P4016R0 — Canonical Parallel Reduction: A Fixed Expression Structure for Run-To-Run Consistency

Document Number:	P4016R0
Date:	2026-02-23
Target:	SG6 (Numerics), LEWG
Reply-to:	Andrew Drakeford <andrew.drakeford@gmail.com>
Project:	Programming Language C++

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 differ across conforming evaluations.

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 (hereafter, the *abstract expression*) — 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.

The approach generalizes a technique already present in the Standard Library: `std::accumulate` obtains determinism by fixing its 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 key decision points are captured in the **Polls for LEWG Direction** (immediately following this 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	Polls, §1, §2, §4, §5	§3 (design rationale)	Appendices
Implementer	Polls, §4, §5, Appendix B, N	§3.8 (tree shape rationale)	
Numerical analyst	Polls, §6, §4, §5, Appendix C, K	Appendix P (throughput data)	

Direction requested (this R0). This paper asks whether the standard library should provide a **canonical reduction semantic contract**: for a given input order and semantic parameter `L`, the algorithm denotes a **single fixed abstract expression** (specified in §4–§5). Implementations may execute using threads/SIMD/work-stealing/GPU kernels, but must produce results **as-if evaluating that abstract expression**.

Non-goals. This revision does not standardize arithmetic determinism (rounding mode, contraction, FTZ/DAZ, etc.), and it does not propose final API naming, placement, or overload sets.

Where to look. The normative semantics are in §4–§5; the remainder provides motivation, design-space discussion, and validation evidence.

Polls for LEWG Direction

[*Note:* The polls below are written for LEWG review. When presented to SG6, the chair may rephrase polls to focus on numerics-specific concerns — for example, whether the canonical tree structure is appropriate for floating-point error analysis, whether the lane count parameter is a suitable topology coordinate for numerical reproducibility, and whether the `GENERALIZED_NONCOMMUTATIVE_SUM`/`GENERALIZED_SUM` distinction is correctly handled. *—end note*]

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

Poll 1 — Semantics-first scope (direction)

Question: We agree that the C++ Standard Library lacks a parallel reduction facility with a specified abstract expression, that this is a gap worth closing, and that validating the expression structure before committing to API design is the right approach for this facility. We direct the authors to focus this revision on the semantic model and return with a separate paper for API surface if the semantic direction is positive.

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

Poll 2 — Approve the semantic framework (fixed abstract expression)

Question: We agree that a canonical reduction facility should define a single fixed abstract expression for a given input order and semantic parameters (including lane count `L`), and that the two-stage structure in §4 (interleaved lane partition, per-lane canonical tree, cross-lane canonical tree) is a suitable framework for this contract. Implementations may execute using threads/SIMD/work-stealing/GPU kernels, but must produce results as-if evaluating the specified abstract expression. This poll is about the framework; the specific tree shape is considered separately in Poll 2A.

[*Note:* Init placement is specified in §4.5; if desired, LEWG can poll init placement separately. *—end note*]

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

Poll 2A — Tentative preference for the canonical tree shape

Question: Assuming Poll 2 passes, we have a tentative preference that the canonical tree shape be the iterative pairwise ("binary-counter / carry") tree described in §4.2.3, on the grounds that it is deterministic, SIMD/MT/GPU implementable, and has clean ragged-tail handling without requiring identity elements.

[*Note:* Recursive bisection ("balanced") was considered as an alternative tree construction. It is documented in Appendix O (informative) for comparison and historical record. *—end note*]

If consensus is not reached on tree shape, the two-stage semantic framework (Poll 2) remains valid; authors will return with a follow-up comparing tree shape alternatives.

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

Poll 3 — Proceed to API surface (conditional direction)

Question: If Poll 2 is positive, we direct the authors to return with a proposed API surface (names, signatures, constraints, header placement), preserving the Poll 2 semantic contract.

Scope of expected API revision:

- New algorithm and/or execution policy approach
- Proposed naming and header placement
- Initial overload set (iterators/ranges, with/without execution policy)
- Constraints and mandates
- Topology selection mechanism (including any named presets and defaulting rules)

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

0. Executive Overview

There is a gap in the C++ Standard Library. Sequential reductions (`std::accumulate`) have a fully specified abstract expression structure — and their returned value is defined by that structure — but cannot be parallelized. Parallel reductions (`std::reduce`) can be parallelized but leave the expression unspecified: for non-associative operations such as floating-point addition, both the order in which operands are combined and how they are grouped (parenthesized) may produce different observable results for the same input solely due to execution strategy selection.

This proposal addresses that gap. It specifies an interleaved binary reduction tree in two stages. The input is distributed across `L` parallel lanes (where `L` is a user-chosen lane count, typically matching SIMD width); each lane is independently reduced by an iterated near-balanced binary tree (Stage 1), and the `L` lane results are combined by the same tree rule (Stage 2). The lane count `L` is a semantic topology coordinate; choosing it to match hardware SIMD width is optional guidance, not required. The tree is built bottom-up, permitting implementations to begin evaluation incrementally as elements arrive; however, the returned value is defined with respect to the completed input range of length `N`. Streaming is an implementation technique only; the specified abstract expression is defined over the final input sequence of length `N`. Fixing the parenthesization and operand order gives a fixed abstract expression (expression identity) that makes results reproducible for a fixed input order, topology coordinate `L`, and floating-point evaluation model. Implementations remain free to schedule evaluation using threads, SIMD, or GPU kernels, provided the returned value matches the specified expression. API design is deferred; this paper seeks validation of the expression structure only. No changes to existing algorithms are proposed in this revision.

What this paper adds

- Fixed abstract expression for parallel reduction, parameterized by lane count `L`.
- Two-stage semantic structure: per-lane tree + cross-lane tree.
- Execution remains unconstrained; only the expression is specified.

What this paper does not do

- Guarantee cross-architecture bitwise identity (requires matching `L` and floating-point model; see §6).
- Constrain execution, scheduling, or parallelism.
- Propose a final API surface.

---

1. The Semantic Gap in C++ Reduction Facilities

The table in §1.1 classifies existing reduction facilities by expression ownership — whether the Standard fully specifies the abstract expression or leaves it (partially or wholly) unspecified. The proposed canonical reduction completes the family by providing a specified expression that admits parallel evaluation.

[*Note:* In this paper, **canonical** refers strictly to the abstract expression structure. The terms "canonical expression," "canonical tree," and "canonical reduction" are used interchangeably to refer to the fixed abstract expression defined in §4. Implementations retain freedom in evaluation schedule; the facility provides a stable, specified baseline topology. The term "canonical" is descriptive and carries no relation to `std::filesystem::canonical`; the facility may be named `specified_reduce`, `deterministic_reduce`, or otherwise in an API revision. *—end note*]

This work was presented to SG14 (Low Latency/Games/Embedded/Financial Trading), who encouraged a formal paper. The approach has been validated with working implementations across x86 AVX2, ARM NEON, and CUDA (Appendix K), producing expression-identical results for fixed topology coordinates (bitwise-identical under sufficiently aligned floating-point evaluation environments; see §6). A summary comparison with HPC frameworks and industry practice appears in §7.

1.1 Why This Needs to Be in the Standard Library

Parallel algorithms literature commonly identifies a small set of fundamental primitives for combining elements of a sequence, notably reduction and scan (prefix computation) [Blelloch1989]. These primitives are defined in terms of combining values using a binary operation over an ordered sequence.

The C++ Standard Library provides corresponding facilities:

- `std::accumulate`, which combines all values in a sequence,
- `std::inclusive_scan` and `std::exclusive_scan`, which compute prefix combinations of values, and
- `std::reduce`, which combines all values in a sequence and permits parallel execution.

These facilities provide different guarantees about the order in which operations are performed. `std::accumulate` specifies that values are combined from left to right, fully determining the abstract expression. `std::inclusive_scan` and `std::exclusive_scan` preserve operand order but permit reassociation (arbitrary parenthesization). `std::reduce`, by contrast, permits implementations to both reorder operands and reassociate to enable parallel execution.

This leads to the following distinction:

[*Note:* Terminology in this table is intended to match the specification models defined in [numerics.defns] (`GENERALIZED_NONCOMMUTATIVE_SUM` and `GENERALIZED_SUM`). *—end note*]

Facility	Specification model	Operand order	Parenthesization	Unique expression?
`std::accumulate`	Sequential left fold	Fixed (left-to-right)	Fixed (left fold)	Yes
`std::ranges::fold_left` / `fold_right`	Sequential fold	Fixed	Fixed	Yes
`std::inclusive_scan`	`GENERALIZED_NONCOMMUTATIVE_SUM`	Fixed (left-to-right)	Unspecified	No — reassociation permitted
`std::exclusive_scan`	`GENERALIZED_NONCOMMUTATIVE_SUM`	Fixed (left-to-right)	Unspecified	No — reassociation permitted
`std::reduce`	`GENERALIZED_SUM`	Unspecified	Unspecified	No — reordering + reassociation permitted
`std::transform_reduce`	`GENERALIZED_SUM`	Unspecified	Unspecified	No — reordering + reassociation permitted
Proposed `canonical_reduce`	Canonical reduction expression (§4)	Fixed (lane assignment)	Fixed (canonical tree)	Yes, for a given lane count L

The Standard already provides expression ownership for sequential reduction (`accumulate`) and preserves operand order for parallel prefix computation (scan). Parallel reduction is the remaining primitive for which no specified expression exists. This proposal completes that family.

