative* learning mode instead adjusts the existing H incrementally and is mass-preserving per row. For each punishment event on edge (i, j) with H_{ij} above a fixed floor, the implementation first adds one unit to *every* off-diagonal entry of row i – including H_{ij} itself – and then subtracts $n - 1$ units from H_{ij} specifically:

$$H_{i,c} \leftarrow H_{i,c} + 1 \;\; \forall\, c \neq i, \qquad H_{ij} \leftarrow H_{ij} - (n-1). \tag{7}$$

Because H_{ij} is one of the $n - 1$ entries that received the preceding $+1$, its net change is $-(n-2)$, not $-(n-1)$; every other non-diagonal, non-target entry nets $+1$; row mass is conserved: $(n-2)(+1) - (n-2) + 0 = 0$. This matches the algorithm's original specification for the Edge model, confirmed by direct comparison against the reference update rule rather than assumed. Each reward event on (i, j) moves one unit into H_{ij} from every other row-i entry currently above the floor,

$$H_{ij} \leftarrow H_{ij} + 1, \quad H_{i,c} \leftarrow H_{i,c} - 1 \;\text{ for each } c \text{ with } H_{i,c} > n, \tag{8}$$

applied once per qualifying competitor c. Both operations only ever read and write row i of H – never any other row – for every event drawn from row i's own reward/punishment histograms. This row-locality is what makes the conservative update, like reconstruction, decomposable by row ownership (Section IV); it is also what distinguishes conservative COIN from an ordinary rebuilt histogram, since it accumulates signed evidence *across* generations rather than replacing the model each time.

E. Dependency taxonomy

Collecting the above into the taxonomy this paper's decompositions are derived from (Figure 2):

- **Population-level independence.** Individuals are independent given a fixed model H (sampling and evaluation).
- **Sequential position dependency.** Positions inside one individual's permutation are sequentially dependent (the unused-symbol constraint, Equation 2).
- **Generation-level synchronization.** Ranking, cohort selection, and the model update form a per-generation barrier: no individual in generation $g{+}1$ can be sampled from a model that has not finished incorporating generation g's statistics.

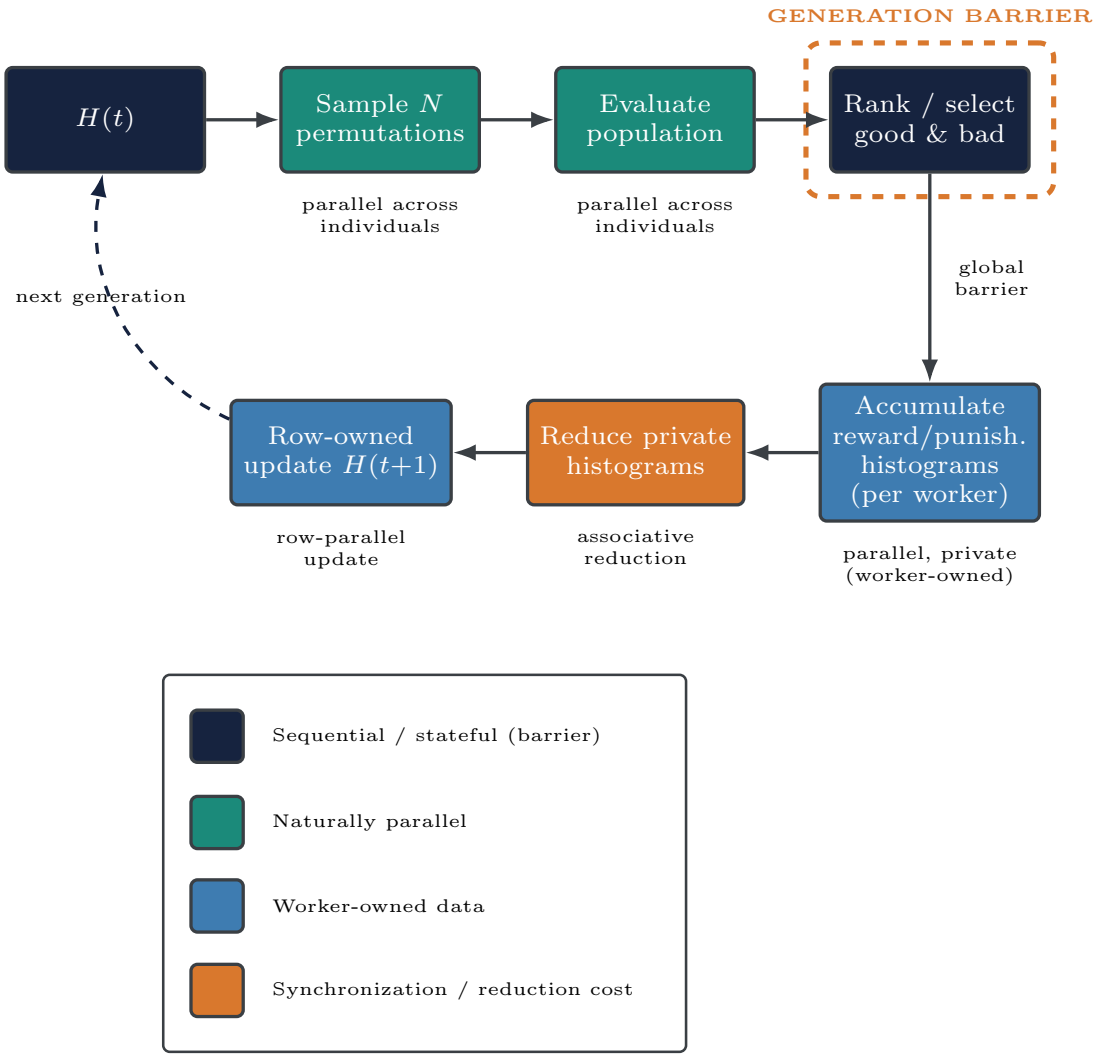

Fig. 2. The COIN generation pipeline and the dependency class of each stage. Solid arrows are data flow; the dashed arrow is the generational feedback loop through the updated generator matrix $H(t{+}1)$. Sampling and evaluation are independent across individuals (teal); ranking/cohort selection is a global generation barrier (navy, dashed outline); reward/punishment accumulation is parallel but worker-private (blue); reduction is associative (orange); and the generator-matrix update is row-parallel (blue). Position-to-position sampling within one individual is sequential and is not parallelized in this work (Figure 1).

- **Associative histogram/model-update reduction.** Reward and punishment accumulation is associative and reducible (Equation 5); rows of the generator matrix can be exclusively owned by one worker, under both learning modes.

Independent runs (different seeds or problem instances) are embarrassingly parallel with no cross-run communication at all, but this is a coarser axis than the four above and is treated separately in Section IV's discussion of proposed cube batching.

F. Amdahl interpretation

Let f be the fraction of one generation's wall-clock time spent in the strictly sequential barrier stages (ranking and, in the current implementation, the reduction handoff) and $1-f$ the fraction spent in stages that scale with W workers. Amdahl's law [18], [19] bounds attainable speedup at

$$S_p \leq \frac{1}{f + \frac{1-f}{W}}. \tag{9}$$

Because the fixed per-generation overhead (kernel launch, thread-pool dispatch, reduction bookkeeping) does not shrink as population size N shrinks, f is effectively *larger* at small N and smaller at large N for the same worker count – exactly the population-size-versus- efficiency pattern reported in Section VII.

IV. Parallel Designs

A. Horizontal population decomposition

Each worker owns a contiguous or interleaved block of complete individuals for both sampling and evaluation (Figure 3A).

