Canonical Parallel Reduction

A Fixed Expression Structure for Run-To-Run Consistency

Document:	DxxxxR0
Date:	2026-02-14
Reply-To:	Andrew Drakeford <andreedrakeford@hotmail.com>
Audience:	LEWG, SG6 (Numerics), WG14

Canonical Parallel Reduction

Abstract

C++ today offers two endpoints for reduction:

- `std::accumulate`: a specified left-fold expression, inherently sequential.
- `std::reduce`: scalable, but permits reassociation; for non-associative operations (e.g., floating-point addition), the returned value may vary.

This paper specifies a **canonical reduction expression structure**: for a given input order and topology coordinate (lane count `L`), the expression — its parenthesization and operand order — is unique and fully specified. Implementations are free to schedule evaluation using parallelization, vectorization, or any other strategy, provided the returned value matches that of the specified expression.

The proposal standardizes the expression structure only. API design is deferred.

This proposal generalizes the Standard Library technique used by `std::accumulate`: determinism is obtained by fixing the abstract expression structure, not by constraining execution strategy or floating-point arithmetic (see Appendix D). Bitwise identity of results additionally depends on the floating-point evaluation model (§6).

Reading guide. The normative content of this paper is §4–§5 (~8 pages). Everything else is informative rationale and appendices.

Reader	Read	Skim	Reference as needed
LEWG reviewer	§1, §2, §4, §5, Polls	§3 (design rationale)	Appendices
Implementer	Add Appendix B, N	§3.8 (tree shape rationale)	
Numerical analyst	Add §6, Appendix C, K	§3.8 (throughput data)	

1. The Semantic Gap in C++ Reduction Facilities

C++ specifies two reduction semantics, but there is a gap:

Facility	Expression Specified	Scalable	Semantic Model
`std::accumulate`	✓ (left-fold)	✗	Sequential, specified grouping
`std::reduce`	✗ (unspecified)	✓	Parallel, unspecified grouping
Canonical reduction (this proposal)	✓ (canonical tree)	✓	Parallel, specified grouping

A summary comparison with HPC frameworks and industry practice appears in §7.

`std::accumulate` specifies a left-fold expression and therefore a deterministic result for any given `binary_op`, but it is inherently sequential. `std::reduce` enables scalable execution by permitting reassociation; as a consequence, the abstract expression is not specified, and for non-associative operations the returned value may vary.

This proposal closes that gap by defining a **canonical, parallel-friendly expression structure**: a deterministic lane partitioning determined by the topology coordinate `L` together with a canonical iterative-pairwise tree over lane results, as defined in §4.

Because the expression is fixed, the facility also supports **heterogeneous verification**: the same topology coordinate can be used to reproduce an accelerator-produced result on a CPU by evaluating the identical abstract expression, provided the floating-point evaluation models are aligned.

[*Note:* In this paper, **canonical** refers strictly to the abstract expression structure (parenthesization and operand order). Implementations retain freedom in evaluation schedule; the facility provides a stable, specified baseline topology. —*end note*]

1.1 Why This Paper Now

Each standard cycle without a specified parallel reduction compounds fragmentation. Frameworks like Kokkos, TBB, CUB, and oneMKL each provide their own solutions with different semantics. A standard semantic foundation enables convergence.

Why standardize, not just use a library? Today, you cannot write a generic library component that demands a specified reduction expression from the standard toolkit. Standardizing the semantic definition lets vendors and users align on a common contract.

1.2 A tiny motivating example (informative)

Consider the same 6 floating-point inputs evaluated with the same `+` operator:

```
[1e16, 1, 1, -1e16, 1, 1]
```

Because floating-point addition is not associative, different parenthesizations can legitimately produce different results.

- A **left-to-right fold** (as with `std::accumulate`) groups as `(((((1e16 + 1) + 1) + -1e16) + 1) + 1)` which may yield a different result (e.g., `4` on many IEEE-754 double implementations) because small terms can survive until after the large cancellation.
- A plausible **tree reduction** (as may happen inside `std::reduce`) can group as `((1e16 + 1) + (1 + -1e16)) + (1 + 1)`, in which some `+1` terms may be lost early, yielding a different result (e.g., `2`), depending on evaluation strategy and rounding behavior.

Today, both outcomes are consistent with the Standard because `std::reduce` does not fix the abstract expression (it requires associativity/commutativity to be well-defined). This paper's goal is to define a **single standard-fixed expression** ("canonical") so that the returned value is reproducible within a fixed consistency domain (fixed input order, fixed topology coordinate `L`, and a fixed floating-point evaluation model), while still permitting parallel execution.

Appendix K.3.1 provides a cancellation-heavy dataset in which `std::reduce` exhibits run-to-run variability while the canonical reduction remains stable for fixed `L`, and in which varying `L` intentionally yields different results (confirming that topology selection is semantic, not a hint).

1.3 Determinism via Expression Ownership in Existing Practice

The C++ Standard already achieves deterministic results for reductions by **fixing the abstract expression structure**, not by restricting floating-point arithmetic or execution strategy.

`std::accumulate` is the canonical example. Its specification mandates a left-to-right fold, fully determining the parenthesization and left/right operand order of the reduction expression. As a consequence, implementations are not permitted to reassociate operations, even when the supplied `binary_op` is non-associative. This guarantee is structural, not numerical: the Standard does not promise bitwise identity across platforms or builds, but it does fully specify the abstract expression being evaluated.

This proposal applies the same semantic technique to parallel reduction. Rather than permitting reassociation (as `std::reduce` explicitly does), it standardizes a single canonical reduction expression that admits parallel evaluation. The novelty of this proposal lies in the topology of the expression, not in the mechanism by which determinism is obtained. Appendix D analyzes the `std::accumulate` precedent in detail and explains why the same conformance model applies to the canonical reduction specified in §4.

2. Scope and Non-Goals

This paper proposes **semantics only**. It seeks LEWG validation of the fixed expression structure defined in §4 before committing to API design. This proposal introduces no new requirements on `binary_op` beyond invocability and convertibility, and does not modify existing algorithms.

This is not "just `std::reduce` + an execution policy": `std::reduce` deliberately leaves the abstract evaluation expression unspecified (and requires associativity/commutativity for meaning under parallelization), whereas this paper standardizes a **single fixed canonical expression** that enables opt-in reproducible results across implementations while still permitting parallel execution.

The facility defines the returned value for a chosen topology coordinate (lane count `L`). Execution strategy remains unconstrained (§4).

Reproducibility under this facility is defined relative to a chosen topology coordinate `L`. Different topology selections intentionally define different abstract expressions and may therefore produce different results for non-associative operations. Determinism is guaranteed for a fixed input sequence, fixed topology coordinate `L`, and fixed floating-point evaluation model.

For clarity: §4–§5 are normative and define the semantic contract of the facility. All other sections and all appendices are informative and do not introduce additional requirements. Appendix K records external demonstrator programs (Compiler Explorer links) used to validate the semantics on multiple architectures; they are semantic witnesses, not reference implementations and not part of the proposal.

In scope (this paper):

- Semantic definition of the canonical compute sequence (fixed abstract expression)
- Parameterization by lane count `L`, with a derived span spelling `M` (§9.2)
- Invariance guarantees (§5)
- Rationale for design choices

Deferred to subsequent revision (pending LEWG direction):

- API surface (new algorithm vs execution policy vs both)
- Range-based overloads
- Scheduler/executor integration (sender/receiver-based execution models)

2.1 Opt-in reproducibility and performance trade-off (informative)

This facility is an **opt-in** choice for users who require a stable, specified abstract expression structure (e.g., for debugging, verification, regression testing, or reproducible numerical workflows). It is not intended to replace existing high-throughput facilities. Users who prioritize maximum throughput over expression identity should continue to use `std::reduce` (or domain-specific facilities) where unspecified reassociation is acceptable.

2.2 Traversal and sizing requirements (informative)

Evaluation of the canonical expression requires a well-defined `N` (the number of elements in the input range). The iterative pairwise algorithm is single-pass, but the normative definition (§4) is stated in terms of `K = ceil(N/L)`, so the specification as written requires `N` to be known. No implicit materialization or allocation is performed by the facility. See Appendix L for further discussion of ranges compatibility.

2.3 Exception safety

Exception handling follows the corresponding Standard Library rules for the selected execution mode:

- When evaluated without an execution policy, if `binary_op` throws, the exception is propagated to the caller ([algorithms.general]).
- When evaluated with an execution policy: if execution of `binary_op` exits via an uncaught exception and the policy is one of the standard execution policies, `std::terminate` is called ([algorithms.parallel.exceptions]).

The expression-equivalence (returned-value) guarantee applies only when evaluation completes normally. If `std::terminate` is called under a policy-based evaluation, the state of any outputs and any externally observable side effects is unspecified.

2.4 Complexity

Evaluation of the canonical reduction for an input range of `N` elements performs `O(N)` applications of `binary_op`.

Specifically, the canonical expression contains exactly `N - 1` applications of `binary_op` when `N > 0`; absent-operand positions (§4.2.2) do not induce additional applications.

This matches the work complexity required of `std::reduce`. Only work complexity is normative in this paper. The canonical abstract expression has O(log N) height; this paper does not require implementations to realize that depth without auxiliary storage. No guarantees are made about evaluation depth, degree of parallelism, or auxiliary storage.

[*Note:* The natural shift-reduce evaluation strategy (§4.2.3) maintains a stack of depth O(log K) per lane, where K = ceil(N/L). For L lanes this implies O(L · log(N/L)) intermediate values of type A as working storage. This is modest in practice (e.g., 8 lanes × 30 stack entries for a billion elements) but is not zero. —*end note*]

2.5 Expression identity vs. bitwise identity

This facility guarantees **expression identity**: for fixed input order and topology coordinate, the abstract reduction expression is identical across conforming implementations. Bitwise identity of results additionally depends on the floating-point evaluation model; §6 discusses the distinction in detail.

3. Design Space (Informative)

This section catalogs key design alternatives for a reproducible reduction facility and explains why this proposal chooses a user-selectable, interleaved-lane topology with a standard-defined canonical expression.

The goal is to close the “grouping gap” for parallel reductions by providing a facility that:

1. Specifies a single abstract expression (parenthesization and operand order) for a chosen topology.
2. Achieves scalable depth (O(log N)) suitable for parallel execution.
3. Is topology-stable: the expression is not implicitly determined by thread count or implementation strategy.
4. Remains implementable efficiently on modern hardware (SIMD, multicore, GPU).

The following alternatives are evaluated against these requirements.

3.0 Expression, Algorithm, and Execution

Every reduction computes an *abstract expression*: a parenthesized combination of operands with a defined left/right operand order. This expression exists independently of how it is evaluated in time. In C++, the abstract expression has historically been implicit — specified only indirectly through algorithm wording — and has never been named as a distinct semantic concern.

In practice, three concerns determine the behavior of a reduction:

- **Expression structure** — the abstract computation being performed: grouping, parenthesization, and operand order.
- **Algorithm** — the semantic contract that determines which aspects of the expression are specified or intentionally left unspecified.
- **Execution** — how evaluation of the expression is scheduled: sequentially, in parallel, vectorized, or otherwise.

The C++ Standard Library already relies on this separation, but does not articulate it explicitly. As a result, expression structure is routinely conflated with execution strategy, producing persistent confusion.

Existing facilities through this lens. Seen through this model, existing facilities differ primarily in *expression ownership*, not in parallelism:

- `std::accumulate` specifies a left-to-right fold. The algorithm fully defines the abstract expression, yielding a deterministic result for a given input order and operation.
- `std::reduce` explicitly declines to specify the expression structure. It defines a *generalized sum*, permitting reassociation to enable scalable execution.
- Execution policies operate exclusively on the execution dimension. They constrain scheduling and concurrency, but do not define or refine the abstract expression being evaluated. In particular, even `execution::seq` does not impose a specific grouping or forbid reassociation; it affects *how* an algorithm runs, not *what* expression it computes.

This explains why `std::reduce(execution::seq, ...)` is still permitted to produce different results for non-associative operations: the algorithm has not specified the expression, and the policy does not add semantic guarantees.

Why execution policies cannot carry expression semantics. It is natural to ask whether a canonical reduction could

be expressed as an execution policy rather than a new algorithm. Under the current standard model, execution policies are deliberately non-semantic with respect to the returned value:

1. Policies are designed to *relax* constraints (permitting concurrency, vectorization), not to *add* new semantic guarantees about grouping or parenthesization.
2. Policies compose freely and orthogonally. Encoding topology in a policy would require dominance rules to resolve conflicts between policies that specify different topologies — a semantic hierarchy the standard does not have.
3. `std::reduce` already proves the limitation: `std::reduce(execution::seq, ...)` still has generalized-sum semantics. If `seq` were sufficient to fix the expression, `std::reduce(seq, ...)` would collapse into `std::accumulate` — but the standard explicitly keeps them distinct.
4. A topology parameter is a *semantic* parameter that intentionally changes observable results for non-associative operations. That is fundamentally different from an execution hint.

Encoding expression structure in a policy would therefore require a fundamental change to the execution-policy model. This proposal intentionally avoids that scope.

Consequence. Because expression structure is a semantic concern that execution policies cannot express, and because views and range adaptors can reorder traversal but cannot constrain combination (§3.7), expression ownership must reside in the algorithm. This proposal makes the abstract expression explicit and assigns ownership of it to the algorithm, restoring a clean separation between *what* is computed (expression), *what* is guaranteed (algorithm), and *how* it is executed (execution policy).

This separation also clarifies the relationship to executors (§3.10).

3.1 Why Not Left-Fold?

A strict left-to-right fold has a fixed evaluation order:

```
// Left-fold
T result = init;
for (auto it = first; it != last; ++it)
 result = op(result, *it);
```

This is `std::accumulate`. It provides run-to-run stability for a given input order, but it cannot parallelize effectively because each operation depends on the prior result. The reduction depth is O(N) rather than O(log N), making it unsuitable for scalable parallel execution.

A left-fold therefore solves stability but not scalability, and it already exists in the standard library.

3.2 Why Not Blocked Decomposition?

A common parallelization strategy is blocked decomposition: assign contiguous chunks to workers and reduce each chunk:

```
// Blocked (illustrative)
// Thread 0: E[0..N/4)
// Thread 1: E[N/4..N/2)
// Thread 2: E[N/2..3N/4)
// Thread 3: E[3N/4..N)
```

Blocked decomposition can be efficient, but it does not by itself provide a topology-stable semantic contract. To obtain a fully specified abstract expression, the standard would need to specify:

- the exact chunk boundaries (including handling of non-power-of-two sizes),
- how chunk results are combined (the second-stage reduction tree), and
- how these choices relate to the execution policy and degree of parallelism.

Absent such specification, the resulting expression is naturally coupled to execution strategy (thread count, scheduler, partitioner), which varies across implementations and runs. Fully specifying these details effectively defines a new algorithm and topology — at which point the question becomes: which topology should be standardized?

This proposal selects a topology that is simple to specify canonically and maps well to common hardware structures (see §3.5).

3.3 Why Not a Fixed Standard Constant?

Another approach is to standardize a fixed topology constant. This can appear attractive for simplicity and uniformity, but it ages poorly:

- If the fixed constant is too small, future wider hardware may be underutilized.
- If the fixed constant is too large, current narrow hardware may incur unnecessary work or overhead.
- Any fixed value becomes a standard revision pressure point as hardware evolves.

A fixed constant trades long-term efficiency and flexibility for initial simplicity and becomes an ongoing "default value" debate. This proposal instead makes topology selection an explicit part of the semantic contract (§3.5).

3.4 Why Not Implementation-Defined?

Allowing implementations to choose the reduction topology is precisely the status quo problem. For parallel reductions such as `std::reduce`, the Standard permits reassociation, and therefore permits different abstract expressions across implementations and settings.

An implementation-defined topology does not close the grouping gap; it merely re-labels it. A reproducibility facility must standardize the expression structure for a chosen topology, rather than leaving that structure implementation-defined.

3.5 The proposed design

The facility proposed here is defined by a **family of canonical expressions** parameterized by a user-selected topology coordinate. For a chosen coordinate, the Standard defines a single abstract expression, and the returned value is the result of evaluating that expression (as-if), independent of execution strategy.

Primary topology coordinate. The canonical expression is defined in terms of the **lane count** `L`. For intuition: `L = 1` degenerates to a single canonical tree over the input sequence (no lane interleaving). Larger `L` introduces SIMD/GPU-friendly lane parallelism, but still denotes one Standard-defined canonical expression fully determined by `(N, L)`. The normative definition of the two-stage reduction (lane partitioning, per-lane canonical tree, cross-lane canonical tree) appears in §4.

Why interleaved lanes, not contiguous blocks? A contiguous-block partition (elements `[0, N/L)` to lane 0, `[N/L, 2N/L)` to lane 1, etc.) would also be topology-stable. The interleaved layout is chosen because it maps directly to SIMD register filling: a single aligned vector load of `L` consecutive elements places one element into each lane simultaneously. Contiguous blocks would require gathering from `L` distant memory locations to fill a register. Additionally, interleaving guarantees that all lanes contain the same number of elements (to within one), so all lanes execute the same canonical tree shape — enabling SIMD lockstep execution without per-lane branching. This uniform tree shape across lanes is what makes the two-stage decomposition (§4.4) efficient in practice.

A byte span `M` may be provided as a *derived convenience* by mapping `L = M / sizeof(value_type)`, but because `sizeof(value_type)` is implementation-defined (and may vary across platforms/ABIs), specifying `M` does not, in general, select the same canonical expression across implementations. For layout-stable arithmetic types, the `M` spelling is a convenient way to align topology with SIMD register width; it does not change the semantic definition of the expression (§9.2).

3.6 Industry precedents for constrained computation structure

A common counter-argument is that because bitwise identity across different ISAs (e.g., x86 vs. ARM) cannot be guaranteed by the C++ Standard alone, the library should not attempt to provide run-to-run stability guarantees. This "all-or-nothing" view overlooks existing industry practice.

A widely used mechanism for improving reproducibility is to **constrain computation structure** (topology, kernel choice, or execution path) to remove sources of run-to-run variability introduced by parallel execution. This proposal targets one such source: unspecified reassociation inside standard parallel reductions. Fixing the abstract expression is therefore a necessary building block for reproducible parallel reductions; additional conditions (e.g., floating-point evaluation model and environment constraints) remain outside the scope of this paper.

3.6.1 Examples (informative)

Library	Feature	Mechanism (documented)	Scope (typical)
Intel oneMKL	Conditional Numerical Reproducibility (CNR)	Constrains execution paths / pins to a logical ISA	Reproducibility across specified CPU configurations
NVIDIA CUB	Deterministic reduction variants	Uses a fixed reduction order for deterministic mode	Reproducibility on the same architecture
PyTorch / TensorFlow	Deterministic algorithms flags	Disables nondeterministic kernels / selects deterministic kernels	Reproducible training runs (scope varies)

3.6.2 References (informative)

Intel oneMKL CNR: Intel documents that parallel algorithms can produce different results based on thread counts and ISA, and provides CNR modes to constrain execution paths. See: Intel oneMKL Developer Guide, “Introduction to Conditional Numerical Reproducibility” [IntelCNR].

NVIDIA CUB: NVIDIA distinguishes between “fast” nondeterministic reductions and “deterministic” variants that use a fixed-order reduction. See: NVIDIA CUB API Documentation [NvidiaCUB].

These libraries address reproducibility through different mechanisms and with different scope. This proposal does not claim to replicate their exact guarantees, but draws on the same insight: fixing the computation structure is a useful building block for reproducibility.

3.6.3 Conclusion

By fixing the abstract expression structure, this proposal provides a **necessary** (but not sufficient) condition for reproducibility. Sufficient conditions depend on the program’s floating-point evaluation model and environment. Without this proposal, even a user with strict control over their floating-point environment cannot achieve reproducible parallel results in general, because the standard library itself permits unspecified reassociation in parallel reductions.

3.7 The structural necessity of a new algorithm

A central tenet of this proposal is that run-to-run stability of the returned value is a **property of the evaluation**, not the data source. To achieve both logarithmic scalability and topological determinism, the algorithm itself must “own” the reduction tree. This logic cannot be injected into existing algorithms via a View or a Range adaptor.