The standard defines the `GENERALIZED_NONCOMMUTATIVE_SUM` and `GENERALIZED_SUM` specification models in [numerics.defns]. `GENERALIZED_NONCOMMUTATIVE_SUM` preserves operand order but allows arbitrary parenthesization (the split point K is unspecified). `GENERALIZED_SUM` additionally permits operand reordering.

[*Note:* Section references such as [numerics.defns] refer to the C++ Working Draft; draft number at the time of writing: N5032. *—end note*]

Sequential reduction specifies a unique expression. Parallel prefix algorithms (scan) preserve operand order but admit multiple conforming parenthesizations. Parallel reduction (`std::reduce`) is unique among these primitives: it specifies neither operand order nor parenthesization — two independent semantic freedoms that together make the abstract expression fully unspecified.

For operations that are not associative, such as floating-point addition, different orders of combination may produce different results. Controlling the floating-point evaluation model alone is therefore insufficient to ensure reproducible parallel reduction: without a specified order of combination, different valid implementations of `std::reduce` may compute different results for the same input.

This proposal introduces a parameter L that selects a specified grouping and operand order for combining values during parallel reduction. For a fixed input sequence and chosen L, conforming implementations must produce the same result as if the values were combined in that specified order, while remaining free to evaluate those combinations using parallel execution strategies. Providing such a facility enables portable parallel reduction with well-defined evaluation semantics, analogous to those already provided for prefix algorithms.

Why this cannot be achieved outside the standard. This facility cannot be obtained merely as an external wrapper around existing standard components. `std::reduce` explicitly permits reassociation (and, under `GENERALIZED_SUM`, operand reordering); no wrapper, view, or range adaptor can remove those permissions while still using `std::reduce` as the semantic basis. A standalone library can of course provide its own fixed-tree reduction algorithm, but it cannot retroactively change what the Standard specifies for `std::reduce`, nor can it provide a single portable *standard* contract that all standard execution mechanisms are required to honor. Standardization provides that common, portable semantic foundation — a specified abstract expression — that library and framework authors can target.

Heterogeneous execution. Modern C++ programs increasingly execute on heterogeneous platforms — CPU threads, GPU kernels, and accelerators within a single application. The sender/receiver model (P2300) enables scheduling work to different executors, but without a standardized expression structure, the same reduction dispatched to different devices may produce different results simply because each executor chooses a different grouping. By fixing the abstract expression in the standard, this proposal provides a common expression contract across executors; returned values match when floating-point evaluation conditions are sufficiently aligned and the same topology coordinate `L` is selected (see §6).

1.2 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.

[*Note:* This example is illustrative; the exact numeric outcomes depend on the floating-point evaluation conditions in effect (see §6). *—end note*]

- A **left-to-right fold** (as with `std::accumulate`) groups as `(((((1e16 + 1) + 1) + -1e16) + 1) + 1)` which yields `4` on IEEE-754 double implementations with round-to-nearest, 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 are lost early, yielding `2` under the same rounding mode.

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 stays 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

`std::accumulate` is the clearest precedent. Its specification mandates a left-to-right fold, fully determining the abstract expression. Consequently, 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 expression being evaluated.

This proposal applies the same structural technique to parallel reduction: it standardizes a single canonical expression that admits parallel evaluation. Appendix D analyzes the `std::accumulate` precedent in detail.

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 *algebraic* requirements on `binary_op` (associativity/commutativity/identity). Iterator/type constraints and API surface are deferred; the semantic requirements on `binary_op` are captured normatively in §4–§5.

This facility differs from `std::reduce` composed with an execution policy: `std::reduce` deliberately leaves the abstract expression unspecified regardless of policy (§1), whereas this paper standardizes a single fixed canonical expression for a chosen topology coordinate.

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 define the semantic contract of the facility. Other sections and appendices provide rationale, motivation, and alignment notes with existing Standard Library rules; they are not intended to introduce independent semantic requirements beyond what is stated in §4–§5. 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 reduction expression (fixed abstract expression)
- Parameterization by lane count `L` (§9)
- Invariance guarantees (§5)
- Rationale for design choices

Deferred to later revision (pending LEWG direction):

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

Scan and reduce differ in their degree of nondeterminism. The nondeterminism admitted by scan algorithms is strictly weaker than that admitted by `std::reduce`. Scan preserves the operand sequence and permits only reassociation of the binary operation (via `GENERALIZED_NONCOMMUTATIVE_SUM`). By contrast, `std::reduce` additionally permits reordering of operands (`GENERALIZED_SUM`). A canonical reduction (§4) therefore fixes two dimensions of variation (operand order and parenthesization), while a corresponding scan facility would need to fix only parenthesization for each prefix.

Related algorithms (informative). Because scan algorithms already preserve operand order ([numerics.defns]: `GENERALIZED_NONCOMMUTATIVE_SUM`), applying the tree rule (§4) independently to each prefix yields a deterministic scan without introducing a lane parameter. The missing semantic in scan is therefore limited to fixed parenthesization; the reduction defined in this paper provides the necessary machinery. A future `canonical_scan` would apply a standard-fixed grouping to each prefix in sequence order; unlike reduction, this must not permute operands. Scan is out of scope for this paper.

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 normative definition (§4) is stated in terms of `K = ceil(N/L)`, but `N` need not be known in advance: the iterative pairwise (shift-reduce) evaluation strategy consumes elements incrementally in a single pass, discovering `N` when the input is exhausted and completing the canonical tree at that point. No implicit materialization or allocation is performed by the facility.

However, the expression is defined for the completed range of length `N`; implementations may consume elements incrementally (streaming) provided the returned value equals that of the canonical expression for that final `N` (see §3.8 for architectural discussion). See Appendix L for further discussion of ranges compatibility.

2.3 Exception safety

Informative summary: intended to align with existing rules in `[algorithms.parallel.exceptions]`; the semantic contract for returned values remains §4–§5.

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

- When evaluated without an execution policy, if `binary_op` throws, the exception is propagated to the caller (standard sequential semantics).
- When evaluated with an execution policy (including `execution::seq`, `execution::par`, `execution::unseq`, or `execution::par_unseq`): if execution of `binary_op` exits via an uncaught exception, `std::terminate` is called.

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

Informative summary: intended to be consistent with `[reduce]`; this summary is non-normative (API wording is deferred).

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` (the init combination in §4.5, if present, adds one additional application); absent-operand positions (§4.2.2) do not induce additional applications.

This matches the work complexity required of `std::reduce`. This paper specifies work complexity only; depth and storage are not normative. The 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.

This facility introduces no additional guarantees regarding floating-point rounding behavior, contraction (e.g., FMA), extended precision, or evaluation environment beyond those already permitted by the C++ Standard.

2.6 Constant evaluation

When evaluated in a constant-evaluation context, the returned value is as-if obtained by evaluating the canonical abstract expression defined in §4. Constant evaluation does not require parallelism. The expression is a pure functional specification with no dependence on runtime state, so it is structurally amenable to constant evaluation provided `binary_op` is itself usable in constant expressions. Detailed `constexpr` specification (including iterator constraints) is deferred to the API design paper.

2.7 Freestanding

The canonical reduction requires no heap allocation and uses no heap allocation; typical working storage is O(L · log(N/L)) intermediate values of type A. It is a candidate for freestanding implementations; detailed freestanding specification is deferred to the API design paper.

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 on the execution dimension. They constrain scheduling and concurrency, but do not define grouping/parenthesization semantics for the abstract expression being evaluated. 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 are composed 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. This differs in kind 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 with 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 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 resolve the grouping gap; it moves the source of variation from the algorithm's reassociation freedom to the implementation's topology choice. 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 expression is defined in terms of the **lane count `L`**. For intuition: `L = 1` degenerates to a single tree over the input sequence (no lane interleaving). Larger `L` introduces SIMD/GPU-friendly lane parallelism but still denotes one Standard-defined expression fully determined by `(N, L)`. The normative definition of the two-stage reduction (lane partitioning, per-lane tree, cross-lane 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 have the same number of elements (to within one), so all lanes execute the same 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.

The lane count `L` is the sole topology coordinate that defines the semantics. Earlier explorations considered a byte-span parameter `M` (where `L = M / sizeof(value_type)`), but this creates a portability trap: `sizeof(value_type)` varies across platforms, so the same `M` value silently produces different expressions on different targets. §9.2 discusses this rationale in detail.

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. However, existing industry practice demonstrates the value of partial guarantees — specifically, fixing the computation topology — even when full bitwise identity is not achievable.

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 an important 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, “Getting Started with Conditional Numerical Reproducibility” [IntelCNR].

NVIDIA CUB / Thrust (CUDA ecosystem): NVIDIA’s CUDA reduction libraries include documented techniques and configuration choices intended to improve determinism and reproducibility of reductions (typically by avoiding schedule-dependent combination mechanisms and by selecting reduction strategies with a more stable, implementation-defined combination order). The scope of these guarantees is library- and version-dependent and is not standardized across platforms. See: NVIDIA CUB documentation [NvidiaCUB].

