Canonical Parallel Scan: Expression and Observation Contracts for Reproducible Parallel Algorithms

Document number:	P4229R0
Date:	2026-05-12
Audience:	SG6, SG1, LEWG

Author:	Andrew Drakeford
Reply-to:	Andrew Drakeford <andreedrakeford@hotmail.com>

Contents

Abstract

Blelloch [Blelloch1990] identifies `reduce`, `scan`, and `transform` as core parallel primitives. P4016R0 demonstrated that efficient, reproducible reduction is achievable

using a tree-based abstract expression.

This paper addresses reproducible `scan`. The same expression model can define reproducible `scan`, but unlike element-wise `transform`, `scan` exposes cross-element expression structure at every prefix, so reproducibility cannot be treated as an implementation detail alone. Implementation experience shows that the choice of expression materially affects performance, scalability, streamability, accuracy, and hardware suitability.

Reproducibility is a contract specified by an abstract expression. Agreement follows from choosing a named expression that defines the result. Named abstract expressions provide the missing specification layer: selectable reproducibility contracts for parallel `reduce` and `scan`, each admitting multiple conforming implementations.

---

1. Introduction

The C++ Standard Library's parallel algorithms rely on implementation freedom for performance. Implementations may split ranges, combine partial results, vectorize, use multiple threads, or target accelerator hardware. For exact associative operations, this freedom may be invisible. For floating-point and other order-sensitive operations, it can change the result.

The issue is not merely non-associativity. That is the symptom. The deeper problem is that the expression evaluated by a parallel algorithm is usually implicit. Without an explicit expression, users cannot know whether results are repeatable, stable under different execution choices, or consistent with related algorithms.

P4016R0 addressed this for `reduce` by proposing a fixed expression structure. `scan` exposes the next distinction: `reduce` observes one value, while `scan` returns a sequence of prefix values. Relating reproducible `reduce` and `scan` therefore requires specifying both what expression defines the values and which values of that expression are returned.

This paper separates three concerns: execution freedom, expression contract, and observation contract. Existing wording gives implementations broad freedom in the first. P4016R0 addressed the second for reduction. A reproducible `scan` requires the third.

The implementation artifacts in the appendices are proof by existence. They show that the model is executable, that scan/reduce bit-consistency can be achieved, and that cross-platform bitwise reproducibility — including cross-platform scan/reduce consistency — is possible under constrained floating-point evaluation rules. They are evidence for feasibility, not proposed algorithms or proposed wording.

1.1 The gap in existing scan facilities

The standard definitions of `inclusive_scan` and `exclusive_scan` preserve the left-to-right order of operands, but permit different parenthesizations. For associative operations this distinction is usually unobservable. For floating-point addition and other

order-sensitive operations, however, different conforming parenthesizations may produce different results.

A reproducible scan therefore requires more than a stable traversal order: it requires a specified expression for each observed prefix.

This is the relevant distinction among the existing facilities:

All rows preserve left-to-right operand order; the relevant distinction is parenthesization and whether the expression contract is named.

Facility	Parenthesization	Named expression contract?
`std::partial_sum`	left fold	effectively fixed
`std::inclusive_scan`	not fixed as a named expression	no
`std::exclusive_scan`	not fixed as a named expression	no
`reproducible_scan<E>`	named expression `E`	yes

The proposed model does not remove implementation freedom. It identifies the missing semantic layer. Existing scan wording preserves operand order; a reproducible scan must also name the expression that defines each prefix value.

---

2. Scopes of determinism

"Deterministic" does not name a single property. A proposal for deterministic `scan` or `reduce` is ambiguous unless it states the scope of determinism being promised.

At the weakest level, determinism may mean only run-to-run repeatability: the same program, input, implementation, and environment produce the same output on repeated runs. This is useful for testing, debugging, and audit, but it does not protect against changes in thread count, alignment, chunk size, vector width, or target hardware.

A stronger claim is same-platform repeatability: the implementation yields stable results across ordinary execution choices on a single platform. This is often what users expect from a deterministic library implementation, but if the expression remains implicit, the guarantee is still implementation-defined. A change in partitioning may alter the expression being evaluated and, therefore, the floating-point result, even though each individual execution is repeatable.

The next scope is expression-based reproducibility. Here, the result is defined by a named expression rather than by the execution shape used to compute it. The implementation may choose different schedules, chunk sizes, vector widths, or thread counts, but it must return the values defined by the expression contract. This is the point at which reproducibility becomes a specification property rather than merely an implementation habit.

Cross-ISA and cross-implementation reproducibility require still more care. A named expression is necessary, but not always sufficient: contraction, extended precision,

rounding, reassociation, exception behavior, and library-function differences may also have to be controlled. A portable named expression must therefore be specified precisely enough that independent conforming implementations produce the same result. The complementary problem of controlling the floating-point environment so that a fixed expression evaluates identically across implementations is the subject of [P3375R3].

There is also a separate property: algorithmic consistency. Two algorithms may each be deterministic while disagreeing with each other. For example, a left-fold deterministic `scan` and a pairwise deterministic `reduce` may both be repeatable, yet the final scan element need not equal the reduce result over the same input.

A reproducible scan need not automatically be consistent with a reproducible reduction. Consistency is an additional property. For a policy `E`, scan/reduce consistency means that each scan output is the same observation that would be produced by applying the corresponding reduction policy to that prefix:

```
inclusive_scan<E>(xs)[i] == reduce<E>(xs[0..i+1))
```

and, for the final prefix:

```
inclusive_scan<E>(xs)[n - 1] == reduce<E>(xs)
```

subject to the same operator, value type, initialization rules, and specified arithmetic environment.

Throughout this paper, expressions such as `inclusive_scan<E>(xs)[i] == reduce<E>(xs[0..i+1))` are semantic identities: both sides denote the same abstract evaluation of the named expression `E`. They are not equalities up to mathematical associativity. For floating-point and other order-sensitive operations, bitwise equality follows only under the floating-point evaluation constraints associated with the policy; the complementary problem of specifying those constraints is the subject of [P3375R3]. For user-defined operations with observable order, the same caveat applies: `==` means the same expression is observed, not merely that two values compare equal under some later equivalence relation.

The iterated pairwise expression described later has this property by definition. Other deterministic scan policies may be reproducible without providing this consistency guarantee.

Heterogeneous execution makes these distinctions unavoidable. A computation distributed across x64, ARM, GPU, and edge devices may require a portable expression contract, or it may prefer the fastest implementation-defined deterministic expression on each target. Neither choice is wrong, but the choice must be explicit.

The remainder of this paper is concerned with the scopes that require an expression contract, and in particular with the consistency property that arises when `reduce` and `scan` are specified as related algorithms rather than isolated implementations.

---

3. Expression is not execution

Specifying an expression does not specify an execution schedule.

An expression contract defines the construction of return values. It does not constrain evaluation order, thread assignment, materialization of intermediates, vector width, or tile size. The implementation remains free to use any evaluation strategy that preserves the specified observable values.

This separation is essential for standardization. The Standard must not over-constrain how implementations schedule and map work to hardware. CPU SIMD, many-core CPUs, and GPUs differ in their natural scheduling, vectorization, and memory-traffic patterns; those choices belong to the implementation, not the Standard.

The Standard Library already uses this separation. A sorting algorithm specifies the order relation that must hold in the output, not the comparison schedule. A deterministic expression contract should specify the result-defining structure, not the hardware schedule.

---

4. The `accumulate` and `partial_sum` example

The simplest way to introduce expression and observation is the relationship between `accumulate` and `partial_sum`.

Ignoring the separate `init` parameter for the moment, the ordinary sequential expression is a left fold. `accumulate` observes the final value of that expression family. `partial_sum` observes every prefix value.

