Manuscript prepared for Journal of Parallel and Distributed Computing

berkeDet: data-parallel lexicographic permutation generation and streaming exact determinant evaluation via block replication and index mapping

Berke Gülmen[a], Meryem Gülmen[a], Ömer Gülmen[a]

[a]Independent Researcher, Ankara 06560, Turkey

Email: gulmenberkemeryem@gmail.com

* Corresponding author.

Abstract

Every classical permutation generator—the lexicographic steppers (naive, Narayana Pandita) and the swap-based methods (Steinhaus–Johnson–Trotter, Heap)—is an incremental chain: permutation $r{+}1$ is computed from the value of permutation r, so the critical path is $\Omega(n!)$ and vector or parallel hardware is given no independent work. We present a compositional alternative built purely from block replication and index mapping. Its two subroutines, meryemPer (all $n!$ permutations, lexicographic order, $O(n \cdot n!)$ work, no sorting, reversal, or scanning) and meryemSign (all $n!$ signs, $\Theta(n!)$ work, no inversion counting), form a factorial tree in which every element of a layer is a pure function of one element of the previous layer; in the work–span model both have span $O(n)$, giving parallelism $\Theta(n!)$. To our knowledge meryemPer is the first inherently data-parallel lexicographic permutation generator: a single-core vectorized realization already outruns the C-implemented `itertools` chain by up to 6.4×. We then develop a streaming execution scheme that exploits the tree's rank locality: fixed chunks of $m!$ lexicographically consecutive lines are emitted through a block-size law $m! \cdot \Pi_{\max} < 2^{62}$ that makes every 64-bit chunk sum provably overflow-safe, with chunk partials reduced exactly in arbitrary-precision integers. Applied to the Leibniz determinant as a certified stress test, the scheme computes the exact 14×14 determinant (87.2 billion terms) on a free Tesla T4 in 194 MB of working memory—a 6,300× reduction over materialization—and, with a fused register-resident kernel on one NVIDIA B300, sums 20,922,789,888,000 terms ($n = 16$) exactly in 361 s at a flat 57.9×10^9 lines/s, every value certified equal to the independent fraction-free Bareiss computation, while LU floating-point error drifts from 10^{-16} to 10^{-15} over the same range. Measured horizons are reported honestly: $n = 17$ at 1.7 h, $n = 18$ at 30.7 h, $n = 19$ at 584 h on the same device.

Keywords: permutation generation · lexicographic order · data parallelism · work–span model · GPU computing · exact computation · Leibniz determinant

1 Introduction

Listing the permutations of $\{1, \ldots, n\}$ is one of the oldest algorithmic problems [2, 1], and lexicographic order is its canonical form: the order symbolic algebra systems expect, the order that supports reproducible testing and rank-addressable partitioning, the order in which the subject is taught. The classical toolkit for producing it, however, shares a single structural commitment that this paper is about: it is *incremental*. The naive stepper sorts a suffix at every step; Narayana Pandita's algorithm (Algorithm L of Knuth [1]; the modern `std::next_permutation`) replaces the sort by a reversal but still scans, swaps, and reverses per step; the Steinhaus–Johnson–Trotter (SJT) [3, 4] and Heap [5] methods produce each permutation by one swap applied to the previous one. In every case, permutation $r{+}1$ is computed *from the value of* permutation r.

That commitment has a cost invisible to sequential analysis: the $n!$ outputs form a dependency chain, so the critical path is $\Omega(n!)$ regardless of processor count and regardless of implementation language. On vector units, GPUs, and array runtimes—hardware that rewards large numbers of identical independent operations—a chain offers nothing to parallelize. As enumeration workloads (exhaustive search, test generation, exact and symbolic computation) meet hardware whose

growth is now almost entirely in width rather than clock speed, the chain structure itself becomes the bottleneck.