This proposal does not standardize vendor-specific reproducibility modes; it standardizes a portable semantic building block: a fixed abstract reduction expression.

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.

The heterogeneous convergence argument: Each vendor — Intel (CNR), NVIDIA (CUDA reduction libraries), and framework authors (PyTorch, Kokkos) — has independently built a proprietary determinism facility because the C++ Standard provides no canonical expression to target. As P2300 senders/receivers enable heterogeneous dispatch across CPUs, GPUs, and accelerators, this fragmentation worsens: a developer seeking deterministic reduction must use vendor-specific APIs on each target. A standard-defined canonical compute sequence provides the **common mathematical contract** that all implementations — regardless of execution substrate — can converge on. The standard defines *what expression* is computed; vendors retain full freedom in *how* to evaluate it on their hardware.

3.6.3 Conclusion

By fixing the abstract expression structure, this proposal provides an **important building block** for reproducibility — but not a sufficient one. Sufficient conditions depend on the program’s floating-point evaluation model and environment (§6).

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

A range adaptor (View) can change which values are presented to an algorithm and in what iteration order they are produced. However, a view does not and cannot constrain the combination semantics of the consuming algorithm.

`std::reduce` is specified in terms of `GENERALIZED_SUM`, which intentionally permits the implementation to reassociate the expression and (under `GENERALIZED_SUM`) to reorder operands. Consequently, even if a view presents a deterministic sequence, a call to `std::reduce` may still evaluate a different abstract expression than any fixed parenthesization intended by the adaptor.

Therefore, a “deterministic reduction” contract cannot be obtained by composing views with `std::reduce`: it requires an algorithm whose specification owns the reduction expression and defines the returned value as-if evaluating that expression.

```
        // 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 cannot reside in execution policies, views, or executors — the algorithm is the only standard mechanism that can own expression semantics.

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.

The discussion in this subsection concerns implementation strategies only and does not alter the semantic requirement that the returned value be that of the canonical expression defined in §4 for the completed input sequence of length `N`. The specification in §4–§5 does not require incremental evaluation, streaming, or any particular execution strategy.

3.8.1 Candidate comparison

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 comparable pairwise-summation error bounds (see [Higham2002]). 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:

```
Iterative Pairwise (IPR)             Recursive Bisection


           op                                   op
         /    \                               /    \
       op      op                           op      op
      /  \    /  \                         /  \    /  \
    op   op  op  e₆                      e₀   op  op   op
   / \  / \  / \                              / \  / \  / \
  e₀ e₁ e₂ e₃ e₄ e₅                        e₁ e₂ e₃ e₄ e₅ e₆
```

IPR: `((e₀⊕e₁)⊕(e₂⊕e₃))⊕((e₄⊕e₅)⊕e₆)` · RB: `(e₀⊕(e₁⊕e₂))⊕((e₃⊕e₄)⊕(e₅⊕e₆))`

Both trees have depth 3 (= $\lceil \log_2 7 \rceil$). Both perform exactly 6 applications of `binary_op`. The error bounds are comparable (see [Higham2002]). The only difference is which element carries: IPR carries `e₆` (the last element, locally determined), while recursive bisection splits at `floor(7/2) = 3`, changing the pairing of `e₀` (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.

Throughput considerations and regret analysis for this choice are discussed in Appendix P.

3.8.2 Streaming evaluation

Motivation only: nothing in §4–§5 requires streaming or incremental composition; this subsection discusses implementation alignment.

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 the 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 does not naturally compose incrementally without materializing (or at least sizing) the range.
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 be evaluated without materializing the entire sequence is the one structurally aligned with the future of C++ execution. This is an architectural property that determines whether the canonical expression can participate in streaming pipelines; the returned value is nonetheless defined with respect to the final N.

[*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*]

Lane-locality of streaming evaluation. Because lane assignment is determined solely by the input index (`i mod L`), each element belongs to exactly one lane, and the shift-reduce stack maintained for that lane captures the complete canonical subtree for all elements of that lane observed so far. Streaming evaluation therefore preserves the expression: inter-lane ordering is determined only by Stage 2 (§4.4), which is itself evaluated by the same tree rule after all per-lane reductions complete. Chunked or incremental delivery of elements does not alter the abstract expression, provided that elements are presented in increasing input index order.

These observations describe implementation properties of the iterative pairwise formulation. They do not impose added semantic requirements beyond those stated in §4–§5.

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 later 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 the same separation this proposal formalizes for reduction expressions. In the absence of a specified expression, different schedulers may legitimately induce different groupings for a reduction, leading to scheduler-dependent results.

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 expression using different physical strategies, but they must 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 consistency across schedulers when the same canonical expression (fixed `L`) and floating-point evaluation model are used. 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 materialization of the entire sequence prior to evaluation; the returned value is defined with respect to the final sequence length N.

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

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

Mandates: `L >= 1`.

Summary. For input sequence `X[0..N)`, binary operation `op`, and lane count `L`:

1. **Partition** `X` into `L` interleaved lanes by `i mod L`. Element `X[i]` is assigned to lane `i mod L`, preserving input order within each lane.
2. **Stage 1 — Per-lane reduction.** For each lane `j` in `[0, L)`, evaluate `CANONICAL_TREE_EVAL(op, Y_j)` where `Y_j` has `K = ceil(N/L)` positions. Absent trailing positions (when `N` is not a multiple of `L`) propagate without invoking `op`.
3. **Stage 2 — Cross-lane reduction.** Reduce the `L` lane results `R[0..L)` using `CANONICAL_TREE_EVAL(op, R[0..L))`.
4. **Init integration.** If `init` is provided and `N == 0`, return `A{init}`. If `init` is provided and `N > 0`, let `I` be `A{init}` and let `R` be the value extracted from `R_all`; return `op(I, R)`. Otherwise (no `init`, `N > 0`), return the value extracted from `R_all`.

`CANONICAL_TREE_EVAL` is an iterative pairwise (shift-reduce) tree: pair adjacent elements left-to-right, carry any odd trailing element, repeat. The tree shape is fully determined by the element count. Only the abstract expression structure is specified. When evaluated with an execution policy, independent subexpressions may be evaluated concurrently or vectorized, provided the returned value matches that of the canonical expression. The remainder of §4 defines each component precisely.

[*Note:* The semantic contract in §4 is defined in terms of the returned value. The invocation count is specified (§4.2.2). —*end note*]

Sequencing of `binary_op` evaluations. For all overloads, the returned value shall be as if obtained by evaluating the canonical expression defined in §4.

Without an execution policy, the returned value is defined as-if evaluating the canonical reduction expression specified in this section. With an execution policy, the relative order of evaluations of `binary_op` — and thus the relative order of any side effects — is unspecified; evaluations may be unsequenced or concurrent, as permitted by `[algorithms.parallel.exec]`, provided the returned value matches that of the canonical expression. In both cases, the relative order of side effects of `binary_op` is unspecified (see §5, Scope of guarantee).

Non-commutative operations. The canonical expression fixes the left and right operand at every `op(a, b)` node in the tree. For non-commutative `binary_op`, the operand order is fully determined by the tree structure defined in §4.2–§4.4; scheduling may vary (under an execution policy) but must produce the same returned value as evaluating the operand-ordered tree. Init placement (§4.5) is likewise fixed: `op(init, R_all)`, with `init` as the left operand.

Overview: Two-stage reduction

The 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 the canonical tree shape. In Stage 2, the `L` lane results are themselves reduced using the same tree rule. Both stages use `CANONICAL_TREE_EVAL` (§4.2.3), ensuring a fully determined expression from input to result.

```
Example: N = 12, L = 4 (3 elements per lane)

Input:  e₀  e₁  e₂  e₃  e₄  e₅  e₆  e₇  e₈  e₉  e₁₀ e₁₁
Lane:    0   1   2   3   0   1   2   3   0   1   2   3

               ┌─── Stage 1: Per-lane canonical trees ───┐

        Lane 0          Lane 1          Lane 2          Lane 3
          op              op              op               op
         /  \            /  \            /  \             /  \
        op  e₈          op  e₉          op  e₁₀          op  e₁₁
       / \             / \             / \              / \
      e₀  e₄          e₁  e₅          e₂  e₆           e₃  e₇

        ↓ R₀            ↓ R₁            ↓ R₂             ↓ R₃

           ┌─── Stage 2: Cross-lane canonical tree ──┐

                               op
                             /    \
                           op      op
                          /  \    /  \
                         R₀  R₁  R₂  R₃

                               ↓
                            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`.

[*Note:* `L = 1` denotes a single canonical tree over the full input, not `std::accumulate`’s left-fold; for non-associative operations these expressions yield different results. —*end note*]

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 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 smallest possible merge state. Adjacent chunk results can then be combined in index order, recovering the expression regardless of thread count. A detailed parallel realization strategy is described in Appendix N.