For the input:

```
1, 2, 3, 4
```

the left-fold expression family is:

```
1
(1 + 2)
((1 + 2) + 3)
(((1 + 2) + 3) + 4)
```

So the two algorithms can be described as:

```
accumulate  = final_observation(left_fold_expression(xs));
partial_sum = prefix_observation(left_fold_expression(xs));
```

The following small example makes the distinction explicit. It is not intended as an efficient implementation; it is executable notation for the semantic model. The vector represents the values denoted by the expression family, not a required runtime representation.

```
#include <cassert>
#include <cstddef>
#include <vector>
```

```
std::vector<int> left_fold_expression(const std::vector<int>& xs)
{
    std::vector<int> prefixes;

    if (xs.empty()) {
        return prefixes;
    }

    int acc = xs[0];
    prefixes.push_back(acc);

    for (std::size_t i = 1; i < xs.size(); ++i) {
        acc = acc + xs[i];
        prefixes.push_back(acc);
    }

    return prefixes;
}

int final_observation(const std::vector<int>& expression)
{
    return expression.back();
}

std::vector<int> prefix_observation(const std::vector<int>& expression)
{
    return expression;
}

int main()
{
    std::vector<int> xs = {1, 2, 3, 4};

    auto expression = left_fold_expression(xs);

    auto accumulated = final_observation(expression);
    auto partial_sum = prefix_observation(expression);

    assert(accumulated == 10);
    assert((partial_sum == std::vector<int>{1, 3, 6, 10}));

    // The final prefix agrees with the final observation.
    assert(partial_sum.back() == accumulated);
}
```

The expression family is defined once, and the algorithms differ only in the observation taken from it. A real implementation is not required to materialize the expression

in this form. The point is semantic: the result is defined as if the named expression family had been constructed and the specified observation had been taken.

This example is deliberately simple because the left-fold expression exposes each prefix as the running accumulator. The more important case is the one that follows: for pairwise and blocked expressions, the root observation and the prefix observations are not merely different elements of a materialized sequence. They may require different observation rules over richer expression state. That is why `scan` forces the distinction between expression and observation.

This gives the bridge to deterministic parallel algorithms:

```
reduce<E> = root_observation(E(xs));
scan<E>   = prefix_observation(E(xs));
```

Here `E` denotes a named expression family, not merely a single root expression. It need not be a left fold. It might be a pairwise expression, a blocked dyadic expression, or an implementation-defined deterministic expression. The expression changes, but the model remains the same.

---

5. Reduce observes the root; scan observes prefixes

A deterministic `reduce` can be described as evaluating a named expression and returning its root value:

```
reduce<E>(first, last, init, op)
    = root_observation(expression_family<E>(first, last, init, op));
```

A deterministic `scan` is richer. It must return one value for each prefix of the input range. Here `E` denotes a prefix expression family, not merely a single full-range tree:

```
inclusive_scan<E>(first, last, out, op)
    = prefix_observations(expression_family<E>(first, last, op));
```

These equations are schematic. The treatment of `init`, aggregation type, and empty ranges is part of the expression contract and is discussed in §5.1.

For `reduce`, the returned value is the root of the full-range expression. For `scan`, the returned values are the prefix values defined by the same named expression family.

If `reduce` and `scan` use the same named expression `E`, the intended consistency property can be stated directly:

```
inclusive_scan<E>(first, last, out, op);

out[i]     == reduce<E>(first, first + i + 1, op);
out[n - 1] == reduce<E>(first, last, op);
```

This is not just a useful test. It is a semantic property. It says that `reduce` and `scan` are not unrelated deterministic algorithms: `reduce` returns the root value of the expression, while `scan` returns the corresponding prefix values.

For a simple left fold, those prefix values are naturally present:

```
1
(1 + 2)
((1 + 2) + 3)
(((1 + 2) + 3) + 4)
```

For a balanced tree or blocked dyadic expression, the situation is more subtle. A full-range pairwise reduction tree contains only the values produced by its specific topology; many input prefixes do not correspond to a single node of that tree. A blocked expression may contain block roots, local prefixes, and a specified rule for composing carries into those local prefixes. The expression contract must therefore define not only the root value, but the prefix values that `scan` returns.

That is the distinction `scan` makes unavoidable:

```
What expression defines the values?
Which values does the algorithm return?
```

5.1 Inclusive scan, exclusive scan, and initialization

Inclusive and exclusive scans differ in which prefixes they return. An inclusive scan returns the prefix including the current element. An exclusive scan returns the prefix before the current element, usually composed with an initial value.

For deterministic expression policies, initialization must be part of the contract. Where `init` is attached changes the expression. The final convention must therefore be chosen together with the API surface that exposes it. The examples below are not proposed semantics; they illustrate why initialization is a contract-level decision rather than a harmless overload detail.

Consider:

```
xs   = [1, 2, 3, 4]
init = 10
```

under the iterated pairwise expression of §6.3. The integer values are used only to make the expression structure readable. For ordinary integer addition the conventions below may evaluate to the same mathematical value. For floating-point and other order-sensitive operations, the same distinction can change the observable result.

One possible convention is *init-as-first-element*. The initial value is prepended to the input and the canonical pairwise expression is taken over the augmented sequence `[10, 1, 2, 3, 4]`. The inclusive scan expressions are then:

```
S[0] = 10 + 1
S[1] = (10 + 1) + 2
S[2] = (10 + 1) + (2 + 3)
S[3] = ((10 + 1) + (2 + 3)) + 4
```

Under this convention, supplying `init` changes the canonical bracketing of the input values themselves, because the expression is formed over `[init, x0, x1, x2, x3]`.

Another possible convention is *init-outside-prefix*. The canonical pairwise expression is taken over the input prefix alone, and `init` is attached outside that prefix expression as the left operand of a final `op`:

```
S[0] = 10 + 1
S[1] = 10 + (1 + 2)
S[2] = 10 + ((1 + 2) + 3)
S[3] = 10 + ((1 + 2) + (3 + 4))
```

Under this convention, the canonical tree of each input prefix is preserved; `init` is not part of that tree.

These are not different implementations of the same contract. They are different expression contracts. A reproducible scan policy must say which convention it uses.

Exclusive scan exposes the same issue. Under the init-as-first-element convention, the exclusive scan observes prefixes of the augmented sequence before the current input element:

```
S[0] = 10
S[1] = 10 + 1
S[2] = (10 + 1) + 2
S[3] = (10 + 1) + (2 + 3)
```

Under the init-outside-prefix convention, the exclusive scan observes the input prefix before the current element and attaches `init` outside that prefix expression:

```
S[0] = 10
S[1] = 10 + 1
S[2] = 10 + (1 + 2)
S[3] = 10 + ((1 + 2) + 3)
```

Again, these may agree for exact associative operations and disagree for floating-point or other order-sensitive operations. The distinction is not surface syntax; it is the expression being specified.

The specification must define where `init` is attached, what aggregation type is used, how empty ranges behave, how exclusive prefixes are represented, and whether the final scan value is required to agree with `reduce` under the same convention.

This paper does not choose between these conventions. A future wording paper must align the selected initialization rule with the selected API: for example, whether `init` is supplied as a seed, as a reduction-like initial value, as part of an expression policy, or through some other surface. The point here is only that initialization is semantic. It changes the expression contract.

5.2 Returned values are not necessarily expression state

An implementation may compute part of an expression and extract a value from it. That returned value is not automatically a valid replacement for the expression state from which it came.

This matters for `scan`. A local prefix value may be numerically equal to the root of a subexpression, but treating that scalar as composable state at the next level can change the specified expression. For exact associative arithmetic this may be harmless. For deterministic floating-point semantics, it can change the result.

The rule is:

```
expression state composes
returned values are observed
```

A root value may be the result of a subexpression, but the specification should not accidentally permit implementations to compose observed prefix values as if they were summaries when doing so would change later observable results.

The purpose of a named expression is therefore not merely to define the final reduction result. For `scan`, it must also define the prefix values and the rules by which those values are obtained.