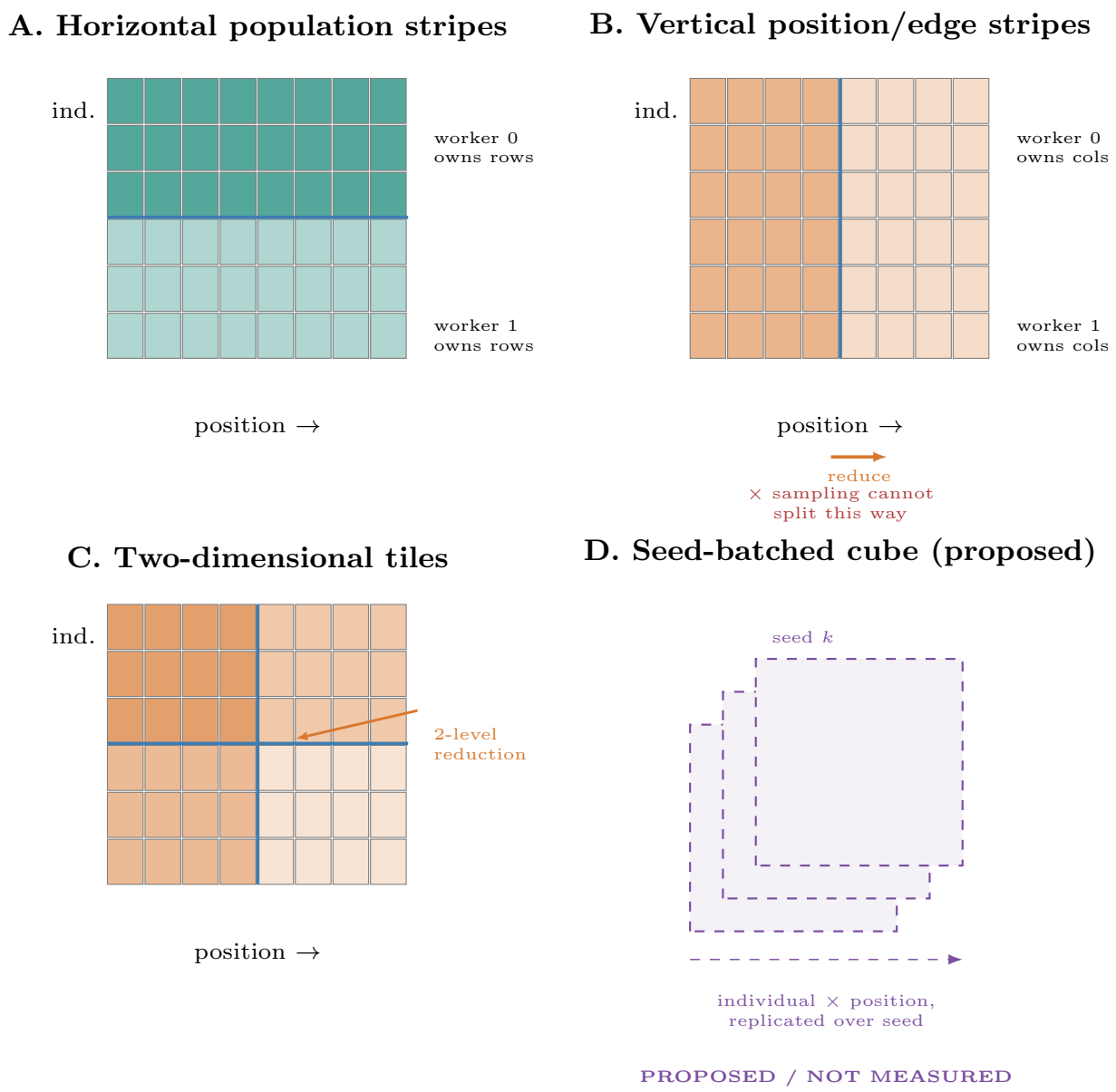

Fig. 3. Four ways to partition the individual × position matrix across workers, using the same grid in every panel. (A) Horizontal: each worker owns complete individuals – matches independence exactly, no partial-fitness reduction, and is the strongest CPU decomposition measured (Table IV). (B) Vertical: each worker owns a position range within every individual; valid for evaluation but not for Markov sampling, which cannot resolve position $k{+}1$ without position k's outcome. (C) Two-dimensional tiles: both axes partitioned, trading locality for a two-level reduction. (D) Seed-batched cube: an added seed axis, proposed and not implemented or measured.

This matches the independence structure of Section III-E exactly: no communication is needed until every worker has finished its block. Logical work unit: one individual (a complete permutation). Grain size: coarse. Synchronization point: none beyond a final join. Expected overhead: lowest of the three implemented CPU decompositions. It is the strongest CPU decomposition measured in this paper (Table IV).

B. Vertical position/edge decomposition

Positions are partitioned into ranges, and each worker produces a partial result for its range, followed by a reduction across ranges (Figure 3B). For *evaluation* (summing legal-move or adjacency scores along a fixed, already-sampled permutation), this is valid: the score contribution of position range $[a, b)$ does not depend on any other range once the permutation is fixed. For *sampling*, this decomposition is not valid: position $k + 1$'s candidate set depends on the full used-symbol history from positions $1, \ldots, k$ (Section III-E), so a worker assigned range $[a, b)$ cannot begin without already knowing the outcome of every earlier position, which is precisely what a different worker would be computing

concurrently. This paper's vertical decomposition is therefore evaluation-only. Logical work unit: a position range across all individuals. Grain size: fine. Synchronization point: a partial-sum reduction after every worker's range completes. Expected overhead: higher than horizontal, dominated by the reduction and by smaller per-worker work at fixed population size.

C. Two-dimensional tiled decomposition

Both the individual and position axes are partitioned into a grid of tiles (Figure 3C), applied to evaluation for the same reason vertical stripes are evaluation-only. Tiling can improve cache locality (a worker's tile fits a smaller working set than a full row or column) and accelerator occupancy, at the cost of an additional partial-result buffer and a two-level reduction (within a tile's position range, then across an individual-range's tiles). Logical work unit: one tile (an individual-range $\times$ position-range block). Grain size: medium. Synchronization point: two-level reduction. Expected overhead: between vertical and horizontal, and sensitive to tile shape.

D. Seed-batched cube decomposition (proposed)

A further axis – independent seeds or runs – can in principle be stacked behind the individual $\times$ position matrix (Figure 3D), so a single accelerator dispatch processes several independent seeds' populations at once, amortizing fixed kernel-launch and transfer overhead across seeds rather than across individuals alone. Logical work unit: one seed's individual $\times$ position batch. Grain size: coarse across the batch. Synchronization point and expected overhead: not established – **this decomposition is proposed and is not implemented or measured in this paper**; Section IX discusses what implementing it would require.

E. Private histogram reduction and row-owned update

As formalized in Equation 5, each worker accumulates its own private reward/punishment histograms from its chunk of the ranked population, with no shared writes and therefore no atomic operations or locks; a final reduction pass sums the private histograms into the shared C^+, C^- used by the update step (Figure 4). Every row i of H is then assigned to exactly one worker for the update step; no worker writes another worker's rows. Because both the conservative and reconstruction update rules only ever touch row i when processing events drawn from row i's own statistics (Section III), this ownership boundary follows the algorithm's actual read/write structure rather than being an arbitrary partition imposed on top of it – unlike splitting individual matrix *cells* across workers, which would require synchronization inside a single row's update (Figure 4's crossed-out inset).

This trades memory (each worker's private histogram is the full $n \times n$ shape, so total private memory scales as Wn^2) for the complete absence of write contention, in both the reduction and the update step.

F. Pipeline and heterogeneous execution

A generation can be viewed as a pipeline of stages – CPU-side preparation, accelerator-side kernel execution, result transfer, and host-side ranking – each of which could in principle overlap with the next generation's earlier stages, or run concurrently on CPU and GPU. **No pipelined or concurrent CPU/GPU speedup is measured in this paper**; Section IX discusses this, together with the Intel integrated GPU's unified-memory access pattern versus the AMD discrete GPU's separate-memory pattern, as future work.