This paper develops a compositional alternative and carries it from theory to trillion-scale certified computation. The contributions are:

1. **A data-parallel lexicographic generator.** meryemPer builds the complete lexicographic listing of S_n layer by layer from S_{n-1}, purely by block replication and order-preserving index remapping: no sorting, no reversal, no pivot scanning, no in-place modification at any step (Section 3). The construction is a factorial tree whose child rule is local—each element of a layer depends on exactly one element of the previous layer—so in the work–span model [6, 7, 8] the generator has work $O(n \cdot n!)$ and span $O(n)$, i.e. parallelism $\Theta(n!)$, against the chains' $\Omega(n!)$ critical path (Section 4). To our knowledge it is the first inherently data-parallel lexicographic permutation generator; a vectorized realization on a *single core* already generates all permutations up to 6.4× faster than the C-implemented `itertools` chain.
2. **Sign generation without inversion counting.** The companion subroutine meryemSign emits the signs of all $n!$ permutations, in the same order, in $\Theta(n!)$ total work—amortized $O(1)$ per sign—by block replication with alternating flips. Isolated measurements show ∼100× over per-permutation inversion counting, 350–400× over merge-based parity, and 15–18× over the strongest classical baseline, a Narayana stepper with $O(1)$ incremental parity updates (Section 7.1).
3. **A streaming execution scheme with provable overflow safety.** Exploiting the tree's rank locality, we emit fixed windows of $m!$ lexicographically consecutive lines from a cached order-m suffix table and an $O(n)$ prefix state. A *block-size law*, $m! \cdot \Pi_{\max} < 2^{62}$, guarantees every 64-bit chunk sum is overflow-free; chunk partials are reduced in arbitrary-precision integers, so results are exact at any order. Working memory falls from $\Theta(n \cdot n!)$ to $\Theta(m! \cdot n)$ (Section 5). A prepare/execute double-buffered pipeline is specified, implemented, and measured, with an honestly negative single-GPU finding explained by roofline reasoning.
4. **Trillion-scale certified evaluation.** Using the Leibniz determinant as a stress test whose every output can be certified against an independent exact algorithm (fraction-free Bareiss elimination [9]), a fused register-resident CUDA kernel sums all 20,922,789,888,000 signed products of a 16×16 integer matrix exactly in 361.4 s on one NVIDIA B300—a flat 57.9 Glines/s from $n = 13$ through 16—inside a ∼6.3 GB working set, where materialization would require 335 TB (Section 7.3). On a free Tesla T4 the same scheme computes the exact $n = 14$ determinant in 194 MB. LU floating-point error is measured drifting from 3.5×10^{-16} to 7.1×10^{-15} over the same range, while the streamed results remain exact integers. Horizons are reported, not romanticized: at the measured rate, $n = 17$ costs 1.7 h, $n = 18$ costs 30.7 h, and $n = 19$ costs 584 h on the same device.

Throughout, every claim is backed by executed, seeded, openly published scripts, and every benchmarked determinant is asserted—in the running code—equal either to an independently computed exact reference or to all competing implementations on the same instance.

2 Related work

2.1 Incremental permutation generation

The naive lexicographic stepper advances by locating the rightmost increasable position and sorting the suffix, $O(n \log n)$ per step. Narayana Pandita's fourteenth-century procedure [1, 2] observes the suffix is always descending, replacing the sort by a reversal; each step still performs a pivot scan, a successor scan, a swap, and a reversal ($O(n)$ worst case, $O(1)$ amortized). SJT [3, 4] and Heap [5] generate each permutation by a single swap—SJT's adjacent transposition additionally flips the sign, making it the classical choice for Leibniz evaluation—but neither is lexicographic. All four are chains in the sense of Section 1.

Table 1: Complete-enumeration Leibniz implementations. $T_{\text{perm}}/T_{\text{sign}}$: per-step costs. "Lex.": lexicographic order. "Sort-free": no sorting or suffix reversal. "Parallel": sublinear-span execution (Section 4).

Method	T_{perm}	T_{sign}	**Total**	**Lex.**	**Sort-free**	**Parallel**
Naive lex. + inversion count	$O(n \log n)$	$O(n^2)$	$O(n^2 \cdot n!)$	Yes	No	No
Narayana [1] + inversion count	$O(n)$	$O(n^2)$	$O(n^2 \cdot n!)$	Yes	No†	No
Narayana + merge-sort sign	$O(n)$	$O(n \log n)$	$O(n \log n \cdot n!)$	Yes	No†	No
Narayana + $O(1)$ parity flip	$O(n)$	$O(1)$	$O(n \cdot n!)$	Yes	No†	No
SJT [3, 4] + sign flip	$O(n)$	$O(1)$	$O(n \cdot n!)$	No	Yes	No
Heap [5] + bookkeeping	$O(1)$ am.	$O(1)$ am.	$O(n!)$	No	Yes	No
This work	$O(n)$ am.	$O(1)$ am.	$O(n \cdot n!)$	**Yes**	**Yes**	**Yes**

†Suffix reversal replaces the sort; per-step pivot scanning, swapping, and reversal remain.