---

6. Candidate expression structures

The proposal should not assume a single natural expression for all deterministic algorithms. Different expression structures provide different reproducibility contracts, and occupy different points in the design space: sequential compatibility, parallelism, accuracy, streamability, hardware locality, and consistency between `reduce` and `scan`.

For `scan`, the important point is that the input order of each prefix is preserved. The variation is in the parenthesization. A scan does not turn:

```
a0, a1, a2, a3
```

into an arbitrary permutation. It computes prefix values over that sequence. But for order-sensitive operations, different valid parenthesizations of the same prefix may produce different results:

```
(((a0 op a1) op a2) op a3)
```

and

```
((a0 op a1) op (a2 op a3))
```

are not necessarily equivalent. Reproducible `scan` therefore needs to name the parenthesization rule.

6.1 Left-fold expression

The left-fold expression is:

```
(((a0 op a1) op a2) op ... op aN)
```

Its strength is clarity. It matches the traditional sequential mental model used by `partial_sum` and by many simple scan implementations. It also naturally exposes scan prefixes: every inclusive prefix is the next value on the left spine.

Its weakness is the same structure. The expression has a linear dependency chain, exposes little parallelism, and for floating-point summation has the rounding behavior of a long sequential chain. It is therefore a poor default for high-performance parallel reduction, even though it gives the simplest scan/reduce consistency story.

The left fold remains useful as a teaching example and for users who require compatibility with traditional sequential accumulation. But it should not be the only deterministic expression.

6.2 Pairwise expression

A pairwise expression recursively combines neighboring elements in a balanced tree. For eight elements:

```
(((a0 op a1) op (a2 op a3)) op ((a4 op a5) op (a6 op a7)))
```

Its strengths are logarithmic depth, natural parallelism, and, for floating-point summation, the familiar numerical advantage of balanced summation over a long left fold [Higham2002]. Pairwise expression families are therefore attractive for deterministic `reduce`.

The difficulty appears when `scan` is added. A whole-range pairwise tree makes the root cheap to observe, but many prefixes are not nodes of that tree. A scan must therefore construct or observe prefix expressions separately. This can make a strict pairwise scan much more expensive than a scan expression designed around prefixes.

This is one of the main implementation lessons: an expression that is excellent for `reduce` may be expensive for `scan`.

6.3 Iterated pairwise expression

The iterated pairwise expression is the direct scan counterpart of canonical pairwise reduction. For an input sequence `X[0..N)`, it defines each inclusive scan output as the canonical reduction of the corresponding prefix:

```
S[i] = CANONICAL_TREE_EVAL(op, X[0..i+1))
```

This expression gives scan/reduce consistency by construction. It is the most direct scan analogue of the canonical reduction expression in P4016R0.

`canonical_reduce` observes the root of the canonical expression for a range. `canonical_scan` observes the canonical expression for each prefix of that range. They are different observations of the same expression family.

For the same operation, value type, and initialization convention:

```
canonical_scan(first, last, out, op);

out[i]     == canonical_reduce(first, first + i + 1, op);
out[n - 1] == canonical_reduce(first, last, op);
```

For eight elements, the scan expressions are:

```
S[0] = e0

S[1] = op(e0, e1)

S[2] = op(op(e0, e1), e2)

S[3] = op(op(e0, e1), op(e2, e3))

S[4] = op(op(op(e0, e1), op(e2, e3)), e4)

S[5] = op(op(op(e0, e1), op(e2, e3)),
          op(e4, e5))

S[6] = op(op(op(e0, e1), op(e2, e3)),
          op(op(e4, e5), e6))

S[7] = op(op(op(e0, e1), op(e2, e3)),
          op(op(e4, e5), op(e6, e7)))
```

The final scan value is the canonical pairwise reduction of the full range:

```
S[7] =
    ((e0 op e1) op (e2 op e3))
        op
    ((e4 op e5) op (e6 op e7))
```

This expression can be generated incrementally with a shift-reduce stack. Each new element is pushed as a singleton expression. When adjacent entries have the same span, they are combined in canonical left-to-right order.

The current scan value is then obtained by applying the canonical prefix-observation rule to the current stack state. The stack is expression state; observing it is not the same as performing an arbitrary fold over its entries. For non-associative operations, the observation rule must preserve the same parenthesization as `CANONICAL_TREE_EVAL` over the full prefix.

The strength of this expression is semantic clarity. It proves that canonical `reduce` and canonical `scan` can be specified as consistent algorithms. It is also online: it does not require foreknowledge of the full input length.

Its weakness is observation cost, but the asymptotic is not what it first appears. A naive observation strategy that folds the full shift-reduce stack at each emission gives `O(log n)` work per output and `O(N log N)` overall. By maintaining auxiliary observation state alongside the stack, and updating that state incrementally as elements are pushed and equal-sized spans are combined, observation can be made `O(1)` amortised per emission and the whole scan `O(N)`, while preserving the same canonical observation rule.

The naive `O(N log N)` cost is therefore a property of one realization, not an inherent property of the iterated pairwise expression. What does vary with the choice of expression is the constant factor of prefix observation, its streamability, and its mapping

to SIMD and tile execution. This is why §6.4 introduces the blocked dyadic expression as a different named contract rather than as a faster implementation of this one.

The iterated pairwise expression is an excellent semantic reference for consistency. It is not necessarily the best high-performance scan expression.

A fuller definition of the iterated pairwise expression, including exclusive scan, initialization, ragged tails, and a worked shift-reduce example, is given in Appendix A. The main text uses it only as one candidate named expression.

6.4 Blocked dyadic expression

A blocked dyadic expression is a practical compromise. It uses deterministic local structure inside fixed-size blocks, then deterministic dyadic composition between completed block states.

At a high level:

```
1. Partition the range into fixed-size blocks of B elements.
2. Evaluate a canonical local expression within each block.
3. Compose completed block states using a deterministic inter-block expression.
4. Define root and prefix results by a specified observation rule.
```

This gives a two-tier expression:

```
local block expression    deterministic within a block
inter-block expression    deterministic across block states
```

The block size `B` is part of the expression contract, not merely a performance hint. The blocked dyadic expression is a *family* parameterized by `B`; each value of `B` instantiates a distinct expression contract with its own bracketing rule and, for floating-point and other order-sensitive operations, its own observable results. Two implementations agree on results only if they share both the named expression family and the same value of `B`.

For example, with block size four:

```
B0 = ((a0 op a1) op (a2 op a3))
B1 = ((a4 op a5) op (a6 op a7))
```

The reduction result is:

```
B0 op B1
```

The scan prefixes in the second block are formed from the completed first block and local prefixes of the second block:

```
out[4] = B0 op a4
out[5] = B0 op (a4 op a5)
out[6] = B0 op ((a4 op a5) op a6)
out[7] = B0 op B1
```

This is still a scan: it returns one prefix value per input element in the original sequence. What has changed is the named expression family used to parenthesize those prefix values.

The blocked dyadic expression is attractive because it maps well to SIMD blocks, GPU tiles, cache-local execution, and streaming implementations. It also gives deterministic inter-block structure, efficient root observation, and a precise rule for composing scan carries.

The blocked dyadic expression is not a pure pairwise tree over all elements, and it is not merely a faster implementation of the iterated pairwise expression. It is a different named expression with different operational and numerical properties. A blocked dyadic scan is consistent with the corresponding blocked dyadic reduction using the same block size and expression rule. It is therefore not required to agree bitwise with an unblocked canonical pairwise reduction.

The standards-relevant lesson is that blocked expressions name an expression suited to tiled, vector, or GPU execution, rather than treating every efficient implementation as a schedule for the same abstract tree.

6.5 Precise and fast scan over a blocked expression

The blocked expression exposes a useful distinction between precise and fast scan variants.