3.7.1 Why std::reduce + Views cannot provide this contract

- Views can reorder iteration; they cannot constrain combination. A view may present elements in a deterministic order, but it cannot force the algorithm to evaluate a particular parenthesization.
- `std::reduce` explicitly permits reassociation. Therefore, even with a deterministic view and a fixed input order, `std::reduce` may legally evaluate a different abstract expression (see [numeric.ops.reduce]).

```
// A deterministic view does not make std::reduce deterministic:
auto view = data | views::transform(f); // preserves iteration order
auto r1 = std::reduce(std::execution::par, view.begin(), view.end(), init, op);
// r1 may still use an implementation-chosen reassociation.
```

3.7.2 Consequence: the algorithm must own the expression

To provide the guarantee defined in §4, the reduction facility itself must specify and “own” the abstract expression tree. As established in §3.0, expression structure is a semantic concern that cannot reside in execution policies (which constrain scheduling, not grouping), views (which reorder traversal, not combination), or executors (which determine *how* an expression is evaluated, not *what* it is). The algorithm is the only standard mechanism that can own expression semantics. This is the semantic gap addressed by this proposal.

3.8 Rationale for tree shape choice

Given that a new algorithm must own its expression tree (§3.7.2), the next design question is: which tree construction rule should the Standard specify? Two well-known candidates exist: iterative pairwise (shift-reduce) and recursive

bisection. This subsection records the considerations that inform the choice.

Arguments favoring iterative pairwise (recommended):

1. **Industry alignment:** SIMD and GPU implementations commonly use iterative pairwise because it maps directly to hardware primitives (warp shuffle-down, vector lane pairing). This includes NVIDIA CUB's deterministic reduction modes.
2. **Implementation naturalness:** For implementers familiar with GPU and SIMD programming, iterative pairwise matches the mental model of "pair adjacent lanes, carry the odd one" — the same pattern used in existing high-performance reduction kernels.
3. **Adoption ease:** Libraries that already implement deterministic reductions using iterative pairwise would require no algorithmic changes to conform to this specification. This lowers the barrier to adoption.
4. **Direct SIMD mapping:** The iterative pairwise pattern corresponds directly to shuffle-down operations on GPU warps and SIMD vector lanes, enabling efficient implementation without index remapping.

Arguments favoring recursive bisection:

1. **Specification clarity:** The recursive definition is three lines with no special cases. The iterative definition requires explicit handling of odd-count carry logic.
2. **Tree symmetry:** For non-power-of-two k, recursive bisection produces more balanced subtrees. The "heavier" subtree (with more elements) is always on the right, following the `m = floor(k/2)` split.
3. **Academic precedent:** Recursive bisection corresponds to the "pairwise summation" algorithm analyzed in numerical analysis literature ([Higham2002] §4).

Why the choice matters:

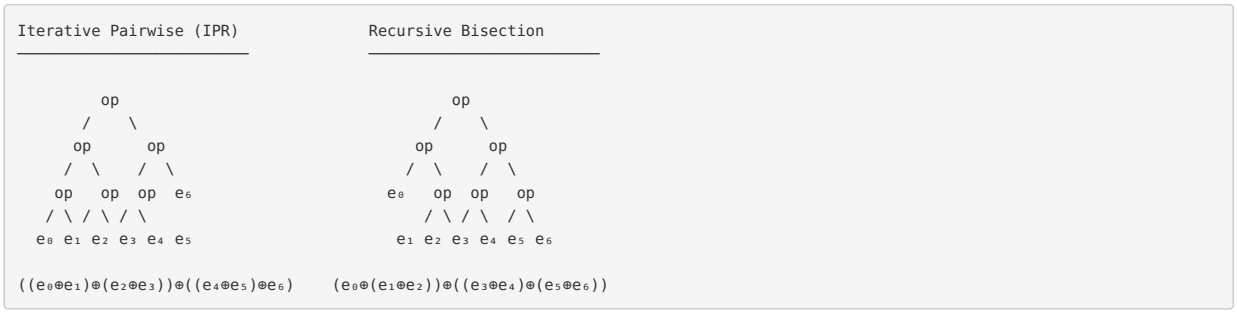
For non-associative operations (e.g., floating-point addition), the two algorithms produce different numerical results for non-power-of-two k. Once standardized, the choice cannot be changed without breaking existing code that depends on specific results.

Why the choice is bounded:

Both algorithms have identical O(log k · ε) error bounds. Both produce identical trees for power-of-two k. The practical impact is limited to: - Non-power-of-two per-lane element counts (when `ceil(N/L)` is not a power of two) - Non-power-of-two lane counts (when L itself is not a power of two) - Cross-lane reduction when N does not evenly divide L

For the common case of power-of-two L and large N, the per-lane trees are dominated by power-of-two cases where both algorithms agree.

Visual comparison (k = 7): For power-of-two k, both algorithms produce identical trees. The difference is visible only for non-power-of-two k. The following side-by-side comparison for k = 7 illustrates the full extent of the difference:



Both trees have depth 3 (= $\lceil \log_2 7 \rceil$). Both perform exactly 6 applications of `binary_op`. The error bounds are identical. The only difference is which element carries: IPR carries e_6 (the last element, locally determined), while recursive bisection splits at `floor(7/2) = 3`, changing the pairing of e_0 (a globally determined property — the pairing of the first element depends on the total count).

This paper specifies iterative pairwise because it aligns with existing practice in high-performance libraries that already provide deterministic reduction modes. This minimizes disruption for implementers and users who have existing workflows built around these libraries.

Streaming evaluation: the fundamental structural difference. The most significant distinction between the two

candidates is not performance or symmetry — it is that iterative pairwise is an *online* algorithm and recursive bisection is a *batch* algorithm.

Recursive bisection's first operation is to compute a midpoint: `mid = N/2`. The entire split tree is determined top-down from the global sequence length. Evaluation cannot begin until N is known, and the split structure depends on a global property of the input.

Iterative pairwise (shift-reduce) processes elements incrementally: each element is shifted onto an O(log N) stack, and reductions are triggered by a branchless bit-test on the running count (`ntz(count)`). The stack captures the complete computation state at any point. N is not needed until termination, when remaining stack entries are collapsed.

This structural property has concrete consequences for how C++ is evolving:

1. **Forward ranges without `size()`:** IPR can begin reducing immediately with a single forward pass. Recursive bisection requires either a counting pass or random access to compute split points.
2. **Chunked executor evaluation:** An executor delivering data in chunks can feed each chunk into the shift-reduce loop. The stack state is the continuation — evaluation can pause and resume at any chunk boundary. Recursive bisection requires the global N before reduction can begin, forcing a synchronization barrier.
3. **P2300 sender/receiver composition:** P2300 (`std::execution`, adopted for C++26) explicitly separates *what* to execute from *where* to execute it, and senders describe composable units of asynchronous work. A streaming reduction using IPR is naturally expressible as a sender that consumes elements as they flow through a pipeline. Recursive bisection requires knowing the complete input before constructing the expression — it cannot compose incrementally.
4. **Heterogeneous and distributed execution:** Data arriving asynchronously from GPU kernels, network reduction, or distributed aggregation can be consumed incrementally by IPR. Recursive bisection must buffer the complete sequence before computation begins.

In a standard moving toward sender/receiver pipelines and composable asynchronous execution, the tree shape that can operate without global knowledge of N is the one structurally aligned with the future of C++ execution. This is not a minor implementation convenience — it is an architectural property that determines whether the canonical expression can participate in streaming pipelines at all.

[*Note:* The current normative definition (§4) is stated in terms of `K = ceil(N/L)`, so the specification as written requires a well-defined N (§2.2). However, the streaming property is inherent to the iterative pairwise algorithm: lane assignment (`i mod L`) is known per-element, and the shift-reduce procedure within each lane builds the tree incrementally without foreknowledge of the lane's element count. N is needed only to determine when to stop and collapse remaining stack entries. Future API revisions may exploit this property to weaken iterator and sizing requirements. —*end note*]

Performance comparison: iterative pairwise vs recursive bisection. Because both trees have identical depth for power-of-two sizes and nearly identical depth otherwise, their throughput is similar on modern hardware. Recursive bisection can be implemented with a direct unrolled mapping for small sub-problems (e.g., a flat switch for $N \leq 32$ eliminates recursive call overhead), and unaligned SIMD loads on contemporary microarchitectures are essentially free when data does not cross a cache line boundary — reducing the alignment advantage that earlier analyses relied on [Dalton2014]. The iterative formulation retains structural advantages (loop-based with a branchless bit-test for reduction, natural streaming order), but these translate to modest rather than dramatic throughput differences against a well-engineered recursive implementation. This approximate throughput equivalence between the two candidates means that performance alone cannot distinguish them — and the decision appropriately falls to the axes where they do differ: industry practice alignment, executor compatibility, and the fact that iterative pairwise is what existing SIMD and GPU reduction libraries already ship. The paper does not claim iterative pairwise is faster than recursive bisection; it claims that, at equivalent performance and identical error bounds, the tree that matches existing practice is the safer standard choice.

Executor compatibility. Once the expression is separated from execution (§3.0), the question becomes: which canonical expression can executors naturally evaluate? Iterative pairwise produces a local, lane-structured expression that maps directly to executor chunking and work-graph execution without requiring the full tree to be materialized. Recursive bisection produces a globally-recursive structure that is harder to realize incrementally. In executor terms, iterative pairwise defines an expression that executors can evaluate without first building the tree.

Standard regret. Once standardized, the tree shape becomes part of the language contract and cannot be changed without breaking code that depends on specific results. The worst consequence of choosing iterative pairwise is not incorrectness or inefficiency — it is commitment. For power-of-two sizes the two candidates are identical; the commitment applies only to non-power-of-two boundary cases, where iterative pairwise matches existing SIMD/GPU practice. This is the expression shape least likely to be regretted as execution hardware evolves.

Upper bound on regret. Regardless of tree shape, any balanced binary reduction has the same $O(\log N \cdot \varepsilon)$ error

bound [Higham2002]. No alternative tree can improve the asymptotic accuracy. On the throughput side, iterative pairwise already achieves 89% of the theoretical peak [Dalton2014]. Even if a superior tree shape were discovered in the future, the maximum possible throughput improvement over IPR is therefore bounded at approximately 11%. The regret is capped: the committee is not choosing between a good answer and an unknown potentially-much-better answer — it is choosing between 89% and at most 100%, with identical error bounds. That is a narrow window in which to find regret.

Measured throughput cost. The practical performance cost of iterative pairwise (shift-reduce) summation has been measured by Dalton, Wang & Blainey [Dalton2014]. Their SIMD-optimized implementation achieves 89% of the throughput of unconstrained naïve summation when data is not resident in L1 cache (86% from L2, 90% streaming from memory), while providing the O(log N · ε) error bound of pairwise summation — and twice the throughput of the best-tuned compensated sums (Kahan-Babuška).

The more telling comparison is against the current deterministic baseline. Today, the only standard facility with a fully specified expression is `std::accumulate`, which is a strict left fold with a loop-carried dependency chain. It cannot utilize SIMD parallelism: on AVX-512 hardware, a left fold occupies 1 of 8 `double` lanes (~12% SIMD utilization) or 1 of 16 `float` lanes (~6%). The canonical iterative pairwise tree, by contrast, is inherently SIMD-friendly — all lanes active, with measured throughput at ~89% of peak. This represents approximately a **7× improvement** in the deterministic reduction path for `double` on AVX-512 (and ~14× for `float`). The question for LEWG is therefore not "what is the cost of owning the expression?" but "how much performance does a specified expression *recover* compared to the only specified expression we have today?" The answer is: nearly all of it.

3.9 Deferral of API surface

This paper acknowledges the importance of a Ranges-first model in modern C++, but the specific API surface (new algorithm, new execution policy, or range-based overload) is deliberately deferred to a subsequent revision. Expression structure is orthogonal to API surface; fixing it first allows API alternatives to be evaluated against a stable semantic baseline. Once LEWG reaches consensus on this semantic foundation, a follow-on revision can propose an API that aligns with modern library patterns.

3.10 Relationship to Executors

The expression/algorithm/execution separation described in §3.0 aligns naturally with the sender/receiver execution model adopted for C++26 (P2300).

P2300 (`std::execution`) addresses the execution concern: *where* work runs, *when* it runs, *how* it is scheduled, and what progress guarantees apply. Its stated design principle — "address the concern of *what* to execute separately from the concern of *where*" — is precisely the separation this proposal formalizes for reduction expressions. Senders and receivers are intentionally agnostic to the abstract computation being performed. In the absence of a specified expression, different schedulers may legitimately induce different groupings for a reduction, leading to scheduler-dependent results. This is not a defect in the execution model; it reflects the fact that the algorithm has not fixed the computation.

By fixing the abstract expression in the algorithm, this proposal provides executors with a stable semantic target:

- The **algorithm** defines *what* computation must be performed.
- The **executor** determines *how* that computation is evaluated.

Different schedulers — CPU thread pool, GPU stream, or distributed sender chain — may execute the canonical expression using different physical strategies, but they are required to produce the same returned value for a fixed expression and floating-point evaluation model.

In this sense, the proposal is orthogonal and complementary to P2300. It does not constrain execution; it enables cross-scheduler consistency by giving the execution model a deterministic expression to execute. The streaming property of iterative pairwise (§3.8) is particularly relevant here: the canonical expression can be evaluated incrementally as data flows through a sender pipeline, without requiring a synchronization barrier to determine N before reduction begins. This also explains why expression semantics could never have resided in execution policies or schedulers: those mechanisms are designed for the execution dimension, and encoding expression structure in them would require conflating the two concerns that this proposal separates.

4. Fixed Expression Structure (Canonical Compute Sequence) [canonical.reduce]

Normative. Sections §4–§5 specify the semantic contract of this proposal. All other sections are informative.

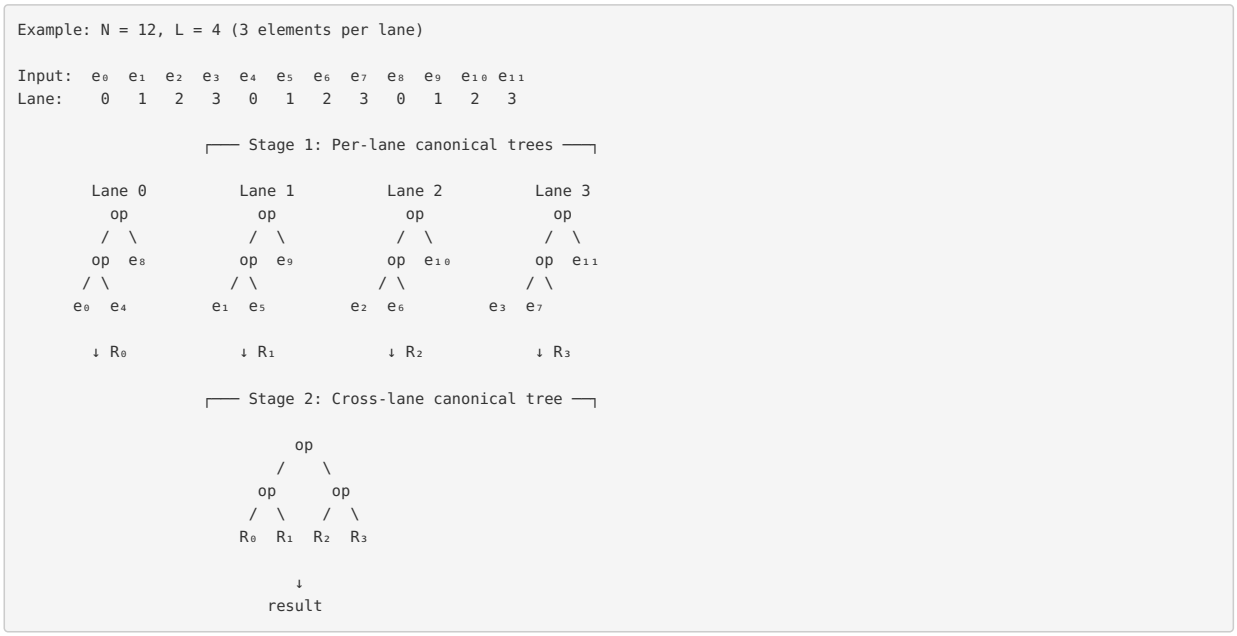
The contract in this section specifies the **returned value only**; the evaluation schedule (including the relative ordering of independent subexpressions) is not specified. For `N > 0`, the standard-fixed abstract expression contains exactly `N - 1` applications of `binary_op`; this paper does not specify which subexpression is evaluated first or how evaluation is scheduled.

This section is the core of the proposal. It defines the fixed abstract expression — the exact grouping and left/right operand order — that determines the returned value for a given input order, `binary_op`, and topology coordinate `L`. A conforming implementation shall produce a result as-if it evaluates this standard-fixed abstract expression; implementations remain free to schedule evaluation using threads, vectorization, work-stealing, GPU kernels, etc., provided the returned value matches.

[*Note:* The semantic contract in §4 is defined in terms of the returned value. It does not specify scheduling, does not guarantee a deterministic number of invocations of `binary_op`, and does not guarantee deterministic side effects. —*end note*]

Overview: Two-stage canonical reduction

The canonical reduction proceeds in two stages over an interleaved lane partition. The input sequence of `N` elements is distributed across `L` lanes by index modulo (element `i` → lane `i mod L`). In Stage 1, each lane independently reduces its elements using a single canonical tree shape. In Stage 2, the `L` lane results are themselves reduced using the same canonical tree rule. Both stages use `CANONICAL_TREE_EVAL` (§4.2.3), ensuring a fully determined expression from input to result.



The lane count `L` is the single topology parameter that determines the shape of the entire computation. When `L = 1`, the two-stage structure collapses to a single canonical tree over the entire input sequence (one lane, no cross-lane reduction). Sections §4.1–§4.4 define each component precisely; §4.3.4 addresses ragged tails when `N` is not a multiple of `L`.

This specification defines the canonical expression structure only; it does not prescribe an evaluation strategy. However, the iterative pairwise formulation can be evaluated efficiently across threads while preserving the canonical tree. When each thread processes a power-of-two-sized chunk of the input, the shift-reduce process within that chunk collapses to a single completed subtree — the minimum possible merge state. Adjacent chunk results can then be combined in index order, recovering the canonical expression regardless of thread count. A detailed parallel realization strategy is described in Appendix N.

4.1 Canonical tree shape [canonical.reduce.tree]

To mirror the style used in the Numerics clauses (e.g. `GENERALIZED_SUM`), this paper introduces definitional functions used solely to specify required parenthesization and left/right operand order. These functions do not propose Standard Library names.

For a fixed input order, lane count `L`, and `binary_op`, the abstract expression (parenthesization and left/right operand order) is uniquely determined by:

1. **Interleaved lanes**: for a lane count `L`, the input positions are partitioned into `L` logical subsequences (lanes) based on `i % L`, preserving input order within each lane (§4.3), followed by a second canonical reduction over lane results (§4.4).
2. **A canonical iterative pairwise tree** defined by a standard-fixed algorithm (§4.2.3).

[*Note:* The iterative pairwise-with-carry evaluation order used by this paper corresponds to the well-known **shift-reduce** (carry-chain / binary-counter stack) formulation of pairwise summation described by Dalton, Wang, and Blainey [Dalton2014]. —*end note*]

A near-balanced tree over a non-power-of-two number of leaves implicitly contains operand positions for which no input element is present. This paper makes that *absence* explicit in the semantic model: when the tree geometry requires combining two operands but one operand is absent, `binary_op` is not applied and the present operand is propagated unchanged (§4.2.2). This avoids padding values and imposes no identity-element requirements on `binary_op`.

4.2 Canonical tree reduction with absent operands [canonical.reduce.tree.absent]

4.2.1 Canonical split rule (pairing count per round) [canonical.reduce.tree.split]

Given a working sequence of length `n`, define the number of pairs formed in each round:

- let `h = floor(n / 2)`.

The split rule `h = floor(n / 2)` is normative and governs the number of pairs formed in each round of the iterative pairwise algorithm (§4.2.3). It does not by itself determine the tree shape; the complete tree is determined by the iterative pairing and carry logic defined in §4.2.3.

4.2.2 Lifted combine on absent operands [canonical.reduce.tree.combine]

Let `maybe<A>` be a conceptual domain with values either `present(a)` (for `a` of type `A`) or `ø` (“absent”). The `maybe<A>` and `ø` notation are purely definitional devices used to specify the handling of non-power-of-two inputs without requiring an identity element; they do not appear in any proposed interface and implementations need not model absence explicitly.

