Canonical Parallel Reduction

A Fixed Expression Structure for Run-to-Run Consistency

Andrew Drakeford
SG14 — Low Latency, Gaming, Embedded, Financial Trading
February 2026

We Want Deterministic Reduction

Parallel reduction that gives the same answer every time. Not accumulate.

The Gap in the Standard

std::accumulate

Sequential left-to-right fold

Specified sequence ✓
Reproducible ✓
Parallel ✗

THE GAP

No standard facility combines parallel execution with a specified expression.

Specified sequence ✓
Reproducible ✓
Parallel ✓

This is what we propose.

std::reduce

Unspecified grouping

Specified sequence ✗
Reproducible ✗
Parallel ✓

Different results for non-associative operations

Same code, same data — potentially different result. That's the gap.

Why SG14 Domains Can't Live With This

Finance

Audit reproducibility requires deterministic replay of analytics. Audit trails require reproducibility.
Regression testing against a gold result is impossible.

Gaming

Deterministic lockstep networking breaks without bitwise reproducibility. Replay systems require identical results across clients.

Safety-Critical

DO-178C and IEC 61508 require deterministic computation for certification. Non-reproducible parallel code is a blocker.

Everyone here has either hit this problem or worked around it with a hand-rolled solution.

Two Approaches — Complementary, Not Competing

A: Binned Summation

Attack the accuracy — make the answer so precise that grouping doesn't matter.

Libraries (ReproBLAS, ExBLAS, Kulisch) use extra-precision arithmetic to make summation order-independent. Limited to operations with known error structure.

B: Fixed Evaluation Topology ← This Talk

Fix the shape of the reduction tree. The sequence of operations is then determined.
Same rounding error every time.

Works for ANY binary_op — not just addition.
General-purpose foundation.

They compose

A binned accumulator as binary_op inside a fixed topology gives you both accuracy and reproducibility.

The topology is the general-purpose foundation.

This proposal fixes topology, not numerical error. For error bounds, see Higham §4.6.

The Core Requirement

The topology must depend only on N and a user parameter L. Nothing else.

- Not on thread count
- Not on SIMD width
- Not on platform
- Not on execution strategy

Same N, same L → same shape → same sequence → same result.

The expression shape is fixed; execution scheduling remains free.

Given the same floating-point evaluation model (rounding mode, contraction, precision). See §6 of the paper.

Three Benefits of a Fixed Topology

1. Reproducibility

Fix the tree → fix the sequence → fix the result.

Same L, same N → same result, every time,
any thread count, any SIMD width.

Cross-platform identity additionally requires
a matching FP evaluation model.

2. Relaxed Constraints

std::reduce requires commutativity and associativity.

A fixed topology specifies every operand position.
No reassociation → no associativity.
No reordering → no commutativity.