For a prefix inside block `k`, the logical prefix consists of:

```
all complete blocks before k
plus the local prefix inside block k
```

The precise observation rule is determined by the selected expression. For the iterated pairwise expression, the reference value for a prefix is the canonical reduction of that prefix. For a blocked dyadic expression, the reference value is the prefix observation defined by that blocked expression, using its block size and carry-composition rule.

A precise scan composes the inter-block state for the previous blocks with the local prefix using the expression rule required for consistency with `reduce`.

A fast scan may compute a block carry once and combine it with each local prefix using a cheaper bracketing. For exact associative arithmetic these may agree. For floating-point arithmetic they may differ; for example, `(1e16 + -1e16) + 1` evaluates to `1.0` while `1e16 + (-1e16 + 1)` evaluates to `0.0` under ordinary `double` round-to-nearest evaluation.

This distinction matters. A fast scan can be deterministic without being consistent with the corresponding `reduce`. A consistent scan must share the required expression relationship with `reduce`.

The useful slogan is:

```
Reproducibility is not consistency.
```

A proposal should not hide this difference. If consistency is required, the precise observation is the semantic reference. If a faster deterministic scan is useful, it should be named as a different expression or observation policy.

6.6 Implementation-defined deterministic expression policies

An implementation-defined deterministic expression is an implementation-provided expression designed for a particular target. It may use tile shapes, vector widths, warp-level primitives, or carry propagation schemes that reflect the hardware.

This category matters because forcing one portable expression onto every target may be unreasonably expensive. A CPU-friendly expression may be poor for a GPU; a GPU-friendly expression may be unnecessary or awkward on a CPU.

The Standard should distinguish two categories of expression policy:

```
portable named expression policy
    semantics specified by the Standard;
    supports cross-implementation reproducibility.

implementation-defined deterministic expression policy
    semantics documented by the implementation;
    may be target-shaped; not portable across implementations by default.
```

Both categories may be parameterized families. For example, `canonical_block_dyadic<B>` is a portable parameterized family; an implementation-defined GPU expression may itself be parameterized over tile size or warp width. The two categories are distinguished by who specifies the semantics, not by whether the family takes parameters.

An implementation-defined deterministic expression should still be named and documented. Users should be able to distinguish, for example:

```
pairwise_expression                         (portable)
block_dyadic_expression<64>                 (portable, parameterized)
implementation_defined_expression           (implementation-defined)
```

Only the portable expressions can promise cross-implementation reproducibility if their semantics are standardized. An implementation-defined deterministic expression can still provide valuable repeatability, consistency, and performance within the implementation that defines it.

---

7. Relationship to P4016R0

P4016R0 can be understood as the first instance of the more general expression/observation model. It addresses the root observation of a fixed expression. In that vocabulary:

```
P4016 reduce       = root observation of a named expression
deterministic scan = prefix observations of a named expression
```

The scan proposal is therefore not a separate idea bolted onto reduction; it is the natural next step. `reduce` demonstrated why fixing the expression matters. `scan` demonstrates why the observation surface matters too.

Deterministic `scan` is not “parallel scan with a stable schedule”. It is an algorithm whose output is defined by an expression contract and an observation contract.

---

8. Operational and numerical properties of expression choice

For `scan`, expression choice is an operational policy. It is also a numerical policy.

Reproducibility alone is not enough. A reproducibly poor answer may not satisfy users; conversely, the most accurate expression may not be affordable on every hardware target, or may not support streaming `scan` efficiently. The proposal should therefore not imply that deterministic means numerically best. It should make the expression visible so users can choose among concrete goals corresponding to the trade-off axes introduced earlier:

- online evaluation
- bounded buffering
- streaming output
- traditional left-fold compatibility
- pairwise accuracy
- blocked SIMD efficiency
- scan/reduce consistency
- platform-specific implementation-defined deterministic throughput
- portable cross-ISA reproducibility

These are different goals. The Standard should provide a way to select among them.

8.1 Online evaluation, bounded buffering, and streaming output

Three operational properties of a named expression are related but distinct:

Property	Meaning
Online evaluation	The expression can be formed incrementally as inputs arrive; the full range length need not be known in advance.
Bounded buffering	The implementation need retain only bounded state, such as a stack, block, tile, or carry state, rather than materializing the whole input sequence.
Streaming output	Prefix results can be emitted progressively, rather than held until the entire input has been consumed.

These properties matter for `scan`, because `scan` is naturally prefix-producing. Users may expect outputs to be produced as inputs are consumed, and range-based designs should not accidentally require the entire input sequence to be stored before the first result can be formed.

The iterated pairwise expression has an important online property. It can be generated by a shift-reduce stack: each new element is pushed as a singleton expression, adjacent equal-sized spans are combined, and the full input length is not needed in advance. A naive observation strategy may fold the current stack at each output, but implementations can maintain auxiliary observation state and make prefix observation O(1) amortised. The relevant distinction is therefore not asymptotic inevitability, but the constant factors, regularity, buffering, and SIMD/tile mapping of the observation strategy.

A blocked dyadic expression also has a useful online form. A fixed-size block can be evaluated locally, completed block states can be composed by an inter-block expression, and scan values can be emitted using the specified carry-composition rule. This may require buffering only the current block or tile, rather than materializing the whole sequence.

This is another reason expression choice must be visible. Two expressions may both be reproducible, but differ materially in whether they support online evaluation, bounded buffering, or streaming output. A proposal should avoid defining reproducibility in a way that forces whole-sequence materialization unless that is an explicit property of the selected expression.

8.2 Error properties as part of documentation

Expression choice is also numerical policy.

For floating-point summation, expression choice affects numerical error. A left-fold expression has a linear-depth dependency chain: each new value is rounded into the running accumulator. A pairwise expression has logarithmic depth and gives the familiar logarithmic-depth error behavior under the usual floating-point error model [Higham2002]. SIMD-oriented pairwise summation has also been studied as a throughput/accuracy trade-off [Dalton2014].

A blocked dyadic expression occupies another point in the design space: deterministic local structure inside blocks, followed by deterministic composition between blocks. Its numerical behavior is not identical to either a pure left fold or a pure pairwise tree.

A named expression should therefore be documented not only by its tree shape, but also by the numerical consequences a user can reasonably infer.

Expression	Structure
`left_fold_expression`	linear-depth dependency chain
`pairwise_expression`	logarithmic-depth tree
`block_dyadic_expression<B>`	local block structure plus inter-block dyadic tree
`implementation_defined_expression`	implementation-documented structure

The Standard need not promise a particular forward-error bound for arbitrary user-defined operations, but it can specify enough structure that numerical experts can

reason about error behavior for common operations such as floating-point summation.

8.3 Accuracy versus consistency

Accuracy and consistency are distinct properties.

A deterministic `scan`/`reduce` pair can be made consistent by defining both algorithms over the same named expression. The scan and reduce then agree, and both are reproducible. But the chosen expression may have different numerical properties from a full pairwise expression over all elements.

This is a selectable trade-off, not a failure. The proposal should let users request scan/reduce consistency without implying that all deterministic expressions are equally accurate.

---

9. Execution policy and scheduling

An expression contract specifies the value to be observed. It does not specify the order in which an implementation performs the work.

The proposed model separates three concerns:

```
execution policy       how work is mapped and scheduled
expression contract    what abstract expression defines the values
observation contract   which values of that expression are returned
```

A standard facility should specify the latter two. It should avoid specifying the first except where existing execution-policy wording already imposes constraints.

A conforming implementation of a named blocked dyadic scan, for example, may use vector instructions within a block, multiple worker threads across blocks, a GPU block or warp-level primitive, a precomputed tree of block summaries, a look-back scheme, a two-pass implementation, or another strategy entirely, provided that the returned values are the specified observations of the named expression.

This separation is especially important for `scan`. A naive attempt to make scan reproducible by fixing a particular schedule risks specifying an implementation technique rather than a semantic result.

---

10. Standard wording direction

This paper is not proposing final wording. It proposes a direction for wording.