2.2 Rank-based parallel generation

A folklore route to parallel lexicographic generation is unranking: the k-th permutation is decodable independently from the factorial-number-system digits of k [1], so ranks can be distributed across processors. We use unranking gratefully for spot verification (Section 6), but as a bulk generator it is inferior on three counts: the work is $O(n^2 \cdot n!)$ (or $O(n \log n \cdot n!)$ with order-statistic structures) against our $O(n \cdot n!)$; each decode is an independent serial procedure with data-dependent control flow, offering no block structure to a vector unit, whereas our layers are realized by bulk gathers; and signs still require per-permutation inversion counting, which meryemSign eliminates. The streaming scheme of Section 5 can be read as the constructive middle ground: unranking is applied only once per $m!$-line window, to an $O(n)$ prefix, while the window body is produced compositionally.

2.3 Sign computation

Per-permutation inversion counting costs $O(n^2)$ ($O(n \log n)$ via merge sort) [1]. Along a Narayana enumeration the parity change of each step is derivable from the step structure (one swap plus $\lfloor L/2 \rfloor$ reversal transpositions), yielding lexicographically ordered signs at $O(1)$ update cost; we adopt this as the strongest classical baseline.

2.4 Exact determinants

For the determinant *value*, complete enumeration cannot compete asymptotically with elimination: LU decomposition [10] costs $O(n^3)$ in floating point, and Bareiss's fraction-free elimination [9] achieves $O(n^3)$ arithmetic steps *exactly* over the integers. We state this plainly and use it: Bareiss serves as the independent certifier of every large-scale result in this paper. Our claims concern the enumeration problem—which elimination cannot address—with the determinant as its verifiable stress test.

3 The generator

Write S_n for the symmetric group on $\{1, \ldots, n\}$ and $\operatorname{sgn}(\sigma) = (-1)^{\operatorname{inv}(\sigma)}$.

3.1 meryemPer: permutations by block replication and index mapping

Theorem 1. *The following procedure outputs all $n!$ permutations of $\{1, \ldots, n\}$ in lexicographic order:*

1. $a \leftarrow [[1]]$.

2. *For $b = 2, \ldots, n$: let $c = [1, \ldots, b]$ and $d \leftarrow$ empty. For each $f \in c$ in increasing order: let $g = c \setminus \{f\}$ (increasing); for each $h \in a$ in order, append $[f] \circ [\, g[h_1], \ldots, g[h_{b-1}]\,]$ to d. Set $a \leftarrow d$.*

Proof. Induction on n; the base case is immediate. Assume a lists S_{n-1} lexicographically at layer $b = n$, and fix a first element f. The map $h \mapsto (g[h_1], \ldots, g[h_{n-1}])$ arranges the elements of g according to the pattern h; since g is increasing it is a bijection onto the permutations of g and *order-preserving*: if $h < h'$ first differ at position t then $h_t < h'_t$, hence $g[h_t] < g[h'_t]$, so the images first differ at t in the same direction. Thus for fixed f the appended block is exactly the lexicographic listing of permutations beginning with f; as f increases the blocks concatenate into the lexicographic listing of S_n. □

Proposition 2. *meryemPer performs $O(n \cdot n!)$ elementary operations ($\sum_b b \cdot b!$, dominated by the last layer); amortized $O(n)$ per permutation.*

Remark 1 (Compositional, not incremental). There is no predecessor to repair, hence no scan, swap, sort, or reversal anywhere; lexicographic order is a consequence of the order-preserving remapping, not of an optimization applied to a naive stepper. Section 4 shows the contrast is not cosmetic.

3.2 meryemSign: signs by block replication

Theorem 3. *The procedure $b \leftarrow [+1]$; for $e = 1, \ldots, n-1$: $c \leftarrow b$, then for $d = 0, \ldots, e-1$ append $-c$ if d is even else c, outputs $\operatorname{sgn}(\sigma_1), \ldots, \operatorname{sgn}(\sigma_{n!})$ in lexicographic order of σ.*

Proof. Group S_n by first element k; for $\sigma = (k, \tau)$ with standardized suffix $\tau \in S_{n-1}$, the element k contributes exactly $k-1$ inversions and standardization preserves the rest, so $\operatorname{sgn}(\sigma) = (-1)^{k-1} \operatorname{sgn}(\tau)$; standardization is order-preserving on suffixes, so within group k the suffixes run lexicographically over S_{n-1}. The appended block for d corresponds to $k = d + 2$ with factor $(-1)^{d+1}$, matching the negate-when-even rule; induction completes the claim. □