Why `L` is a semantic parameter, not a hardware hint. For non-associative `binary_op` (notably floating-point addition), different reduction topologies produce different results — this is precisely why `std::reduce` leaves its expression unspecified. To *specify* the expression, a topology must be selected. `L` is that selection: it determines the lane interleaving and canonical tree structure, fixing a single abstract expression. Changing `L` intentionally changes the expression and may change the result — just as `std::accumulate`'s left-fold produces a different result from a tree reduction. `L` is not a performance hint; it is the coordinate that makes the expression unique. Safe defaults and named presets are API concerns deferred to a future revision (see §9.2, §9.6).

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 tree structure 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*]

This canonical tree is near-balanced: it is exactly balanced when the number of participating operands is a power of two; otherwise it corresponds to iterative pairwise reduction with carry/absent operands as specified in §4.

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 `∅` ("absent"). The `maybe<A>` and `∅` 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, ∅, x) = x`
- `COMBINE(op, x, ∅) = x`
- `COMBINE(op, ∅, ∅) = ∅`
- `COMBINE(op, present(x), present(y)) = present( static_cast<A>( 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 `∅` 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.

[*Note:* The `maybe<A>` formalism is a specification device analogous to `std::optional<A>`; it is not a proposed API type and implementations need not materialize optional objects. *—end note*]

Implementations need not lift `binary_op` into an optional domain. A common strategy is to apply the canonical tree rule over full SIMD groups and evaluate any ragged tail using a scalar iterative pairwise loop. This naturally satisfies the invocation constraint without introducing additional branches into the vectorized reduction.

Invocation constraint. A conforming implementation shall evaluate `binary_op` only at internal nodes of the canonical tree where both child subexpressions are present (the fourth case of `COMBINE` above). Propagation through an absent operand (the first three cases) does not evaluate `binary_op`. For any input sequence of length `N > 0`, the canonical expression contains exactly `N - 1` evaluations of `binary_op`. When an initial value `init` is provided and `N > 0`, one additional evaluation is performed (§4.5), for a total of `N`. This requirement applies only to evaluations that complete normally. If evaluation exits via an exception or calls `std::terminate` under a policy-based execution mode, no invocation-count guarantee is made.

[*Note:* An implementation that pads absent positions with a value (e.g., an identity element) and applies `binary_op` to the padded operands would produce the same returned value for `binary_op` operations that possess an identity but would invoke `binary_op` more times than specified. Such an implementation does not conform to this specification, because it violates the invocation count and may produce observable differences for stateful `binary_op`. *—end note*]

[*Note:* Internal implementation mechanisms that do not evaluate the user-supplied `binary_op` — such as predicated SIMD lanes, masked vector operations, or compiler-generated code operating on implementation-internal state — are not constrained by the invocation count above. The constraint applies solely to evaluations of the user-supplied callable. *—end note*]

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).

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 `∅`, 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	∅	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
∅	b_0 a_2 e_6 e_7	reduce a_3 = op(e_6, e_7)
∅	b_0 a_2 a_3	reduce b_1 = op(a_2, a_3)
∅	b_0 b_1	reduce c_0 = op(b_0, b_1)
∅	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`:

```
                 op (c₀)
               /         \
         op (b₀)           op (b₁)
         /     \           /     \
     op (a₀) op (a₁)  op (a₂) op (a₃)
      / \     / \       / \     / \
     e₀ e₁   e₂ e₃    e₄ e₅   e₆ e₇
```

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:

```
              op
            /    \
           op      op
          /  \    /  \
         op   op  op  e₆
        / \  / \  / \
       e₀ e₁ e₂ e₃ e₄ e₅
```

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)
∅           : an absent operand position (no input element exists there)
combine(u,v) : lifted combine:
              - combine(∅, x) = x
              - combine(x, ∅) = x
              - combine(∅, ∅) = ∅
              - 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          → propagates left child (right is ∅)
            /       \
        COMBINE      ∅       carried from round 2 (odd n=3)
         /    \
     COMBINE  COMBINE         → left: evaluates op(X0, X1); right: propagates X2
      / \     / \
     X0  X1  X2  ∅
```

Result: `op(op(X0, X1), X2)` — two evaluations of `binary_op` (both children present at two nodes) from three present elements. All internal nodes are `COMBINE`; when both children are present, `COMBINE` evaluates `binary_op`; when one child is `∅`, `COMBINE` propagates the present child unchanged. The two `∅` positions at different tree levels each induce no evaluation of `binary_op`.

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` (equivalently, `K = (N + L − 1) / L` for integers). (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] = ∅`

Thus **all lanes use the same canonical tree shape over `K` leaf positions**. Lanes with fewer than `K` elements simply have trailing `∅` 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>` with non-negative inputs, for which `+0.0` is a suitable padding value; this shortcut does not generalize (e.g., IEEE-754 signed zero). *—end note*]

When `N < L`, some lane indices `j` correspond to no input positions: `Y_j[t] == ∅` for all `t`, yielding `R_j == ∅`. 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 `∅`). 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. In this regime, multiple lane counts may denote identical abstract expressions; this does not affect the determinism guarantee. *—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), ∅           ]
Y_3: [ present(X3), present(X7), ∅           ]
```

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 `∅` 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,  ∅        (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      ∅
    /   \
  X3    X7
```

`COMBINE(op(X3, X7), ∅) = 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]

The returned value of the canonical reduction is specified as if by the following sequence of evaluations. This two-stage decomposition is a definitional structure; conforming implementations may use any execution strategy (threading, SIMD, GPU, or otherwise) provided the returned value matches that of the specified expression.

[*Note:* This paper uses “Stage 1” and “Stage 2” as conceptual labels for the per-lane and cross-lane phases of the canonical expression. Future standard wording may adopt different presentational conventions (e.g., “steps” or “equivalent to”) per established [numerics.ops] practice; the normative content — the specified expression — is independent of the labeling. —*end note*]

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, ∅, R2, ∅ ]

       combine
       /      \
    combine   combine
    /    \    /    \
   R0    ∅   R2    ∅

combine(R0, ∅) = R0
combine(R2, ∅) = 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.

4.4.5 Value propagation [canonical.reduce.twostage.propagation]

Each application of `binary_op` within the canonical expression is performed as if by:

```
op(lhs, rhs)
```

where `lhs` and `rhs` are the values produced by evaluating the corresponding child subexpressions, each of type `A` (§4.6).

Propagation through an absent operand (§4.2.2) does not create additional copies or moves of the present operand; the present value is propagated to the parent node unchanged.

[*Note:* Implementations are free to move from intermediate values and to elide copies where permitted by the as-if rule; this specification does not constrain value categories of intermediate operands beyond the returned-value contract. *—end note*]

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

The returned value depends on whether an initial value `init` is provided and on `N`:

Case 1 — `init` provided, `N == 0`: The result is `A{init}`. No evaluations of `binary_op` are performed.

For Cases 2–3 (`N > 0`), let `R` be the value extracted from `R_all = CANONICAL_TREE_EVAL(op, Z)` in §4.4.2.

Case 2 — `init` provided, `N > 0`: - Let `I` be a value of type `A` initialized from `init` (where `A` is defined in §4.6). - The result is `op(I, R)`.

Case 3 — no `init` provided, `N > 0`: - The result is `R`.

Case 4 — no `init` provided, `N == 0`: This case is not defined by the expression structure in §4. A conforming API specification shall require `N > 0` as a precondition for any no-init overload (or else not provide such an overload).

The placement of `init` is normative. In particular, `init` is not interleaved into lanes and does not participate in the tree expression. Combining `init` with the tree result in a single final application of `binary_op` ensures that the tree shape is independent of whether an initial value is provided. Placing `init` as the left operand preserves lane-assignment invariance under extension of the input sequence and ensures that the abstract expression stays stable when additional elements are appended to the range.

Appending additional elements to the input sequence does not change the induced abstract subexpression over the original prefix elements; newly appended elements only add additional operator nodes that combine previously formed partial results with the appended values. The left placement of `init` preserves this stability property for non-associative operations.

Treating `init` as an input element would change the lane assignment of all subsequent elements and therefore alter the abstract expression.

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 this 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):

if N == 0: Result = A{init}
else:
  I = A{init}
  R = value( CANONICAL_INTERLEAVED_REDUCE(L, op, X[0..N)) )
  Result = op(I, R)

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 )
```

[*Note:* This differs structurally from `std::accumulate`, where `init` is the leftmost operand of a left-fold and participates in every step of the evaluation. In this proposal, `init` is applied exactly once, as the left operand to the completed tree result. While both conventions place `init` on the left, `std::accumulate` threads `init` through `N` applications of `binary_op`, whereas this reduction applies `binary_op(I, R)` a single time after the tree evaluation. This is an intentional design choice (see Appendix E for rationale): integrating `init` into the tree would alter the tree shape and break the independence of the expression from the presence or absence of an initial value. *—end note*]

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 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 = remove_cvref_t<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.

- The expression `static_cast<A>(E[i])` shall be well-formed for each element `E[i]` of the input range.
- When an initial value `init` is provided, the initialization `A{init}` shall be well-formed.
- `binary_op` shall be invocable with two arguments of type `A`, and the result shall be convertible to `A`.
- 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.

[*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 abstract expression for fixed `(N, L)`. Determinism is obtained by fixing the expression, not by restricting `binary_op`.