The central standardization question is not whether one particular tree should be mandated for all deterministic algorithms. The question is whether the Standard should provide a way to name reproducibility contracts.

A possible direction is to specify a family of expression policies. Each expression policy would define:

1. the abstract expression associated with a range, operator, and optional initial value;
2. the observations provided by `reduce`, `inclusive_scan`, and `exclusive_scan`;
3. the requirements on the operator and value types;
4. the treatment of empty ranges and initialization;
5. the consistency properties, if any, between algorithms using the same policy.

Illustratively, one might describe such a facility as:

```
reproducible_reduce(policy, first, last, init, op);

reproducible_inclusive_scan(policy, first, last, result, op);

reproducible_exclusive_scan(policy, first, last, result, init, op);
```

This syntax is illustrative only and is not proposed API. The important point is not the exact spelling. The important point is that the reproducibility contract is explicit and selectable.

An expression policy should not be merely an execution hint. It names the expression contract that defines the result.

For a policy `E`, the specification may state properties such as:

```
reduce<E>(xs)
    returns the root observation of E(xs)

inclusive_scan<E>(xs)[i]
    returns the inclusive prefix observation of E(xs) at position i
```

This notation is illustrative and is not proposed API syntax.

For policies that provide scan/reduce consistency, the specification may additionally state:

```
inclusive_scan<E>(xs)[i]
    is equal to reduce<E>(xs[0..i+1))
```

where equality means that both sides denote the same abstract evaluation of the named expression `E`, under the specified arithmetic environment. It is not equality up to mathematical associativity.

This distinction matters for floating-point operations, saturating arithmetic, user-defined operations with observable order, and other operations for which associativity cannot be assumed.

The wording should also be explicit that returned scan values need not be reusable expression states. A scan result is an observation. An implementation may maintain richer internal state than the value written to the output iterator.

That distinction prevents accidental wording that requires implementations to compose observed prefix values as if they were summaries. Summaries compose. Observations are returned.

---

11. Informative implementation experience

The framework is supported by implementation work on CPU SIMD targets (AVX2, NEON) and CUDA. The details below are informative; they are not proposed wording.

The most important lesson from that work is that `scan` exposes expression choice in a way reduction alone does not: a full pairwise tree attractive for `reduce` may be awkward for `scan` because many prefix values are not nodes of the full-range tree. This motivates blocked dyadic and implementation-defined deterministic expressions, which preserve a named expression contract while making prefix observation more implementation-friendly. CUDA work additionally separates *readiness* from *expression*: which metadata is ready is an execution fact, while which subexpressions may be combined is a semantic fact. Appendix B.8 reports a measured Tesla T4 result showing scan/reduce consistency at the bit level across 1M elements, alongside an existing GPU baseline whose scan and reduce do not agree with each other on the same input. The detail is in Appendix B.

The standardization-relevant observation is general:

> Different named expressions have different operational and numerical properties. A reproducible facility should allow the contract to be named rather than hidden inside an implementation schedule.

---

12. Design principles

The preceding sections suggest the following design principles.

Principle	Consequence
Specify observable results, not execution schedules.	A reproducible algorithm should define the expression and observations that determine its results. It should not prescribe a thread schedule, vector width, tile shape, carry-propagation protocol, or materialization strategy.
Make reproducibility an explicit contract.	A user who requires reproducible results should be able to select a named expression contract rather than rely on an implementation habit.
Treat expression choice as operational and numerical policy.	The selected expression affects buffering, streaming behavior, parallel structure, numerical error, performance, and hardware suitability.

Principle	Consequence
Distinguish reproducibility from consistency.	A deterministic scan may be reproducible without being consistent with a deterministic reduce. Consistency should be an explicit property of the selected expression and observation contracts.
Do not treat returned observations as composable expression state.	A scan output is an observed value. It is not necessarily the internal state required to continue the expression.
Allow both portable and implementation-defined contracts.	Portable named expressions can support cross-implementation agreement. Implementation-defined deterministic expressions can provide documented, stable behavior within a stated implementation or target domain.

Two distinctions are especially important.

Reproducibility asks:

`Will this algorithm produce the same specified result?`

Consistency asks:

`Do two algorithms observe the same expression contract in compatible ways?`

A fast deterministic scan may always produce the same result for a given implementation and policy, yet still differ from a deterministic reduce over the same range if the two algorithms use different expression contracts. Conversely, a scan/reduce pair can be made consistent by specifying both algorithms as compatible observations of the same named expression family.

The same care is needed for observations and expression state. A returned prefix value may be numerically equal to the root of some subexpression, but it is not automatically valid as composable state for a larger expression. The specification should not accidentally permit implementations to compose observed prefix values as if they were summaries when doing so changes later observable results.

Together, these principles lead to the same standardization direction: name the expression and observation contracts that define the result, while preserving implementation freedom for scheduling, vectorization, tiling, buffering, and hardware mapping.

---

13. Analogy with random-number generation

The Standard Library already contains a useful precedent: random-number generation.

The library does not provide only one source of randomness. It provides named engines and distributions. A user can choose an engine because its state transition and output properties matter.

For example, users may care about:

- reproducibility
- period
- state size
- statistical quality
- speed
- portability
- suitability for a given simulation

The same pattern applies to reproducible numeric algorithms.

There need not be one universal deterministic expression for all uses. Instead, the Standard can provide named expressions with documented properties.

A user might choose:

```
left_fold_expression
    for compatibility with sequential accumulation

pairwise_expression
    for balanced reduction structure

iterated_pairwise_expression
    for direct scan/reduce consistency

block_dyadic_expression<B>
    for bounded local structure and practical scan implementation

implementation_defined_expression
    for implementation-documented deterministic throughput
```

The names above are illustrative, not proposed final API.

The analogy is not exact. Random-number engines generate sequences, while named expressions define abstract computations. But the standardization pattern is similar: expose the contract that determines reproducibility-relevant behavior, rather than leaving it implicit.

---

14. Non-goals

This proposal does not attempt the following:

- **Make all existing parallel algorithms reproducible by default.** That would be a substantial semantic change. Existing parallel algorithms intentionally allow implementation freedom, and for many uses that freedom is valuable.

- **Require deterministic algorithms to be bitwise identical to existing sequential algorithms.** For order-sensitive operations, different expression structures may legitimately produce different values. The purpose of a named expression is to make the chosen value reproducible, not to assert that it is the only reasonable value.
- **Mandate a single expression for all implementations and all hardware.** A single mandated expression may be useful for some portable facilities, but it should not preclude other named contracts with different trade-offs.
- **Specify a GPU algorithm, SIMD algorithm, thread schedule, or buffering strategy.** Implementation techniques are evidence that the model is practical. They are not the specification.
- **Solve numerical accuracy in general.** A reproducible expression may be more accurate, less accurate, or differently accurate than another expression. Accuracy properties should be documented where relevant, but reproducibility and accuracy are not the same property.
- **Require users to abandon existing `reduce`, `inclusive_scan`, or `exclusive_scan`.** The intent is to provide an explicit facility for users who require reproducibility, consistency, or documented expression semantics.

---

15. Questions for SG6 and SG1

The paper asks for direction rather than presenting a fully closed design. If SG6 and SG1 agree with the model, a later revision can bring API alternatives to LEWG.

The questions below are divided into primary direction questions and secondary design questions.

The primary direction questions are:

- Is expression/observation the right abstraction for specifying reproducible `reduce` and `scan`? (§15.1)
- Should reproducibility contracts be named and selectable? (§15.2)
- Should scan/reduce consistency be a separately named property of some expression policies? (§15.4)

Positive direction on these questions would indicate that a follow-on wording paper should explore named expression and observation policies. Negative direction on any of them should be discussed before spending meeting time on the more detailed design questions.