Define the lifted operation `COMBINE(op, u, v)` on `maybe<A>`:

- `COMBINE(op, ø, x) = x`
- `COMBINE(op, x, ø) = x`
- `COMBINE(op, ø, ø) = ø`
- `COMBINE(op, present(x), present(y)) = present( op(x, y) )`

[*Note:* In implementation terms, `COMBINE` is the familiar SIMD tail-masking or epilogue pattern: when an input sequence does not fill the last group evenly, the implementation skips the missing positions rather than fabricating values. The formalism above specifies this behavior precisely without prescribing the implementation technique (predicated lanes, scalar epilogue, masked operations, etc.). —*end note*]

This lifted operation does not require `binary_op` to have an identity element. Absence is a property of the expression geometry (whether an operator application exists), not a property of `binary_op`.

The use of `ø` is a definitional device. For any given `(N, L)`, the locations of absent leaves are fully determined (they occur only in the ragged tail, §4.3.4). Implementations can therefore handle the tail using an epilogue or masking, without introducing per-node conditionals in the main reduction.

4.2.3 Canonical tree evaluation: Iterative Pairwise [canonical.reduce.tree.eval]

This subsection defines the canonical tree-building algorithm using iterative pairwise reduction. The algorithm pairs adjacent elements left-to-right in each round, carrying the odd trailing element to the next round. This corresponds to the shift-reduce summation pattern described by Dalton, Wang, and Blainey [Dalton2014] and matches the shuffle-down pattern used in CUDA CUB and GPU warp reductions (see §3.8 for rationale).

Definition. Define `CANONICAL_TREE_EVAL(op, Y[0..k))`, where `k >= 1` and each `Y[t]` is in `maybe<A>`:

```
CANONICAL_TREE_EVAL(op, Y[0..k)):
    if k == 1:
        return Y[0]

    // Iteratively pair adjacent elements until one remains
```

```
    let W = Y[0..k)  // working sequence (conceptual copy)
    while |W| > 1:
        let n = |W|
        let h = floor(n / 2)
        // Pair elements: W'[i] = COMBINE(op, W[2i], W[2i+1]) for i in [0, h)
        let W' = [ COMBINE(op, W[2*i], W[2*i + 1]) for i in [0, h) ]
        // If n is odd, carry the last element
        if n is odd:
            W' = W' ++ [ W[n-1] ]
        W = W'
    return W[0]
```

When `CANONICAL_TREE_EVAL` returns `ø`, it denotes that no present operands existed in the evaluated subtree.

Shift-reduce state table (k = 8):

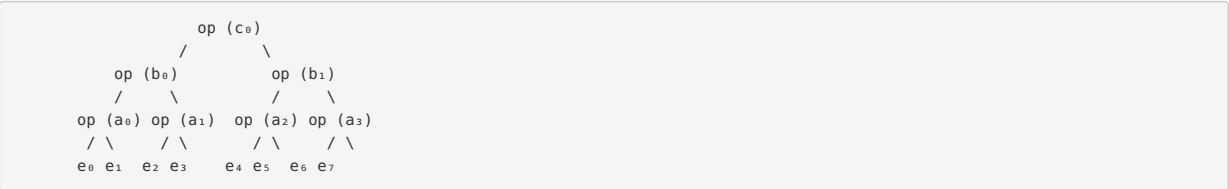
The following table illustrates the iterative pairwise algorithm step by step for `k = 8` elements, adapted from [Dalton2014] Figure 4. Each step either *shifts* (pushes an element onto the stack) or *reduces* (combines the top two stack entries of the same tree level). Lowercase letters denote intermediate results at successively higher levels of the tree: `a` terms are sums of two elements, `b` terms are sums of two `a` terms, and so on.

Sequence	Stack	Operation
$e_0\ e_1\ e_2\ e_3\ e_4\ e_5\ e_6\ e_7$	ø	shift e_0
$e_1\ e_2\ e_3\ e_4\ e_5\ e_6\ e_7$	e_0	shift e_1
$e_2\ e_3\ e_4\ e_5\ e_6\ e_7$	$e_0\ e_1$	reduce $a_0 = op(e_0, e_1)$
$e_2\ e_3\ e_4\ e_5\ e_6\ e_7$	a_0	shift e_2
$e_3\ e_4\ e_5\ e_6\ e_7$	$a_0\ e_2$	shift e_3
$e_4\ e_5\ e_6\ e_7$	$a_0\ e_2\ e_3$	reduce $a_1 = op(e_2, e_3)$
$e_4\ e_5\ e_6\ e_7$	$a_0\ a_1$	reduce $b_0 = op(a_0, a_1)$
$e_4\ e_5\ e_6\ e_7$	b_0	shift e_4
$e_5\ e_6\ e_7$	$b_0\ e_4$	shift e_5
$e_6\ e_7$	$b_0\ e_4\ e_5$	reduce $a_2 = op(e_4, e_5)$
$e_6\ e_7$	$b_0\ a_2$	shift e_6
e_7	$b_0\ a_2\ e_6$	shift e_7
ø	$b_0\ a_2\ e_6\ e_7$	reduce $a_3 = op(e_6, e_7)$
ø	$b_0\ a_2\ a_3$	reduce $b_1 = op(a_2, a_3)$
ø	$b_0\ b_1$	reduce $c_0 = op(b_0, b_1)$
ø	c_0	done

The final result c_0 is the value of the canonical expression. The number of reductions after the n-th shift is determined by the number of trailing zeros in the binary representation of n [Dalton2014].

Tree diagram (k = 8):

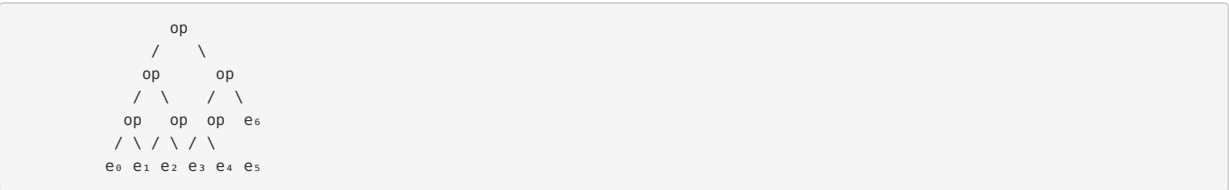
The state table above produces the following canonical expression tree — a perfectly balanced binary tree for power-of-two `k`:



Expression: $((e_0 \oplus e_1) \oplus (e_2 \oplus e_3)) \oplus ((e_4 \oplus e_5) \oplus (e_6 \oplus e_7))$

Tree diagram (k = 7, non-power-of-two):

When `k` is not a power of two, the odd trailing element is carried forward, producing a slightly unbalanced tree. This is where the carry logic in the algorithm definition above determines the canonical shape:



Expression: $((e_0 \oplus e_1) \oplus (e_2 \oplus e_3)) \oplus ((e_4 \oplus e_5) \oplus e_6)$

The left subtree is identical to the `k = 8` case with the last element removed. The carry of e_6 at round 1 (odd `n = 7`) places it as the right child of the right subtree's right branch.

4.2.4 Canonical tree diagrams (informative)

The following diagrams illustrate the fixed abstract expression structure only; they do not imply any particular evaluation order, scheduling, or implementation strategy.

```
Legend (informative)

present(x)  : a present operand holding value x (type A)
ø           : an absent operand position (no input element exists there)
combine(u,v) : lifted combine:
               - combine(ø, x) = x
               - combine(x, ø) = x
               - combine(ø, ø) = ø
               - combine(x, y) = op(x, y) when both present
op(a,b)     : the user-supplied binary_op, applied only when both operands exist
```

Example: absence propagation with k = 5, Y = [X0, X1, X2, ∅, ∅]

```
             COMBINE          COMBINE(op(op(X0,X1), X2), ø) = op(op(X0,X1), X2)
            /       \
         COMBINE      ø       carried from round 2 (odd n=3)
          /    \
        op     COMBINE          COMBINE(X2, ø) = X2
       / \      / \
      X0  X1   X2  ø
```

Result: `op(op(X0, X1), X2)` — two `binary_op` calls from three present elements. The two `ø` positions at different tree levels each induce no application of `binary_op`; the lifted COMBINE rule absorbs them uniformly.

A near-balanced tree is not necessarily full; missing leaves may induce absent operands at internal combine points as the tree reduces. The lifted combine rule handles this uniformly.

4.3 Interleaved topology [canonical.reduce.interleaved]

Let `E[0..N)` denote the input elements in iteration order, and let `X[0..N)` denote the corresponding conceptual terms of the reduction expression (materialization and the reduction state type are defined in §4.6).

4.3.1 Lane partitioning by index modulo [canonical.reduce.interleaved.partition]

For each lane index `j` in `[0, L)`, define:

```
I_j = < i in [0, N) : (i mod L) == j >, ordered by increasing i.
X_j = < X[i] : i in I_j >.
```

This preserves the original input order within each lane (equivalently, `X_j` contains positions `j, j+L, j+2L, ...` that are less than `N`).

4.3.2 Fixed-length lane leaves (single shape per lane) [canonical.reduce.interleaved.leaves]

Define `K = ceil(N / L)` when `N > 0`. (The `N == 0` case is handled by §4.5 and does not form lanes.)

For each lane `j` in `[0, L)`, define a fixed-length conceptual sequence `Y_j[0..K)` of `maybe<A>` leaves:

- for `t` in `[0, K)`:
  - let `i = j + t*L`
  - if `i < N`: `Y_j[t] = present( X[i] )`
  - otherwise: `Y_j[t] = ø`

Thus **all lanes use the same canonical tree shape over `K` leaf positions**. Lanes with fewer than `K` elements simply have trailing `ø` leaves; these do not introduce padding values and do not require identity-element properties of `binary_op`.

[*Note:* Implementations must not pad absent operand positions with a constant value (e.g., `0.0` for addition) unless that value is the identity element for the specific `binary_op` and argument types. The lifted `COMBINE` rules (§4.2.2) define the

correct handling of absent positions for arbitrary `binary_op`. The demonstrators in Appendix K use zero-padding only because they test `std::plus<double>`, for which `0.0` is the identity. —*end note*]

When `N < L`, some lane indices `j` correspond to no input positions: `Y_j[t] == ø` for all `t`, yielding `R_j == ø`. No applications of `binary_op` are induced for such lanes under the lifted `COMBINE` rules.

[*Note:* When `L > N`, `K = 1` and each lane holds at most one element. Stage 1 performs no applications of `binary_op` (each lane result is either a single present value or `ø`). Stage 2 then applies `CANONICAL_TREE_EVAL` over the `L` lane results, of which only `N` are present; the `COMBINE` rules propagate the `L − N` absent entries without invoking `binary_op`. The result is therefore equivalent to `CANONICAL_TREE_EVAL` applied directly to the `N` input elements. In this regime `L` has no observable effect on the returned value. This is intentional: no diagnostic is required, and implementations need not special-case it. —*end note*]

4.3.3 Interleaving layout diagrams (informative)

Example: `N = 10`, `L = 4` ⇒ `K = ceil(10/4) = 3`

```
Input order (i):   0   1   2   3   4   5   6   7   8   9
Elements X[i]:    X0  X1  X2  X3  X4  X5  X6  X7  X8  X9
Lane (i mod L):    0   1   2   3   0   1   2   3   0   1

Lanes preserve input order within each lane:

Lane 0: X0  X4  X8
Lane 1: X1  X5  X9
Lane 2: X2  X6
Lane 3: X3  X7
```

Fixed-length leaves with absence (no padding values):

```
Y_0: [ present(X0), present(X4), present(X8) ]
Y_1: [ present(X1), present(X5), present(X9) ]
Y_2: [ present(X2), present(X6), ø           ]
Y_3: [ present(X3), present(X7), ø           ]
```

4.3.4 Ragged tail handling [canonical.reduce.interleaved.ragged]

When `N` is not a multiple of `L`, the final group of input elements is incomplete: some lanes receive one fewer element than others, producing a ragged trailing edge across the lane partition. The canonical expression handles this uniformly through the `maybe<A>` formalism defined in §4.2.2. Every lane uses the same tree shape over `K = ceil(N/L)` leaf positions, but lanes whose element count falls short of `K` have trailing `ø` leaves. The `COMBINE` rules propagate these absences without invoking `binary_op` and without requiring padding values or identity elements from the caller.

Example: `N = 11`, `L = 4`, so `K = ceil(11/4) = 3`.

The input is distributed across lanes by `i mod L`:

```
Input:   X0  X1  X2  X3 | X4  X5  X6  X7 | X8  X9  X10
Block:   ——— full ——— | ——— full ——— | — ragged ——
```

Lane assignment:

```
Lane 0:  X0,  X4,  X8       (3 elements — full)
Lane 1:  X1,  X5,  X9       (3 elements — full)
Lane 2:  X2,  X6,  X10      (3 elements — full)
Lane 3:  X3,  X7,  ø        (2 elements + 1 absent)
```

All four lanes evaluate the same canonical tree shape over `K = 3` leaf positions. For lanes 0–2, every leaf is present and the tree evaluates normally. For lane 3, the tree encounters an absent leaf:

```
     COMBINE
      /    \
    op      ø
   /   \
  X3    X7
```

`COMBINE(op(X3, X7), ø) = op(X3, X7)` — no `binary_op` application occurs for the absent position. The result is

identical to reducing only the present elements `[X3, X7]`.

This mechanism generalizes to any `(N, L)` pair. The number of ragged lanes is `L - (N mod L)` when `N mod L ≠ 0`; these lanes each have exactly one trailing `⌀`. The remaining `N mod L` lanes have all `K` positions present. When `N` is a multiple of `L`, no lanes are ragged and no `⌀` entries arise.

In implementation terms, the ragged tail corresponds to the familiar SIMD epilogue or tail-masking pattern: the final group of elements is narrower than the full lane width, and the implementation must avoid reading or combining nonexistent data. The `maybe<A>` formalism specifies the required behavior without prescribing the implementation technique (masking, scalar epilogue, predicated lanes, etc.).

4.4 Two-stage reduction semantics [canonical.reduce.twostage]

Let `L >= 1` be the lane count and let `op` denote the supplied `binary_op`.

4.4.1 Stage 1 — Per-lane canonical reduction (single tree shape) [canonical.reduce.twostage.perlane]

For each lane index `j` in `[0, L)`, define:

```
R_j = CANONICAL_TREE_EVAL(op, Y_j)  // returns maybe<A>
```

4.4.2 Stage 2 — Canonical reduction over lane results (in increasing lane index) [canonical.reduce.twostage.crosslane]

Define a conceptual sequence `Z[0..L)` by:

- `Z[j] = R_j` for `j` in `[0, L)`.

Then define:

```
R_all = CANONICAL_TREE_EVAL(op, Z)  // returns maybe<A>
```

When `N > 0`, at least one lane contains a present operand, therefore `R_all` is `present(r)` for some `r` of type `A`. The interleaved reduction result is that `r`.

Therefore, the overall expression is uniquely determined by the canonical tree rule applied first within each lane and then across lanes in increasing lane index order. When `L = 1`, there is a single lane containing all `N` elements; Stage 2 receives one input and returns it unchanged, so the result is simply `CANONICAL_TREE_EVAL(op, Y_0)`.

Summary definition. For convenience, define the composite definitional function:

```
CANONICAL_INTERLEAVED_REDUCE(L, op, X[0..N)):
    Partition X into L lanes by index modulo (§4.3.1).
    Form fixed-length leaf sequences Y_j[0..K) for each lane j (§4.3.2).
    For each lane j: R_j = CANONICAL_TREE_EVAL(op, Y_j).
    Form Z[0..L) where Z[j] = R_j.
    Return CANONICAL_TREE_EVAL(op, Z).
```

When `N == 0`, `CANONICAL_INTERLEAVED_REDUCE` is not invoked; the result is determined by §4.5.

4.4.3 Two-stage diagrams (informative)

Stage 1 summary (`N = 10`, `L = 4`, `K = 3`; all lanes use the same shape; `⌀` propagates):

```
Y_0 = [X0, X4, X8] --(canonical tree k=3)--> R_0
Y_1 = [X1, X5, X9] --(canonical tree k=3)--> R_1
Y_2 = [X2, X6, ⌀ ] --(canonical tree k=3)--> R_2
Y_3 = [X3, X7, ⌀ ] --(canonical tree k=3)--> R_3
```

Stage 2 (cross-lane canonical reduction; same rules apply):

```
Example: L = 4, Z = [R0, R1, R2, R3]

       combine
       /     \
    combine   combine
    /    \    /    \
```

```
   R0    R1  R2    R3
```

Conceptual completeness: the same lifted rule handles absence in Stage 2:

```
Example: Z = [ R0, ø, R2, ø ]

       combine
       /     \
    combine   combine
    /    \    /    \
   R0    ø   R2    ø

combine(R0, ø) = R0
combine(R2, ø) = R2
combine(R0, R2) = op(R0, R2)
```

4.4.4 Equivalence to reducing only present terms (informative)

For any lane `j`, `CANONICAL_TREE_EVAL(op, Y_j)` evaluates the same abstract expression as applying the canonical split rule (§4.2.1) to the subsequence `X_j` containing only present terms, with the understanding that absent operand positions do not create `binary_op` applications. The explicit absence notation does not affect the returned value; it makes the "absent operand" behavior precise and enables a single tree shape for all lanes.

4.5 Integration of an initial value (if provided) [canonical.reduce.init]

If an initial value `init` is provided, the abstract expression is:

- If `N == 0`: return `init`.
- Otherwise:
  - let `R =` the value extracted from `R_all = CANONICAL_TREE_EVAL(op, Z)` in §4.4.2
  - let `I` be a value of type `A` initialized from `init` (where `A` is defined in §4.6)
  - return `op(I, R)`.

The placement of `init` is normative. In particular, `init` is not interleaved into lanes and does not participate in the canonical tree expression. Combining `init` with the tree result in a single final application of `binary_op` ensures that the canonical tree shape is independent of whether an initial value is provided. The left-operand placement is consistent with existing fold-style conventions; because this proposal does not require commutativity of `binary_op`, the position of `init` is specified. In particular, `std::accumulate` places `init` as the left operand at every step (`op(op(init, x₀), x₁)...`); this proposal preserves that convention so that non-commutative operations produce consistent results when migrating from `std::accumulate` to the canonical reduction.

[*Note:* Whether a convenience form without an explicit `init` is provided, and what default it uses, is an API decision deferred to a future revision. *—end note*]

```
With init (conceptual):

Result = op( init, CANONICAL_INTERLEAVED_REDUCE(L, op, X[0..N)) )

init is not interleaved into lanes and is combined once with the overall result.
```

Informative contrast:

```
std::accumulate:
  (((init op X0) op X1) op X2) ... op XN-1

This proposal:
  init op ( fixed canonical tree expression over X0..XN-1 )
```

4.6 Materialization of conceptual terms and reduction state (introducing V and A) [canonical.reduce.types]

The preceding sections (§4.1–§4.4) define the canonical expression structure over abstract sequences. This section specifies the type rules that bridge the abstract expression to C++ evaluation.

Let:

- `V` be the value type of the input sequence elements,

- `A` be the reduction state type:
  - if an initial value `init` of type T is provided, `A = remove_cvref_t<T>`;
  - otherwise `A = V`.

Define the conceptual term sequence `X[0..N)` of type A by converting each input element:

`X[i] = static_cast<A>(E[i])` for `i in [0, N)`.

All applications of `binary_op` within the definitional functions in §4 operate on values of type A.

Constraints: Let `A` be the reduction state type defined above.

When an initial value `init` is provided, the initialization `A{init}` shall be well-formed.

- Each conceptual term `X[i]` is formed by conversion to `A` as specified above.
- If an initial value `init` is provided, it is materialized as a value `I` of type `A` initialized from `init` and participates in the expression as `op(I, R)` per §4.5.
- `binary_op` shall be invocable with two arguments of type `A`, and the result shall be convertible to `A`.