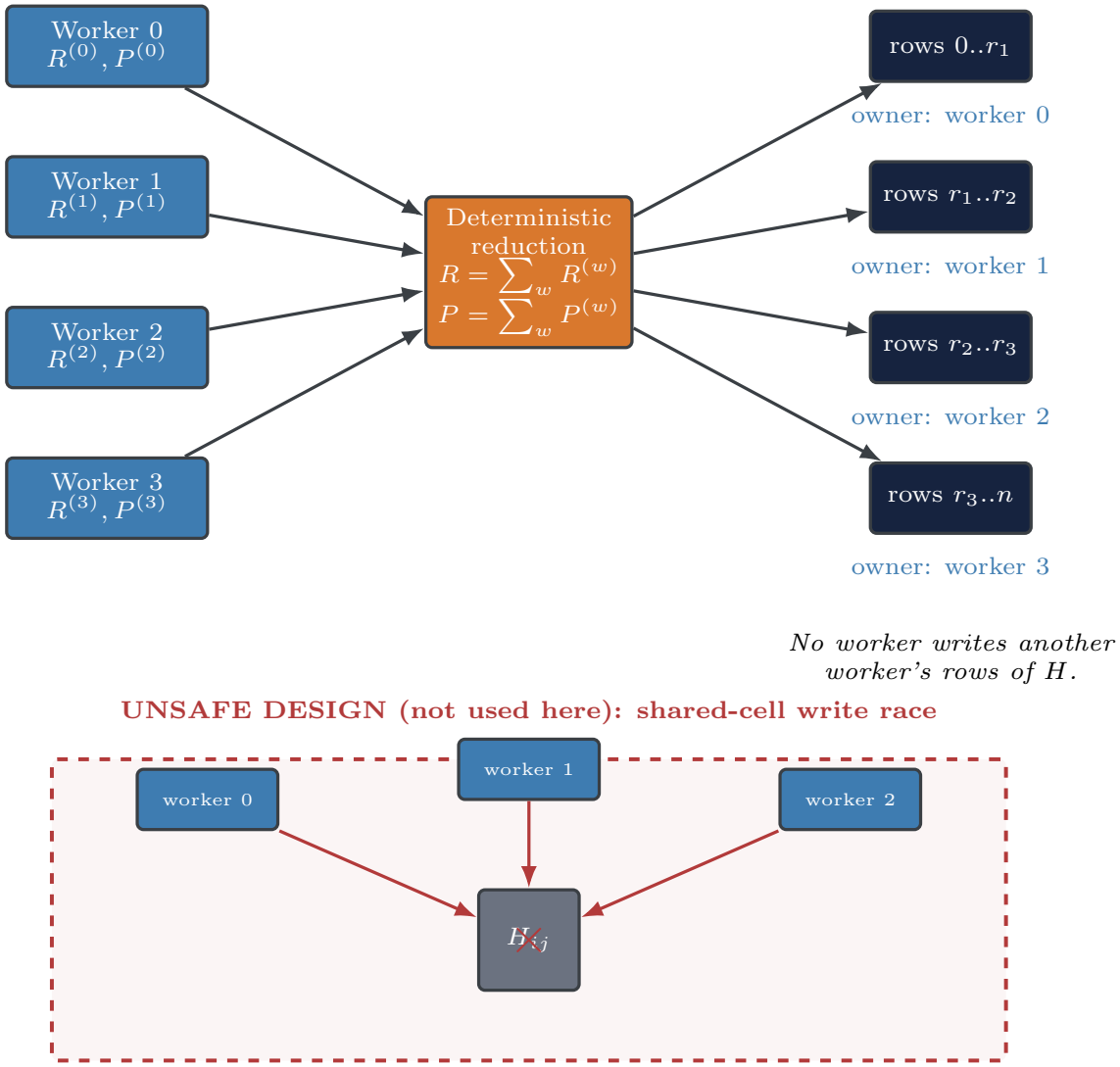

Fig. 4. Race-free learning architecture. Four workers accumulate private reward/punishment histograms $R^{(w)}, P^{(w)}$ from their population chunks; a deterministic reduction sums them into shared R, P; H is then partitioned by owned row ranges, one worker per range, so no worker writes another worker's rows. Inset: the unsafe alternative – multiple workers writing the same matrix cell – which this design avoids entirely, not merely mitigates.

V. Deterministic Implementation and Verification

A. Logical RNG stream

Each individual in generation g is assigned its own SplitMix64-style stream, seeded as a pure function of (g, i) – generation and individual index – via a fixed golden-ratio increment and the standard SplitMix64 finalizer mixing constants [12]. Figure 5 makes this mapping explicit: $\text{seed}_{g,i} = \text{Mix}(\text{master\_seed}, g, i)$. Because this stream does not depend on which worker processes (g, i) or in what order workers finish, sequential execution and parallel execution under any worker schedule draw *exactly the same random values* for the same (g, i) pair – the stream belongs to the logical individual, not to a worker or an execution schedule.

B. Parity invariants

Every benchmark script used in this paper asserts equality – `numpy.array_equal` on populations and fitness, plus exact equality of the best score and the final generator matrix for the full-pipeline benchmark – between its sequential and parallel code paths *before* timing either one, for every population size tested. This is an inline, benchmark-time parity check, not a separate regression-test suite, a distinction that bears on how strong a correctness claim this paper is entitled to make (Section VIII).

C. Race-free implementation

The learning step's implementation follows Sections III and IV directly: a chunked private-histogram counting pass (each of W chunks writes only its own $n \times n$ private array) followed by a reduction pass, and a row-owned update pass