The remaining questions — which expression policies to consider, how initialization should be specified, how portable and implementation-defined policies should be distinguished, and how expression policies should relate to execution policies — are secondary design questions for a follow-on wording paper. Informal direction on them is still useful, but they need not be settled before deciding whether the model is worth pursuing.

15.1 Is the expression-and-observation model the right abstraction?

Do SG6 and SG1 agree that reproducible `reduce` and `scan` should be specified by separating:

```
expression contract
observation contract
execution strategy
```

rather than by specifying an implementation schedule?

15.2 Should reproducibility contracts be named and selectable?

Should the Standard expose named expressions through expression policies, rather than defining only one deterministic algorithm?

15.3 Which expression policies are worth standardizing?

Possible portable candidates include:

```
left_fold_expression
pairwise_expression
iterated_pairwise_expression
block_dyadic_expression<B>
```

A separate question is whether the Standard should also provide a place for documented implementation-defined deterministic expression policies, and what requirements such policies must satisfy.

The question is not whether these are final names. The question is which semantic contracts are useful enough to standardize.

15.4 Should scan/reduce consistency be a separately named property?

For some policies, it is valuable to guarantee:

```
inclusive_scan<E>(xs)[i] == reduce<E>(xs[0..i+1))
```

For other policies, deterministic throughput may be more important than this consistency property.

Should the Standard distinguish consistency-preserving expression policies from throughput-prioritized deterministic policies?

15.5 How should initialization be specified?

Initialization affects the expression.

The paper should ask how much should be fixed for:

```
reduce with init
inclusive_scan with and without init
exclusive_scan with init
```

```
empty ranges
type conversion and the aggregation type
```

These details are necessary for a real wording paper.

15.6 How should portable and implementation-defined expressions be distinguished?

A portable expression policy can provide cross-implementation agreement. An implementation-defined deterministic policy can provide deterministic results with better use of hardware-specific features. Both categories are valuable. The question is how the Standard should distinguish them — for example, by reserving certain names, requiring documentation, or scoping conformance claims to the originating implementation.

15.7 What is the right relationship to existing execution policies?

The design should not confuse expression policy with execution policy.

A possible design may allow both:

```
execution policy:   where and how work is scheduled
expression policy:  what result is specified
```

SG6 and SG1 should advise whether these should be separate parameters, combined policy objects, customization points, or a different mechanism.

---

16. Conclusion

P4016R0 showed that reproducible parallel reduction can be specified by naming the expression whose root defines the result.

Parallel scan exposes the next layer of the problem. A scan does not observe only the root of an expression. It observes prefix values. Therefore a reproducible scan needs both an expression contract and an observation contract.

This distinction also clarifies the relationship between `reduce` and `scan`.

A deterministic `reduce` is a root observation of a named expression. A deterministic `scan` is a family of prefix observations of a named expression. When both algorithms use the same named expression and compatible initialization rules, scan/reduce consistency can be specified directly.

The choice of expression is not merely formal. Implementation experience suggests that it affects performance, scalability, buffering, streamability, numerical behavior, and hardware suitability. Some expressions are natural for reduction but costly for scan. Others are shaped to make prefix observation efficient. This is a design space, not a single point.

The Standard should specify the result by naming the expression and the observations, leaving implementations free to choose efficient schedules that preserve those observable values.

Named abstract expressions provide the missing specification layer: selectable reproducibility contracts for parallel `reduce` and `scan`, each admitting multiple conforming implementations.

Appendix A: Direct Iterated Pairwise Scan Policy

This appendix records the most direct scan policy derived from P4016R0. The reduction expression rule is unchanged: each scan output is defined as the canonical reduction of the corresponding prefix. What changes is the observation pattern. Reduction observes one root value; scan observes one root value for each prefix.

The purpose of this appendix is not to propose that this policy is the only useful reproducible scan contract. It is a semantic reference point: a policy that gives scan/reduce consistency by construction.

A.1 Inclusive iterated pairwise scan

For an input sequence `X[0..N)` and binary operation `op`, the inclusive iterated pairwise scan produces an output sequence `S[0..N)` where:

```
S[i] = CANONICAL_TREE_EVAL(op, X[0..i+1))
    for i in [0, N)
```

`CANONICAL_TREE_EVAL` is the canonical reduction expression defined by P4016R0. Each scan output is therefore the value of a fixed abstract expression over the corresponding prefix.

For a given input order, operation, value type, and floating-point evaluation model, the expression at every output position is uniquely determined.

A.2 Exclusive iterated pairwise scan

For an input sequence `X[0..N)`, binary operation `op`, and initial value `init`, one possible exclusive convention is:

```
S[0] = init_A

S[i] = op(init_A, CANONICAL_TREE_EVAL(op, X[0..i)))
    for i in [1, N)
```

Here `init_A` denotes the initial value after any conversion to the aggregation type required by the selected convention. This convention places the initial value outside the tree as the left operand of a final application of `op`.

Final wording would need to align the treatment of initialization, aggregation type, empty ranges, and conversion rules with the corresponding `reduce` convention.

A.3 Scan/reduce consistency

The iterated pairwise expression gives scan/reduce consistency by definition:

```
iterated_pairwise_scan(op, X[0..N))[i]
    == canonical_reduce(op, X[0..i+1))
```

and, in particular:

```
iterated_pairwise_scan(op, X[0..N))[N-1]
    == canonical_reduce(op, X[0..N))
```

This is not a performance claim. It is a semantic identity. Both sides evaluate the same abstract expression.

A.4 Reference shift-reduce construction

The iterated pairwise expression can be generated online by a shift-reduce stack.

Each new element is pushed as a singleton expression. Adjacent entries of the same span are combined in canonical left-to-right order. The stack therefore represents expression state for the prefix consumed so far.

The scan output is obtained by applying the prefix-observation rule to that stack state. This observation must preserve the same parenthesization as `CANONICAL_TREE_EVAL` over the full prefix. It is not an arbitrary fold over the stack entries.

This construction is executable notation for the policy. It does not constrain implementation strategy.

A.5 Ragged tails and absent operands

In vectorized or blocked implementations, a final partial block may be represented using absent operands. The semantic input remains the actual sequence `X[0..N)`. No padding value is introduced and no identity element is required.

A conceptual implementation may use a maybe-domain rule:

```
COMBINE(present x, present y) = present op(x, y)
COMBINE(present x, absent)    = present x
COMBINE(absent, present y)    = present y
COMBINE(absent, absent)       = absent
```

This is an implementation aid for fixed-size local trees. The observable scan values are emitted only for present input positions.

A.6 Numerical properties for floating-point summation

For floating-point summation under the usual error model, the canonical pairwise expression has logarithmic-depth error behavior for each prefix [Higham2002], in

contrast to the linear-depth chain of a left fold.

This illustrates the main paper's point: expression choice is also numerical policy.

The Standard need not specify such a bound for arbitrary `binary_op`, but a named expression gives users enough structure to reason about numerical consequences.

A.7 Limitations of the direct policy

The iterated pairwise expression is simple and scan/reduce consistent. It is also useful as a semantic reference.

Its primary limitation is not asymptotic. Naive observation that folds the full stack at each emission is `O(log n)` per emission and `O(N log N)` overall. Observation that carries per-level running-prefix state alongside the shift-reduce stack and updates it incrementally can be `O(1)` amortised per emission and `O(N)` overall, while preserving the same canonical observation rule.

The amortised argument is the same as for binary-counter increments: over `N` pushes, the total number of equal-span combine events is `O(N)`, even though a single push may occasionally cascade through `O(log N)` levels.

The real limitation is operational. Strict iterated pairwise observation maintains a stack of spans of mixed sizes, which does not map naturally onto fixed-width SIMD blocks, GPU tiles, or other regular execution units. This motivates blocked dyadic and implementation-defined deterministic expressions. Such expressions are not merely alternative implementations of the same iterated pairwise expression. They are distinct named contracts with different operational and numerical properties, chosen for streamability, SIMD/tile mapping, and bounded local buffering rather than for asymptotic reasons.