For a chosen **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: The canonical abstract expression defined in §4 depends only on the sequence of values obtained from the iterator range in iteration order and the lane count `L`; it does not depend on memory addresses, alignment, or physical data placement. The returned value additionally depends on the floating-point evaluation model (§6).

Execution independence: The returned value is determined solely by the canonical expression specified in §4. Implementations may evaluate using any strategy provided the returned value equals that of evaluating the canonical expression for the completed input sequence of length `N`.

Cross-invocation reproducibility: Given the same lane count `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. The relative order of side effects of `binary_op` is unspecified for all overloads. For execution-policy overloads, side effects may additionally be concurrent (as with existing parallel algorithms).

Constraints on `binary_op`: Let `A` be the reduction state type (§4.6). No algebraic requirements — associativity, commutativity, or identity element — are imposed on `binary_op`. The requirements on `binary_op` are invocability with two arguments of type `A` and convertibility of the result to `A`. The algorithm evaluates `binary_op` exactly `N - 1` times for `N > 0` (and `N` total when `init` is provided; see §4.2.2 and §4.5). For overloads with an execution policy, `binary_op` is additionally an element access function subject to `[algorithms.parallel.exec]`.

The returned-value guarantee applies only when evaluation completes normally. When evaluated with an execution policy, if an evaluation of `binary_op` exits via an exception, `std::terminate` is called, consistent with `[algorithms.parallel.exceptions]`. The canonical expression contains O(N) evaluations of `binary_op` (specifically, `N - 1` when `N > 0`; see §4.2.2).

The remaining requirements on iterators, value types, and side effects match those of the corresponding `std::reduce` facility ([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 lane count `L` 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 lane count `L` and an equivalent floating-point evaluation model (§2.5, §6).

6. Floating-Point Considerations (Informative)

This facility removes one specific source of variability in parallel reductions: implementation-permitted reassociation and reordering of the reduction expression. It does not attempt to address all sources of floating-point non-determinism.

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 expression 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 use an identical lane count `L` across the intended platforms.

Relationship to P3375 (Reproducible floating-point results): Davidson's P3375 [P3375R3] addresses reproducible floating-point arithmetic across implementations, proposing mechanisms to specify sufficient conformance with ISO/IEC 60559:2020 (including correctly rounded functions). 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 two key conditions for cross-platform bitwise reproducibility of parallel reductions. Neither paper alone is sufficient.

Relationship to `std::simd` (P1928): The canonical expression defined in §4 is designed to compose naturally with SIMD execution. The interleaved topology maps directly onto vertical operations over `basic_vec` values: each contiguous load feeds one element per lane, and the canonical tree over each lane's subsequence corresponds to successive vertical accumulations. An implementation may use `std::simd` operations to evaluate the canonical expression efficiently; the semantic contract does not depend on `std::simd` but is designed to be SIMD-friendly.

The horizontal `reduce` overload for `basic_vec` ([simd.reductions]) serves a different role: it reduces a single SIMD value with `GENERALIZED_SUM` semantics (both operand order and parenthesization unspecified). This proposal's cross-lane combination (Stage 2 of §4.4) specifies the horizontal reduction order that `GENERALIZED_SUM` leaves open.

Relationship to P0350 (Integrating simd with parallel algorithms): Kretz's P0350 [P0350R4] proposes an `execution::simd` policy that processes data in chunks whose width is determined by the target's native SIMD register (i.e., `native_simd<T>::size()`). The two proposals address orthogonal concerns: P0350 specifies an *execution strategy* for SIMD hardware; this paper specifies an *expression structure* for deterministic results.

The designs compose naturally. When the user's chosen lane count `L` equals the native SIMD width, or is a whole multiple of it, the canonical expression's lane structure aligns directly with the physical vector registers — each contiguous vector load feeds exactly the lanes that the canonical tree expects. This is the intended usage pattern: choose `L` to match (or be a multiple of) the deployment target's SIMD width. The mechanism by which `L` is supplied to the algorithm (template parameter, policy argument, or composition of algorithm and policy) is an API question deferred to a subsequent revision.

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	Same-config deterministic; not across thread counts or backends	Library-internal tree; expression varies with runtime configuration
oneTBB `parallel_reduce` / `parallel_deterministic_reduce`	Yes	Schedule-deterministic (not expression-specified)	Fixed join order with `deterministic` variant; tree shape is library-defined, not standardized
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.

Standard specification models for unordered summation. The Standard defines two recursive summation models in [numerics.defns] that differ in how much freedom the implementation has:

`GENERALIZED_NONCOMMUTATIVE_SUM` preserves the operand order of the input sequence but leaves the choice of split point unspecified — the implementation may parenthesise the expression any way it likes. This is the model used by `inclusive_scan` and `exclusive_scan`.

`GENERALIZED_SUM` goes further: it permits any permutation of the operands *and* any parenthesisation. This is the model used by `std::reduce` and `std::transform_reduce`.

The canonical reduction expression proposed here fixes both degrees of freedom: operand ordering is determined by interleaved lane assignment (§4), and parenthesisation is determined by the canonical iterative pairwise-with-carry tree (near-balanced; exactly balanced for power-of-two leaf counts) (§4.1). For a given lane count `L`, there is exactly one conforming abstract expression.

Table: Relationship of proposed facility to existing `<numeric>` algorithms

Property	`accumulate`	`inclusive_scan` / `exclusive_scan`	`reduce` / `transform_reduce`	Proposed `canonical_reduce`
Specification model	Sequential left fold	`GENERALIZED_NONCOMMUTATIVE_SUM`	`GENERALIZED_SUM`	Canonical reduction expression (§4)
Operand order	Fixed (left-to-right)	Fixed (left-to-right)	Unspecified (any permutation)	Fixed (interleaved lane assignment)
Parenthesization	Fixed (left fold)	Unspecified (any split)	Unspecified (any split)	Fixed (canonical iterative pairwise-with-carry tree, §4.1)
Unique expression?	Yes	No — reassociation permitted	No — reordering + reassociation permitted	Yes, for a given lane count L
Parallel execution?	No (inherently sequential)	Yes	Yes	Yes
User parameter	—	—	—	Lane count L

[*Note:* “Unique expression” means the standard fully specifies a single abstract expression — one parenthesization and one operand ordering of `binary_op` applications. `accumulate` specifies a unique left fold. The proposed `canonical_reduce` specifies a unique canonical tree for a given lane count L. Scan and reduce each admit multiple conforming abstract expressions; an implementation may choose any member of that set. For non-associative operations (e.g., floating-point addition), different members of the set may yield different results. —*end note*]

[*Note:* This paper uses `canonical_reduce<L>(...)` as the illustrative spelling, reflecting that lane count `L` is the semantic topology coordinate. 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 foundational design constraint is that the semantic topology parameter is the lane count `L`. Any API surface must expose `L` explicitly to ensure that the user’s chosen abstract expression stays stable across different compilers, operating systems, and architectures.

Two primary approaches exist.

9.1 New Algorithm Approach (Illustrative)

We propose exposing the semantics as a new algorithm parameterized by `L` via a non-type template parameter (NTTP). This ensures the chosen expression topology is a compile-time property of the algorithm call.

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

namespace std {
  // Sequential / Unsequenced evaluation (constexpr friendly)
  template <size_t L, class InputIt, class T, class BinaryOp>
  constexpr T canonical_reduce(InputIt first, InputIt last, T init, BinaryOp op);

  // Execution policy overloads
  template <size_t L, class ExecutionPolicy, class ForwardIt, class T, class BinaryOp>
  T canonical_reduce(ExecutionPolicy&& policy,
                     ForwardIt first, ForwardIt last, T init, BinaryOp op);
}
```

Rationale: The choice of topology `L` alters the abstract expression, which intentionally affects observable results for non-associative operations (e.g., floating-point addition). This argues for an API whose contract explicitly includes the topology coordinate in the type system, rather than treating topology as an implementation detail or an execution hint.

9.2 Rationale: Why Lane Count (L) is the Sole Coordinate (The Portability Trap)

Earlier explorations of this facility considered offering an added "numerics convenience" parameter `M` (representing a span in bytes), where the algorithm would derive the lane count via `L = M / sizeof(value_type)`.

This paper explicitly rejects a byte-span (`M`) API due to the portability trap it creates.

If a user requests a 64-byte span (e.g., `canonical_reduce<64>`), the derived `L` depends entirely on `sizeof(T)`. Consider `long double`:

- On MSVC (x86_64), `sizeof(long double)` is 8 bytes. `L` becomes 8.
- On GCC/Clang (x86_64 Linux), `sizeof(long double)` is 16 bytes. `L` becomes 4.

Changing `L` inherently changes the parenthesization and operand order of the abstract tree. Therefore, an API based on `M` would silently generate fundamentally different mathematical expressions on different platforms for the exact same source code, destroying the cross-platform run-to-run stability this paper seeks to provide.

By enforcing `L` as the sole topology coordinate, the C++ Standard guarantees expression identity: `canonical_reduce<16>(...)` evaluates the exact same tree on an ARM processor as it does on an x86 processor, regardless of how the underlying types are represented in memory.

If users wish to align `L` with a specific cache-line or hardware width, they can explicitly calculate it at the call site (e.g., `canonical_reduce<64 / sizeof(T)>(...)`), making the representation-size dependency explicit in source code.

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 L>
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 same conflation that §3.0 identifies as a source of 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<L>(...)` as the illustrative spelling, reflecting that lane count `L` is the semantic topology coordinate. 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)

To prevent silent semantic drift across targets, the Standard should not leave the default topology implementation-defined. Instead, a small set of standard-fixed named presets for the lane count `L` may be provided.