[*Note:* How `V` is derived from the input — whether as `iter_value_t<InputIt>` for an iterator-pair interface, `range_value_t<R>` for a range interface, or otherwise — is an API decision deferred to a future revision. The semantic contract requires only that `V` is well-defined and that the conversion `static_cast<A>(E[i])` is well-formed. Proxy reference types (e.g., `std::vector<bool>::reference`) and their interaction with `V` are likewise API-level concerns. — *end note*]

5. Invariance Properties [canonical.reduce.invariance]

This proposal does **not** impose associativity or commutativity requirements on `binary_op`. Instead of permitting implementations to reassociate or reorder (which can make results unspecified for non-associative operations), this proposal defines a single canonical abstract expression for fixed `(N, L)`. Determinism is obtained by fixing the expression, not by restricting `binary_op`.

See §2.3 for exception/termination behavior; in particular, when `std::terminate` is called under a policy-based evaluation, the state of outputs and any externally observable side effects is unspecified.

See §2.4 for complexity; guarantees are intentionally limited to work complexity, consistent with `std::reduce`.

For a chosen **topology coordinate** (lane count `L`), the fixed expression structure provides:

Topological determinism: For fixed input order, `binary_op`, lane count `L`, and `N`, the abstract expression (grouping and left/right operand order) is fully specified by §4. It does not depend on implementation choices, SIMD width, thread count, or scheduling decisions.

Layout invariance: Results are independent of memory alignment and physical placement, given the same input sequence as observed through the iterator range.

Execution independence: Implementations may evaluate independent subtrees in any order or concurrently. Only the grouping is specified, not the schedule.

Cross-invocation reproducibility: Given the same topology coordinate `L`, input sequence, `binary_op`, and floating-point evaluation model, the returned value is stable across invocations (it is the value of the same specified expression under the same evaluation model).

Scope of guarantee (returned value): The run-to-run stability guarantee applies to the **returned value** of the reduction. If `binary_op` performs externally observable side effects, the order and interleaving of those side effects is not specified by this paper and may vary between invocations.

Constraints on `binary_op`: Let `A` be the reduction state type (§4.6). The only requirements on `binary_op` are invocability with two arguments of type `A` and convertibility of the result to `A`. No associativity, commutativity, or identity-element requirements are imposed. Because the canonical expression is fixed for a given `(N, L)`, determinism is obtained by fixing the expression structure, not by restricting `binary_op`.

The remaining requirements on iterators, value types, and side effects match those of the corresponding `std::reduce` facility ([numeric.ops.reduce]):

- When evaluated without an execution policy, `binary_op` shall not invalidate iterators or subranges, nor modify elements in the input range.

- When evaluated with an execution policy, `binary_op` is an element access function subject to the requirements in [algorithms.parallel.exec].

When evaluated without an execution policy, `binary_op` is invoked as part of a normal library algorithm call; this paper does not require concurrent evaluation. When evaluated with an execution policy, the requirements of [algorithms.parallel.exec] additionally apply.

The run-to-run stability guarantee applies to the returned value when `binary_op` is functionally deterministic — that is, when it returns the same result for the same operand values. If `binary_op` reads mutable global state, uses random number generation, or is otherwise non-deterministic, the returned value may vary even with fixed topology coordinate and input.

[*Note:* Functional determinism of `binary_op` is not a formal requirement (the standard cannot enforce functional purity), but an observation about when the stability guarantee is meaningful. —*end note*]

Cross-platform reproducibility requires users to ensure an identical topology coordinate `L` and an equivalent floating-point evaluation model (§2.5, §6).

6. Floating-Point Considerations (Informative)

This section discusses what is and is not guaranteed about floating-point results under the canonical expression defined in §4.

Terminology: This paper uses **floating-point evaluation model** to mean the combination of the program's runtime floating-point environment (e.g., `<cfenv>` rounding mode) and the translation/target choices that affect how floating-point expressions are evaluated (e.g., contraction/FMA, excess precision, subnormal handling, fast-math).

What is specified: For a given topology coordinate `L`, input sequence, `binary_op`, and `init`, the result is the value obtained by evaluating the canonical compute sequence defined in §4, in the floating-point evaluation model in effect for the program.

What this enables: By removing library-permitted reassociation, repeated executions of the same program under a stable evaluation model can obtain the same result independent of thread count, scheduling, or SIMD width.

What it does not attempt to specify: Cross-architecture bitwise identity is not a goal of this paper. Users who require bitwise identity must additionally control the relevant evaluation-model factors and ensure that `sizeof(V)` (and thus lane count) is stable across the intended platforms.

Relationship to P3375 (Reproducible floating-point results): Davidson's P3375 [P3375R2] proposes a `strict_float` type that specifies sufficient conformance with ISO/IEC 60559:2020 to guarantee reproducible floating-point arithmetic across implementations. This proposal and P3375 are complementary: this paper fixes the *expression structure* (parenthesization and operand order) of a parallel reduction, while P3375 addresses the *evaluation model* (rounding, contraction, intermediate precision) for individual operations. Together, they would provide both necessary conditions for cross-platform bitwise reproducibility of parallel reductions. Neither paper alone is sufficient.

Relationship to `std::simd` (P1928): This proposal is orthogonal to `std::simd`. `std::simd::reduce` performs a horizontal reduction within a single SIMD value with unspecified order; this facility defines a deterministic expression structure over an arbitrary-length input range. An implementation may use `std::simd` operations internally, but the semantic contract does not depend on `std::simd`.

The C++ `<cfenv>` floating-point environment covers rounding mode and exception flags; many other factors that affect floating-point results (such as contraction/FMA and intermediate precision) are translation- or target-dependent and are not fully specified by the C++ abstract machine. This proposal therefore guarantees expression identity, not universal bitwise identity.

7. Relationship to Existing Facilities (Informative)

This paper specifies a canonical expression structure for parallel reduction. The goal is to complete the reduction "semantic spectrum" in the Standard Library: from specified but sequential, to parallel but unspecified, to parallel and specified.

Facility	Parallel	Expression specified	Notes
`std::accumulate`	No	Yes (left-fold)	Fully specified; sequential
`std::reduce`	Yes	No (generalized sum)	Unspecified grouping; results may vary for non-associative ops
HPC frameworks (Kokkos, etc.)	Yes	No	Strategy-dependent grouping; FP results may vary
oneTBB `parallel_reduce`	Yes	No	Join order varies; scheduler-dependent results
Canonical reduction (this proposal)	Yes	Yes (§4 tree)	Fixed parenthesization for chosen topology coordinate L; free scheduling

What this proposal adds: a standard-specified expression for parallel reduction, closing the third cell in the table above.

[*Note:* This paper uses `canonical_reduce_lanes<L>(...)` as the primary illustrative spelling, reflecting that lane count `L` is the semantic topology coordinate. For layout-stable numeric types, an API may additionally provide a span spelling `canonical_reduce<M>(...)` as a convenience (where `L = M / sizeof(V)` when well-formed; see §9.2). No Standard Library API is proposed in this paper. —*end note*]

8. Motivation and Use Cases (Informative)

This proposal is motivated by workloads where run-to-run stability matters, but existing parallel reductions are intentionally free to choose an evaluation order (and thus may vary with scheduling). Typical use cases include:

- **CI regression testing:** A reduction that is stable across runs eliminates intermittent test failures and enables "golden value" comparisons.
- **Debugging and bisection:** A stable result makes it practical to reproduce and minimize numerical regressions without chasing schedule-dependent drift.
- **Auditability / reproducible analytics:** Regulatory or scientific workflows often require re-running computations and obtaining the same result within a defined environment.
- **Large-scale simulation and ML training:** Stable aggregation of large sums (e.g., gradients, risk factors) improves repeatability of model training and scenario analysis.
- **Heterogeneous execution:** A single semantic contract that can be implemented on CPU and GPU enables consistent verification, even when the execution strategy differs.

Detailed examples (with code) are collected in Appendix M.

9. API Design Space (Informative)

This section sketches possible directions for exposing the canonical expression defined in §4. The intent is to build consensus on the semantic contract before committing to an API surface.

A key design point is that the **semantic topology parameter is lane count L**. A byte span M is a numerics convenience coordinate that derives `L = M / sizeof(V)` when well-formed (§9.2). An API may choose to expose one or both coordinates.

Two primary approaches exist.

9.1 New Algorithm Approach (Illustrative)

Expose the semantics as a new algorithm (illustrative spelling only):

```
      // Illustrative only: name/signature not proposed in this paper

      // Lane-based topology (portable across ABIs for a fixed L)
      template <size_t L, class InputIt, class T, class BinaryOp>
      constexpr T canonical_reduce_lanes(InputIt first, InputIt last, T init, BinaryOp op);

      // Span-based topology (numerics convenience; derives L from sizeof(V))
      template <size_t M, class InputIt, class T, class BinaryOp>
      constexpr T canonical_reduce(InputIt first, InputIt last, T init, BinaryOp op);
```

Rationale: The choice of topology affects observable results for non-associative operations (e.g., floating-point addition). This argues for an API whose contract explicitly includes the topology coordinate, rather than treating topology as an implementation detail.

9.2 Span Convenience: Deriving L from a Byte Span M (Informative)

Some environments prefer selecting topology using a byte-based span coordinate `M` rather than specifying the lane count `L` directly. Such a coordinate is derived only and does not change the semantic definition in §4, which is defined entirely in terms of `L`.

Let `value_type` denote `iter_value_t<InputIt>`. When a span coordinate `M` is used and is well-formed for `value_type`, derive:

- `L = M / sizeof(value_type)`.

Interpret the span spelling using the same definitional function name:

`CANONICAL_INTERLEAVED_REDUCE(M, value_type, op, X) = CANONICAL_INTERLEAVED_REDUCE(L, op, X)`, with `L` as above.

M is well-formed for `value_type` only when:

- `M >= sizeof(value_type)`, and
- `M % sizeof(value_type) == 0`.

A future API should reject invalid M rather than silently rounding to a different L.

[*Note:* While the span spelling `M` provides convenient alignment with SIMD register widths for layout-stable arithmetic types, users requiring cross-platform expression identity (e.g., for UDTs or heterogeneous verification) must specify topology via lane count `L`. The mapping `L = M / sizeof(value_type)` is platform-dependent when `sizeof(value_type)` varies across targets. *—end note*]

Mapping the span spelling M to lanes L (informative)

The span spelling selects topology by deriving `L = M / sizeof(value_type)` when well-formed.

```
Example: M = 64 bytes

Case A: value_type = float  (sizeof(float) = 4) => L = 64 / 4 = 16 lanes
Case B: value_type = double (sizeof(double) = 8) => L = 64 / 8 = 8 lanes

Interleaving rule (semantic): element i belongs to lane (i % L).

For L = 8 (e.g. double, M=64):
Input indices:    0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 ...
Lane index (i%8): 0 1 2 3 4 5 6 7 0 1  2  3  4  5  6  7 ...

Lanes (preserving original order within each lane):
  lane 0: X[0], X[8], X[16], ...
  lane 1: X[1], X[9], X[17], ...
  lane 2: X[2], X[10], X[18], ...
  ...
  lane 7: X[7], X[15], X[23], ...
```

When topology is selected via `M`, expression identity across platforms additionally depends on `sizeof(value_type)` being stable within the intended reproducibility domain. Users who require topology identity independent of representation should specify `L` directly.

Convenience spelling example (informative):

```
// Primary spelling: select topology via lane count L
auto r_lanes = canonical_reduce_lanes<8>(first, last, init, op); // L = 8
// Convenience spelling: select topology via a byte span M (numeric domains)
// For value_type = double, sizeof(double) == 8, so M = 64 derives L = 8:
auto r_span = canonical_reduce<64>(first, last, init, op); // M = 64 bytes
```

9.3 Execution Policy Approach (Illustrative)

Expose the semantics as a new execution policy (illustrative spelling only):

```
// Illustrative only: policy type/spelling not proposed in this paper
template <size_t M>
struct canonical_policy { /* ... */ };
```

This approach integrates naturally with the existing parallel algorithms vocabulary. However, as established in §3.0, execution policies in the current standard are designed to constrain *scheduling*, not *expression structure*. Encoding topology in a policy would require the policy to carry semantic guarantees about the returned value — a role that policies do not currently play. It also raises unresolved questions about policy composition: what happens when two policies specify conflicting topologies, or when a topology-carrying policy composes with one that permits reassociation?

9.4 Trade-offs (Informative)

The expression/algorithm/execution analysis in §3.0 informs this trade-off:

- A dedicated algorithm places expression ownership where the standard already locates semantic contracts: in the algorithm. The topology coordinate is explicit, composition questions do not arise, and the facility can be taught as "this algorithm computes *this* expression" — parallel to how `std::accumulate` computes a left fold.
- A policy-based approach may reduce algorithm surface area but conflates expression and execution — the very conflation that §3.0 identifies as the source of existing confusion around `std::reduce` and `execution::seq`. It would also require new rules for policy dominance and semantic interaction that do not exist in the current standard.

9.5 Naming Considerations (Informative)

This paper uses `canonical_reduce_lanes<L>(...)` as the primary illustrative spelling, reflecting that lane count `L` is the semantic topology coordinate. A span spelling `canonical_reduce<M>(...)` may also appear as a numerics convenience (where `L = M / sizeof(V)` when well-formed). Final naming should communicate that:

- the return value is defined by a specified abstract expression; and
- the topology coordinate is part of the semantic contract.

9.6 Topology Defaults and Named Presets (Informative)

If an eventual API offers a no-argument default topology selection, it should avoid "silent semantic drift" across targets or toolchain versions.

In this paper, a **topology preset** (also referred to as a **span preset**) is a named, standard-fixed constant used to select the topology coordinate—typically the interleave span M (bytes), and thereby the derived lane count L for layout-stable numeric types—in a way that remains stable across targets and toolchain versions.

Option	Example spelling	Trade-off
No default	`canonical_reduce<M>(...)` required	Most explicit; no surprises; but higher user burden
Implementation-defined default	`canonical_reduce(...)` selects an implementation-defined topology	Easy to teach; can silently change returned values across targets/versions
Standard-fixed named presets	`canonical_reduce<std::canonical_span_small>(...)`	Readable; coordinated semantics; stable across targets/versions
Explicit literal	`canonical_reduce_lanes<128>(...)`	Maximum control; most verbose; requires users to pick a number

A committee-robust approach is to provide **standard-fixed named presets** for the span coordinate, and (if a default is adopted for numeric types) define it in terms of such a preset constant.

Illustrative named presets (standard-fixed):

```
namespace std {
 // Named semantic presets (bytes)
 inline constexpr size_t canonical_span_small = 128; // baseline coordination preset
 inline constexpr size_t canonical_span_large = 1024; // wide coordination preset

 // The golden reference span for a given V (derives L == 1)
 template<class V>
 inline constexpr size_t canonical_span_max_portability = sizeof(V);

 // Evolution rule: existing preset values never change.
 // Future standards may add new presets under new names.
}

// This paper does not propose a global default topology.
// A follow-on API paper may propose a default in terms of a named preset.
```

[*Note:* Appendix J provides an indicative “straw-man” API that uses these standard-fixed named presets (Option 3) without committing the committee to a final spelling or placement. *—end note*]

For types where `sizeof(V)` is not stable across the intended reproducibility domain, users requiring cross-platform topology identity should select topology by lane count L (or enforce representation as part of the contract).

Practical topology selection guidance (informative)

The semantic coordinate is `L`. When using the span convenience `M`, `L = M / sizeof(value_type)`. Performance is often improved when `L` aligns with the target’s preferred execution granularity, but the Standard does not prescribe hardware behavior.

- **CI/CD and cross-platform verification baseline:** choose `L = 1` (equivalently, `M = sizeof(value_type)`), yielding a single global canonical tree.
- **Typical CPU deployment:** choose `L` matching the target SIMD lane count (e.g., `L = 4` for AVX2/double, `L = 8` for AVX-512/double).
- **GPU warp-level consistency:** choose `L = warp_width` (e.g., `L = 32` on NVIDIA GPUs).

Use case	L (double)	L (float)	M (bytes)	Notes
Golden reference	1	1	`sizeof(V)`	Single tree; for debugging/golden values
Portability baseline	2	4	16	SSE/NEON width
AVX / AVX2	4	8	32	Desktop/server AVX2
AVX-512	8	16	64	AVX-512 servers
CUDA warp (double)	32	—	256	32-thread warp

What LEWG is being asked to agree to (this paper)

We seek direction on the semantic contract and tree shape; API surface is explicitly deferred.

- We are **not** approving an API.
- Authors recommend selecting **iterative pairwise** in Poll 2A to align the canonical shape with common SIMD/GPU reduction practice.
- We are approving the **fixed expression contract in §4** (canonical compute sequence) as a semantic building block: implementations may schedule using threads/SIMD/work-stealing/GPU kernels, but must return the value **as-if evaluating the standard-fixed expression**.
- We intend the canonical expression to have **O(N)** applications of `binary_op` (accounting for carry/propagation rules) and **O(log N)** depth for the canonical shape; the semantic definition imposes **no workspace requirement** (implementations may use workspace).
- A future revision will return with an **API surface** and **constraints/range categories** aligned with `std::reduce` and execution-policy requirements.

Polls for LEWG Direction (this paper)

Vote categories for all Favor/Against polls: **Strongly Favor / Weakly Favor / Neutral / Weakly Against / Strongly Against**

Poll 1 — Semantics-first scope

Question: We agree this paper proposes semantics only, and we want LEWG to validate the fixed expression structure before committing to API design.

Vote: Strongly Favor / Weakly Favor / Neutral / Weakly Against / Strongly Against

Poll 2 — Core semantic contract

Question: We agree the canonical expression defined in §4 is a suitable semantic contract: implementations may execute using threads/SIMD/work-stealing/GPU kernels, but must produce results as-if evaluating the canonical compute sequence defined in §4. (Approval of this poll validates the semantic contract only and does not approve an API.)

Vote: Strongly Favor / Weakly Favor / Neutral / Weakly Against / Strongly Against

Poll 2A — Canonical tree shape

Question: We agree that iterative pairwise reduction with carry (§4.2.3) is an acceptable canonical tree shape for this proposal.

Vote: Strongly Favor / Weakly Favor / Neutral / Weakly Against / Strongly Against

[*Note:* Recursive bisection (“balanced”) was considered as an alternative tree construction. It is documented in Appendix X (informative) for comparison and historical record, but is not proposed as an alternative in this paper. —*end note*]

If consensus is not reached on tree shape, authors will return with a follow-up comparing alternatives; lack of consensus on this poll does not block agreement on the core semantic contract (Poll 2).

Poll 3 — Proceed to API surface (clarified)

Question: We agree the authors may return with one or more API surface proposals (new algorithm and/or execution policy), including proposed naming and header placement, an initial overload set (iterators/ranges, with/without execution policy as applicable), constraints/mandates, and a proposal for topology selection (including any named topology presets and defaulting rules), contingent on favorable outcomes for Polls 1–2, without committing LEWG to adopt a specific surface.

Vote: Strongly Favor / Weakly Favor / Neutral / Weakly Against / Strongly Against

Poll 4A — Straw poll: API approach (new algorithm)

[*Note:* Polls 4A and 4B are straw polls to guide the direction of a future API revision, contingent on favorable outcomes for Polls 1–3. They do not approve an API in this paper. —*end note*]

Question: For the API surface, we support pursuing a **new algorithm** approach (topology parameter is visible in the

algorithm contract; e.g. `canonical_reduce_lanes<L>(...)` (primary, semantic coordinate) with an optional span spelling `canonical_reduce<M>(...)` as convenience for numeric domains).

Vote: Strongly Favor / Weakly Favor / Neutral / Weakly Against / Strongly Against

Poll 4B — Straw poll: API approach (execution policy)

Question: For the API surface, we support pursuing an **execution policy** approach (topology is encoded in a policy type and composes with existing parallel algorithm forms).

Vote: Strongly Favor / Weakly Favor / Neutral / Weakly Against / Strongly Against

Interpretation (non-normative): If both Poll 4A and Poll 4B are favored, return with both surfaces (or an algorithm-first surface plus a policy mapping) and ask LEWG to converge on a primary surface after reviewing wording/teachability/composition.

Acknowledgements

The author thanks Bryce Adelstein Lelbach for a helpful discussion on the algorithm-vs-execution-policy design trade-off.

Revision History

DxxxxR0: Initial draft.