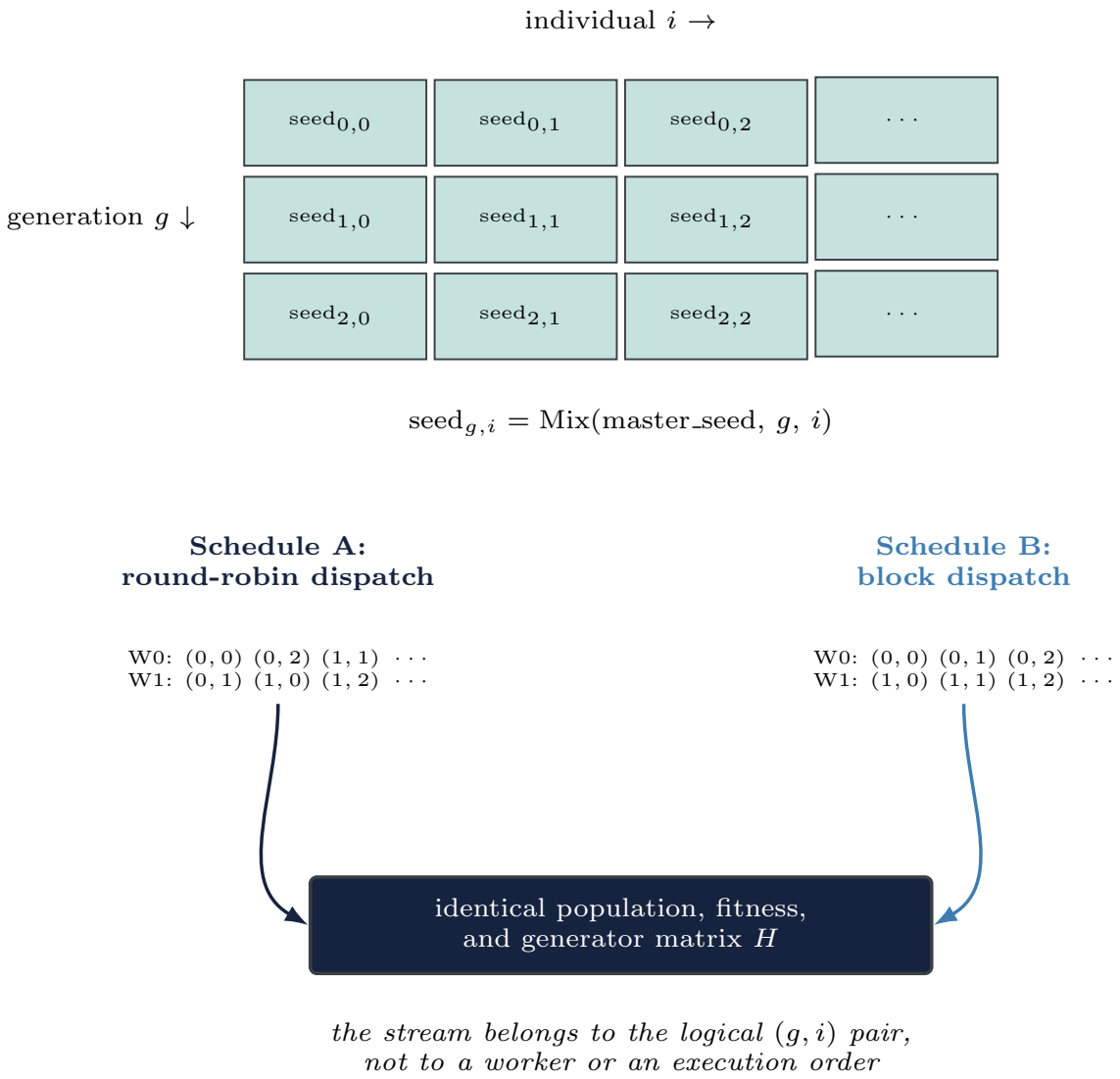

Fig. 5. Deterministic logical random-stream mapping. Each (generation, individual) cell owns a stream derived only from the master seed and its own indices. Two different worker dispatch schedules – A (round-robin) and B (block) – draw the same per-cell streams in a different order, so both produce an identical population, fitness, and generator matrix H.

(each worker's parallel loop iterates over the row index and only reads/writes that row). No worker ever writes a memory location another worker also writes, under either learning mode, which is why no lock or atomic operation appears anywhere in the parallel learning path.

D. Verification tests

Verification in this paper means direct equality, not statistical comparison: identical populations, identical fitness values, an identical best score, and an identical final generator matrix H between the sequential and parallel/alternative-implementation code paths under a shared master seed. This is a strictly stronger check than a distributional similarity test would be, and any divergence under it is a correctness bug, not sampling noise. No automated, independently runnable test suite currently enforces these invariants outside benchmark time (Section VIII).

VI. Experimental Method

A. Hardware and software environment

All reported pilot measurements were executed on the single workstation described in Table I. Every run checked deterministic parity before recording a timing, and used a fixed random seed.

B. Timing boundaries

Three benchmark scripts define the timing boundaries used in Section VII. The evaluator implementation-strategy baseline times seven Knight's Tour evaluator implementations – pure-Python list iteration, a typed-array variant, scalar NumPy

TABLE I
TEST ENVIRONMENT AND MEASUREMENT PROTOCOL (AUTHOR-REPORTED).

Component	Value
CPU	Intel Core Ultra 7 155H (16c/22t)
Discrete GPU (OpenCL)	AMD, device `gfx1101`
Integrated GPU (OpenCL)	Intel Arc iGPU
OS	Windows
Python / NumPy / Numba	3.10.6 / 2.2.6 / 0.67.0
PyOpenCL	2026.1.4
CPU workers	4
Random seed	42
Repetitions	5–10 per config. (benchmark-dependent)
Warm-up policy	One untimed generation excluded
Deterministic parity	Checked and passed before every timing
CPU affinity / power	Not fixed or recorded
Thermal monitoring	Not performed
Raw output archived	No (console output only)

indexing, fully vectorized NumPy, sequential Numba, Numba with `parallel=True`/`prange` (horizontal), and a multi-process variant – against the same populations, reporting the median of 10 repetitions, all outputs checked for equality against the pure-Python reference before timing. The full-pipeline benchmark measures one complete generation – deterministic population generation, fitness evaluation, stable ranking, cohort selection, private reward/punishment histogram accumulation, reduction, and row-owned conservative update – for the Knight's Tour problem (an 8×8 board, 64-symbol permutations, fitness = number of legal consecutive knight moves plus a closing-move bonus for a complete tour), median of 5 repetitions. The evaluator-only decomposition benchmark isolates the evaluation stage's decomposition choice: sequential Numba, horizontal, vertical (stripe width 16), 32×16 tiled, and, at the largest tested population only, two OpenCL kernels (Intel integrated, AMD discrete), reported as end-to-end offload measurements including allocation, transfer, and kernel launch (Figure 8), median of 5 repetitions.

C. Warm-up policy and repetitions

Every benchmark excludes one untimed warm-up generation or call to remove JIT-compilation cost from the timed region; the OpenCL program-build step specifically was not separately confirmed as excluded in this pass. Repetition counts are 5 (full-pipeline, evaluator-decomposition) or 10 (implementation-strategy baseline) – sufficient for a pilot median, not for a defensible confidence interval.

D. Populations and evaluator-only vs. full-pipeline distinction

The implementation-strategy baseline and evaluator-only decomposition benchmarks were run at populations 100, 400, 1,000, and 10,000 (the implementation-strategy baseline has no population-400 row; it was not run at that size, and no value is estimated for it). The full-pipeline benchmark was run at 100, 400, and 1,000 only; population scaling beyond 1,000 for the full pipeline is not measured in this pass. The evaluator-only and full-pipeline benchmarks answer different questions and are not interchangeable: the former isolates the evaluation stage's decomposition cost in isolation, the latter measures one complete generation including sampling, ranking, and the row-owned update, so a decomposition's evaluator-only speedup does not by itself predict its full-pipeline speedup.

E. Author-pilot limitations

Every measurement in Section VII is an author-reported pilot measurement: executed directly on the workstation of Table I with deterministic parity checked before timing, at 5–10 repetitions and a single optimization seed, with no archived raw output file. Timing repetitions and optimization seeds answer different questions – the former characterizes wall-clock variance for a fixed computation, the latter characterizes solution-quality variance across the algorithm's own randomness – and this paper reports only the former. Section VIII states the full confirmatory protocol this motivates; later sections refer back to it rather than restate it.

VII. RESULTS

Every number in this section is an author-reported pilot measurement (Section VI). We report every measured value, including values that make a decomposition or backend look unfavorable, and do not backfill a missing cell by estimation or interpolation.

A. Implementation ladder

Figure 6 and Table II report the seven-implementation baseline, median of 10 repetitions, at populations 100, 1,000, and 10,000 (no population-400 measurement exists for this benchmark). Every step's output is checked for equality against the Python-list reference before timing, so semantics do not change along the ladder – only representation, vectorization, compilation, and the parallelism model do.

Two patterns are worth naming before the decomposition results below. First, compilation – not container replacement – is responsible for essentially all of the speedup in this table: the typed-array and scalar-NumPy variants, which change data representation without compiling to native code, are *slower* than plain Python lists at every population size, while vectorized NumPy and either Numba variant are the only strategies that help. Second, Numba's parallel variant is *slower* than sequential Numba at population 100 (fixed thread-dispatch overhead exceeds the available per-worker work at that size) and only overtakes it at populations 1,000 and 10,000, the same fixed-overhead-versus-population-size pattern the Amdahl interpretation of Section III predicts for the full pipeline.