Appendix B: Implementation Experience

This appendix records implementation experience relevant to the proposal. The details are informative. They are evidence for the expression/observation model, not proposed wording.

B.1 CPU SIMD experience

CPU SIMD implementations suggest that blocked expressions can be made both efficient and semantically clean.

A fixed block size allows a deterministic local expression to be evaluated using vector registers and straight-line code. Completed block states can then be composed using a deterministic higher-level expression. The resulting implementation may provide:

- reproducible local scan values;
- deterministic block summaries;
- scan/reduce consistency for the named blocked expression;

- efficient use of SIMD lanes;
- bounded local buffering.

The important observation is that block size and local expression shape are semantic choices if they affect the result. They should therefore be part of the named expression, not hidden as incidental tuning parameters.

B.2 CUDA scan as a stress test

CUDA scan is a useful stress test for the expression/observation model. A reduction observes one root value. A scan must produce every prefix value. On GPU hardware this interacts strongly with global memory traffic, occupancy, synchronization, and carry propagation.

A conventional efficient GPU scan tries to stay close to the ideal traffic model: read the input once and write the output once. Hierarchical deterministic scans can easily require additional global passes: local scan output, block totals, a scan of block totals, and a carry-add pass. The extra traffic may dominate.

This matters for standardization. A reproducible scan facility should not mandate a CPU-centric expression or a particular multi-pass schedule. It should specify observable values and leave implementations free to use hardware-appropriate execution strategies.

B.3 Readiness is not expression

Efficient GPU scans often use look-back protocols, notably the decoupled look-back family of Merrill and Garland [Merrill2016]. A tile computes a local scan, publishes metadata, and later tiles determine their carry by looking back through published summaries.

For deterministic semantics, the critical issue is not when metadata becomes ready. The critical issue is which expression the metadata represents.

Readiness is an execution fact. Expression is a semantic fact. A deterministic look-back protocol may poll, wait, or use available metadata in different ways, but all permitted paths must produce the same specified expression. If using whichever fragment is ready changes the bracketing or operand order, scheduling has leaked into the result.

A deterministic protocol must therefore specify, or be constrained by, rules such as:

- which spans may be used;
- how spans are combined;
- which operand order is used;
- when a parent summary is valid;
- whether children may be used instead of a parent;
- how fallback paths preserve the same expression.

B.4 Owner publication and ready-only observation

One class of CUDA prototypes used owner publication: responsibility for publishing parent summaries was assigned deterministically. Consumers could observe ready metadata, but the protocol was constrained so that observations preserved the intended dyadic expression.

This was semantically attractive. It avoided opportunistic synthesis in which a consumer combines available fragments in a schedule-dependent order. The cost was additional metadata traffic and waiting in the carry path.

This experience supports the paper's distinction between expression and execution. The implementation may schedule work asynchronously, but the expression being observed must remain fixed.

B.5 Chunk-level expressions

A later class of prototypes grouped tiles into chunks. The chunk owner constructed and published chunk-level summaries, reducing the amount of global look-back work required by individual tiles.

This changed the engineering trade-off. Chunk-level ownership reduced carry attempts, metadata reads, and parent waits in the prototypes that were measured. However, some diagnostic versions paid additional local scan cost, moving the implementation away from the ideal traffic model.

The important standards lesson is not the particular chunk size or CUDA protocol. The lesson is that a tile or chunk expression may be a better deterministic contract for a GPU than a strict unblocked pairwise expression. That expression should be named and documented rather than hidden inside an implementation schedule.

B.6 Accuracy, consistency, and throughput

The CUDA work also illustrates that accuracy, consistency, reproducibility, and throughput are distinct properties.

A full pairwise expression may provide attractive numerical properties for floating-point summation. A tile-based expression may map better to GPU scan. A deterministic scan and deterministic reduce can be made mutually consistent by using the same tile or chunk expression, even if that expression is not the same as an unblocked pairwise expression over all elements.

This is not a defect. It is a policy choice. The user should be able to know which expression was selected and which properties follow from it.

B.7 Correctness reference

The correctness criterion for these prototypes was bit equality with the selected expression reference. CUB-style scans (referring to NVIDIA's CUB primitive library) are useful performance baselines, but unless they expose the same named expression, they are not semantic references for this proposal.

The right comparison has two dimensions:

```
semantic comparison:
    does the implementation match the named expression?

performance comparison:
    how close is it to a highly tuned scan baseline?
```

B.8 Measured CUDA results

A standalone CUDA harness implements a blocked dyadic expression aligned to the `canonical_block_dyadic` semantics described in §6.4. The implementation uses a three-pass structure (block roots, global tree, prefix emission) deliberately, so that readiness does not leak into the expression. Measurements were taken on an NVIDIA Tesla T4 with `N = 1,048,576` `double` inputs, block size `B = 256`, and `--fmad=false` to prevent fused multiply-add contraction.

The semantic results agree exactly with the expression contract. CUB and the canonical implementation evaluate different expressions. Bit-disagreement between them is therefore expected and is the point being witnessed, not a defect in CUB.

Property	Result
canonical scan back element vs canonical reduce	bit-exact (`0x406fef4dbe54a0f8`)
canonical scan back element vs host canonical reduce	bit-exact (`0x406fef4dbe54a0f8`)
sampled mid-range prefixes vs host reference	22 of 22 bit-exact
CUB `inclusive_scan` back element vs CUB `reduce`	differ (`0x...a0eb` vs `0x...a0d7`)
CUB `inclusive_scan` vs canonical `inclusive_scan`	975,698 of 1,048,576 elements differ
canonical scan throughput vs CUB scan	5.115× slower
canonical reduce throughput vs CUB reduce	1.922× slower

Two observations are relevant to the proposal. First, the canonical implementation demonstrates that scan/reduce consistency is achievable in practice on GPU hardware: the final scan output and the reduce output denote the same value of the same named expression, and the agreement is bit-exact. Second, CUB's existing `inclusive_scan` and `reduce` are each individually deterministic on this hardware, yet they do not agree with each other on the same input. This is precisely the gap §2 and §6.5 describe between reproducibility and scan/reduce consistency: each algorithm is reproducible under its own implicit expression, but the two expressions are not the same.

The performance gap is informative but not part of the proposal. The canonical implementation is 5× slower than CUB on scan because it uses additional global memory

traffic to keep the expression contract independent of look-back readiness. Closing that gap is an implementation question within the contract, not a question about the contract itself. The standardization-relevant lesson is that an implementation can pay a measured cost for an explicit expression contract, and that the cost is bounded enough to be a real engineering trade-off rather than a prohibitive one.

B.9 Lesson for the proposal

The implementation work supports the main proposal: `scan` makes expression choice visible in a way reduction alone does not. Pairwise, blocked dyadic, tile-based, and implementation-defined deterministic expressions occupy different points in the design space.

The Standard should therefore not prescribe a memory-traffic pattern, carry protocol, tile size, or look-back scheme. It should specify observable values. Scan/reduce consistency remains a valuable additional property, but it is a selectable property of the chosen expression and observation contracts.

Appendix C: Executable CUDA artifact

The following Compiler Explorer artifact is informative and is not proposed wording:

P4016 canonical CUDA scan/reduce witness
URL: https://godbolt.org/z/qMa9bs3zh

The artifact demonstrates the expression/observation model using a P4016-style canonical prefix expression. The central relationship is:

```
canonical reduce = root observation of a named expression family
canonical scan   = prefix observations of the same expression family
```

The CUDA code treats the input as 128 semantic lanes, groups rows into fixed-size chunks, builds a canonical chunk expression, and then observes that expression in two ways. The reduction observes the root. The scan observes prefixes. The key semantic check is therefore bitwise equality of the final scan observation and the corresponding reduction root:

```
bits(scan.back()) == bits(canonical_reduce)
```

The artifact also checks selected prefix observations directly:

```
scan[i] == canonical_reduce(input[0..i+1))
```

In the shown hostile floating-point case, the artifact reports that `scan.back()` agrees bitwise with the canonical reduction, repeated scans produce the same full output array, and 512 selected prefix probes match direct canonical prefix reductions. CUB is included only as a performance contrast; it is not the semantic reference for the named P4016 expression. On hostile floating-point inputs, CUB and the selected expression are expected to produce different bit patterns.