Proposition 4. *Total work $\Theta(n!)$; amortized $O(1)$ per sign. No inversion is ever counted.*

3.3 The determinant as application

Given $\mathbf{A} \in \mathbb{Z}^{n \times n}$, the Leibniz formula $\det \mathbf{A} = \sum_\sigma \operatorname{sgn}(\sigma) \prod_i a_{i,\sigma(i)}$ is evaluated by pairing the two streams; total cost $O(n \cdot n!)$ time (Corollary of Propositions 2–4). The determinant's role in this paper is that of a *certifiable stress test*: unlike raw enumeration, its single-number output can be checked against an independent exact algorithm at any scale (Section 2, Bareiss).

4 Parallel structure

4.1 The factorial tree and its local child rule

The construction of Theorem 1 is a layered tree: layer b holds $b!$ nodes, each node of layer $b-1$ spawning b children. The decisive property is *locality*: the child at global rank r of layer b is determined by $f = \lfloor r/(b-1)! \rfloor$ and $r' = r \bmod (b-1)!$ alone, and equals $[f{+}1]\circ$ (the order-preserving relabeling of the layer-$(b{-}1)$ node at rank r'). No sibling reads another sibling; no shared state exists within a layer. The same holds for meryemSign, where block k is $(-1)^{k-1}$ times the previous list.

Proposition 5. *At cell granularity, meryemPer has work $O(n \cdot n!)$ and span $O(n)$; meryemSign has work $\Theta(n!)$ and span $O(n)$. Both therefore have parallelism $\Theta(n!)$ up to a factor n.*

Proof. Cell (r, j) of layer b is a pure function of at most one cell of layer $b-1$ (namely $g_f[A_{b-1}(r', j-1)]$, with f, r' computable in $O(1)$), so each layer has depth $O(1)$ with unlimited processors and there are $n-1$ layers. Work is Propositions 2–4. □

Observation 6 (Chains are inherently serial)**.** *Every incremental generator of Section 2 computes permutation $r+1$ as a function of the* value *of permutation r. Under this strategy the $n!$ outputs form a dependency chain, so the critical path is $\Omega(n!)$ regardless of processor count and of implementation language. The limitation belongs to the strategy, not to any particular code.*

5 Streaming execution

Materializing the full tree costs $\Theta(n \cdot n!)$ memory (1.2 TB at $n = 14$; 335 TB at $n = 16$). Rank locality removes the need.

5.1 Windows from rank locality

Fix $m < n$ and cache, once, the order-m tables P_m (all $m!$ suffix patterns, Theorem 1) and S_m (their signs, Theorem 3). Global lexicographic rank decomposes as $r = q \cdot m! + s$: the *prefix index* q fixes the first $n-m$ elements (decodable in $O(n)$ via the factorial number system, or streamed in lexicographic order at C speed as the $(n-m)$-arrangements of $\{1, \ldots, n\}$), and s indexes the cached pattern. Window q's $m!$ lines are then

$$\text{perm} = \text{pre} \circ R[\, P_m[s]\,], \qquad \text{sign} = \varepsilon(\text{pre}) \cdot S_m[s],$$

where R is the increasing list of remaining elements and $\varepsilon(\text{pre}) = (-1)^{\text{inv}(\text{pre}) + \sum_i \#\{x \in R : x < \text{pre}_i\}}$ costs $O(n^2)$ once per $m!$ lines. Each window is one bulk gather; working memory is $\Theta(m! \cdot n)$.

5.2 The block-size law

Law 7 (Overflow-safe chunk sums)**.** *Let $\Pi_{\max} = \prod_i \max_j |a_{ij}|$. If m is chosen so that $m! \cdot \Pi_{\max} < 2^{62}$, then every window's partial sum of signed products lies strictly within the int64 range, for every input matrix and every window.*

The law couples the two resources that matter: $m!$ is the vector width offered to the hardware (and a multiple of the 32-lane warp quantum for $m \geq 8$), while the bound guards the arithmetic. Window partials are then reduced in arbitrary-precision integers on the host, so the final result is exact at any order. With entries in $[-3, 3]$, $m = 12$ (window = 479,001,600 lines) satisfies the law through $n = 20$ ($12! \cdot 3^{18} = 1.86 \times 10^{17} < 4.61 \times 10^{18}$); a further consequence is that with $\Pi_{\max} < 2^{31}$ each line's product also fits int32, enabling a native 32-bit multiply path with 64-bit accumulation.