B. Full-pipeline results

Table III reports the full-pipeline benchmark, 4 workers, median of 5 repetitions. Speedup is $S_p = T_1/T_p$ and efficiency is $E_p = S_p/p$ for p workers. Worker-count scaling beyond four is not measured in this pass.

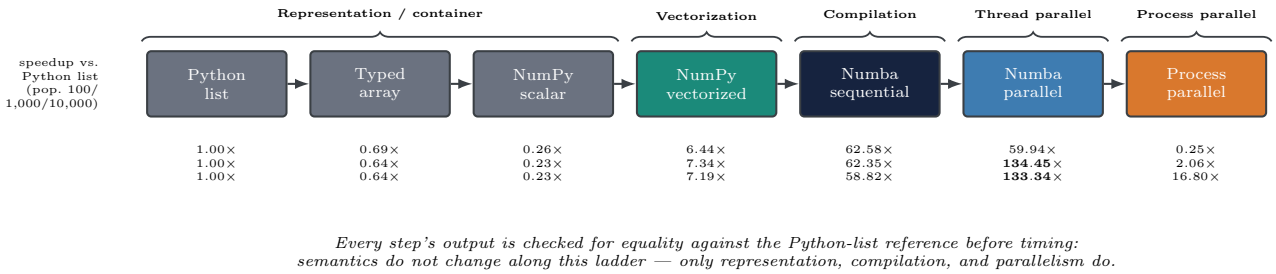

Fig. 6. Implementation ladder from a plain Python list to process parallelism, grouped by what actually changed at each step. Compilation (Numba), not container replacement, is responsible for most of the speedup: typed-array and scalar-NumPy indexing change representation without compiling to native code and are *slower* than plain Python at every population size.

TABLE II
IMPLEMENTATION LADDER: ABSOLUTE TIME AND SPEEDUP VS. PURE-PYTHON-LIST, MEDIAN OF 10 REPETITIONS, SEED 42. ABSOLUTE TIME IS DERIVED FROM THE MEASURED PYTHON-LIST BASELINE TIME AT THAT POPULATION DIVIDED BY THE REPORTED SPEEDUP RATIO; IT IS NOT INDEPENDENTLY TIMED PER ROW. NO POPULATION-400 MEASUREMENT EXISTS FOR THIS BENCHMARK.

Implementation	Pop. 100 ms (×)	Pop. 1,000 ms (×)	Pop. 10,000 ms (×)
Python list	0.394 (1.00)	3.611 (1.00)	34.33 (1.00)
Typed array	0.571 (0.69)	5.64 (0.64)	53.6 (0.64)
NumPy scalar	1.52 (0.26)	15.7 (0.23)	149 (0.23)
NumPy vectorized	0.0612 (6.44)	0.492 (7.34)	4.78 (7.19)
Numba sequential	0.00630 (**62.58**)	0.0579 (62.35)	0.584 (58.82)
Numba parallel	0.00657 (59.94)	0.0269 (**134.45**)	0.257 (**133.34**)
Process parallel	1.58 (0.25)	1.75 (2.06)	2.04 (16.80)

TABLE III
FULL-PIPELINE SPEEDUP AND EFFICIENCY, 4 WORKERS, 50 GENERATIONS, CONSERVATIVE LEARNING, MEDIAN OF 5 REPETITIONS, SEED 42.

Pop.	T_1 (ms)	T_p (ms)	S_p (E_p)
100	62.44	29.90	2.09× (52.2%)
400	236.67	90.04	2.63× (65.7%)
1,000	680.28	242.63	2.80× (70.1%)

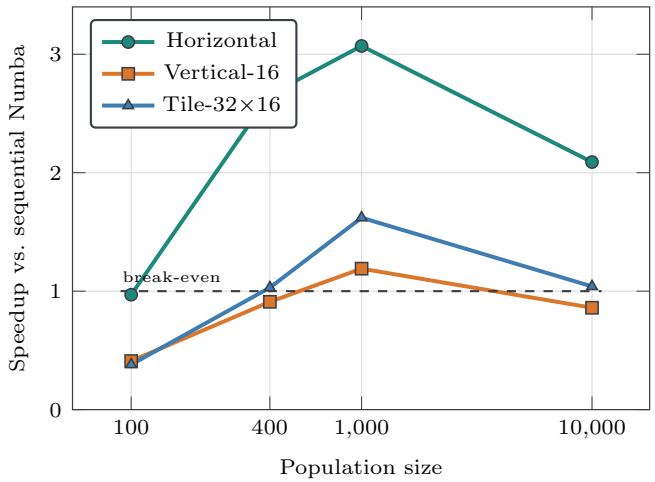

Fig. 7. CPU decomposition speedup vs. sequential Numba across population sizes (Table IV). Horizontal is the strongest tested CPU decomposition at every measured population, but even it falls slightly below break-even at population 100; vertical and tiled fall further below break-even at that size and remain below horizontal throughout.

C. Decomposition results

Table IV and Figure 7 report the evaluator-only decomposition comparison, expressed as speedup relative to sequential Numba, median of 5 repetitions. OpenCL offload was measured only at population 10,000 in this pass; smaller-population OpenCL behavior is left for future confirmatory work.

D. OpenCL pilot

Figure 8 (left) shows the measured path: allocate, host-to-device transfer, kernel, device-to-host transfer, host reduction. This is an end-to-end offload cost, not raw device compute throughput, and this pass does not instrument the stages separately. [2] already moved generation, evaluation, and learning onto the GPU and minimized transfers; this paper does not claim a resident GPU design as new. Figure 8 (right) sketches such a resident pipeline as a proposed architecture, not yet benchmarked; the possible novelty relative to [2] is a modern deterministic, cross-vendor, common-code implementation together with the controlled decomposition comparison of Table IV, not GPU-resident state itself.

E. Already-documented component results

Two results come from the repository's own committed benchmark documentation, with a reproduction command, rather than from the author-reported pilot measurements above, and are reported with that distinct provenance made explicit. They were measured under Numba 0.66.0, one minor version behind the 0.67.0 used for Tables II–IV. The optimized, Numba-compiled Edge COIN implementation is 134.47× faster per warm generation than the reference implementation on the same 64-symbol Knight's Tour problem, checked for output equality before timing. Separately, on the Flow Shop problem, a Numba-batched evaluator is 17.03× faster than the reference two-dimensional scheduler on a 20×5 instance and 757.24× faster on a 500×20 instance, checked against the reference to an absolute tolerance of 10^{-10}. We report

TABLE IV

DECOMPOSITION COMPARISON: ABSOLUTE TIME AND SPEEDUP VS. SEQUENTIAL NUMBA, MEDIAN OF 5 REPETITIONS, SEED 42. "—" MARKS A CELL NOT MEASURED IN THIS PASS. SPEEDUP IS THE MEDIAN OF PER-REPETITION RATIOS AND DOES NOT ALWAYS EQUAL THE RATIO OF THE TWO MEDIAN ABSOLUTE TIMES SHOWN, SINCE THE TWO STATISTICS ARE COMPUTED INDEPENDENTLY; BOTH ARE REPORTED AS MEASURED RATHER THAN RECONCILED POST HOC.