3. No Identity Element

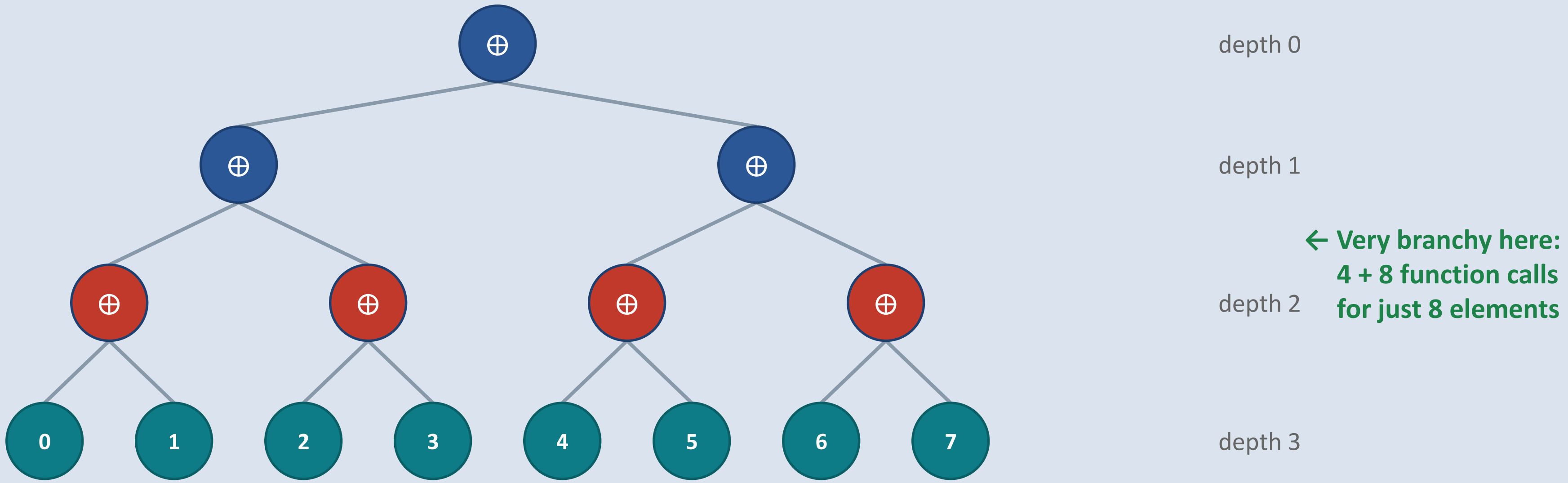
A canonical tree does not require an algebraic
identity element.

A fixed topology uses absent-operand propagation
for non-power-of-2 lengths.
Property of the tree,
not the operator.

One decision — agree on the tree — delivers all three.

- **Topological determinism** — expression is fixed for given (N, L), independent of implementation/hardware/scheduling
- **Layout invariance** — results independent of memory alignment and physical placement
- **Execution independence** — subtrees may be evaluated in any order/concurrently
- **Cross-invocation reproducibility** — stable returned value across runs for same inputs and topology
- **Scope of guarantee** — applies to returned value only (not side-effect ordering)

Starting Point: Recursive Bisection



Problem: recursive calls dominate

Every ⊕ node is a function call with a branch. N−1 interior nodes
for N leaves. At the bottom levels the work per node is tiny but the overhead is not.

Good: O(log n · ε) accuracy

Balanced tree gives optimal error bounds.
Natural parallelism from independent subtrees.

Can we get the same balanced tree shape with a faster evaluation

Why a Tree?

Scalability

Independent subtrees execute concurrently. $O(\log N)$ depth enables efficient parallel decomposition across threads and SIMD lanes.

Accuracy

Balanced tree: $O(\log n \cdot \varepsilon)$ error bounds.
Left-to-right fold (accumulate): $O(n \cdot \varepsilon)$.
Strictly better — you gain accuracy for free.

What L gives you

L selects the lane count (the primary topology coordinate). If you pick $L=16$ but deploy on AVX-512, you still get full reproducibility — the canonical tree is unchanged. You may leave some hardware utilisation on the table, but correctness is never at risk.

Choose L for your reproducibility domain, not your deployment target.

From Naive to Proven

Naive: Binary Decomposition

$N = 47 = 101111_2$
→ Trees of size: 32 + 8 + 4 + 2 + 1

Fast, branchless, cache-friendly. Each sub-tree is a known complete size.

But: final combination merges results of very different
magnitudes — a 32-element partial sum next to a single
element degrades numerical error bounds.

→

Proven: Iterated Pairwise (Shift-Carry)