References

C++ Standard References

- **[algorithms.general]** — ISO/IEC 14882, General requirements for algorithms. Specifies exception behavior for sequential algorithm overloads.
- **[algorithms.parallel.exceptions]** — ISO/IEC 14882, Exception handling in parallel algorithms. Specifies that `std::terminate` is called when an exception escapes during parallel execution under standard execution policies.
- **[algorithms.parallel.exec]** — ISO/IEC 14882, Execution policies. Specifies requirements for element access functions and parallel execution semantics.
- **[numeric.ops.accumulate]** — ISO/IEC 14882, Accumulate. Specifies the left-fold semantics of `std::accumulate`.
- **[numeric.ops.reduce]** — ISO/IEC 14882, Reduce. Specifies the generalized sum semantics of `std::reduce`, which permits reassociation for parallel execution.

WG21 Papers

- **[P1928]** — Matthias Kretz. “std::simd — merge data-parallel types from the Parallelism TS 2.” WG21 paper P1928. Available at: https://www.open-std.org/jtc1/sc22/wg21/docs/papers/2023/p1928r8.pdf
- **[P2300]** — Michał Dominiak, Georgy Evtushenko, Lewis Baker, Lucian Radu Teodorescu, Lee Howes, Kirk Shoop, Michael Garland, Eric Niebler, Bryce Adelstein Lelbach. “`std::execution`.” WG21 paper P2300R10, adopted for C++26 at the St. Louis plenary, June 2024. Available at: https://wg21.link/P2300R10
- **[P3375R2]** — Guy Davidson. “Reproducible floating-point results.” WG21 paper P3375R2, 2025. Proposes a `strict_float` type specifying sufficient conformance with ISO/IEC 60559:2020 to guarantee reproducible floating-point arithmetic across implementations. Available at: https://www.open-std.org/jtc1/sc22/wg21/docs/papers/2025/p3375r2.html

Industry References

- **[IntelCNR]** — Intel Corporation. “Introduction to Conditional Numerical Reproducibility (CNR).” *Intel oneAPI Math Kernel Library Developer Guide*. Available at: https://www.intel.com/content/www/us/en/docs/onemkl/developer-guide/current/conditional-numerical-reproducibility.html

- **[NvidiaCUB]** — NVIDIA Corporation. “CUB: CUDA UnBound.” *NVIDIA CUB Documentation*. Describes deterministic reduction variants with fixed-order tree reduction. Available at: https://nvlabs.github.io/cub/
- **[KokkosReduce]** — Sandia National Laboratories. “Custom Reductions: Determinism.” *Kokkos Documentation*. Available at: https://kokkos.github.io/kokkos-core-wiki/API/core/parallel-dispatch/parallel_reduce.html
- **[IntelTBB]** — Intel Corporation. “Specifying a Partitioner.” *Intel oneAPI Threading Building Blocks Developer Guide*. Available at: https://www.intel.com/content/www/us/en/docs/onetbb/developer-guide-reference/current/partitioner.html

Academic References

- **[Dalton2014]** — B. Dalton, E. Wang, and R. Blainey. “SIMDizing pairwise sums: a summation algorithm balancing accuracy with throughput.” *Proceedings of the Workshop on Programming Models for SIMD/Vector Processing (WPMVP)*, 2014. DOI: https://doi.org/10.1145/2568058.2568070
- **[Demmel2013]** — James Demmel and Hong Diep Nguyen. “Fast Reproducible Floating-Point Summation.” *Proceedings of the 21st IEEE Symposium on Computer Arithmetic (ARITH)*, April 2013, pp. 163–172. DOI: https://doi.org/10.1109/ARITH.2013.9
- **[Higham2002]** — Nicholas J. Higham. *Accuracy and Stability of Numerical Algorithms*, Second Edition. Society for Industrial and Applied Mathematics (SIAM), 2002. ISBN: 978-0-89871-521-7. Chapter 4 discusses pairwise summation and error analysis of floating-point summation algorithms.

Appendix A: Illustrative Wording (Informative)

This appendix is illustrative; no API is proposed in this paper (§2). It shows one way the semantic definition could be expressed for a future Standard Library facility. Names, headers, and final API shape are intentionally provisional.

A.1 Example Algorithm Specification

The following shows how the semantic definition could be expressed for a hypothetical algorithm that exposes both topology coordinates:

- **L (lane count):** the semantic topology coordinate (§4.3).
- **M (interleave span, bytes):** a derived numerics convenience coordinate that maps to `L = M / sizeof(V)` when well-formed.

```
// Lane-based topology (portable across ABIs for a fixed L)
template<size_t L, class InputIterator, class T, class BinaryOperation>
constexpr T canonical_reduce_lanes(InputIterator first, InputIterator last, T init,
                 BinaryOperation binary_op);

template<size_t L, class ExecutionPolicy, class ForwardIterator, class T, class BinaryOperation>
T canonical_reduce_lanes(ExecutionPolicy&& policy,
            ForwardIterator first, ForwardIterator last, T init,
            BinaryOperation binary_op);

// Span-based topology (numerics convenience; derives L from sizeof(V))
template<size_t M, class InputIterator, class T, class BinaryOperation>
constexpr T canonical_reduce(InputIterator first, InputIterator last, T init,
              BinaryOperation binary_op);

template<size_t M, class ExecutionPolicy, class ForwardIterator, class T, class BinaryOperation>
T canonical_reduce(ExecutionPolicy&& policy,
         ForwardIterator first, ForwardIterator last, T init,
         BinaryOperation binary_op);
```

Constraints: - For overloads with an ExecutionPolicy, ForwardIterator meets the Cpp17ForwardIterator requirements.

Mandates (lane form): - `L >= 1`.

Mandates (span form): - Let V = `iter_value_t<Iter>` for the relevant iterator type. - `M >= sizeof(V)`. - `M % sizeof(V) == 0`. - The span form is equivalent to the lane form with `L = M / sizeof(V)`.

A.2 Semantics (as-if)

For the span form, let `L = M / sizeof(V)`.

A conforming implementation returns a value that is as-if evaluation of the canonical abstract expression defined in §4, which may be constructed conceptually as follows:

1. **Materialize terms.** Form the conceptual sequence of terms `X[0..N)` by converting each input element to the reduction state type `A` as specified in §4.6.
2. **Partition into lanes.** Choose a lane count `L` (directly, or via `M→L`), and partition terms into `L` logical lanes by index modulo as specified in §4.3.1.
3. **Form fixed-length lane leaves with absence.** Let `K = ceil(N / L)` (for `N > 0`). For each lane index `j` in `[0, L)`, form a fixed-length leaf sequence `Y_j[0..K)` of `maybe<A>` where `Y_j[t]` is `present(X[j + t*L])` when `j + t*L < N`, and `ø` otherwise (§4.3.2).

Informative: When `N < L`, some lane indices have no corresponding input positions; such lanes are best understood as "no lane data exists". In the canonical expression this is represented by `Y_j[t] == ø` for all `t`, yielding `R_j == ø`.

4. **Per-lane canonical reduction.** For each lane index `j`, compute `R_j = CANONICAL_TREE_EVAL(binary_op, Y_j)` using the canonical balanced tree shape and the lifted `COMBINE` rules for `ø` (§4.2–§4.4.1).
5. **Cross-lane canonical reduction.** Form `Z[0..L)` where `Z[j] = R_j` and compute `R_all = CANONICAL_TREE_EVAL(binary_op, Z)` (§4.4.2).
6. **Integrate `init` (if provided).** If an initial value is provided, the result is `init` when `N == 0`, otherwise `binary_op(init, value(R_all))`, as specified in §4.5.

[*Note:* `init` is combined once with the total result and is not treated as an additional input element; treating it as an extra element would shift lane assignments and change the selected canonical expression topology. —*end note*]

Informative: An implementation may skip forming absent leaves and lanes that contain no elements, provided the returned value is as-if evaluation of the canonical expression (since `ø` does not induce applications of `binary_op`).

Appendix B: Implementation Guidance (Informative)

This appendix provides detailed guidance for implementers. It is informative only; the normative specification is the canonical compute sequence defined in §4.

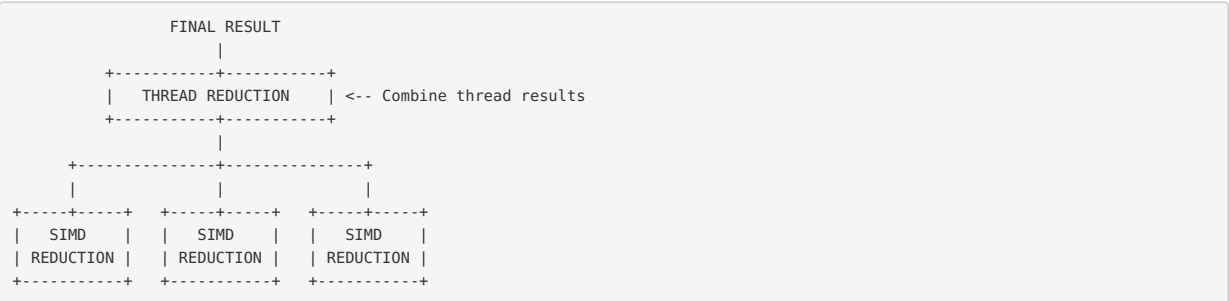
General principle: Implementations need not instantiate lanes for empty subsequences; only the logical reduction order must be preserved. For example, if L = 1000 but N = 5, the implementation need not allocate 1000 accumulators — only the 5 non-empty subsequences participate in the final reduction.

B.1 Two Degrees of Parallelism

Real implementations exploit two orthogonal forms of parallelism:

Parallelism	Mechanism	Typical Scale
SIMD	Vector registers, GPU warps	4–64 lanes
Threads	CPU cores, GPU blocks	4–thousands

This leads to two levels of reduction:



Both levels must produce results matching the canonical expression.

B.2 The Efficient SIMD Pattern

The canonical expression supports efficient vertical addition:

For L = 4, N = 16:

```
Load V0 = {E[0], E[1], E[2], E[3]} → Acc = V0
Load V1 = {E[4], E[5], E[6], E[7]} → Acc = Acc + V1
Load V2 = {E[8], E[9], E[10], E[11]} → Acc = Acc + V2
Load V3 = {E[12], E[13], E[14], E[15]} → Acc = Acc + V3

Acc now holds {R_0, R_1, R_2, R_3}
Final horizontal reduction: op(op(R_0, R_1), op(R_2, R_3))
```

This is contiguous loads plus vertical adds — optimal for SIMD.

B.3 Thread-Level Partitioning

With multiple threads, each processes a contiguous chunk:

```
Thread 0: E[0..256) → {R_0^T0, R_1^T0, R_2^T0, R_3^T0}
Thread 1: E[256..512) → {R_0^T1, R_1^T1, R_2^T1, R_3^T1}
...

Thread partial results are combined per lane, then across lanes:
Global R_0 = CANONICAL_TREE_EVAL(op, {R_0^T0, R_0^T1, ...})
Global R_1 = CANONICAL_TREE_EVAL(op, {R_1^T0, R_1^T1, ...})
...
Final = CANONICAL_TREE_EVAL(op, {R_0, R_1, R_2, R_3})
```

B.4 GPU Implementation

The sequence maps to GPU architectures:

```
        // Vertical addition within block
        for (size_t base = 0; base < n; base += L) {
          size_t idx = base + lane;
          if (idx < n) acc = op(acc, input[idx]);
        }

        // Horizontal reduction using canonical tree
        for (size_t stride = L/2; stride > 0; stride /= 2) {
          if (lane < stride) shared[lane] = op(shared[lane], shared[lane + stride]);
          __syncthreads();
        }

        // Cross-block: fixed slots, not atomics
        block_results[blockIdx.x] = shared[0]; // Deterministic position
```

Key constraints: - Static work assignment (no work stealing) - Fixed block result slots (no atomicAdd) - Canonical tree for horizontal reduction

[*Note:* This GPU sketch is illustrative and non-normative; it demonstrates one possible mapping of the canonical expression to a GPU kernel and is not intended to constrain implementation strategy. —*end note*]

B.5 Cross-Platform Consistency

The proposal defines the expression structure. All conforming implementations evaluate as-if by the same canonical parenthesization and operand ordering for a given topology coordinate `L`.

Within a single, controlled environment (same ISA/backend, same compiler and flags, and equivalent floating-point evaluation models), this removes the run-to-run variability of `std::reduce` by fixing the reduction tree.

Across architectures/backends (CPU ↔ CPU, CPU ↔ GPU): the facility guarantees expression/topology equivalence — i.e., the same abstract tree is specified. Achieving bitwise identity additionally requires aligning the floating-point evaluation environment (§2.5, §6).

B.5.1 Requirements for Cross-Platform Bitwise Reproducibility

This proposal standardizes the abstract expression structure (parenthesization and operand order). Bitwise-identical results across different platforms, compilers, or architectures generally require that fixed expression structure and an equivalent floating-point evaluation environment.

[*Note:* For fundamental floating-point types with non-associative operations (e.g., float, double with `std::plus`), factors that commonly affect bitwise results include (non-exhaustive):

1. **Floating-point format and evaluation rules:** whether the type and operations follow ISO/IEC/IEEE 60559 (IEEE 754) and whether intermediate precision differs (e.g., extended precision).
2. **Rounding mode:** both executions use the same rounding mode (typically round-to-nearest ties-to-even).
3. **Contraction / FMA:** whether multiply-add sequences are fused/contracted, and whether contraction behavior matches across backends.
4. **Subnormal handling:** whether subnormals are preserved or flushed to zero (FTZ/DAZ).
5. **Transforming optimizations:** "fast-math" style options that permit reassociation or relax IEEE behavior.
6. **Library math and user operations:** if `binary_op` (or upstream transformations such as `views::transform`) call implementation-defined math libraries, results may differ.
7. **Topology selection and input order:** both sides use identical topology coordinate (`L`, or `M→L`) and the same input order.

—end note]

What this proposal guarantees: for fixed topology coordinate, input order, and `binary_op`, the abstract expression (parenthesization and operand order) is identical across all conforming implementations.

What remains platform-specific: the floating-point evaluation model (items above). Aligning it may require toolchain controls (for example, disabling contraction and avoiding fast-math; exact spellings are toolchain-specific).

Verification workflow: the demonstrators in Appendix K use a fixed seed and publish expected hex outputs for representative topology coordinates. Matching those values across platforms validates both (a) correct expression structure (this proposal) and (b) sufficiently aligned floating-point environments (user responsibility).

Illustrative CPU ↔ GPU check:

```
        // Compile and run both sides under equivalent FP settings (toolchain-specific).
        double cpu = canonical_reduce_lanes<16>(data.begin(), data.end(), 0.0, std::plus<>{});
        double gpu = cuda_canonical_reduce_lanes<16>(d_data, N, 0.0);

        // Bitwise equality is achievable when the FP evaluation environments are aligned:
        assert(std::bit_cast<uint64_t>(cpu) == std::bit_cast<uint64_t>(gpu));
```

This enables "golden result" workflows where a reference evaluation can act as a baseline for accelerator correctness verification.

B.6 When Physical Width ≠ Logical Width

Physical > Logical (e.g., 256 GPU threads, L = 8): Multiple threads cooperate on each logical lane. Partial results combine using the canonical tree.

Physical < Logical (e.g., 4 physical SIMD lanes, L = 16 logical lanes): Process logical lanes in chunks across multiple physical iterations. The logical sequence is unchanged; only the physical execution differs.

B.7 Performance Characteristics

Representative measurements appear in Appendix N.12. With proper SIMD optimization (8-block unrolling), the reference implementation indicates that the canonical expression structure can be evaluated at throughput comparable to unconstrained reduction, suggesting that conforming implementations need not incur prohibitive overhead. Observed overhead is workload- and configuration-dependent; see also the demonstrators in Appendix K for platform observations.

Appendix C: Prior Art and Industry Practice (Informative)

The tension between parallel performance and numerical reproducibility is a well-documented challenge in high-performance computing (HPC) and distributed systems. This appendix summarizes how existing frameworks address the "Grouping Gap."

C.1 Kokkos Ecosystem (HPC Portability)

Kokkos is a widely used C++ programming model for exascale computing. Its documentation explicitly warns users about the lack of run-to-run stability (for FP) in reductions:

> *"The result of a parallel_reduce is not guaranteed to be bitwise identical across different runs on the same hardware, nor across different numbers of threads, unless the joiner operation is associative and commutative." [KokkosReduce]*

In practice, HPC users requiring reproducibility in Kokkos must often implement "two-pass" reductions or use fixed-point arithmetic, which imposes a significant developer burden. The interleaved topology proposed in §4.3 of this paper mirrors the "vector-lane" optimization patterns used in Kokkos's backend, but elevates it to a deterministic semantic contract.

C.2 Intel oneAPI Threading Building Blocks (oneTBB)

Intel TBB is the industry standard for task-parallelism on CPUs. Its `parallel_reduce` implementation uses a recursive splitting strategy that relies on a "Join" pattern:

1. **Splitting:** The range is split into sub-ranges until they reach a "grain size."
2. **Reduction:** Each sub-range is reduced locally.
3. **Joining:** Sub-results are combined using the user-provided `join()` method.

Because TBB uses a work-stealing scheduler, the order in which these `join()` operations occur is non-deterministic — it depends on which thread becomes free first. While TBB offers a `static_partitioner` to minimize this, it does not guarantee a canonical tree topology, making bitwise identity fragile across different thread counts [IntelTBB].

C.3 NVIDIA Thrust and CUB

For GPU architectures, NVIDIA provides the Thrust and CUB libraries.

Thrust: Generally favors throughput and uses atomic operations in many reduction paths. These atomics are processed in an order determined by hardware thread scheduling, leading to non-deterministic floating-point results.

CUB: Provides more granular control through "Block-level" and "Warp-level" primitives. The interleaved subsequences (S_j) defined in §4.3 are mathematically equivalent to the "blocked-arrangement" and "striped-arrangement" memory patterns used in CUB to achieve peak bandwidth on SIMD-intensive hardware [NvidiaCUB].

C.4 Academic and Research Solutions

Significant research has been conducted into "Reproducible Summation" (e.g., [Demmel2013]). These solutions typically fall into two categories:

1. **Exact Summation:** Using very wide fixed-precision accumulators (e.g., 400+ bits) to ensure associativity. These are bit-accurate but carry a 2x–10x performance penalty.
2. **Fixed-Topology Summation:** Enforcing a specific reduction tree.

This proposal follows the Fixed-Topology school of thought. It recognizes that while we cannot easily standardize "Exact Summation" due to its cost, we can standardize the topology, which provides reproducibility for a given platform at a much lower performance cost (10–15% overhead relative to `std::reduce` in representative configurations; see Appendix H).

See References for full citations of [KokkosReduce], [IntelTBB], [NvidiaCUB], and [Demmel2013].

Appendix D: Standard Wording for Sequential Evaluation Order (Informative)

In the C++ Standard, sequential algorithms like `std::accumulate` and `std::ranges::fold_left` have a mandated evaluation order — the grouping of operations is fully specified as a left-fold.

Important distinction: The Standard specifies evaluation order, not bitwise reproducibility. Even with identical evaluation order, results may differ across compilers or platforms due to floating-point evaluation model factors (see §D.5). However, a fixed evaluation order is a necessary precondition for reproducibility — without it, reproducibility is impossible even under identical floating-point evaluation models.

D.1 std::accumulate

The evaluation order guarantee for `std::accumulate` is found in the Numerics library section of the Standard.

Section: [numeric.ops.accumulate]/2 (In N4950/C++23: §27.10.3)

The Text: The Standard defines the behavior of `accumulate(first, last, init, binary_op)` as:

> *"Computes its result by initializing the accumulator acc with the initial value init and then modifies it with `acc = std::move(acc) + *i` or `acc = binary_op(std::move(acc), *i)` for every iterator i in the range [first, last) in order."*

What this guarantees: The expression structure is fixed as `(((init + a) + b) + c)...`. The grouping is fully specified — there is no implementation freedom in how operations are combined.

What this does not guarantee: Bitwise identical results across compilers or platforms. The same grouping may produce different bits due to floating-point evaluation model differences.

Contrast with std::reduce: `std::reduce` uses a generalized sum ([numerics.defns]/1–2, [numeric.ops.reduce]/7), allowing the implementation to group elements as `((a + b) + (c + d))` or any other valid tree. This makes even the grouping non-deterministic.

D.2 std::ranges::fold_left (C++23)

For the newer Ranges-based algorithms, the specification is even more explicit about its algebraic structure.