Population	Sequential (ms)	Horizontal	Vertical-16	Tile-32×16	Intel OpenCL / AMD OpenCL
100	—	— (0.97×)	— (0.41×)	— (0.38×)	— / —
400	—	— (2.66×)	— (0.91×)	— (1.03×)	— / —
1,000	0.0666	0.0252 (**3.07**×)	0.0649 (1.19×)	0.0478 (1.62×)	— / —
10,000	0.6903	0.3438 (2.09×)	0.8370 (0.86×)	0.6897 (1.04×)	0.3875 (≈1.85×) / 4.5291 (≈0.16×)

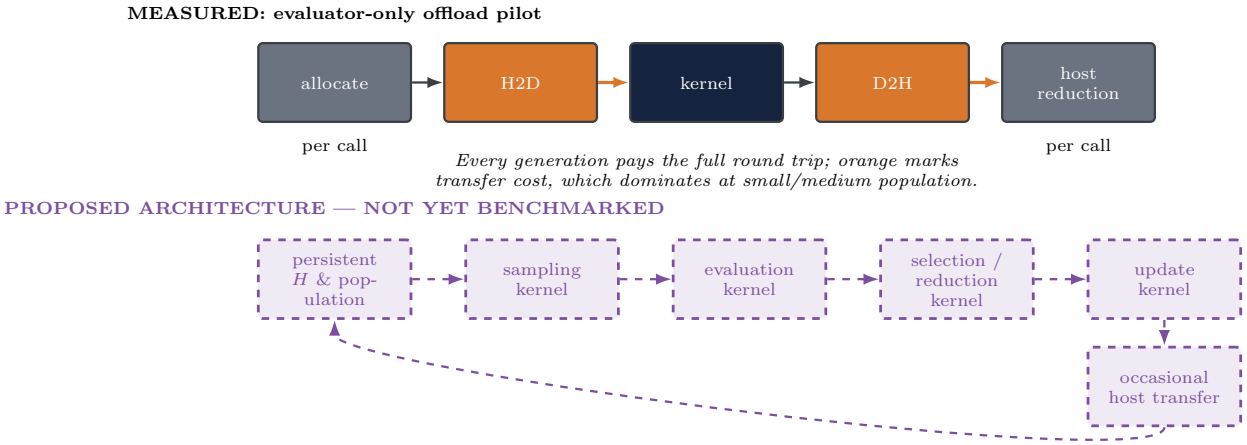

Fig. 8. Two ways to place a permutation EDA on an accelerator. Top, measured: the evaluator-only pilot benchmarked in this paper – every call pays allocation, transfer, kernel execution, and host reduction, so the reported OpenCL numbers are this end-to-end offload cost. Bottom, proposed and not yet benchmarked: a resident design that keeps the population and generator matrix on the device across generations and transfers only occasional results.

this second result as corroborating evidence that evaluator-side batching matters across this codebase's permutation problems, not as a COIN-parallelization number specifically.

F. Comparison with predecessor architecture

Table V compares design choices and evidentiary status, not raw performance, across machines separated by roughly a decade of hardware and software evolution; a numeric speedup ranking across such different platforms would not be meaningful and is not attempted.

TABLE V
ARCHITECTURAL COMPARISON WITH THE TWO 2012 PREDECESSOR PAPERS. ROWS DESCRIBE DESIGN CHOICES AND EVIDENTIARY STATUS, NOT RAW PERFORMANCE.

	Srimook & Chongstitvatana [1]	Tongsiri & Chongstitvatana [2]	This paper
Stages parallelized	Population generation, evaluation, matrix update	Population generation, evaluation, model update	Population generation, evaluation (CPU + OpenCL pilot); row-owned matrix update (CPU)
Decomposition	Row-wise matrix update across CPU cores	Kernel-based population/evaluation/update on GPU	Horizontal population (primary); vertical/tiled evaluation compared, not adopted; row-owned update
Memory residency	CPU-resident throughout (not applicable)	GPU-resident across generations, as reported	Evaluator-only offload, not resident (measured); resident design proposed, not implemented
Synchronization	Intel TBB thread synchronization, as reported	GPU block/thread synchronization and host coordination, as reported	Private-histogram reduction plus row-owned update; no locks or atomics
Hardware	CPU, quad-core (Intel Core i5)	GPU (NVIDIA GeForce 540M)	Many-core CPU plus a cross-vendor OpenCL pilot (Intel integrated, AMD discrete)
Deterministic parity	Not reported	Not reported	Identical populations, fitness, and generator matrix under a shared deterministic logical stream, checked before every timing
Timing boundary	Not specified in the available record	Not specified in the available record	Full allocate/H2D/kernel/D2H/reduce round trip every generation, measured (evaluator-only); resident boundary proposed, not implemented

VIII. DISCUSSION AND THREATS TO VALIDITY

Figure 9 anchors this section: it separates what this pass measured, what a predecessor paper reported on different hardware, and what remains proposed with no implementation at all, so the discussion below does not need to re-qualify every sentence with its evidentiary tier.

A. Compilation dominates

Table II shows compilation, not container replacement, gives the largest evaluator-only improvement of any strategy tested: typed arrays and scalar NumPy indexing do not help, and actively hurt, relative to plain Python; vectorized NumPy substantially improves over Python loops, but Numba compilation provides a further order of magnitude beyond that.

B. Horizontal decomposition follows independence

Horizontal population decomposition is the strongest tested CPU decomposition at every population size (Table IV, Figure 7), consistent with Section III's dependency analysis: it is the only decomposition that requires no reduction step at all. Even so, at population 100 horizontal itself measures at $0.97\times$ – marginally below sequential-equivalent performance – and vertical-16 and tile-32×16 fall further below it ($0.41\times$ and $0.38\times$), because at this size the fixed per-dispatch overhead a reduction step imposes is large relative to the small amount of per-worker work available.

C. Fine decomposition suffers synchronization

By population 1,000, all three CPU decompositions exceed sequential performance and horizontal peaks at its best measured ratio ($3.07\times$); at population 10,000, all three *fall back* from their population-1,000 values rather than continuing to improve. This non-monotonic shape is consistent with Amdahl-style overhead dominating at small populations and a second, currently uninstrumented effect (plausibly memory-bandwidth or cache pressure at larger array sizes) emerging at the largest one; isolating which requires the per-stage instrumentation Figure 10 calls for, and we do not claim to have identified the mechanism from an aggregate timing number alone. We also note that at population 1,000 specifically, the horizontal/vertical/tile ranking places tile-32×16 above vertical-16 – an ordering that reversed the corresponding ranking in an earlier, superseded pilot pass at the same population size, which we read as evidence that 5 repetitions do not yet fix a stable ranking.

D. The OpenCL pilot measures transfer-heavy offload

The evaluator-only OpenCL numbers, measured only at population 10,000 in this pass ($\approx1.85\times$ Intel, $\approx0.16\times$ AMD), are preliminary end-to-end observations and do not rank intrinsic GPU compute capability. As Figure 8 makes explicit, the measured path is allocate $\rightarrow$ transfer $\rightarrow$ kernel $\rightarrow$ transfer $\rightarrow$ reduction, a cost that can be dominated by transfer and launch overhead rather than by kernel execution time, though this pass does not instrument the stages separately and so cannot attribute the two devices' differing ratios to a specific stage. Intel's integrated part becoming competitive with the CPU baseline at this population, while AMD's discrete part does not, is consistent with – but not proof of – the discrete part paying a larger transfer cost, since it does not share system memory with the CPU; we state this as the most parsimonious reading available, not as a measured transfer-time breakdown. A negative GPU speedup at this population does not by itself establish that GPUs are unsuitable for this workload; it establishes that this particular transfer-bound offload pattern, at this population, on this device, is not yet competitive.