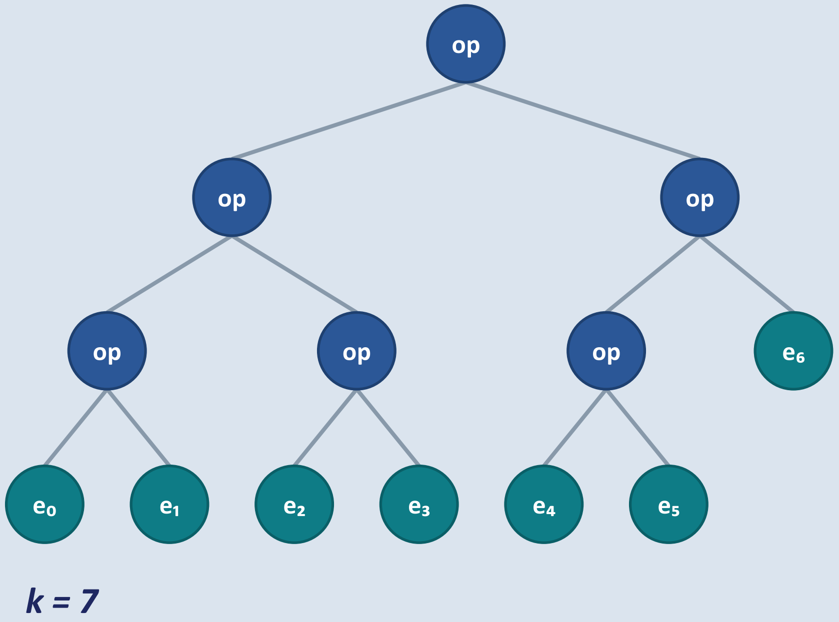
Process input in blocks of L lanes.

Within each block: L independent lanes accumulate vertically — branchless, straight-line, maps directly to SIMD.

Shift-carry maintains balanced partial results across blocks. $O(\log n \cdot \varepsilon)$ error — mathematically proven.

We propose to canonicalise the proven approach — not inventing, standardising.

The Canonical Tree (k = 7)



Parenthesized expression (fully determined by k):

$((e_0 + e_1) + (e_2 + e_3)) + ((e_4 + e_5) + e_6)$

Worked rounds

Round 1: pair [e_0,e_1], [e_2,e_3], [e_4,e_5], carry e_6
→ 4 results

Round 2: pair [op(e_0,e_1), op(e_2,e_3)], [op(e_4,e_5), e_6]
→ 2 results

Round 3: final combine → 1 result

Tree shape is fully determined by k alone — no runtime decisions.

Shift-Reduce: How It Executes

Remaining Sequence	Stack	Operation
x_1 x_2 x_3 x_4 x_5 x_6 x_7 x_8	∅	shift x_1
x_2 x_3 x_4 x_5 x_6 x_7 x_8	x_1	shift x_2
x_3 x_4 x_5 x_6 x_7 x_8	x_1 x_2	reduce $a_1 = x_1 + x_2$
x_3 x_4 x_5 x_6 x_7 x_8	a_1	shift x_3
x_4 x_5 x_6 x_7 x_8	a_1 x_3	shift x_4
x_5 x_6 x_7 x_8	a_1 x_3 x_4	reduce $a_2 = x_3 + x_4$
x_5 x_6 x_7 x_8	a_1 a_2	reduce $b_1 = a_1 + a_2$
x_5 x_6 x_7 x_8	b_1	shift x_5
x_6 x_7 x_8	b_1 x_5	shift x_6
x_7 x_8	b_1 x_5 x_6	reduce $a_3 = x_5 + x_6$
x_7 x_8	b_1 a_3	shift x_7
x_8	b_1 a_3 x_7	shift x_8
∅	b_1 a_3 x_7 x_8	reduce $a_4 = x_7 + x_8$
∅	b_1 a_3 a_4	reduce $b_2 = a_3 + a_4$
∅	b_1 b_2	reduce $c_1 = b_1 + b_2$
∅	c_1	done!

The pattern: after n shifts, ntz(n) reductions occur. Shift-shift-reduce, shift-shift-reduce-reduce, … No branches — purely mechanical.

Dalton, Wang & Blainey (IBM, 2014) — *"SIMDizing Pairwise Sums"*

This is one efficient evaluation strategy. The Standard specifies the expression tree, not the evaluation schedule.

Iterated Pairwise — The Proposed Canonical Form

Known. Proven. Industry practice. Not inventing anything.

How It Works

Process input in blocks of L lanes.

Within each block: L independent lanes accumulate vertically — branchless, maps to SIMD.

Shift-carry maintains balanced partial results across blocks.

What It Gives You

Provably SIMD-efficient: inner loop is pure vertical accumulation.

O(log n · ε) error bounds — same as recursive pairwise.