Section: [alg.fold] (In N4950/C++23: §27.6.18)

The Text:

> *"The range fold algorithms are sequential operations that perform a left-fold... `ranges::fold_left(R, init, f)` is equivalent to: `cpp auto acc = init; for (auto&& e : R) acc = f(std::move(acc), e); return acc;`"*

What this guarantees: By defining the algorithm via an explicit for loop, the Standard fully specifies the evaluation order. The state of the accumulator at step N depends exactly and only on the state at step N-1 and the Nth element.

D.3 Sequential vs. Parallel: Contrasting Guarantees

Property	Sequential (accumulate/fold_left)	Parallel (reduce)
Standard Section	[numeric.ops.accumulate] / [alg.fold]	[numeric.ops.reduce]
Grouping	Mandated: Left-to-right	Generalized Sum (Unspecified)
Complexity	O(N) operations	O(N) operations
Evaluation Order	Fully specified	Not specified

Key insight: `std::accumulate` and `std::reduce` differ in whether the grouping is specified, not in whether they guarantee "bitwise reproducibility" (neither does, strictly speaking).

D.4 Init Placement: Comparison with std::accumulate

This proposal specifies init placement as `op(I, R)` (where `I` is the initial value materialized as type `A` per §4.6) — the initial value is combined once at the end with the tree result. The table below compares this to `std::accumulate`:

Aspect	std::accumulate	This Proposal (`op(I, R)`)
Init handling	Folded at every step	Combined once at end
Structure	`(((init ⊕ a) ⊕ b) ⊕ c)`	`init ⊕ tree_reduce(a,b,c,d)`
Init participates in	N operations	1 operation

Implication for non-associative operations: For associative operations, these approaches produce equivalent results. For non-associative operations (e.g., floating-point addition), the results may differ. Users migrating from `std::accumulate` should be aware of this distinction.

See Appendix E for rationale behind the `op(I, R)` design choice.

D.5 Important Caveat: Evaluation Order ≠ Bitwise Identity

The Standard specifies evaluation order, not bitwise results. Even when the grouping of operations is fully specified (as in `std::accumulate`), bitwise identity across different compilers, platforms, or even different runs is not guaranteed due to:

1. **Intermediate precision:** The Standard permits implementations to use higher precision for intermediate results (e.g., x87 80-bit registers vs SSE 64-bit). See [cfenv.syn] and implementation-defined floating-point behavior.
2. **FMA Contraction:** A compiler might contract `a * b + c` into a single fused multiply-add instruction on one platform but not another, changing the result bits. This is controlled by `#pragma STDC FP_CONTRACT` and compiler flags like `-ffp-contract`.
3. **Rounding mode:** Different default rounding modes across implementations can affect results.

What this proposal provides: A fully specified expression structure (grouping and operand order). Combined with user control of the floating-point evaluation model, this enables reproducibility.

What this proposal does not provide: Automatic cross-platform bitwise identity. Users must also control their floating-point evaluation model (see §6).

D.6 Why Compilers Cannot Reassociate Mandated Evaluation Order

A potential concern is whether compilers might reassociate the reduction operations defined by this proposal, defeating the run-to-run stability guarantee. This section explains why such reassociation would be non-conforming.

D.6.1 The As-If Rule

The C++ Standard permits compilers to perform any transformation that does not change the observable behavior of a conforming program ([intro.abstract]/1). This is commonly called the "as-if rule."

For floating-point arithmetic, reassociation (changing `(a + b) + c` to `a + (b + c)`) generally changes the result due to rounding. Therefore, a compiler cannot reassociate floating-point operations under the as-if rule unless:

1. It can prove the result is unchanged (generally impossible for FP), or
2. The user has explicitly opted into non-standard semantics via compiler flags

D.6.2 Existing Precedent: std::accumulate

`std::accumulate` mandates a specific evaluation order ([numeric.ops.accumulate]/2):

> *"Computes its result by initializing the accumulator acc with the initial value init and then modifies it with `acc = std::move(acc) + *i` or `acc = binary_op(std::move(acc), *i)` for every iterator i in the range [first, last) in order."*

This wording constrains implementations to a left-fold: `(((init ⊕ a) ⊕ b) ⊕ c) ...`

Compilers respect this constraint. A compiler that reassociated `std::accumulate` into a tree reduction would be non-conforming, because the observable result would differ for non-associative operations (including floating-point addition).

D.6.3 This Proposal Follows the Same Model

This proposal mandates a specific expression structure with the same normative force. The "Generalized Sum" semantics of `std::reduce` explicitly grant permission to reassociate; this proposal removes that permission — exactly as `std::accumulate` mandates a specific left-fold expression.

[*Note:* For `std::accumulate`, "expression structure" and "evaluation order" coincide because the left-fold is inherently sequential. For this proposal, the expression structure (parenthesization) is fixed, but independent subexpressions may be evaluated concurrently. —*end note*]

The only difference from `std::accumulate` is the shape of the mandated expression:

Algorithm	Mandated Expression Shape
std::accumulate	`(((init ⊕ a) ⊕ b) ⊕ c)` Linear fold
This proposal	`init ⊕ ((a ⊕ b) ⊕ (c ⊕ d))` Balanced binary tree

Both specifications constrain the parenthesization. Both are subject to the same as-if rule. A compiler that reassociates one would equally violate conformance by reassociating the other.

D.6.4 What About -ffast-math?

Compiler flags like `-ffast-math`, `-fassociative-math`, or `/fp:fast` explicitly opt out of IEEE 754 compliance. Under these flags:

- The compiler may reassociate floating-point operations
- `std::accumulate` may not produce left-fold results
- `std::reduce` may not match any particular grouping
- This proposal's guarantee would also not apply

This is not a defect in the proposal — it is the documented behavior of these flags. Users who enable `-ffast-math` have explicitly traded determinism for performance. The same trade-off applies to all floating-point code, not just reductions.

Recommendation for users requiring run-to-run stability: Compile with `-ffp-contract=off` (or equivalent) and avoid `-ffast-math`. This applies equally to `std::accumulate` and to this proposal.

D.6.5 Summary

Concern	Resolution
"Compilers might reassociate the tree"	Violates as-if rule, same as for std::accumulate
"What about -ffast-math?"	User has opted out of IEEE 754; all FP guarantees void
"Is this proposal different from existing algorithms?"	No — same conformance model as std::accumulate

The expression-structure guarantee in this proposal has the same normative force (with respect to parenthesization and operand order) as the mandated left-fold structure in `std::accumulate`. Compilers that respect the latter will respect the former.

Appendix E: Init Placement Rationale (Informative)

This appendix provides rationale for the init placement design choice. This proposal specifies **Option A: op(I, R)** — the initial value (materialized as a value `I` of type `A` per §4.6) is applied as the left operand to the result of the canonical tree reduction. This appendix explains why this option was chosen over alternatives.

[*Note:* `init` is combined once with the total result and is not treated as an additional input element; treating it as an extra element would shift lane assignments and change the selected canonical expression topology. —*end note*]

E.1 Design Options Considered

Option A: Post-reduction (`op(I, R)`) — CHOSEN

```
canonical_reduce<M>(E[0..N), init, op):
  if N == 0:
    return init
  let R = interleaved_reduce<M>(E[0..N))
  let I = A(init)  // materialize init as the reduction state type
  return op(I, R)
```

Pros: - init is not part of the canonical reduction tree — clear separation of concerns - Simplifies parallel implementation (tree can complete before init is available) - Matches `std::reduce`'s treatment of init ([numeric.ops.reduce]) - Empty range handling is trivial: return init

Cons: - Differs from `std::accumulate`'s left-fold semantics - For non-associative operations, results differ from left-fold expectations

Option B: Post-reduction (op(R, I))

Same as Option A but with init as the right operand.

Pros: - Same implementation simplicity as Option A

Cons: - Less intuitive for users expecting init to be "first" - Still differs from `std::accumulate`

Option C: Treat init as element 0 (prepend to sequence)

```
canonical_reduce<M>(E[0..N), init, op):
  let E' = {init, E[0], E[1], ..., E[N-1]}
  return interleaved_reduce<M>(E'[0..N+1))
```

Pros: - init participates in the canonical tree with a known position - More predictable for non-associative operations

Cons: - Shifts all element indices by 1 - init assigned to lane 0, changing topology when L > 1 - Complicates the interleaving definition

Option D: Leave implementation-defined

The standard specifies that init participates in exactly one `binary_op` application with the tree result, but does not specify the order.

Pros: - Maximum implementation flexibility - Avoids contentious design decision - For associative operations (the common case), result is unaffected

Cons: - Non-deterministic for non-associative operations - Users cannot rely on specific init behavior

E.2 Analysis

For **associative operations** (the vast majority of use cases), all options produce equivalent results. The choice only matters for non-associative operations.

For non-associative operations, users already face the fact that the tree reduction differs from left-fold. The init placement is one additional degree of freedom that must be specified for full run-to-run stability.

E.3 Why Option A Was Chosen

This proposal specifies Option A (`op(I, R)`) for the following reasons:

1. **SIMD purity:** Folding init into lanes would require broadcast and can amplify init influence in non-associative operations.
2. **Task independence:** A "pure" tree over the range can be computed before init is available in async/distributed contexts.
3. **Algebraic clarity:** A balanced tree has no distinguished "first" element; init is most naturally a post-reduction adjustment.
4. **Precedent:** This matches `std::reduce`'s treatment of init ([numeric.ops.reduce]) (as part of the generalized sum, not folded sequentially).
5. **Full determinism:** Specifying the placement ensures that the complete evaluation order — including init — is fully determined, consistent with the paper's run-to-run stability guarantee.

E.4 Why Not Treat init as Element 0?

Treating init as "element 0" (Option C) introduces complications:

- With L > 1, init would be assigned to lane 0, but all other element indices would shift
- The meaning of init would depend on lane topology in ways that are harder to explain and reason about
- It conflates two distinct roles: "initial state" vs "operand in the tree"

The post-reduction design keeps these roles separate.

E.5 Migration Path for Users Expecting accumulate-style Semantics

Users who require init to participate as a leaf within the tree (rather than post-reduction) can achieve this through composition:

```cpp
// To get init as element 0 in the tree:
auto extended = concat_view(single_view(init), data);
auto result = canonical_reduce<M>(extended.begin(), extended.end(),
                identity_element, op);

// Or manually:
auto tree_result = canonical_reduce<M>(data.begin(), data.end(),
                  identity_element, op);
auto result = op(init, tree_result); // explicit post-reduction (matches this proposal)
```

This flexibility allows users to achieve alternative semantics when needed while the standard provides a single, well-defined default.

Appendix F: Design Evolution (Informative)

This proposal underwent significant internal development before committee submission. The design space was explored systematically, with key decisions documented in §3 (Design Space) and the appendices. This section summarizes the major evolution points for reviewers interested in the design rationale.

F.1 Core Design Decisions

Decision	Alternatives Considered	Chosen Approach	Rationale
Topology	Left-fold, blocked, N-ary tree	Interleaved balanced binary	O(log N) depth, SIMD-friendly
Parameterization	Fixed constant, lane count, implementation-defined	User-specified M (bytes)	Type-independent, future-proof
Init placement	As leaf, implementation-defined	Post-reduction `op(I, R)`	Algebraic clarity, matches std::reduce ([numeric.ops.reduce])
Split rule	⌈k/2⌉, variable	⌊k/2⌋ (normative)	Unique grouping specification

F.2 Key Design Iterations

Why bytes, not lanes: Initial designs parameterized by lane count L. `L` is the semantic topology coordinate; `M` is a convenience spelling for numeric domains. `M` is convenient for layout-stable arithmetic types because it scales across element sizes (e.g., `float` vs `double`), but it does not change the semantic definition of the canonical expression.

Why interleaved, not blocked: Blocked decomposition creates topology that varies with thread count and alignment. Interleaved assignment (index mod L) produces stable topology regardless of execution strategy. See §3.2.

Why user-specified M: Fixed M ages poorly as hardware evolves; implementation-defined M defeats determinism. User specification places control where it belongs. See §3.3.

Init placement: Treating init as “element 0” would shift all indices and change lane assignment. Post-reduction application keeps the tree “pure” and matches `std::reduce` semantics ([numeric.ops.reduce]). See §4.5 and Appendix E.

F.3 Industry Context

The core design (fixed topology, user-controlled width) draws on approaches seen in production libraries: - Intel oneMKL CNR (Conditional Numerical Reproducibility) - NVIDIA CUB deterministic overloads - PyTorch/TensorFlow deterministic modes

These libraries address similar problems through different mechanisms. Their existence suggests the design space is viable and addresses real needs. See §3.6 for references.

Appendix G: Detailed Design Rationale (Informative)

This appendix contains detailed rationale for design decisions that were summarized in Section 4. It is provided for reviewers seeking deeper understanding of the trade-offs.

G.1 Why the Convenience Spelling Uses Bytes

M is expressed in bytes rather than lanes for type independence: lane count is derived from `sizeof(V)`, so a single M works across element types.

M (bytes)	double (8B)	float (4B)	int32_t (4B)
16	2 lanes	4 lanes	4 lanes
32	4 lanes	8 lanes	8 lanes
64	8 lanes	16 lanes	16 lanes

M corresponds directly to SIMD register width:

Target	Register Width	M
SSE / NEON	16 bytes	16
AVX / AVX2	32 bytes	32
AVX-512	64 bytes	64
Future (1024-bit)	128 bytes	128

Implementations with narrower physical registers execute the canonical expression through multiple iterations. The logical topology — which operations combine with which — is unchanged.

G.2 Why Interleaved Topology Supports Efficient SIMD

The interleaved topology supports the simplest and most efficient SIMD implementation pattern:

```
Memory: E[0] E[1] E[2] E[3] | E[4] E[5] E[6] E[7] | E[8] ...
    └── Vector 0 ──┘   └── Vector 1 ──┘

Iteration 1: Load V0 = {E[0], E[1], E[2], E[3]} → Acc = V0
Iteration 2: Load V1 = {E[4], E[5], E[6], E[7]} → Acc = Acc + V1
...
Final: Acc = {R_0, R_1, R_2, R_3}
```

This pattern achieves: - One contiguous vector load per iteration (optimal memory access) - One vector add per iteration (single instruction) - Streaming sequential access (spatially local) - No shuffles or gathers until final horizontal reduction

A blocked topology would require gather operations or sequential per-lane processing, losing the SIMD benefit.

G.3 Information Density and Spatial Locality

The divisibility constraint and formula `L = M / sizeof(V)` act as a self-regulating mechanism for memory efficiency. For M = 64 and double (8 bytes), L = 8, maintaining exactly 64 bytes of independent partial-accumulator state per logical stride.

For large types where `sizeof(V) > M`, the user specifies `M = sizeof(V)`, giving L = 1. This degenerates to a single-lane balanced binary tree with perfect spatial locality.

G.4 Cross-Architecture Expression-Parity

GPU architectures achieve peak efficiency when reduction trees align with warp width (typically 32 threads). For double (8 bytes), a warp-level reduction operates on 32 × 8 = 256 bytes.

By specifying L=32 (equivalently M=256 for double), users define a canonical expression that maps to warp-level operations on GPU while CPU evaluates the same mathematical expression by iterating over narrower registers.

What expression-parity guarantees: All platforms evaluate the same mathematical expression — same parenthesization, same operand ordering, same reduction tree topology.

What it does not guarantee: Bitwise reproducibility requires equivalent floating-point semantics across architectures, which is difficult due to differences in FTZ/DAZ modes, FMA contraction, and rounding behavior.

G.5 The Golden Reference (L = 1)

Specifying `L = 1` (equivalently `M = sizeof(V)`) collapses the algorithm to a single global balanced binary tree. This serves as a hardware-agnostic baseline for CI/CD testing and debugging numerical discrepancies.

G.6 Divergence from std::accumulate

`std::accumulate` specifies a strict linear left-fold with depth O(N). `canonical_reduce` specifies a balanced binary tree with depth O(log N). For non-associative operations, these represent fundamentally different algebraic expressions.

For floating-point summation, the tree structure is often numerically advantageous: error growth is O(log N · ε) versus O(N · ε) for left-fold. This is well-established as "pairwise summation" in numerical analysis [Higham2002].

G.7 Init Placement Determinism

If init placement were implementation-defined, two conforming implementations could produce different results for the same inputs. For non-commutative operations:

```
op(I, R) = 5.0 * 2 + 10.0 = 20.0
op(R, I) = 10.0 * 2 + 5.0 = 25.0
```

Therefore, this proposal normatively specifies `op(I, R)` — init as left operand of the final combination.

Appendix H: Performance Feasibility (Informative)

This appendix provides representative prototype measurements to support the claim that enforcing a fixed expression structure is practical. These measurements are not a performance guarantee.

H.1 Prototype test conditions

- **Floating-point evaluation model controls:** `-ffp-contract=off`, `-fno-fast-math` (GCC/Clang); `/fp:precise` (MSVC)
- **FMA explicitly disabled** via compiler flags where applicable
- **Input sizes:** 10K to 10M elements, uniformly distributed random values

H.2 Representative observations

- Observed overhead of approximately 10-15% compared to `std::reduce` for typical inputs in the tested configurations.
- Overhead is primarily attributable to enforcing a fixed parenthesization (removing reassociation freedom) and maintaining per-lane state.
- Results are configuration-dependent; different compilers, CPUs/ISAs, and choices of M can shift the trade-off materially.

H.3 Interpretation

Users opt into a canonical reduction when they value reproducibility and auditability over peak throughput. Users requiring maximum throughput can continue to use `std::reduce` (or domain-specific facilities) where unspecified reassociation is acceptable.

Appendix I: Rationale for Semantic Span Presets (Informative)

This appendix records rationale for providing a small set of **standard-fixed span preset constants** (in bytes) as coordination points for topology selection (§9.5). The goal is to provide readable, stable choices that do not vary with platform properties and therefore avoid "silent semantic drift" in returned values for non-associative operations.

The span coordinate is a numerics convenience: for layout-stable numeric types, it derives the semantic lane count `L = M / sizeof(V)`.

I.1 Rationale for 128 Bytes (Small Span)

A “small” preset should provide useful lane counts for the main scalar sizes while remaining narrow enough that overhead does not dominate for moderate N.

For common scalar sizes this yields: - **float (4 bytes):** L = 128 / 4 = 32 - **double (8 bytes):** L = 128 / 8 = 16 - **64-bit integers (8 bytes):** L = 16

This width is a practical coordination point for implementations that exploit instruction-level parallelism through vectorization and batching while preserving the canonical expression structure.

I.2 Rationale for 1024 Bytes (Large Span)

A “large” preset is an explicit opt-in for throughput-oriented structures and heterogeneous verification workflows.

For common scalar sizes this yields: - **float (4 bytes):** L = 1024 / 4 = 256 - **double (8 bytes):** L = 1024 / 8 = 128

Such widths provide substantial independent work per lane and can support aggressive batching and parallel evaluation of independent reduction nodes while preserving the same abstract expression.

I.3 Cross-Domain Verification

Because these presets are standard-fixed, a user can select the same span constant when executing on different hardware or in different deployment environments. When the floating-point evaluation model is aligned (and the same `binary_op` and input order are used), the abstract expression structure is identical; any remaining divergence is attributable to differences in the underlying arithmetic environment rather than to re-association or topology choice.

[*Note:* This appendix is informative. These values are coordination points for a stable abstract expression; they do not impose any particular scheduling, threading, or vectorization strategy. —*end note*]

Appendix J: Indicative API Straw Man (Informative)

This appendix is illustrative; no API is proposed in this paper (§2). It records one indicative way an eventual Standard Library facility could expose the semantics defined in §4, while adopting the standard-fixed named preset approach (Option 3 in §9.5). The intent is to give LEWG something concrete to react to, while keeping spelling and header placement explicitly provisional.

Presentation note (informative): - The **semantic topology coordinate** is the lane count `L`. - A byte span `M` is a **derived convenience** that maps to `L` using `sizeof(value_type)`. - When cross-platform expression stability is required, prefer APIs that specify `L` directly.

J.1 Design goal

- Preserve the core semantic contract: for fixed topology selection, the returned value is as-if evaluating the canonical expression.
- Avoid “silent semantic drift” for a no-argument default: defaults must be stable across targets/toolchains.
- Make “coordination choices” readable: users should be able to say “small / large span” in code reviews, not “128 / 1024”.