```
namespace std {
  inline constexpr size_t canonical_lanes_narrow = 16;
  inline constexpr size_t canonical_lanes_wide   = 128;
  inline constexpr size_t canonical_lanes_single = 1;
}
```

Typical call sites:

```
auto a = std::canonical_reduce<std::canonical_lanes_narrow>(
    v.begin(), v.end(), 0.0, std::plus<>{});

auto b = std::canonical_reduce<std::canonical_lanes_wide>(
    v.begin(), v.end(), 0.0, std::plus<>{});

auto reference = std::canonical_reduce<std::canonical_lanes_single>(
    v.begin(), v.end(), 0.0, std::plus<>{});
```

[*Note:* Appendix J provides an indicative “straw-man” API using these presets without committing the committee to a final spelling or placement. —*end note*]

Practical topology selection guidance (informative)

The semantic coordinate is `L`. 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`, 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)	Notes
Golden reference	1	1	Single tree; for debugging/golden values
Portability baseline	2	4	SSE/NEON width
AVX / AVX2	4	8	Desktop/server AVX2
AVX-512	8	16	AVX-512 servers
CUDA warp	32	32	32-thread warp; L maps to thread count, not data width

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, and any eventual interface may differ materially.

A.1 Example Algorithm Specification

The following shows how the semantic definition could be expressed for a hypothetical algorithm that exposes the lane count `L`:

- **L (lane count):** the sole semantic topology coordinate. The canonical expression is fully determined by `(N, L)`.

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

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

[*Note:* The illustrative wording marks the non-policy overloads `constexpr`. The canonical expression is a pure functional specification with no dependence on runtime state — it is structurally amenable to constant evaluation. The tree evaluation corresponds to a finite canonical expression (a finite set of `binary_op` applications) whose structure is statically determined, which is compatible with constexpr evaluation provided `binary_op` is itself constexpr. Full constexpr feasibility depends on whether the API ultimately uses iterators (which have constexpr limitations for some container types) or ranges. Detailed constexpr specification is deferred to the API design paper. *—end note*]

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

Mandates: - `L >= 1`.

A.2 Semantics (as-if)

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.** 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 `∅` 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] == ∅` for all `t`, yielding `R_j == ∅`.

4. **Per-lane canonical reduction.** For each lane index `j`, compute `R_j = CANONICAL_TREE_EVAL(binary_op, Y_j)` using the pairwise reduction tree shape and the lifted `COMBINE` rules for `∅` (§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 `∅` does not induce applications of `binary_op`).

Appendix B: Implementation Guidance (Informative)

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

General principle: Implementations need not instantiate lanes for empty subsequences; only the logical reduction structure (the canonical expression) must be preserved. For example, if L = 1000 but N = 5, the implementation need not allocate 1000 accumulators — only the 5 non-empty subsequences take part 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:

```
                  FINAL RESULT
                        |
            +-----------+-----------+
            |   THREAD REDUCTION    | <-- Combine thread results
            +-----------+-----------+
                        |
        +---------------+---------------+
        |               |               |
+-----+-----+   +-----+-----+   +-----+-----+
|   SIMD    |   |   SIMD    |   |   SIMD    |
| REDUCTION |   | REDUCTION |   | 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 (assumes L is a power of two)
        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 abstract expression 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`) 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<16>(data.begin(), data.end(), 0.0, std::plus<>{});
        double gpu = cuda_canonical_reduce<16>(d_data, N, 0.0);

        // Bitwise equality is achievable when the FP evaluation environments are aligned (see §6):
        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 (Performance feasibility). 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 long-standing challenge in high-performance computing (HPC) and distributed systems. In practice, many frameworks prioritize throughput by permitting schedule-dependent reduction structures; when operations are non-associative (e.g., floating-point addition), different reduction trees can yield different results. This appendix summarizes representative approaches and their limitations relative to the “Grouping Gap.”

C.1 Kokkos Ecosystem (HPC Portability)

Kokkos is a widely used C++ programming model for performance portability in HPC. Kokkos does not generally specify a fixed reduction expression for `parallel_reduce` or `parallel_scan`: the documentation notes that neither concurrency nor order of execution are guaranteed. As a result, floating-point results for non-associative operations may vary across thread counts, backend configurations, or architectures as the library-internal reduction structure changes.

In practice, users who require stronger reproducibility across configurations (e.g., scaling studies, regression testing across cluster generations, or reproducible validation pipelines) often adopt workarounds such as fixed-topology reductions, compensated/exact summation techniques, or application-level deterministic accumulation strategies — at non-trivial implementation cost.

This paper’s lane-interleaved topology (§4.3) is motivated by common “lane/warp” optimization patterns in HPC backends, but elevates the topology to a standardized semantic contract: the reduction expression is determined by the user-chosen topology coordinate `L`, not by runtime configuration.

C.2 Intel oneAPI Threading Building Blocks (oneTBB)

oneTBB is a widely used library for task-parallel programming on CPUs. Its `parallel_reduce` facility uses a recursive splitting strategy to partition work into sub-ranges reduced locally, then combines partial results. When scheduling is dynamic (e.g., work-stealing), the timing and grouping of joins can vary across runs, and thus the realized reduction tree can be schedule-dependent. This can produce different floating-point results for non-associative operations even when the input order is unchanged.

To address this, oneTBB provides `parallel_deterministic_reduce`, which aims to make the reduction deterministic with respect to an internal split/join structure for a given configuration (range + partitioner/grain choices), reducing run-to-run variability caused purely by scheduling. However, the tree shape remains library-defined and configuration-dependent: changing the partitioner, grain size, or other configuration choices may change the internal tree and thus the result. The key contrast with the canonical reduction proposed in §4 is that this proposal specifies a standard-defined expression determined solely by (input order, `N`, `L`) — independent of thread count, partitioner, or runtime environment.

C.3 NVIDIA Thrust and CUB

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

Thrust: Typically prioritizes throughput. Depending on the backend and algorithm, implementations may employ techniques (including schedule-dependent mechanisms) that can lead to run-to-run variability for non-associative operations. Consequently, floating-point results may vary across runs and configurations.

CUB: Provides lower-level “block-level” and “warp-level” primitives and data-movement utilities designed for high bandwidth. CUB documentation describes striped access patterns (where consecutive threads access consecutive items) used to achieve coalesced memory behavior. The interleaved subsequences (S_j) defined in §4.3 closely correspond to this striped access/layout pattern (without implying identical algorithms or guarantees) [NvidiaCUB].

C.4 Academic and Research Solutions

A substantial literature addresses reproducible summation and reproducible reductions. One class of techniques targets reproducibility independent of summation order (often via error-free transformations, compensated methods, or controlled rounding/error-bound strategies). These methods can provide strong reproducibility properties but may impose significant overhead and/or require more complex accumulator representations.

A second class enforces a specific reduction topology (fixed expression structure). This paper follows the fixed-topology approach: rather than attempting to standardize an “exact summation” method with potentially large and architecture-dependent cost, it standardizes the reduction expression structure. This enables reproducible results for a given platform under a specified expression contract, while permitting efficient realization via threads/SIMD/work-stealing/GPU kernels.

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

Appendix D: Standard Wording for Fixed Abstract Expression Structure (Informative)

In the C++ Standard, some sequential algorithms such as `std::accumulate` and `std::ranges::fold_left` define a fully specified **abstract expression structure** for the returned value. Informally, they define a left fold over the input range.

Important distinction: This paper follows that *expression identity* model. The guarantee is about the value being defined by a fixed abstract expression, **not** about portable sequencing of side effects, and **not** about bitwise reproducibility. Even with the same abstract expression, results may differ across compilers or platforms due to floating-point evaluation conditions (see §6 and §D.5). Conversely, when floating-point conditions are sufficiently aligned, a fixed abstract expression removes parenthesization/topology as a source of variation.

The guarantee being sought here is: **the returned value is defined by the abstract expression**. The relative order/timing of side effects of `binary_op` is not the objective of this paper.

D.1 std::accumulate

The expression-structure guarantee for `std::accumulate` is found in the Numerics library section of the Standard.

Section: [accumulate]/2

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 abstract expression structure is fixed as `(((init + a) + b) + c)...`. The grouping is fully specified for the purpose of defining the returned value — there is no freedom to choose a different parenthesization/topology for that value computation.

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. It also does not attempt to standardize the timing or relative order of side effects of `binary_op` beyond what is required to compute the returned value.