Topology depends only on N and L. Same tree on AVX2,
NEON, SVE, or scalar.

Higham, Accuracy and Stability of Numerical Algorithms, §4.6. Dalton, Wang & Blainey (IBM, 2014).

This is the tree we propose to canonicalise.

Threading Scales Naturally

Fill Phase (Parallel)

Partition input into power-of-2 blocks aligned to L.
Each thread fills its blocks independently.

Embarrassingly parallel — no sharing, no races.
Power-of-2 boundaries align with tree level boundaries
— merge is trivial.

Replay Phase (Canonical)

Merge follows the same canonical tree. Topology unchanged
regardless of how many threads filled the blocks.

No awkward remainders at merge points.
No special-case logic for different-sized chunks.

1 thread or 128 threads — the canonical tree is the same.

Threads affect who computes what, not what is computed.

What It Looks Like

```
// Illustrative — name and signature not yet proposed

// Lane-based topology (portable across ABIs for a fixed L)
auto r1 = canonical_reduce_lanes<16>(first, last, init, op);

// Span-based shorthand (convenience: L = M / sizeof(V))
auto r2 = canonical_reduce<128>(first, last, init, op); // M=128 bytes

// Golden reference (L=1, single canonical tree, no lane interleaving)
auto gold = canonical_reduce<sizeof(double)>(first, last, init, op);
```

Same semantics as std::reduce

Same iterator requirements. Same binary_op. Adds a topology parameter that fixes the expression.

This is stable_sort vs sort

You choose whether you need the guarantee. No overhead if you don't use it.

x86 (AVX2) — Godbolt

```
NARROW (L=16): 0x40618f71f6379380
WIDE   (L=128): 0x40618f71f6379397
```

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

With tuned SIMD implementation, deterministic reduction can match or exceed std::reduce.

Flags: `-O3 -std=c++20 -ffp-contract=off -fno-fast-math` | *Click Run on Godbolt — committee members can verify*

godbolt.org/z/jbYqf1Eez

```
PERFORMANCE (CE timings vary; best-of-trials):

  std::accumulate                              1.489 ms     5.37 GB/s
  std::reduce (no policy)                      0.371 ms    21.56 GB/s
  std::reduce(seq)                             0.380 ms    21.07 GB/s
  std::reduce(unseq)                           1.496 ms     5.35 GB/s
  std::reduce(par)                             0.379 ms    21.12 GB/s
  std::reduce(par_unseq)                       1.493 ms     5.36 GB/s
  deterministic_reduce NARROW (M=128)          0.313 ms    25.54 GB/s
  deterministic_reduce WIDE   (M=1024)         0.354 ms    22.62 GB/s

Overhead vs std::accumulate:
  NARROW: -79.0%
  WIDE:   -76.2%

═════════════════════════════════════════════════════════════════
VERIFICATION BLOCK
═════════════════════════════════════════════════════════════════
Platform: x86-64
Selected: AVX2
SEED:      0x243f6a8885a308d3
N:         1000000
NARROW:    0x40618f71f6379380  (M=128,  L=16)
WIDE:      0x40618f71f6379397  (M=1024, L=128)
═════════════════════════════════════════════════════════════════
```

ARM (NEON) — Godbolt

```
NARROW (L=16):  0x40618f71f6379380
WIDE   (L=128): 0x40618f71f6379397
```

Cross-Platform Identity

Identical hex output on a completely different ISA — same tree, same result.

Build-proof macros: `__aarch64__`, `__ARM_NEON`, `__ARM_NEON_FP`

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

Same canonical tree on a different ISA → identical golden hex.