5.3 Prepare/execute pipelining: a measured negative

The window structure invites a producer/consumer pipeline in the style of a processor's fetch–execute overlap: while stream E computes window q's products, stream P builds window $q+1$ into the alternate buffer, with event-ordered ping-pong reuse. We implemented this with dual CUDA streams and report the honest result: on a Tesla T4 the gain was 0.99× at $n = 11$–13—none. Two roofline facts explain it: both stages are memory-bandwidth-bound and share a single DRAM bus, so overlapping them hides nothing; and preparation costs only ∼10% of execution precisely because the gather-based generator is so cheap, leaving little to hide. The finding delimits, rather than damages, the design: the pipeline's claim condition is *disjoint* resources (host/device stages, or multiple devices), and the fused kernel below subsequently reduced preparation to microseconds, mooting single-device overlap entirely.

5.4 The fused kernel

The remaining bandwidth cost was the materialized window itself. It, too, is unnecessary. Because the window body is pre $\circ\, R[P_m[s]]$, a per-window $m \times m$ table $T[j][k] = a_{n-m+j,\, R[k]}$ and the scalar $\pi = \prod_{i<n-m} a_{i,\mathrm{pre}_i}$ suffice: one GPU thread per line computes, entirely in registers,

$$\pi \cdot \prod_{j=1}^{m} T[\,j\,][\,P_m[s][j]\,] \cdot \varepsilon(\mathrm{pre})\, S_m[s],$$

followed by an in-block reduction and one atomic add into the window partial. The only per-line memory traffic is the pattern row and its sign ($m+1$ bytes); storing P_m transposed makes consecutive threads read consecutive bytes (fully coalesced), and T ($\leq 13^2$ words) resides in shared memory. Windows are batched (K per launch, the table array uploaded per batch) so launch overhead amortizes; segments of windows checkpoint the running exact total to disk, making multi-hour runs crash- and preemption-safe, and an optional global rank range splits a run across independent devices—the distributed evaluation promised by rank locality, available as a command-line parameter. The complete kernel is Appendix C.

6 Verification

All checks below were executed in the published scripts; failures abort the run. *Structural.* meryemPer and meryemSign were verified exhaustively—every line and sign, in order—through $n = 9$ against independent enumeration and inversion counting (403,200 lines in the streaming form; 46,233 pairs in the monolithic form through $n = 8$, plus three-way sign agreement through $n = 10$, all 3,628,800 signs). The streaming generator was additionally spot-checked at 150–25 random ranks per order through $n = 16$ against factoradic decoding and direct inversion counting, on CPU and read back from GPU windows. *Cross-implementation.* At every benchmarked instance, all competing implementations (serial, gather-based GPU, fused GPU, both pipeline modes) were asserted to return identical values. *Independent certification.* Every determinant reported in Sections 7.2–7.3, including all trillion-scale results, was asserted equal to the fraction-free Bareiss value [9] computed in arbitrary-precision integers—an algorithm sharing no code, no ordering, and no arithmetic path with the enumeration. LU comparisons additionally matched to floating-point rounding, with the drift reported in Table 5.

7 Experimental evaluation

Conventions. Median of three runs where runtimes permit; fixed seeds (42 benchmark, 7 verification); pseudorandom integer matrices with entries in $[-4, 4]$ for the single-core study (Section 7.1) and $[-3, 3]$ for the streaming/GPU study (required by Law 7 at $m = 12$; both conventions stated with their tables). Single-core: Intel Xeon 2.10 GHz, Python 3.12.3, NumPy 2.4.4. GPU: Tesla T4 (16 GB, CuPy 14.0.1) and NVIDIA B300 SXM6 (268 GB, 148 SMs, CuPy 14.1.1; kernels compiled to PTX at `compute_100` and driver-JITted to `sm_103`). Scripts, seeds, and raw outputs: `https://detsignper.org` and Zenodo.