J.2 Favored approach: standard-fixed preset constants (Option 3)

This approach exposes preset names as standard-fixed constants, and (optionally) defines a default in terms of one of those preset constants.

Illustrative (provisional) names:

```
namespace std {
  // Standard-fixed span presets (bytes). Values never change.
  inline constexpr size_t canonical_span_small = 128; // "narrow" preset in prototypes
  inline constexpr size_t canonical_span_large = 1024; // "wide" preset in prototypes

  // This paper does not propose a default. A follow-on API paper may propose one.
}
```

Clarification (informative): The preset span values above are intended to be fixed, `constexpr` constants, not implementation-tunable parameters, so that the same canonical expression shape is selected across implementations.

This does not imply bitwise-identical results across different floating-point evaluation models.

These byte-span presets are a convenience for selecting `L` via `L = M / sizeof(value_type)`; the canonical expression is defined by the resulting `L`. Because `sizeof(value_type)` may vary across ABIs/platforms, users who require the same canonical expression across platforms should prefer specifying `L` directly.

Interpretation (informative): for common arithmetic types, these spans correspond to the following lane counts: - **double (8 bytes):** L_small = 16, L_large = 128 - **float (4 bytes):** L_small = 32, L_large = 256

These correspond to the "small/narrow" and "large/wide" semantics used in the prototype demonstrations and rationalized in Appendix I.

J.3 Straw-man algorithm API (span coordinate)

The simplest surface is an algorithm family parameterized by the span coordinate.

```
namespace std {
  // Primary overloads (illustrative; header placement is out of scope for this paper)
  template<size_t M,
       class InputIterator, class T, class BinaryOperation>
  constexpr T canonical_reduce(InputIterator first, InputIterator last,
                 T init, BinaryOperation binary_op);

  template<size_t M,
       class ExecutionPolicy, class ForwardIterator, class T, class BinaryOperation>
  T canonical_reduce(ExecutionPolicy&& policy,
           ForwardIterator first, ForwardIterator last,
           T init, BinaryOperation binary_op);
}
```

Typical call sites:

```
// Readable coordination points (small/narrow vs large/wide)
auto a = std::canonical_reduce<std::canonical_span_small>(v.begin(), v.end(), 0.0, std::plus<>{});
auto b = std::canonical_reduce<std::canonical_span_large>(v.begin(), v.end(), 0.0, std::plus<>{});

// No default topology is proposed in this paper (users spell a preset or provide literal)
// If a future default is added, it should be standard-fixed and spelled in terms of a preset
auto c = std::canonical_reduce<std::canonical_span_small>(v.begin(), v.end(), 0.0, std::plus<>{});
```

Rationale for this shape (informative): - Keeps the topology selection in the type system (NTTP), which is consistent with "semantic selection", not a performance hint. - Supports compile-time specialization for a fixed topology coordinate (L, or derived span M→L) without requiring new execution-policy machinery. - Makes the coordination presets show up in diagnostics and in code review as names.

J.4 Straw-man algorithm API (lane coordinate)

Where `sizeof(V)` is not stable across the intended reproducibility domain (e.g., user-defined types), the paper already recommends selecting topology by lane count L (§3.5, §4.3.1). An indicative spelling for that coordinate is:

```
namespace std {
  template<size_t L,
       class InputIterator, class T, class BinaryOperation>
  constexpr T canonical_reduce_lanes(InputIterator first, InputIterator last,
                   T init, BinaryOperation binary_op);

  template<size_t L,
       class ExecutionPolicy, class ForwardIterator, class T, class BinaryOperation>
  T canonical_reduce_lanes(ExecutionPolicy&& policy,
              ForwardIterator first, ForwardIterator last,
              T init, BinaryOperation binary_op);
}
```

This keeps the semantic contract ABI-independent: for fixed L, the canonical tree is identical even if `sizeof(V)` differs across targets.

J.5 Notes on naming and evolution

- The values of existing preset constants must never change. Future standards may add new presets under new

names.
- Whether the facility lives as a new algorithm (e.g., `canonical_reduce`) or as a `std::reduce`-adjacent customization is an LEWG design choice; this appendix only shows one straightforward spelling.
- If LEWG prefers to mirror [numerics.defns] ("GENERALIZED_SUM"-style definitional functions), the same preset constants can still be used as stable topology selectors.

[*Note:* This appendix is illustrative. It is intended to reduce "API haze" during discussion while keeping the semantic core of the paper independent of any particular surface spelling. —*end note*]

J.6 Range overloads (straw-man)

Range overloads are deferred per §2. The following sketches are included only to reduce "API haze" and to indicate one illustrative direction consistent with the semantic requirements of §4.

Key observation: the canonical expression in §4 is defined over N elements, so a range surface generally needs N prior to evaluation. A surface can obtain N without allocation (e.g., `sized_range` or a counting pass over a multipass range), or it can require explicit materialization for single-pass sources.

```
namespace std::ranges {
  // Span coordinate — sequential/reference mode
  // Non-sized forward ranges can be supported via a counting pass to determine N.
  template<size_t M,
       forward_range R, class T, class BinaryOperation>
  requires (M >= sizeof(range_value_t<R>)) &&
       (M % sizeof(range_value_t<R>) == 0)
  constexpr T canonical_reduce(R&& r, T init, BinaryOperation op);

  // Span coordinate — execution policy overload (illustrative, conservative constraints)
  // A subsequent API revision could choose conservative constraints (e.g., random_access_range)
  template<size_t M,
       class ExecutionPolicy,
       random_access_range R, class T, class BinaryOperation>
  requires sized_range<R> &&
       is_execution_policy_v<remove_cvref_t<ExecutionPolicy>> &&
       (M >= sizeof(range_value_t<R>)) &&
       (M % sizeof(range_value_t<R>) == 0)
  T canonical_reduce(ExecutionPolicy&& policy, R&& r, T init, BinaryOperation op);

  // Lane coordinate — sequential/reference mode
  template<size_t L,
       forward_range R, class T, class BinaryOperation>
  requires (L >= 1)
  constexpr T canonical_reduce_lanes(R&& r, T init, BinaryOperation op);

  // Lane coordinate — execution policy overload (illustrative, conservative constraints)
  template<size_t L,
       class ExecutionPolicy,
       random_access_range R, class T, class BinaryOperation>
  requires sized_range<R> &&
       is_execution_policy_v<remove_cvref_t<ExecutionPolicy>> &&
       (L >= 1)
  T canonical_reduce_lanes(ExecutionPolicy&& policy, R&& r, T init, BinaryOperation op);
}
```

[*Note:* The final constraints for a range surface (e.g., whether to permit a counting pass for non-sized multipass ranges, and whether to reject single-pass `input_range` to avoid implicit allocation) are design questions for a subsequent API-focused revision once LEWG has accepted the semantic contract in §4. Appendix L records the relevant trade-offs. —*end note*]

Appendix K: Demonstrator Godbolts (Informative)

This appendix is not part of the proposal. It records the demonstrator Compiler Explorer ("Godbolt") programs used to validate the semantics described in §4 on multiple architectures, and to show the gross performance impact of enforcing a fixed abstract expression.

The programs referenced here are semantic witnesses, not reference implementations. They exist to demonstrate that the canonical expression defined in §4 can be evaluated on real hardware (SIMD, GPU, multi-threaded). They are not normative examples and are not intended as implementation guidance for general `binary_op`. All demonstrators except GB-SEQ test only `std::plus<double>` and may use `0.0` for absent operand positions; conforming implementations must handle arbitrary `binary_op` via the lifted `COMBINE` rules of §4.2.2.

§K.0 Golden reference values

The following reference results are used throughout the demonstrators to validate bitwise reproducibility under the specified canonical expression and fixed PRNG seed:

- **NARROW** (`L = 16` , i.e. `M = 128 bytes` for `double`): `0x40618f71f6379380`
- **WIDE** (`L = 128` , i.e. `M = 1024 bytes` for `double`): `0x40618f71f6379397`

[*Note:* These values validate conformance to the canonical expression for the specific demonstrator environment. Bitwise agreement across platforms additionally requires aligned floating-point evaluation models (§2.5, §6). —*end note*]

The intent is pragmatic: give reviewers a “click-run-inspect” artifact that: - validates semantic invariants (tree identity and “golden hex” outputs) rather than trying to “win benchmarks”, - prints a short verification block (seed, N, and the resulting hex value), - checks run-to-run stability and translational invariance (address-offset invariance), - and shows a coarse benchmark table comparing against `std::accumulate` and `std::reduce` variants.

Important caveat: Compiler Explorer run times vary substantially with VM load, CPU model, and throttling. These tables are illustrative only; the repository benchmarks (multi-thread and CUDA) are the authoritative numbers.

K.1 Demonstrator set (gross platform runs)

The following Compiler Explorer (“Godbolt”) demonstrators are intended to be click-run-inspect artifacts for reviewers. Each link contains: - a determinism/verification block (seed, N, and printed result hex), - run-to-run stability checks, - a coarse timing table comparing against `std::accumulate` and `std::reduce` variants under comparable FP settings.

Demonstrator	Platform	Purpose	Arbitrary `binary_op`?	Notes
[GB-SEQ] single-thread reference	Portable	Sequential evaluation of the canonical expression (§4)	Yes (faithful §4 reference)	Debugger-friendly "golden" comparator; single-threaded only.
[GB-x86-AVX2] single-file x86	x86-64	Canonical reduction on x86 with AVX2 codegen; compares against `std::accumulate` and `std::reduce`	No (`std::plus<double>` only)	Uses safe feature gating; suitable for "Run".
[GB-x86-MT] multi-threaded x86	x86-64	Deterministic multi-threaded reduction using shift-reduce stack state + deterministic merge	No (`std::plus<double>` only)	Demonstrates schedule-independent, thread-count-independent results.
[GB-x86-MT-PERF] multi-threaded perf	x86-64	Production-quality multi-threaded reduction with thread pool and per-lane carry pairwise	No (`std::plus<double>` only)	Thread pool amortizes creation cost; shows throughput scaling.
[GB-NEON] single-file NEON	AArch64	Canonical reduction on ARM64 using NEON (exact tree preserved)	No (`std::plus<double>` only)	Prints build-proof macros (`__aarch64__` , `__ARM_NEON` , `__ARM_NEON_FP`).
[GB-NEON-PERF] NEON performance	AArch64	Shift-reduce with 8-block NEON pre-reduction for near-bandwidth throughput	No (`std::plus<double>` only)	8-block straight-line NEON hot loop; carry cascade fires 8× less often.
[GB-CUDA] (optional) CUDA / CUB comparison	NVCC	Illustrates canonical topology evaluation on GPU and compares against CUB reduction	No (`std::plus<double>` only)	Heterogeneous "golden result" workflow demonstrator.

Godbolt links:

- [GB-SEQ] = https://godbolt.org/z/8EEhEqrz6
  (single-threaded reference; faithful §4 implementation including `COMBINE` rules; supports arbitrary `binary_op`)
- [GB-x86-AVX2] = https://godbolt.org/z/Eaa3vWYqb
  (tests `std::plus<double>` only; uses `0.0` for absent positions; proves SIMD vertical-add matches §4 tree)
- [GB-x86-MT] = https://godbolt.org/z/7a11r9o95
  (tests `std::plus<double>` only; uses `0.0` for absent positions; proves thread-count and schedule invariance)
- [GB-x86-MT-PERF] = https://godbolt.org/z/sdxMohT48
  (tests `std::plus<double>` only; thread pool + per-lane carry pairwise; proves throughput scaling with thread

count)

- [GB-NEON] = https://godbolt.org/z/Pxzc3YM7q
  (Arm v8 / AArch64; tests `std::plus<double>` only; uses `0.0` for absent positions; proves NEON vector reduction matches §4 tree)
- [GB-NEON-PERF] = https://godbolt.org/z/sY9W78rze
  (Arm v8 / AArch64; tests `std::plus<double>` only; shift-reduce with 8-block NEON pre-reduction; near-bandwidth throughput)
- [GB-CUDA] = https://godbolt.org/z/5n9EvGoeb
  (CUDA/NVCC; tests `std::plus<double>` only; uses `0.0` for absent positions; proves warp shuffle matches §4 tree; includes L=16 and L=128)

All demonstrators use the same dataset generation (seeded RNG) and report the same canonical results for the two topology coordinates used throughout this paper:

- **Small / narrow:** L = 16 for double (M = 128 bytes)
- **Large / wide:** L = 128 for double (M = 1024 bytes)

K.2 What the demonstrators are intended to prove

Each demonstrator prints:

1. **Data verification:** the first few generated elements (as hex) match a known sequence for the fixed seed.
2. **Correctness:** the returned value for L=16 and L=128 matches known “golden” hex outputs.
3. **Run-to-run stability:** repeated evaluation yields identical results.
4. **Translational invariance:** copying the same values to different memory offsets does not change the result.

These are concrete checks that: - the canonical expression is independent of execution schedule (single-thread vs vectorized), - the topology is a function of the chosen topology coordinate (L, or derived span M→L) and input order, - and the implementation does not accidentally depend on address alignment or allocator behavior.

K.3 Expected verification outputs (for the published demonstrators)

For the canonical seed and N = 1,000,000 doubles, the demonstrators are configured to print the following expected result hex values:

- **L = 16, “NARROW” (M = 128 bytes):** `0x40618f71f6379380`
- **L = 128, “WIDE” (M = 1024 bytes):** `0x40618f71f6379397`

The demonstrators treat a mismatch as a test failure.

K.3.1 Cancellation stress dataset (recommended)

The PRNG dataset validates implementation correctness but may not demonstrate *why* canonical reduction matters. Uniform random values of similar magnitude can produce nearly identical results under different tree shapes, making the golden match appear trivially achievable.

To demonstrate the facility’s core value proposition, demonstrators should additionally include a **cancellation-heavy dataset** where evaluation order visibly affects the result. The pattern from §1.2, scaled to large N:

```
// Cancellation stress: [+1e16, +1.0, -1e16, +1.0, ...] repeated N/4 times
std::vector<double> data;
for (size_t i = 0; i < N/4; ++i) {
    data.push_back(+1e16);
    data.push_back(+1.0);
    data.push_back(-1e16);
    data.push_back(+1.0);
}
```

For this dataset, the demonstrators should show all three of:

1. **`std::reduce` is non-deterministic:** repeated runs (or varying thread counts) produce different hex values, because the scheduler changes the effective parenthesization.
2. **`canonical_reduce` with fixed `L` is deterministic:** repeated runs produce identical hex values, because the tree is fixed.

3. **Different `L` values produce different results:** confirming that the topology coordinate controls the abstract expression and is not merely an implementation hint.

This triple is the paper's entire motivation in one test output. If the PRNG dataset is the "does the implementation work?" test, the cancellation dataset is the "does the facility matter?" test.

K.4 Recommended Compiler Explorer settings

K.4.1 x86 demonstrator ([GB-x86-AVX2])

Compiler: x86-64 clang (trunk) (or GCC)

Recommended flags (for running):

```
-O3 -std=c++20 -ffp-contract=off -fno-fast-math
```

K.4.2 AArch64 demonstrator ([GB-NEON])

Compiler: AArch64 clang (trunk) (or GCC)

Recommended flags (for running):

```
-O3 -std=c++20 -march=armv8-a -ffp-contract=off -fno-fast-math
```

If `std::execution::par` is unavailable or unreliable in the CE environment, the demonstrator can be built with:

```
-DNO_PAR_POLICIES=1
```

K.5 Interpreting the performance tables (gross impacts)

The demonstrators typically report: - `std::accumulate` as a deterministic, sequential baseline; - `std::reduce` variants (no policy, seq, unseq, and optionally par*) as "existing practice" comparators; - deterministic/canonical narrow and wide presets.

Readers should interpret the results as: - **Gross cost of structure:** overhead (or sometimes speedup) relative to `std::reduce(seq)` under similar FP settings. - **Configuration sensitivity:** narrow vs wide spans can trade off cache behavior, vectorization, and bandwidth. - **Non-authoritative:** results on CE do not replace proper benchmarking; they are a convenience for quick inspection.

K.6 Relationship to repository evidence

The accompanying repository (referenced in the paper's artifacts section) contains: - controlled micro-benchmarks on pinned CPUs (repeatable measurements), - multi-threaded execution model comparisons, - and a CUDA demonstrator evaluating the same canonical expression (for chosen topology) to support heterogeneous "golden result" workflows.

[*Note:* The paper's semantic contract is independent of any specific demonstrator; these artifacts exist to make the semantics tangible and reviewable. —*end note*]

[*Note:* These demonstrators depend on specific compiler versions and Compiler Explorer VM configurations available at the time of writing. The normative specification is §4; demonstrator links provide supplementary illustration only. If a link becomes unavailable or produces results inconsistent with the published golden values due to toolchain changes, the specification remains unaffected. —*end note*]

Appendix L: Ranges Compatibility (Informative)

This appendix is not part of the proposal. It records design considerations for eventual C++20/23 ranges overloads that expose the semantics of §4 in a composable way.

L.1 A range surface does not change the semantic contract

A range-based surface would not change the semantic contract: for a fixed topology selector (`L` , or span `M` interpreted as `M→L`), input order, and `binary_op` , the returned value is as-if evaluating the canonical expression defined in §4.

This paper intentionally defers the ranges surface to keep first discussion focused on the semantic contract (see §2). Appendix J.6 contains a straw-man sketch.

L.2 Determining N without hidden allocation

The canonical expression in §4 is defined over N elements in a fixed input order. A range-based surface therefore needs a way to determine N and to preserve the element order used by the expression.

Three implementation strategies exist:

- **sized_range:** obtain N in O(1) via `ranges::size(r)`.
- **Non-sized, multipass (forward_range):** determine N via a counting pass (e.g., `ranges::distance(r)`), then evaluate the canonical expression in a second pass. This avoids allocation but may traverse the range twice.
- **Single-pass (input_range):** the count is not available without consuming the range; this specification does not support single-pass ranges without explicit materialization by the caller.

One conservative design point (for a subsequent API revision) is to avoid implicit allocation:

- **Sequential/reference overloads:** accept `forward_range` and permit a counting pass when N is not otherwise known.
- **Execution-policy overloads:** require a stronger range category (e.g., `random_access_range`) and may also require `sized_range`, keeping buffering explicit.

L.3 Working with non-sized, single-pass sources

For sources that are single-pass (or otherwise cannot be traversed twice), users requiring canonical reduction can materialize explicitly:

```
auto filtered = data
  | std::views::filter(pred)
  | std::ranges::to<std::vector>();

auto r = std::ranges::canonical_reduce<std::canonical_span_small>(
  filtered,
  0.0,
  std::plus<>{});
```

This makes allocation explicit and keeps the cost model under user control.

L.4 Projection parameter

This paper does not attempt to design a projection parameter. Users can express projection by composing with `views::transform`, keeping the algorithm surface orthogonal and consistent with ranges composition:

```
auto sum = std::ranges::canonical_reduce<std::canonical_span_small>(
  data | std::views::transform([](const auto& x) { return x.value; }),
  0.0,
  std::plus<>{});
```

[*Note:* This appendix is informative. It does not commit the proposal to a particular ranges overload set; it documents constraints and trade-offs to inform a subsequent API-focused revision. *—end note*]

Appendix M: Detailed Motivation and Use Cases (Informative)

This appendix is not part of the proposal. It collects the longer use-case narratives (with examples) to keep the main document focused on the semantic contract in §4.

The following use cases illustrate the value of run-to-run stable parallel reduction.

[*Note:* Code examples use a placeholder spelling. No specific API is proposed in this paper. *—end note*]

M.1 CI Regression Testing

```
// Test that fails intermittently with std::reduce
double baseline = load_golden_value("gradient_sum.bin"); // Previously computed
```

```
double computed = std::reduce(std::execution::par,
              gradients.begin(), gradients.end(), 0.0);
EXPECT_DOUBLE_EQ(baseline, computed); // FAILS: result varies by thread scheduling

// With canonical_reduce_lanes: no run-to-run variation
double computed = canonical_reduce_lanes<8>(          // L = 8 (or equivalently M = 64 bytes for double)
  gradients.begin(), gradients.end(), 0.0);
EXPECT_DOUBLE_EQ(baseline, computed); // PASSES: expression structure is fixed
// (baseline must also have been computed with same L on same platform)
```

Value: Eliminate spurious test failures caused by run-to-run variation from unspecified reduction order. Within a consistent build/test environment, the returned value is stable across invocations.