Contrast with std::reduce: `std::reduce` uses a generalized sum ([numerics.defns]/1–2, [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) (informal paraphrase of the specified effect)

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

Section: [alg.fold]

[*Note:* The following code is an informal paraphrase for exposition; the Standard specifies the effect using "Equivalent to:" wording in [alg.fold]. *—end note*]

Paraphrase: The range fold algorithms are sequential left-folds. Conceptually, `ranges::fold_left(R, init, f)` is equivalent to:

```
        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 left-fold, the Standard fully specifies the abstract expression structure (and thus the returned value). The accumulator state 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	[accumulate] / [alg.fold]	[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: Expression Structure ≠ Bitwise Identity

The Standard specifies an abstract fold expression structure, 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 Expression Structure

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` defines a specific left-fold expression ([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`, the fixed expression structure also implies a single sequential chain of accumulator updates. For this proposal, the expression structure (parenthesization) is fixed, but independent subexpressions may be evaluated in any order or 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))` Pairwise reduction 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 consistent with existing behavior: these flags override IEEE 754 compliance for all floating-point algorithms, including `std::accumulate`. Users who enable `-ffast-math` have explicitly traded determinism for performance.

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:* See §4.5 (init placement) for rationale. —*end note*]

E.1 Design Options Considered

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

```
canonical_reduce<L>(E[0..N), init, op):
  if N == 0:
    return init
  let R = interleaved_reduce<L>(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) - Consistent with the reduce family having an explicit `init` parameter; this paper additionally fixes init placement to a single final `op(I, R)` to keep the canonical tree independent of `init` ([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<L>(E[0..N), init, op):
  let E' = {init, E[0], E[1], ..., E[N-1]}
  return interleaved_reduce<L>(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 pairwise reduction tree has no distinguished "first" element; init is most naturally a post-reduction adjustment.
4. **Precedent:** This is consistent with the reduce family having an explicit `init` parameter; this paper additionally fixes init placement to a single final `op(I, R)` to keep the canonical tree independent of `init` ([reduce]).
5. **Full determinism:** Specifying the placement ensures that the complete abstract expression — including init placement — is fully determined. (The evaluation order of independent subexpressions remains unspecified.)

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:

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

        // Or manually:
        auto tree_result = canonical_reduce<L>(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 pairwise reduction tree	O(log N) depth, SIMD-friendly
Parameterization	Fixed constant, byte span, implementation-defined	User-specified lane count L	Portable, ABI-independent (see §9.2)
Init placement	As leaf, implementation-defined	Post-reduction `op(I, R)`	Algebraic clarity; compatible with the reduce-family API shape (explicit `init`), while additionally fixing init placement to a single final `op(I, R)` to keep the canonical tree independent of `init` ([reduce]).
Split rule	⌈k/2⌉, variable	⌊k/2⌋ (normative)	Unique grouping specification for the chosen iterative pairwise tree

F.2 Key Design Iterations

Why lane count (L): The lane count `L` is the semantic topology coordinate. It directly determines the abstract expression tree structure. Earlier designs also considered a byte-span parameter `M`, but this was rejected due to the portability trap described in §9.2: `sizeof(value_type)` varies across platforms, so the same `M` value would silently produce different expressions on different targets.

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 L: Fixed L ages poorly as hardware evolves; implementation-defined L does not provide 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 is compatible with the reduce-family API shape (explicit `init`), while additionally specifying a fixed init placement ([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 has 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 Lane Count and Hardware Alignment

The lane count `L` is specified directly, making the topology explicit and portable. Users who wish to align with hardware SIMD width can compute `L` at the call site (e.g., `64 / sizeof(V)`). This is a performance heuristic only; for cross-target expression identity choose the same explicit `L` across platforms. The following tables show how common `L` values relate to hardware targets:

L	double (8B)	float (4B)	int32_t (4B)	Register width
2	16 bytes	8 bytes	8 bytes	SSE/NEON (double)
4	32 bytes	16 bytes	16 bytes	AVX/SSE (float)
8	64 bytes	32 bytes	32 bytes	AVX-512 (double)
16	128 bytes	64 bytes	64 bytes	AVX-512 (float)

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

When `L` is chosen to match the target SIMD width (e.g., `L = 8` for AVX-512 with `double`), the algorithm maintains exactly `L` independent partial-accumulator lanes per logical stride, which maps directly to register-width execution.

For any type, specifying `L = 1` degenerates to a single-lane canonical pairwise reduction 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, 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` collapses the algorithm to a single global pairwise reduction 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 pairwise reduction tree with depth O(log N). For non-associative operations, these are different algebraic expressions and may produce different results.

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 L can shift the trade-off materially.
- *[Note:* These figures reflect a baseline prototype. The SIMD-optimised implementation in Appendix N (SIMD optimization section) demonstrates throughput competitive with or exceeding `std::reduce` on the same hardware, indicating that the overhead is an artefact of the unoptimised prototype rather than an inherent cost of the canonical semantics. *—end note]*

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 Lane Count Presets (Informative)

This appendix records rationale for providing a small set of **standard-fixed lane count preset constants** as coordination points for topology selection (§9.6). The goal is to provide readable, stable choices that do not vary with platform properties, value type, or ABI, and therefore avoid “silent semantic drift” in returned values for non-associative operations.

The lane count `L` is the semantic topology coordinate. Because `L` is specified directly (not derived from a byte span — see §9.2), the same preset selects the same abstract expression regardless of `sizeof(value_type)`.

I.1 Rationale for L = 16 (Narrow Preset)

A “narrow” preset should provide useful parallelism for the most common scalar sizes while staying small enough that overhead does not dominate for moderate N.

L = 16 is a practical coordination point because: - It matches AVX-512 lane count for `float` (16 × 4 bytes = 64 bytes = one ZMM register). - It provides 2× AVX-512 width for `double` (16 × 8 bytes = 128 bytes), which maps to two-register unrolling — a standard SIMD optimization pattern. - It is a power of two, ensuring the per-lane tree is perfectly balanced for power-of-two K values. - It is wide enough for meaningful instruction-level parallelism on all current architectures yet narrow enough that the cross-lane Stage 2 tree is shallow (4 levels).

I.2 Rationale for L = 128 (Wide Preset)

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

L = 128 provides substantial independent work per lane and can support aggressive batching and parallel evaluation of independent reduction nodes while preserving the same abstract expression. It accommodates GPU warp widths (32) with room for multi-warp cooperation and enables deeply unrolled SIMD loops on CPU.

I.3 Cross-Domain Verification

Because these presets are standard-fixed lane counts, a user can select the same preset 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 reassociation 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. Sender-returning (P2300) surfaces are deferred to the follow-on API paper.

Presentation note (informative): - The **semantic topology coordinate** is the lane count `L`. - The lane count `L` is the sole semantic topology coordinate. The canonical expression is fully determined by `(N, L)`. - When cross-platform expression stability is needed, `L` must be specified directly (not derived from hardware properties).

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 “narrow / wide” in code reviews, not “16 / 128”.

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

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

Illustrative (provisional) names:

Placement in `namespace std` is illustrative only; a follow-on API paper may place any such presets in a dedicated scope (e.g., a nested namespace or tag type) to avoid adding new top-level `std` names.

```
namespace std {
  // Standard-fixed lane presets. Values never change.
  inline constexpr size_t canonical_lanes_narrow = 16;  // baseline lane preset
  inline constexpr size_t canonical_lanes_wide = 128;   // wide lane preset

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

Clarification (informative): The preset lane 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.

J.3 Straw-man algorithm API

The simplest surface is an algorithm family parameterized by lane count `L`.

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

  template<size_t L,
       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 (narrow vs wide)
auto a = std::canonical_reduce<std::canonical_lanes_narrow>(v.begin(), v.end(), 0.0, std::plus<>{});
auto b = std::canonical_reduce<std::canonical_lanes_wide>(v.begin(), v.end(), 0.0, std::plus<>{});

// Explicit literal for hardware-specific alignment
auto c = std::canonical_reduce<8>(v.begin(), v.end(), 0.0, std::plus<>{}); // L = 8
```

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 without requiring new execution-policy machinery. - Makes the coordination presets show up in diagnostics and in code review as names. - For fixed `L`, the canonical tree is identical across all targets, even if `sizeof(V)` differs.