7.1 Single-core study

Three findings (Tables 2–4). *Signs:* meryemSign is $\sim$100$\times$ faster than inversion counting, 350–400$\times$ faster than merge parity, and 14.8–18.1$\times$ faster than even the $O(1)$-flip stepper; the margin over an equal-asymptotic opponent is granularity—the chain pays interpreter overhead per permutation, the compositional method per factorially growing block. *End-to-end serial:* $\sim$2$\times$ over inversion-count lexicographic evaluation; parity with SJT, including an honest crossover at $n = 10$ (5.79 vs 5.27 s) caused by serial materialization and resolved below. *Generation:* the

Table 2: Isolated sign generation, all $n!$ signs in lexicographic order (seconds, pure Python, entries convention irrelevant). Baselines receive their permutations from C-implemented `itertools` at negligible cost.

n	**meryemSign**	inv-count	merge	Narayana+flip
7	**0.0002**	0.0095	0.0337	0.0024
8	**0.0012**	0.0907	0.2822	0.0189
9	**0.0095**	0.9802	3.8015	0.1727
10	**0.1116**	11.3518	39.0508	1.7376
11	**1.2724**	–	–	18.7991

Table 3: Full serial Leibniz determinant, pure Python, entries $[-4, 4]$ (seconds).

n	Naive+inv	Nar.+inv	SJT+flip	**This work**
6	0.0013	0.0011	0.0008	**0.0005**
7	0.0098	0.0102	0.0061	**0.0036**
8	0.0849	0.0784	0.0483	**0.0338**
9	0.9083	0.8225	0.4820	**0.4291**
10	–	–	5.2665	5.7925

vectorized single-core generator beats its own serial form 45×, the Python chain 20–22×, and the *C-implemented* chain 5.9–6.4×—the empirical content of Observation 6: implementation language cannot rescue a chain. Vectorized meryemSign additionally emits all 479,001,600 signs of S_{12} in 0.187 s.

7.2 Streaming on a free GPU (Tesla T4)

Table 5: the $n = 14$ determinant—87.2 billion signed products—is computed exactly in 194 MB, against the 1.2 TB a materialized run would require: a ∼6,300× working-set reduction on hardware available at no cost. Throughput is essentially flat (0.31–0.37 G/s), the architectural signature of pure $n!$ scaling; the elementwise rate in fact *rises* with n as overheads amortize. The LU column shows floating-point error drifting upward by two orders of magnitude across the same range while the streamed integers remain exact. The dual-stream pipeline of Section 5.3 measured 0.99× here—the reported negative. The $n = 10$ serial crossover of Table 3 is resolved on this same class of instance: the vectorized/streamed realization runs it 15.4× faster than serial SJT, and SJT has no vectorized realization to answer with.

7.3 Fused kernel at scale (NVIDIA B300)

Table 6 is the paper's summit. The fused kernel sustains a *flat* 57.9 Glines/s from $n = 14$ through 16—187× the T4 gather engine and roughly $10^5\times$ the serial chains of Table 3—and sums all 20,922,789,888,000 Leibniz terms of a 16×16 matrix exactly in six minutes, inside a ∼6.3 GB working set (5.7 GB transposed pattern table, 0.48 GB signs, batch buffers) where materialization would demand 335 TB: a working-set reduction above 5×10^4. Every value is certified against Bareiss. The horizon rows are stated as measurements of the future, not attempts: with the budget gates that protected the run, $n = 17$ costs 1.7 h, $n = 18$ costs 30.7 h (or proportionally less across rank-partitioned devices, Section 5.4), and $n = 19$ costs 583.6 h on this device.

8 Discussion

The structural insight. The lexicographic listing of S_n and its sign sequence are block-recursive in S_{n-1}; the compositional view makes the chains' per-step machinery unnecessary rather than

Table 4: Generation of all $n!$ permutations, lexicographic order, one CPU core (seconds).

n	serial	Narayana chain	itertools (C)	**vectorized**
8	0.0446	0.0312	0.0085	**0.0016**
9	0.4980	0.2872	0.0819	**0.0135**
10	6.0391	2.9344	0.8601	**0.1334**
11	–	34.1425	10.2221	**1.7461**

Table 5: Streamed exact determinant on Tesla T4 (gather engine, $m = 10$, entries $[-4, 4]$), with LU float64 reference. Peak GPU memory in MB.

n	lines	time	G/s	peak	LU (ms)	LU rel. err	exact value
11	39,916,800	0.11 s	0.353	173	1.388	3.5×10^{-16}	42199559
12	479,001,600	1.31 s	0.367	180	0.067	1.2×10^{-16}	−7810213458
13	6,227,020,800	18.36 s	0.339	187	0.069	5.6×10^{-15}	−10956741216
14	87,178,291,200	279.4 s	0.312	194	0.110	7.1×10^{-15}	363599628590

merely cheaper, and converts the enumeration from a chain into a tree—the sole source of every parallel result above.