M.2 Distributed Training Checkpoints

```
// Machine learning gradient aggregation
// Expression structure fixed by the chosen lane count L, enabling reproducible checkpoints
auto gradient_sum = canonical_reduce_lanes<8>(
  local_gradients.begin(), local_gradients.end(), 0.0);

// Later, on same or equivalent hardware:
auto restored_sum = canonical_reduce_lanes<8>(
  restored_gradients.begin(), restored_gradients.end(), 0.0);

// Reproducible: same inputs + same L + the same floating-point evaluation model → same result
// No longer subject to thread-count or scheduling variation
```

Value: Enable checkpoint/restore with reproducible gradient aggregation, eliminating run-to-run variation from unspecified reduction order. Cross-platform restore requires matching floating-point semantics.

M.3 Regulatory Audit Trails

```
// Financial risk calculation that must be reproducible for auditors
struct RiskCalculation {
  static constexpr size_t L = 8; // Documented in audit trail

  double compute_var(const std::vector<double>& scenarios) {
    return canonical_reduce_lanes<L>(
      scenarios.begin(), scenarios.end(), 0.0,
      [](double a, double b) { return a + b * b; });
  }
};

// Auditor can reproduce exact result years later with same L
```

Value: Meet regulatory requirements for reproducible risk calculations (Basel III, Solvency II).

M.4 Scientific Reproducibility

```
// Climate model energy conservation check
// Paper claims: "Energy drift < 1e-12 per century"
// Reviewers must reproduce this result
auto total_energy = canonical_reduce_lanes<8>(
  grid_cells.begin(), grid_cells.end(), 0.0,
  [](double sum, const Cell& c) { return sum + c.kinetic + c.potential; });

// Published with: "Results computed with L=8, compiled with -ffp-contract=off"
// Reviewer with the same L and evaluation-model settings gets same result
```

Value: Enable peer verification of published computational results when floating-point evaluation model is documented and matched.

M.5 Exascale HPC with Kokkos

```
// DOE exascale application using Kokkos
// Current Kokkos parallel_reduce is non-deterministic
// Today: Results vary across runs, making debugging nearly impossible
Kokkos::parallel_reduce("EnergySum", N, KOKKOS_LAMBDA(int i, double& sum) {
  sum += compute_energy(i);
}, total_energy);
```

```
// With standardized canonical reduction semantics, frameworks could expose a canonical mode
// (illustrative — actual Kokkos API would be determined by Kokkos maintainers)
Kokkos::parallel_reduce<Kokkos::CanonicalLanes<8>>("EnergySum", N, KOKKOS_LAMBDA(int i, double& sum) {
  sum += compute_energy(i);
}, total_energy);

// Reproducible across runs on same platform
// Cross-platform reproducibility requires matching FP semantics
```

Value: Enable reproducible physics in production HPC codes, at least within a consistent execution environment.

M.6 Cross-Platform Development

```
// Game physics: aim for consistency between client platforms
constexpr size_t PHYSICS_L = 16; // 16 lanes (M = 64 bytes for float)

float compute_collision_impulse(const std::vector<Contact>& contacts) {
  return canonical_reduce_lanes<PHYSICS_L>(
    contacts.begin(), contacts.end(), 0.0f,
    [](float sum, const Contact& c) { return sum + c.impulse; });
}

// Expression structure is fixed.
// Cross-platform match requires controlling floating-point evaluation model
// (e.g., disabling FMA, matching rounding modes)
```

Value: Fix expression structure as one component of cross-platform consistency. Full cross-platform bitwise identity additionally requires matching hardware-level FP behaviors (e.g., contraction, intermediate precision, and subnormal handling), which are outside the scope of this proposal.

M.7 Reference and Debugging Mode

A practical requirement for reproducibility is the ability to validate and debug operator logic on a single thread while preserving the same abstract expression as production parallel execution. For that reason, if an eventual Standard Library API is adopted, the overload without an ExecutionPolicy should be specified to evaluate the identical canonical expression tree as the execution-policy overloads for the same L and inputs.

This “reference mode” enables: - Reproducing “golden results” for audit or regression testing on a single thread, - Debugging `binary_op` with conventional tools while guaranteeing expression-equivalence to the parallel workload, - Cross-platform verification (e.g., CPU verification of accelerator-produced results) when evaluation models are aligned.

[*Note:* This paper does not propose a specific API shape; this subsection records a design requirement that follows from the semantic goal. —*end note*]

Appendix N: Multi-Threaded Implementation via Ordered State Merge (Informative)

This appendix describes a multi-threaded implementation strategy that achieves parallel execution while preserving bitwise identity with the single-threaded canonical compute sequence. A reference implementation is available at **[GB-x86-MT]**.

N.1 Overview

The single-threaded shift-reduce algorithm (§4.2) processes blocks sequentially, maintaining a binary-counter stack that implicitly encodes the canonical pairwise tree. To parallelize this while preserving determinism, we observe that:

1. The stack state after processing any prefix of blocks is a complete representation of that prefix’s reduction
2. Two stack states can be merged to produce the state that would result from processing their blocks sequentially
3. Power-of-2 aligned partition boundaries ensure merges are algebraically equivalent to sequential processing

This leads to a three-phase algorithm: parallel local reduction, deterministic merge, and final fold.

N.2 Stack State Representation

An **ordered reduction state** summarizes a contiguous range as an ordered sequence of fully reduced power-of-two blocks.

For a contiguous range `R`, the state may be viewed as a sequence:

```
[B0, B1, ..., Bm]
```

where each `Bi` is the canonical reduction of a contiguous subrange of `R`, the subranges are disjoint and appear in strictly increasing stream order, each block size is `2^k`, and the block sizes are strictly decreasing. The concatenation of these subranges equals `R`.

The stack/bucket representation below is one concrete way to store this ordered block sequence and to implement the canonical "coalesce equal-sized adjacent blocks" rule used by shift-reduce.

A **stack state** compactly represents a partially reduced sequence as a collection of buckets:

```
        template<size_t L>
        struct StackState {
            static constexpr size_t MAX_DEPTH = 32;

            double buckets[MAX_DEPTH][L];  // buckets[k] holds 2^k blocks worth of reduction
            size_t counts[MAX_DEPTH];      // lane counts (L for full, <L for partial)
            uint32_t mask;                 // bit k set iff buckets[k] is occupied
        };
```

Invariant: If `mask` has bits set at positions $\{k_0, k_1, \ldots, k_m\}$ where $k_0 < k_1 < \ldots < k_m$, then the state represents a prefix of length:

```
num_blocks = 2^k₀ + 2^k₁ + ... + 2^kₘ
```

Each `buckets[kᵢ]` holds the fully reduced L-lane vector of a contiguous block of 2^{k_i} input blocks. **Lower-indexed buckets represent more recent (rightward) blocks; higher-indexed buckets represent older (leftward) blocks.**

N.3 The Push Operation

The push operation incorporates a new L-lane vector into the stack state, implementing binary-counter carry propagation:

```
        void push(StackState& S, const double* vec, size_t count, size_t level = 0) {
            double current[L];
            copy(vec, current, count);
            size_t current_count = count;

            while (S.mask & (1u << level)) {
                // Bucket exists: combine (older on left)
                current = combine(S.buckets[level], S.counts[level],
                                  current, current_count);
                S.mask &= ~(1u << level);  // Clear bucket
                ++level;                    // Carry to next level
            }

            S.buckets[level] = current;
            S.counts[level] = current_count;
            S.mask |= (1u << level);
        }
```

Key property: Processing element i triggers carries corresponding to the trailing 1-bits in the binary representation of i. This exactly mirrors the canonical pairwise tree structure.

N.4 The Fold Operation

After all blocks are pushed, the stack is collapsed to a single vector:

```
        double* fold(const StackState& S) {
            double acc[L];
            size_t acc_count = 0;
            bool have = false;

            // CRITICAL: Iterate low-to-high, bucket on LEFT
            for (size_t k = 0; k < MAX_DEPTH; ++k) {
                if (!(S.mask & (1u << k))) continue;

                if (!have) {
```

```
                acc = S.buckets[k];
                acc_count = S.counts[k];
                have = true;
            } else {
                // Higher bucket (older) goes on LEFT
                acc = combine(S.buckets[k], S.counts[k], acc, acc_count);
            }
        }
        return acc;
    }
```

Critical detail: The fold must iterate from low to high indices, placing each bucket on the **left** of the accumulator. This reconstructs the canonical tree's final merges where older (leftward) partial results combine with newer (rightward) ones.

N.5 Power-of-2 Aligned Partitioning

For multi-threaded execution, we partition the block index space among threads. **The critical requirement is that partition boundaries fall on power-of-2 aligned indices.**

Definition: A block index b is *k-aligned* if b is a multiple of 2^k.

Observation: After processing blocks [0, b) where b = m × 2^k, the stack state has no occupied buckets below level k.

The binary representation of b has zeros in positions 0 through k-1. Since bucket[j] is occupied iff bit j is set in the count of processed blocks, buckets 0 through k-1 are empty.

Consequence: When thread boundaries are power-of-2 aligned, the "receiving" state A has no low-level buckets that could collide incorrectly with the "incoming" state B's buckets during merge.

N.6 Partition Strategy

Given B total blocks and T threads, choose chunk size C = 2^k where k is the largest integer such that B / 2^k ≥ T:

```
    size_t choose_chunk_size(size_t num_blocks, size_t T) {
        if (num_blocks <= T) return 1;
        size_t k = bit_width(num_blocks / T) - 1;
        return size_t{1} << k;
    }
```

This yields ceil(B / C) chunks, each containing C blocks (except possibly the last). Thread t processes chunks assigned to it, producing a local stack state for each.

Example: For B = 1000 blocks and T = 4 threads, C = 256 (2^8):

Thread	Chunks	Blocks	Output
0	0	[0, 256)	Single bucket at level 8
1	1	[256, 512)	Single bucket at level 8
2	2	[512, 768)	Single bucket at level 8
3	3	[768, 1000)	Buckets at levels {7, 6, 5, 3} for remainder 232

Full chunks produce exactly one bucket at level k (since 2^k blocks reduce to one bucket). The remainder chunk (232 = 128 + 64 + 32 + 8) naturally decomposes via shift-reduce into multiple buckets.

N.7 The Merge Operation

To combine two stack states where A represents an earlier (leftward) portion of the sequence and B represents a later (rightward) portion:

```
    void merge_into(StackState& A, const StackState& B) {
        // Process B's buckets in increasing level order
        for (size_t k = 0; k < MAX_DEPTH; ++k) {
            if (B.mask & (1u << k)) {
                // Push B's bucket into A, starting at level k (NOT level 0!)
                push(A, B.buckets[k], B.counts[k], k);
```

```
            }
        }
    }
```

Critical detail: `push(A, B.buckets[k], B.counts[k], k)` starts at level k, not level 0. This correctly reflects that B.buckets[k] represents 2^k already-reduced blocks.

Why low-to-high order: Within B's stack state, lower-indexed buckets represent more recent (rightward) elements within B's range. Processing them first ensures they combine before higher-indexed (older) buckets, matching the canonical left-to-right order.

N.8 Correctness argument

Claim: The multi-threaded algorithm produces a result bitwise identical to single-threaded shift-reduce for any number of threads T ≥ 1 and any input size N.

Argument:

1. **Shift-reduce correctness:** Single-threaded shift-reduce evaluates the canonical pairwise-with-carry tree. The push operation's carry pattern exactly mirrors binary increment, corresponding to completing subtrees of size 2^k.
2. **Alignment property:** For a prefix of length b = m × 2^k blocks, the stack state has no occupied buckets below level k (see observation above).
3. **Merge correctness:** `merge_into(A, B)` produces the same state as processing A's blocks followed by B's blocks sequentially. By the alignment property, when B starts at an aligned boundary, A has no buckets below the alignment level. B's buckets, when pushed at their native levels, trigger exactly the carries that would occur from processing B's blocks after A's blocks. The low-to-high iteration order preserves within-B ordering.
4. **Combining these:** Thread boundaries are aligned, so merge correctness applies to each merge. The left-to-right merge order (thread 0, then 1, then 2, ...) matches sequential block order.

N.9 Complete Algorithm

```
template<size_t L>
double deterministic_reduce_MT(const double* input, size_t N, size_t T) {
    const size_t num_blocks = (N + L - 1) / L;
    const size_t C = choose_chunk_size(num_blocks, T);
    const size_t num_chunks = (num_blocks + C - 1) / C;

    vector<StackState<L>> states(num_chunks);

    // Phase 1: Parallel local reductions
    parallel_for(0, num_chunks, [&](size_t chunk) {
        size_t b0 = chunk * C;
        size_t b1 = min(b0 + C, num_blocks);
        states[chunk] = replay_range(input, N, b0, b1);
    });

    // Phase 2: Serial merge (left-to-right order)
    for (size_t i = 1; i < num_chunks; ++i) {
        merge_into(states[0], states[i]);
    }

    // Phase 3: Final fold + cross-lane reduction
    double* lane_result = fold(states[0]);
    return cross_lane_pairwise_reduce(lane_result);
}
```

N.10 Complexity Analysis

The semantics do not require parallel scalability. This section explains that the canonical expression *admits* scalable multi-threaded realizations.

Phase	Work	Span (critical path)
Local reduction	`O(N)` total	`O(N/T)`
Ordered merge (sequential)	`O(T · log N)`	`O(T · log N)`
Ordered merge (tree-structured)	`O(T · log N)`	`O(log T · log N)`
Final fold	`O(log N)`	`O(log N)`

Total work: `O(N + T·log N)` (typically `O(N)` when `T << N`)

Span: With a tree-structured merge, `O(N/T + log T · log N)`.

Space: `O(T · L · log N)` for per-thread states (typically a few KB per thread, depending on `L`).

N.11 SIMD Optimization: 8-Block Unrolling

A significant optimization reduces stack operation overhead by processing 8 blocks at once:

```
        // Instead of pushing one block at a time:
        for (size_t b = 0; b < num_blocks; ++b) {
            push(S, load_block(b), L, 0);  // O(num_blocks) stack operations
        }

        // Process 8 blocks in one SIMD reduction, push at level 3:
        for (size_t g = 0; g < num_groups_of_8; ++g) {
            double result[L];
            reduce_8_blocks_simd(input + g * 8 * L, result);  // 8 blocks → 1 vector
            push(S, result, L, 3);  // Start at level 3 (since 8 = 2^3)
        }
```

This reduces stack operations from N/L to N/(8L), yielding substantial performance improvements. The reference implementation achieves throughput exceeding `std::reduce` while maintaining full determinism.

N.12 Performance Observations

The reference implementation at **[GB-x86-MT]** demonstrates:

Variant	Throughput	vs `std::accumulate`
`std::accumulate`	5.4 GB/s	baseline
`std::reduce`	21.4 GB/s	+297%
Deterministic ST (L=16, 8-block unroll)	**26.5 GB/s**	**+391%**
Deterministic MT (L=16, T=2)	21.2 GB/s	+293%

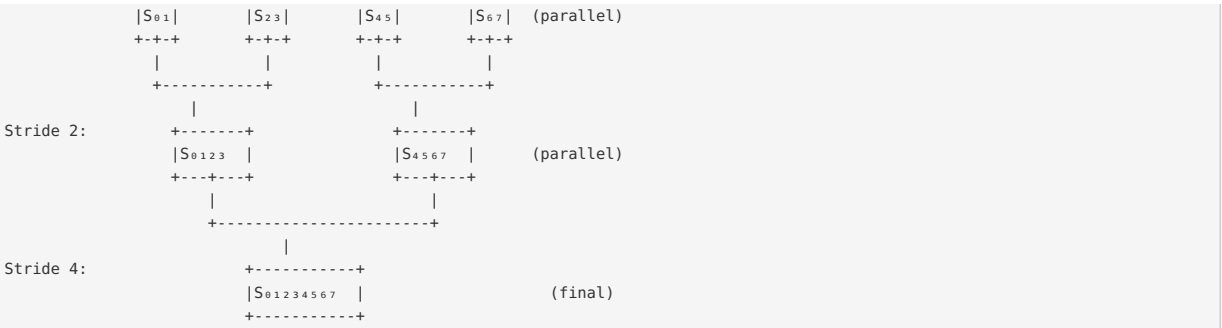
Key finding: With proper SIMD optimization, deterministic reduction is **faster** than non-deterministic `std::reduce` while guaranteeing bitwise reproducibility. The 8-block unrolling minimizes stack overhead, and the interleaved lane structure enables efficient vectorized combines.

N.13 Implementation Notes

Thread pool reuse: For production implementations, reuse a thread pool rather than spawning threads per invocation. The reference limits thread creation for Godbolt compatibility.

Parallel merge (optional): The merge phase can itself be parallelized using a tree structure:

```
Initial:    +---+ +---+ +---+ +---+ +---+ +---+ +---+ +---+
            |S₀ | |S₁ | |S₂ | |S₃ | |S₄ | |S₅ | |S₆ | |S₇ |
            +-+-+ +-+-+ +-+-+ +-+-+ +-+-+ +-+-+ +-+-+ +-+-+
              |     |     |     |     |     |     |     |
              +-----+     +-----+     +-----+     +-----+
                |           |           |           |
Stride 1:     +---+       +---+       +---+       +---+
```



This reduces the merge critical path from O(T) to O(log T), though for typical T values the improvement is marginal compared to the dominant O(N/T) local reduction phase.

Memory efficiency: Stack states are fixed-size (O(L × log N) per state) with no heap allocation required during the hot path. The merge operation modifies A in-place, requiring only register-level temporaries for the carry chain.

N.14 Summary

The multi-threaded stack ordered state merge algorithm achieves:

Property	Guarantee
Determinism	Bitwise identical to single-threaded for any T
Correctness	Provably equivalent to canonical pairwise tree
Efficiency	O(N) work, O(N/T + T log N) span
Practicality	Demonstrated faster than `std::reduce` with SIMD

The key insights enabling this approach are:

1. Power-of-2 aligned partitioning eliminates boundary ambiguity
2. Stack states are complete representations of partial reductions
3. Pushing at native levels during merge preserves the canonical tree structure
4. 8-block SIMD unrolling amortizes stack overhead

This demonstrates that deterministic parallel reduction is not merely *feasible* but can be *optimal* — the constraint of a fixed evaluation order, far from being a performance liability, enables aggressive SIMD optimization that outperforms unconstrained implementations.

Appendix X: Recursive Bisection (“Balanced”) Tree Construction (Informative)

This appendix records a previously-considered alternative canonical tree construction based on recursive bisection. It is **not** part of the normative contract in §4; this paper specifies iterative pairwise (§4.2.3) as the sole canonical tree definition.

X.1 Definition

Define `CANONICAL_TREE_EVAL_RECURSIVE(op, Y[0..k))`, where `k >= 1` and each `Y[t]` is in `maybe<A>`:

```
CANONICAL_TREE_EVAL_RECURSIVE(op, Y[0..k)):
    if k == 1:
        return Y[0]
    let m = floor(k / 2)
    return COMBINE(op,
        CANONICAL_TREE_EVAL_RECURSIVE(op, Y[0..m)),
        CANONICAL_TREE_EVAL_RECURSIVE(op, Y[m..k))
    )
```

X.2 Equivalence on power-of-two sizes

For `k = 2^n` (k = 2, 4, 8, …), recursive bisection and iterative pairwise produce identical trees, and therefore define identical abstract expressions.

X.3 Differences on non-power-of-two sizes

For non-power-of-two `k`, the trees differ in structure (but maintain the same asymptotic depth bounds).

Example (`k = 5`), writing `+` for `op`:

- Recursive bisection:
  - `(e0 + e1) + (e2 + (e3 + e4))`
- Iterative pairwise:
  - `((e0 + e1) + (e2 + e3)) + e4`

X.4 Rationale for selecting iterative pairwise

This paper selects iterative pairwise as the sole canonical tree definition for the following reasons:

- **Industry alignment:** SIMD/GPU deterministic reductions commonly use adjacent pairwise combination patterns.
- **Direct hardware mapping:** iterative pairing corresponds directly to SIMD-lane and warp-lane pairing operations.
- **Adoption continuity:** existing deterministic implementations can conform without structural change.
- **Specification focus:** once a single canonical tree is selected, retaining alternatives in the normative contract increases complexity without adding semantic value.

No other grouping is permitted.