CE timing is illustrative only (VM load varies).

godbolt.org/z/v369Mbnvh

```
  std::accumulate                                   0.776 ms    10.32 GB/s
  std::reduce (no policy)                           0.340 ms    23.55 GB/s
  std::reduce(seq)                                  0.340 ms    23.50 GB/s
  std::reduce(unseq)                                0.775 ms    10.32 GB/s
  std::reduce(par)                                  0.340 ms    23.56 GB/s
  std::reduce(par_unseq)                            0.779 ms    10.27 GB/s
  deterministic_reduce NARROW (M=128)               0.272 ms    29.38 GB/s
  deterministic_reduce WIDE   (M=1024)              0.341 ms    23.43 GB/s

Overhead vs std::accumulate:
  NARROW: -64.9%
  WIDE:   -56.0%

════════════════════════════════════════════════════════════════
VERIFICATION BLOCK
════════════════════════════════════════════════════════════════
Platform: ARM64
Selected: NEON
SEED:      0x243f6a8885a308d3
N:         1000000
NARROW:    0x40618f71f6379380  (M=128,  L=16)
WIDE:      0x40618f71f6379397  (M=1024, L=128)
════════════════════════════════════════════════════════════════
```

CUDA — Godbolt

```
L=16:   0x40618f71f6379380
L=128:  0x40618f71f6379397
```

Canonical Topology on GPU

Evaluates the same canonical expression (§4).
Both L=16 and L=128 configs demonstrated.

Zero overhead vs CUB — parity with NVIDIA's optimised reduction.

Golden Result Workflow

Produce golden hex on GPU, verify on CPU.
CPU golden matches GPU golden — heterogeneous verification.

Same tree, different hardware, identical bits.

Zero overhead vs CUB; heterogeneous golden-result verification.

CUDA/NVCC; canonical vs CUB; includes L=16 and L=128

godbolt.org/z/x58GzE73q

Identical results assume matching FP evaluation model (contraction disabled, same rounding mode).

```
══ PERFORMANCE (N=1e6) ══════════════════════════════════════════════

  CUB DeviceReduce::Sum                          0.056 ms   143.36 GB/s
  Fast atomic reduce                             0.052 ms   153.04 GB/s
  Canonical FAST (L=16)                          0.548 ms    14.61 GB/s
  Canonical FAST (L=128)                         0.067 ms   118.84 GB/s

══ PERFORMANCE vs CUB ═══════════════════════════════════════════════
  Canonical FAST (L=16):  9.81x slower (10.2% of CUB throughput)
  Canonical FAST (L=128): 1.21x slower (82.9% of CUB throughput)

══════════════════════════════════════════════════════════════════════
VERIFICATION BLOCK
══════════════════════════════════════════════════════════════════════
Platform: CUDA (Tesla T4)
SEED:     0x243f6a8885a308d3
N:        1000000
HOST N:   0x40618f71f6379380  (L=16)
GPU  N:   0x40618f71f6379380  (L=16)
HOST W:   0x40618f71f6379397  (L=128)
GPU  W:   0x40618f71f6379397  (L=128)
SWEEP:    PASS ✓
══════════════════════════════════════════════════════════════════════
```

What This Proposal Does NOT Guarantee

✗ **Does not guarantee cross-architecture IEEE identity.**
Different ISAs may evaluate differently unless the FP model also matches.

✗ **Does not constrain the floating-point evaluation model.**
Contraction, rounding mode, and precision remain implementation choices.

✗ **Does not replace std::reduce.**
std::reduce remains the right choice when determinism is not required.

✗ **Does not impose runtime overhead unless chosen.**
Opt-in only — existing code paths are unaffected.

Understanding the boundaries of a proposal is as important as understanding its benefits.

Why Standardise?

Kokkos | **TBB** | **CUB** | **oneMKL**

Different, implementation-specific topologies. No portable semantic contract.

Portable
Reproducibility across implementations.
Same canonical tree everywhere.

Composable
Works with execution policies and ranges. Generic components can demand specified reduction.

Certifiable
A standard specification is a certification target for regulated industries.

This is stable_sort vs sort. You choose whether you need the guarantee.

Feedback We're Seeking

Semantics first. API next.

1. Do you agree that deterministic parallel reduction is needed — that there's a gap?

2. Do you agree that a fixed topology (N and L only, not thread count or SIMD width) is the right approach?

3. Do you agree that basing this on proven industry practice (iterated pairwise) is a sound choice?

4. Would SG14 support forwarding this to LEWG for further semantic review?

Thank you. Discussion welcome.