The generator beyond the determinant. Nothing in Sections 3–5 is specific to determinants. meryemPer with the streaming window scheme is, to our knowledge, the first lexicographic permutation generator whose bulk structure vector hardware can consume: applicable wherever complete or rank-partitioned enumeration in canonical order is needed—exhaustive search and verification, test-vector generation, symbolic computation, reproducible distributed enumeration by rank interval. The determinant served as the stress test precisely because Bareiss could certify it; the generator is the contribution that survives outside the test.

Exactness at scale. The block-size law turns overflow safety from a runtime hope into a provable property chosen at configuration time, and the big-integer window reduction makes exactness unconditional. The measured LU drift ($10^{-16} \to 10^{-15}$ over $n = 11$–14) is small but monotone; the enumeration path pays $n!$ work for a result whose correctness is not a function of conditioning.

Limits, stated plainly. For the determinant *value*, Bareiss at $O(n^3)$ exact and LU at $O(n^3)$ float are and will remain unbeatable by any $\Omega(n!)$ method; our claims are about enumeration, which elimination cannot perform. Single-device prepare/execute overlap bought nothing (Section 5.3); its claim condition is disjoint resources, and multi-device rank partitioning—already supported by the implementation—is the natural continuation, together with $n = 17/18$ as questions of compute budget rather than architecture.

9 Conclusion

A compositional, tree-structured alternative to sixty years of chain-based permutation generation: lexicographic, sort-free, sign-complete, span $O(n)$ against $\Omega(n!)$, streaming in $\Theta(m! \cdot n)$ memory under a provable overflow law, exact at any order, and measured from a single Python core to 20.9 trillion certified terms in six minutes on one GPU—with its remaining horizons priced honestly rather than promised.

CRediT authorship contribution statement

Berke Gülmen: Conceptualization, Methodology. **Meryem Gülmen:** Conceptualization, Methodology. **Ömer Gülmen:** Conceptualization, Methodology, Software, Validation, Investigation, Writing – original draft, Writing – review & editing.

Table 6: Fused streaming on one NVIDIA B300 ($m = 12$, window 479,001,600 lines, entries $[-3, 3]$, int32 product path). Every value asserted equal to the independent exact Bareiss computation. ETAs are measured-rate projections, reported in place of runs exceeding the compute budget.

n	lines	time	G/s	exact value	
13	6,227,020,800	0.11 s	54.2	395452572	[= Bareiss]
14	87,178,291,200	1.51 s	57.9	−1107829026	[= Bareiss]
15	1,307,674,368,000	22.59 s	57.9	86895821433	[= Bareiss]
16	20,922,789,888,000	361.4 s	57.9	−28685347455	[= Bareiss]
17	355,687,428,096,000	ETA 102.4 min at measured rate (budget-gated, not run)			
18	6,402,373,705,728,000	ETA 30.7 h at measured rate (budget-gated, not run)			
19	121,645,100,408,832,000	ETA 583.6 h at measured rate (horizon)			

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Data availability

All verification and benchmark scripts, with fixed seeds and complete raw outputs, are openly available at `https://detsignper.org` and archived on Zenodo (Version 4 DOI: 10.5281/zenodo.21709734; prior versions retained in the same record lineage).

Acknowledgements

The writing of this manuscript was assisted by AI tools for LaTeX formatting, structural organization, and exposition. The mathematical content, algorithms, and proofs are the original work of the authors. This research received no external funding.

References

[1] D. E. Knuth, The Art of Computer Programming, Vol. 4A: Combinatorial Algorithms, Part 1, Addison-Wesley, Boston, 2011.

[2] R. Sedgewick, Permutation generation methods, ACM Comput. Surv. 9 (1977) 137–164.

[3] H. F. Trotter, Algorithm 115: Perm, Commun. ACM 5 (1962) 434–435.

[4] S. M. Johnson, Generation of permutations by adjacent transposition, Math. Comput. 17 (1963) 282–285.

[5] B. R. Heap, Permutations by interchanges, Comput. J. 6 (1963) 293–298.

[6] G. E. Blelloch, Programming parallel algorithms, Commun. ACM 39 (1996) 85–97.

[7] J. JáJá, An Introduction to Parallel Algorithms, Addison-Wesley, Reading, 1992.

[8] T. H. Cormen, C. E. Leiserson, R. L. Rivest, C. Stein, Introduction to Algorithms, 3rd ed., MIT Press, Cambridge, 2009.