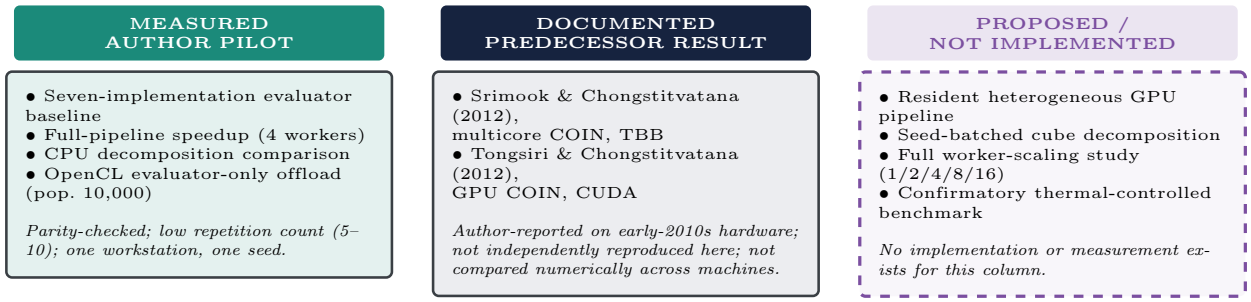

Fig. 9. Experimental validity map. Measured author-pilot results (teal), documented predecessor results on different, older hardware (navy), and proposed designs with no implementation or measurement (dashed purple) are kept visually and textually distinct throughout this paper.

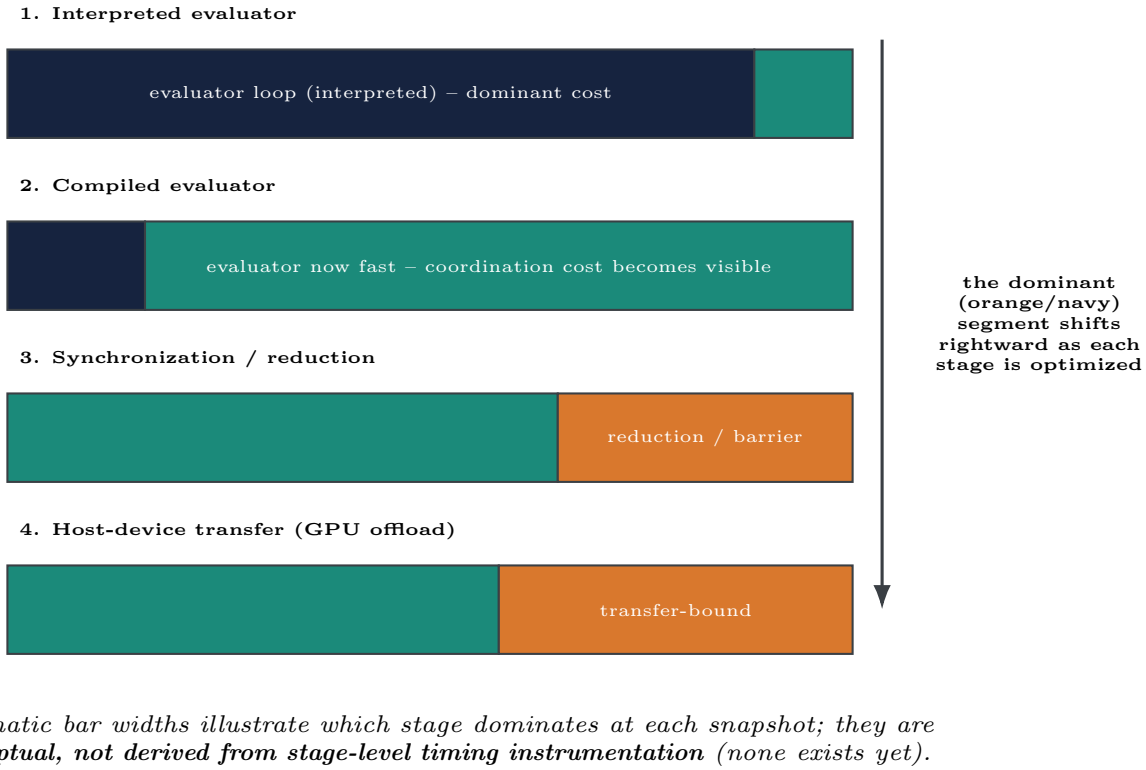

Fig. 10. Conceptual bottleneck migration across implementation stages. Optimizing one stage exposes the next serial or coordination cost: an interpreted evaluator dominates first; once compiled, coordination (synchronization/reduction) becomes visible; once CPU-side coordination is addressed, host-device transfer dominates under GPU offload. Bar proportions are schematic and are not derived from stage-level timing instrumentation, which does not exist for this implementation.

E. Low repetitions, one machine

All timing data comes from one workstation (Table I), at 5–10 repetitions per configuration, a single worker count (4) for the full pipeline, and a single optimization seed. Five to ten repetitions support a pilot median, not a defensible confidence interval, and do not rule out a single favorable or unfavorable repetition dominating that median.

F. Thermal and CPU affinity not controlled; no energy measurement

No CPU-affinity, fixed-power-profile, or thermal-monitoring information was recorded, and this machine has previously been observed to run hot under sustained load; a mid-run throttling event would silently distort a 5-repetition median. No energy-to-solution measurement is reported, because no reliable sensor was available in this pass – we omit rather than approximate this quantity.

G. No stage-level instrumentation

No stage-level timing (sampling, evaluation, ranking, histogram construction/reduction, generator update, host-device transfer) exists in this implementation. Figure 10 is offered as a conceptual architecture figure for exactly this reason, explicitly labeled as not derived from measurement; no per-stage runtime-breakdown chart is reported as data anywhere in this paper.

H. No generalization beyond Knight's Tour

Both the full-pipeline and evaluator-only decomposition benchmarks are specific to the 64-symbol Knight's Tour problem. Flow Shop's evaluator has separately documented batching speedups (Section VII) that suggest the evaluator-batching argument generalizes, but no parallel timing data exists yet for Flow Shop, TSP, or any other permutation problem in this paper.

I. Output provenance and correctness-checking mechanism

The three benchmark scripts print JSON to standard output and do not write a file, so no archived raw-output artifact accompanies the numbers in Tables II–IV; they are transcribed console output. The reported parity guarantee comes from inline equality assertions inside the benchmark scripts at benchmark time, not from a standing, independently runnable regression-test suite.

Collecting what this pilot pass does and does not support: compilation and data-oriented implementation matter more than container replacement alone; horizontal decomposition is the strongest tested CPU decomposition overall, though not uniformly above break-even; vertical and tiled decompositions are sensitive to population size in a way no measured mechanism yet explains; four-worker full-pipeline parallelism gives roughly 2.1–2.8$\times$ speedup across the tested population range; evaluator-only OpenCL offload can lose to the CPU baseline when transfer and launch overhead dominate; and Intel's integrated OpenCL device becomes competitive with the CPU only at the largest population tested. None of these observations establishes universal CPU-versus-GPU superiority, a stable decomposition ranking, or a mechanism for the population-10,000 pattern; they describe this one workstation, this one problem, these population sizes, this worker count, and this repetition count, and should not be read more broadly than that.

IX. Future Work

Resident device state. Keeping the population and generator matrix on the accelerator across generations, transferring only periodic or final results, would remove the per-generation transfer round trip Figure 8 shows dominates the current pilot. This is architecturally similar to what [2] already did on 2012-era CUDA hardware; the open contribution here would be a modern, deterministic, cross-vendor common-code version, not the resident-state idea itself.

Full pipeline kernels. A resident design additionally requires sampling, evaluation, selection/reduction, and update kernels (Figure 8, right), raising open questions about which stages can safely overlap given the per-generation model-update barrier (Section III).

Cube-batched independent seeds. Stacking independent seeds behind the individual $\times$ position matrix (Figure 3D) would amortize fixed kernel-launch and transfer overhead across seeds, targeting the launch/transfer-dominated pattern visible at the one population size where OpenCL was measured. This requires restructuring the sampling and evaluation kernels to accept a batch dimension and has not been attempted.

Worker scaling. Worker-count scaling at 1, 2, 4, 8, and cautiously 16 workers for the full-pipeline benchmark is not measured in this pass (4 workers only); a prior, now-superseded pilot's 8-worker numbers are deliberately not reused alongside this pass's 4-worker data, since they come from a different measurement pass.