J.4 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.5 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 {
  // Lane coordinate — sequential/reference mode
  // Non-sized forward ranges can be supported via a counting pass to determine N.
  template<size_t L,
       forward_range R, class T, class BinaryOperation>
  requires (L >= 1)
  constexpr T canonical_reduce(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(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`): `0x40618f71f6379380`
- **WIDE** (`L = 128`): `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 has: - 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-x86-MT-EXT] extended correctness	x86-64	870 hostile bitwise comparisons: 5 generators × 29 awkward sizes × 3 thread counts × 2 lane widths	No (`std::plus<double>` only)	Strongest evidence: adversarial FP data, all bitwise identical to ST reference under aligned FP settings (e.g., no fast-math / no contraction; see §6). Cross-ISA golden values (AVX2 = AVX-512).
[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:

[*Note:* Except **[GB-SEQ]**, these demonstrators test only `std::plus<double>` and may use `0.0` for absent operand positions as an implementation shortcut; this does not generalize to arbitrary `binary_op` and is not a conforming technique for the general case. *—end note*]

- [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
  (proves SIMD vertical-add matches §4 tree)
- [GB-x86-MT] = https://godbolt.org/z/7a11r9o95
  (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-x86-MT-EXT] = https://godbolt.org/z/no5b5hPo6
  (extended correctness: 5 hostile generators [cancellation, exponential, Kahan, subnormal, uniform] × 29 awkward sizes [N=1 to N=1,000,003] × T=2,3,4 × L=16,128 = 870 bitwise comparisons against ST reference under aligned FP settings (see §6); cross-ISA golden value validation [AVX2 on Godbolt matches AVX-512 locally]; build flags: `-O3 -std=c++20 -ffp-contract=off -fno-fast-math -DFORCE_AVX2=1 -pthread -DNO_PAR_POLICIES=1 -mavx2`)

Hostile data generators ([GB-x86-MT-EXT]): The extended correctness demonstrator uses five data generators chosen to be maximally sensitive to grouping order. *Cancellation* alternates $\pm 10^{15}$ with small perturbations, creating intermediates 15 orders of magnitude larger than the true result — any grouping change alters which perturbation bits survive rounding. *Exponential* spans ~300 orders of magnitude with random signs, so different trees swallow different small values. *Kahan* repeats six +1.0 values followed by one −6.0, creating systematic oscillation with O(N) condition number. *Subnormal/edge* mixes values near 5×10^{-324}, near $\pm 10^{308}$, and signed zeros to exercise floating-point hardware edge cases. *Uniform* draws from [−1, +1) as a baseline. The 29 test sizes include degenerate cases (N < L), lane-boundary ±1 values, primes, and power-of-2 edges — all chosen to force ragged partitions across every thread count and lane width combination.

- [GB-NEON] = https://godbolt.org/z/Pxzc3YM7q
  (Arm v8 / AArch64; 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; 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
- **Large / wide:** L = 128 for double

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 lane count `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”:** `0x40618f71f6379380`
- **L = 128, “WIDE”:** `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 size 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 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 presets 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 lane count `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.5 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. However, the iterative pairwise (shift-reduce) evaluation strategy can consume elements in a single pass, discovering `N` at end-of-sequence and completing the canonical tree at that point. Whether an API surface accepts `input_range` directly is an API decision deferred to a subsequent revision.

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_lanes_narrow>(
  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_lanes_narrow>(
          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: no run-to-run variation
        double computed = canonical_reduce<8>(           // L = 8
          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<8>(
          local_gradients.begin(), local_gradients.end(), 0.0);

        // Later, on same or equivalent hardware:
        auto restored_sum = canonical_reduce<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 Scientific Reproducibility

```
        // Climate model energy conservation check
        // Paper claims: "Energy drift < 1e-12 per century"
        // Reviewers must reproduce this result
        auto total = canonical_reduce<8>(
          grid_cells.begin(), grid_cells.end(),
          Cell{0.0, 0.0},
          [](const Cell& a, const Cell& b) {
            return Cell{a.kinetic + b.kinetic, a.potential + b.potential};
          });
        double total_energy = total.kinetic + total.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.4 Exascale HPC with Kokkos

```
// DOE exascale application using Kokkos
// Kokkos guarantees same-config reproducibility, but results may differ
// across thread counts, backends, or architectures
// In such cases, results can vary across runs, complicating debugging and reproducibility
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.5 Cross-Platform Development

```
// Game physics: aim for consistency between client platforms
constexpr size_t PHYSICS_L = 16; // 16 lanes

float compute_collision_impulse(const std::vector<Contact>& contacts) {
  auto total = canonical_reduce<PHYSICS_L>(
    contacts.begin(), contacts.end(),
    Contact{0.0f},
    [](const Contact& a, const Contact& b) {
      return Contact{a.impulse + b.impulse};
    });
  return total.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.6 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 equality with the single-threaded canonical expression (bitwise identity under the same floating-point evaluation environment; see §6). 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 having 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 = 128 (2^7), giving ⌈1000/128⌉ = 8 chunks:

Thread	Chunks	Blocks	Output
0	0–1	[0, 256)	Two buckets at level 7, merged locally
1	2–3	[256, 512)	Two buckets at level 7, merged locally
2	4–5	[512, 768)	Two buckets at level 7, merged locally
3	6–7	[768, 1000)	Bucket at level 7 + remainder (104 blocks)

Full chunks (128 = 2^7 blocks) produce exactly one bucket at level 7. The remainder chunk (104 = 64 + 32 + 8) naturally decomposes via shift-reduce into buckets at levels {6, 5, 3}.

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 under the same floating-point evaluation environment (§6), 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);  // sequential push() over blocks [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 the canonical expression contract.

N.12 Performance Observations

The reference implementation at **[GB-x86-MT-PERF]** 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 run-to-run bitwise reproducibility within a fixed floating-point evaluation environment (§6). 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:     +---+       +---+       +---+       +---+
              |S₀₁|       |S₂₃|       |S₄₅|       |S₆₇|  (parallel)
              +-+-+       +-+-+       +-+-+       +-+-+
                |           |           |           |
                +-----------+           +-----------+
                    |                       |
Stride 2:         +-------+               +-------+
                  |S₀₁₂₃  |               |S₄₅₆₇  |      (parallel)
                  +---+---+               +---+---+
                      |                       |
                      +-----------------------+
                              |
Stride 4:                 +-----------+
                          |S₀₁₂₃₄₅₆₇  |                    (final)
                          +-----------+
```

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	Expression-identical to the single-threaded canonical expression for any T under the specified partition/merge scheme; bitwise identity additionally requires a matching floating-point evaluation model (§6)
Correctness	Equivalent to the canonical pairwise tree (argument outlined in Appendix N)
Efficiency	O(N) work, O(N/T + T log N) span
Practicality	Demonstrated competitive with, and in some configurations faster than, `std::reduce` with SIMD on the tested platforms/harness

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 shows that a fixed abstract expression structure (canonical expression) is compatible with high-throughput SIMD execution. The constraint of a canonical tree enables aggressive unrolling and predictable memory access, achieving throughput comparable to unconstrained implementations in the tested configurations.

GPU implementations may place the O(log N) working state per lane in shared memory rather than thread-local registers, avoiding register pressure while preserving evaluation of the identical canonical expression. This is a standard GPU reduction pattern and does not require changes to the semantic contract.

Appendix O: 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.

O.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))
    )
```

O.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.

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

For non-power-of-two `k`, the trees differ in structure (but keep 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`

O.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, keeping alternatives in the normative contract increases complexity without adding semantic value.

(Normative rule in §4:) No other grouping is permitted.

Appendix P: Performance and Regret (Informative)

This appendix records performance and regret considerations for the choice of iterative pairwise (shift-reduce) as the canonical tree construction rule (§3.8). These considerations are informative and do not affect the semantic contract specified in §4–§5.

P.1 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 keeps 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. This 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 lower-risk standard choice.

Executor compatibility. Once the expression is separated from execution (§3.0), the question becomes: which 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.

P.2 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 consequence of choosing iterative pairwise is commitment to a specific non-power-of-two boundary behavior. 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. Of the two candidates, iterative pairwise is most closely aligned with existing implementations.

P.3 Upper bound on regret

Regardless of tree shape, any balanced binary reduction has the same O(log N · ε) 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, Dalton et al. report roughly 89% of peak throughput on their setup; this suggests limited headroom in that environment, but the exact gap is configuration-dependent.

P.4 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. In general, that dependency prevents full-width SIMD parallelism without changing the expression, so utilization can be far below the SIMD width (e.g., a single running scalar corresponds to ~1/8 of AVX-512 `double` lanes or ~1/16 of `float` lanes in an idealized utilization model). The canonical iterative pairwise tree, by contrast, exposes independent operations that map naturally to SIMD lanes and can approach memory-bandwidth limits in tuned implementations (e.g., ~89% of peak in the literature for non-L1-resident data). Interpreting these figures as utilization rather than guaranteed speedups, the canonical tree can recover most of the SIMD parallelism that a strict left fold cannot generally access.

The throughput and regret analyses above establish that neither candidate has a decisive performance advantage. The distinguishing property is architectural: iterative pairwise is the only common candidate that can be evaluated incrementally without global sizing — a structural alignment with the sender/receiver execution model adopted for C++26 (see §3.8.2 and §3.10).

Appendix Q: Historical Note on Span-Based Topology (Informative)

Earlier explorations of this facility considered specifying the reduction topology in terms of a fixed byte span `M`, with the corresponding lane count derived as:

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

This formulation aligns with common implementation concerns such as cache-line size, SIMD register width, or hardware batch size, and may be convenient for numerics workflows that target a specific memory or vector width.

However, because `sizeof(value_type)` is implementation-defined, a span-based topology does not, in general, select the same abstract expression across different platforms or ABIs. For example, `sizeof(long double)` may differ between implementations on the same architecture, causing the same source-level span value `M` to produce different lane counts `L` and therefore different parenthesizations of the reduction expression.

Since the topology coordinate intentionally determines the abstract expression evaluated for non-associative operations, such variation would undermine the cross-platform expression identity this proposal seeks to provide.

For this reason, the semantic topology parameter in §4 is defined solely in terms of the lane count `L`. API designs that derive `L` from a byte span may be considered in future revisions as layout-oriented conveniences for environments where representation size is known and fixed, but such facilities would not, in general, denote the same abstract expression across implementations.