[9] E. H. Bareiss, Sylvester's identity and multistep integer-preserving Gaussian elimination, Math. Comput. 22 (1968) 565–578.

[10] G. H. Golub, C. F. Van Loan, Matrix Computations, 4th ed., Johns Hopkins University Press, Baltimore, 2013.

[11] R. A. Horn, C. R. Johnson, Matrix Analysis, 2nd ed., Cambridge University Press, Cambridge, 2013.

[12] C. R. Harris et al., Array programming with NumPy, Nature 585 (2020) 357–362.

[13] R. Okuta, Y. Unno, D. Nishino, S. Hido, C. Loomis, CuPy: A NumPy-compatible library for NVIDIA GPU calculations, in: Proc. Workshop on Machine Learning Systems (LearningSys), NeurIPS, 2017.

A Serial reference implementation (Python)

```
def meryemSign(n):                    # all n! signs, lex order
    b = [1]
    for e in range(1, n):
        c = b[:]
        for d in range(e):
            b.extend([-x for x in c] if d % 2 == 0 else c[:])
    return b

def meryemPer(n):                     # all n! permutations, lex order
    a = [[1]]
    for b in range(2, n + 1):
        c = list(range(1, b + 1)); d = []
        for f in c:
            g = [e for e in c if e != f]
            for h in a:
                d.append([f] + [g[i - 1] for i in h])
        a = d
    return a

def berkeDet(M):                      # exact Leibniz determinant
    n = len(M); det = 0
    for s, p in zip(meryemSign(n), meryemPer(n)):
        prod = 1
        for i in range(n):
            prod *= M[i][p[i] - 1]
        det += s * prod
    return det
```

B Vectorized single-core realization (NumPy)

```
import numpy as np, math

def meryemPer_vec(n):                 # layer = one gather per block
    A = np.zeros((1, 1), dtype=np.int8)
    for b in range(2, n + 1):
        prev = A.shape[0]
        out = np.empty((b * prev, b), dtype=np.int8)
        base = np.arange(b, dtype=np.int8)
        for f in range(b):
            g = np.delete(base, f)
            blk = slice(f * prev, (f + 1) * prev)
            out[blk, 0] = f
            out[blk, 1:] = g[A]       # bulk gather
```

```
        A = out
    return A

def meryemSign_vec(n):                # bulk negate/copy per block
    out = np.empty(math.factorial(n), dtype=np.int8)
    out[0] = 1; size = 1
    for e in range(1, n):
        c = out[:size]; pos = size
        for d in range(e):
            blk = out[pos:pos + size]
            if d % 2 == 0: np.negative(c, out=blk)
            else:          blk[:] = c
            pos += size
        size = pos
    return out
```

C Fused streaming kernel (CUDA, int32 product path)

Per line: registers only; per-window $m \times m$ table in shared memory; transposed pattern table for coalesced reads; window partial via block reduction and one atomic add; window partials reduced exactly on the host.

```
extern "C" __global__
void fused32(const signed char* __restrict__ PmT,   // (m, m!) transposed
             const signed char* __restrict__ Sm,    // m! signs
             const int* __restrict__ Mrem,          // K windows * m * m
             const int* __restrict__ prefprod,      // K prefix products
             const int* __restrict__ psign,         // K prefix signs
             long long* partials,
             const long long rows, const int m, const long long q0)
{
    __shared__ int Mt[169];
    __shared__ long long sh[256];
    int kb = blockIdx.y;
    for (int t = threadIdx.x; t < m * m; t += blockDim.x)
        Mt[t] = Mrem[(size_t)kb * m * m + t];
    __syncthreads();
    long long r = blockIdx.x * (long long)blockDim.x + threadIdx.x;
    long long acc = 0;
    if (r < rows) {
        int prod = prefprod[kb];
        for (int j = 0; j < m; ++j)
            prod *= Mt[j * m + PmT[(size_t)j * rows + r]];
        prod *= (int)Sm[r] * psign[kb];
        acc = (long long)prod;
    }
    sh[threadIdx.x] = acc; __syncthreads();
    for (int s = blockDim.x >> 1; s > 0; s >>= 1) {
        if (threadIdx.x < s) sh[threadIdx.x] += sh[threadIdx.x + s];
        __syncthreads();
    }
    if (threadIdx.x == 0)
        atomicAdd((unsigned long long*)&partials[q0 + kb],
                  (unsigned long long)sh[0]);
}
```