Confirmatory benchmark. Before these numbers can be read as confirmatory: 20–30 repetitions per configuration with warm-up (including the OpenCL program-build step) excluded from every timed region; randomized execution order within each repeat block; documented CPU affinity and a fixed power profile with thermal monitoring; median, interquartile range, mean, standard deviation, and a bootstrap confidence interval per cell; population scaling beyond 1,000 for the full pipeline; OpenCL measurement at populations 100/400/1,000, not only 10,000; per-stage timing instrumentation; throughput in candidates/s and generations/s; memory footprint per configuration; extension to at least one further permutation problem already in the codebase (Flow Shop, or a plain TSP instance, since only the time-windowed TSPTW variant exists currently); and repeated-seed solution-quality parity checks, separate from the single-shared-seed identical-output determinism check already in place.

Energy efficiency. Energy-to-solution is proposed as future work only if a reliable on-machine sensor becomes available; no approximated figure is reported in its place.

X. Conclusion

Parallel hardware does not erase algorithmic dependencies. It exposes them. The best decomposition is the one that respects what can be independent, what must be reduced, and where the model must wait.

The Coincidence Algorithm's parallel opportunities are not uniform: population-wise sampling and evaluation are embarrassingly parallel, within-permutation position sampling is sequential and does not admit the same treatment, and reward/punishment-histogram accumulation and the row-structured model update are associative and row-separable respectively. A decomposition that respects this structure – horizontal population stripes, private per-worker histograms with reduction, and row-owned model updates, all verified for exact output equality against a sequential baseline under a shared deterministic logical random stream – is, on this pilot's evidence, both the simplest and the fastest CPU option tested overall, while decompositions that expose more nominal

parallelism measurably lose to coordination and transfer cost at small population sizes, and even horizontal decomposition itself does not clear break-even at the smallest population tested. These are author-reported pilot measurements in the sense fixed in Section VI: well-grounded on their own terms, but not yet a finished experimental campaign.

Reproducibility

The implementation used in this study is not yet available in a public archival repository. The three benchmarks reported in Section VII were run as:

```
benchmark_knight_seven_implementations.py
--sizes 100,400,1000,10000
--repeats 10 --workers 4 --seed 42
benchmark_knight_parallel_shapes.py
--sizes 100,400,1000,10000
--repeats 5 --workers 4 --seed 42
benchmark_full_coin_parallel.py
--sizes 100,400,1000 --generations 50
--repeats 5 --workers 4
--learning conservative
```

under Python 3.10.6, NumPy 2.2.6, Numba 0.67.0, and (for the evaluator-decomposition benchmark's OpenCL path) PyOpenCL 2026.1.4 (Table I). None of the three scripts currently writes its JSON result to a file; we recommend appending a redirect to a timestamped result file on every future run, so a later revision or an independent reproduction attempt has an archived artifact rather than transcribed console output.

References

[1] W. Srimook and P. Chongstitvatana, "An implementation of coincidence algorithm on multi-core processors," in *International Conference on Computer and Information Technology (ICCIT)*, Bangkok, Thailand, 2012, pp. 76–80, https://www.cp.eng.chula.ac.th/~prabhas/paper/2012/iccit-coin-multi-core.pdf. An M.Sc. thesis of the same title is archived at the Chulalongkorn University Intellectual Repository (CUIR), handle 123456789/21866, 2011.

[2] T. Tongsiri and P. Chongstitvatana, "An implementation of coincidence algorithm on graphic processing units," in *2012 Ninth International Conference on Computer Science and Software Engineering (JCSSE)*, 2012, pp. 126–130.

[3] E. Cantú-Paz, *Efficient and Accurate Parallel Genetic Algorithms*, ser. Genetic Algorithms and Evolutionary Computation. Kluwer Academic Publishers, 2001, covers master-slave (Ch. 3), island/coarse-grained (Ch. 4), and fine-grained/cellular (Ch. 8) parallel GA models used as the classification scheme in this paper's related work.

[4] ——, "Markov chain models of parallel genetic algorithms," *IEEE Transactions on Evolutionary Computation*, vol. 4, no. 3, pp. 216–226, 2000.

[5] E. Alba and M. Tomassini, "Parallelism and evolutionary algorithms," *IEEE Transactions on Evolutionary Computation*, vol. 6, no. 5, pp. 443–462, 2002.

[6] J. Ocenasek and J. Schwarz, "The parallel bayesian optimization algorithm," in *The State of the Art in Computational Intelligence*. Physica-Verlag, 2000, pp. 61–67.

[7] J. Ocenasek, E. Cantú-Paz, M. Pelikan, and J. Schwarz, "Design of parallel estimation of distribution algorithms," in *Scalable Optimization via Probabilistic Modeling*, ser. Studies in Computational Intelligence. Springer, 2006, pp. 187–203.

[8] A. Ferigo and G. Iacca, "A gpu-enabled compact genetic algorithm for very large-scale optimization problems," *Mathematics*, vol. 8, no. 5, p. 758, 2020.

[9] T. V. Luong, N. Melab, and E.-G. Talbi, "GPU-based island model for evolutionary algorithms," in *Proceedings of the 12th Annual Conference on Genetic and Evolutionary Computation (GECCO '10)*, 2010, pp. 1089–1096.

[10] J. Ceberio, E. Irurozki, A. Mendiburu, and J. A. Lozano, "A review on estimation of distribution algorithms in permutation-based combinatorial optimization problems," *Progress in Artificial Intelligence*, vol. 1, no. 1, pp. 103–117, 2012.

[11] J. K. Salmon, M. A. Moraes, R. O. Dror, and D. E. Shaw, "Parallel random numbers: As easy as 1, 2, 3," in *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis (SC '11)*, 2011, pp. 1–12.

[12] G. L. Steele, D. Lea, and C. H. Flood, "Fast splittable pseudorandom number generators," in *Proceedings of the 2014 ACM International Conference on Object Oriented Programming Systems Languages & Applications (OOPSLA '14)*, 2014, pp. 453–472.

[13] M. E. O'Neill, "PCG: A family of simple fast space-efficient statistically good algorithms for random number generation," Harvey Mudd College, Tech. Rep. HMC-CS-2014-0905, 2014.

[14] J. Gómez-Luna, J. M. González-Linares, J. I. Benavides, and N. Guil, "An optimized approach to histogram computation on GPU," *Machine Vision and Applications*, vol. 24, no. 5, pp. 899–908, 2013.

[15] C. R. Harris, K. J. Millman, S. J. van der Walt, R. Gommers, P. Virtanen, D. Cournapeau, E. Wieser, J. Taylor, S. Berg, N. J. Smith *et al.*, "Array programming with NumPy," *Nature*, vol. 585, no. 7825, pp. 357–362, 2020.

[16] S. K. Lam, A. Pitrou, and S. Seibert, "Numba: A LLVM-based Python JIT compiler," in *Proceedings of the Second Workshop on the LLVM Compiler Infrastructure in HPC (LLVM-HPC '15)*, 2015, pp. 1–6.

[17] A. Klöckner, N. Pinto, Y. Lee, B. Catanzaro, P. Ivanov, and A. Fasih, "PyCUDA and PyOpenCL: A scripting-based approach to GPU run-time code generation," *Parallel Computing*, vol. 38, no. 3, pp. 157–174, 2012.

[18] G. M. Amdahl, "Validity of the single processor approach to achieving large scale computing capabilities," in *Proceedings of the April 18–20, 1967, Spring Joint Computer Conference (AFIPS '67)*, 1967, pp. 483–485.

[19] M. D. Hill and M. R. Marty, "Amdahl's law in the multicore era," *Computer*, vol. 41, no. 7, pp. 33–38, 2008.