Appendix D: Executable CUDA witness — canonical_block_dyadic expression

The following Compiler Explorer artifact is informative and is not proposed wording:

canonical_block_dyadic CUDA scan/reduce witness
URL: https://godbolt.org/z/aqfY6TWas

The artifact witnesses the `canonical_block_dyadic` expression of §6.4. Where Appendix C exhibits prefix observation of the iterated pairwise expression, this artifact exhibits prefix observation of the two-tier blocked dyadic expression: a canonical local expression within each fixed-size block, composed across completed block states by a deterministic inter-block expression. The central relationship is:

```
canonical reduce = root observation of the canonical_block_dyadic expression
canonical scan   = prefix observations of the same expression
```

The CUDA code partitions the input into fixed-size blocks of `B` `double` elements, evaluates a canonical pairwise expression within each block, and composes completed block states using a canonical dyadic tree over block roots. Prefix observation follows the carry-composition rule specified by the blocked expression: for a position `j` inside block `b`, the scan output is `init + (D_b + P_b(j))`, where `D_b` is the canonical inter-block prefix of completed blocks `[0..b)` and `P_b(j)` is the canonical local prefix within block `b` up to position `j`. At block boundaries this reduces to `init + D_{b+1}`, attaching the carry as a peer to the next completed block state rather than absorbing it into local prefix arithmetic. Readiness of block roots and inter-block summaries is computed in a deliberately separate pass from prefix emission, so that the named expression is not affected by the order in which metadata becomes available. The key semantic check is therefore bitwise equality of the final scan observation and the corresponding reduction root:

```
bits(scan.back()) == bits(canonical_reduce)
```

The artifact also checks selected prefix observations directly against the canonical_block_dyadic reference computed on the host:

```
scan[i] == canonical_block_dyadic_reduce(input[0..i+1))
```

On the reported Tesla T4 run with `N = 1,048,576` `double` inputs, block size `B = 256`, and `--fmad=false`, the artifact reports that `scan.back()` agrees bitwise with the canonical reduction, sampled mid-range prefix probes match the host canonical_block_dyadic reference, and the canonical scan is approximately 5.292× slower than CUB scan while the canonical reduce is approximately 1.997× slower than CUB reduce. The gap is implementation cost within the contract, not a property of the contract itself: the three-pass structure trades global memory traffic for an expression that is independent of look-back readiness, and closing that gap is an implementation question pursued in a later revision.

Appendix E: Executable CPU witnesses — canonical_block_dyadic expression on x86-64 and AArch64

The following Compiler Explorer artifacts are informative and are not proposed wording:

- canonical_block_dyadic CPU scan/reduce witness — x86-64 (AVX2)
  URL: https://godbolt.org/z/nEae7MGW7
- canonical_block_dyadic CPU scan/reduce witness — AArch64 (NEON)
  URL: https://godbolt.org/z/qEv6YnG54

Both artifacts witness the same `canonical_block_dyadic` expression of §6.4 as the CUDA witness in Appendix D, but on host CPU targets. The contract is unchanged:

```
canonical reduce = root observation of the canonical_block_dyadic expression
canonical scan   = prefix observations of the same expression
```

Each artifact times a single-pass O(N) realization of the named expression against the `std::reduce` and `std::inclusive_scan` baselines on its host. The within-block axis is the canonical pairwise tree evaluated as straight-line code at compile-time-fixed `B`; the across-block axis is an iterated dyadic stack at L=1, drained per block to emit each prefix. Each block emission attaches the local prefix as a peer of the across-block subtotal `D_b` and commits the block total `bt_b` to the stack, in the convention `init + (D_b + P_b(j))` mid-block and `init + D_{b+1}` block-final. The same correctness checks as in Appendix D apply, plus a per-prefix oracle check against the §6.4 literal evaluator:

```
bits(scan.back()) == bits(canonical_block_dyadic_reduce)
scan[i] == canonical_block_dyadic_reduce(input[0..i+1))
```

At `N = 10,000` `double` inputs (in-cache regime), the reduce measurements are: peak `canonical_block_dyadic_reduce<32>` throughput is approximately 41.5 GB/s on x86-64 against an `std::reduce` baseline of approximately 23.3 GB/s, and approximately 41.9 GB/s on AArch64 against an `std::reduce` baseline of approximately 35.5 GB/s. The scan measurements are: peak `canonical_block_dyadic_scan<32>` throughput is approximately 11.8 GB/s on x86-64 against an `std::inclusive_scan` baseline of approximately 10.2 GB/s, and peak `canonical_block_dyadic_scan<16>` throughput is approximately 12.3 GB/s on AArch64 against an `std::inclusive_scan` baseline of approximately 12.1 GB/s. Both artifacts report exact bit-equality at every sampled prefix with the §6.4 literal-evaluator oracle and with the canonical reduction over the same prefix.

The standardization-relevant observation is not the absolute throughput, which depends on the host. It is that the block size at which the named expression performs best is not the same across hosts: `B = 32` is the best scan block size on the x86-64 host, while `B = 16` is the best scan block size on the AArch64 host. This is the same expression family — `canonical_block_dyadic` — instantiated at two different values of `B`, and therefore two distinct expression contracts in the sense of §6.4. The realization is the same single-pass O(N) construction in both cases; the differing contract parameter is precisely the semantic distinction. It is empirical evidence for the §6.6 point that the best instance of a parameterized expression family may dif-

fer across targets, and for the §13 observation that a portable named expression and an implementation-defined deterministic expression are different specification choices rather than the same choice expressed at different granularities. The closeness of `canonical_block_dyadic_reduce` to the autovectorized `std::reduce` baseline on AArch64 — and the wider gap to it on x86-64 — is a measurement, not a property of the contract; it is recorded here only to indicate that the named expression is implementable at rates close to a non-deterministic reduce on present hardware.

Appendix F: Cross-platform expression-reproducibility witness — same expression, same bits, three platforms

The following Compiler Explorer artifacts are informative and are not proposed wording:

- host witness (x86-64, GCC 16.1)
  URL: https://godbolt.org/z/WovKvEh1b
- host witness (AArch64, Clang)
  URL: https://godbolt.org/z/bjd5sfMKv
- CUDA witness
  URL: https://godbolt.org/z/9ME9dYeen

This appendix witnesses cross-platform expression-level reproducibility — the stronger of the determinism scopes distinguished in §2 — for two named expressions:

```
canonical_pairwise            (P4016R0 §4, Appendix A)
canonical_block_dyadic        (§6.4, at B = 16 and at B = 256)
```

The shared input is a fixed hostile dataset of `N = 1,048,576` `double` values generated by an inlined splitmix64 stream and an explicit bits-to-double constructor (sign, biased exponent in `[1e-30, 1e+8]`, mantissa). The generator is pure unsigned bit operations, with no dependence on `std::uniform_real_distribution` or any other library-defined distribution, so it is byte-identical across libc, compiler, host, and device.

Each platform runs an independent implementation of each expression — single-pass O(N) iterated-stack on host, single-thread `<<<1, 1>>>` device kernel on CUDA — and emits, for each (expression, block size):

- the inclusive-scan back element in hex,
- the reduction in hex,
- an FNV-1a 64-bit hash of the full 1M-element scan output,
- 24 individual prefix values at non-block-aligned positions.

Probe-position design. The probe positions are deliberately chosen to test the hard case. They are filtered so that every reported probe lies strictly inside `[1, B-2]` for both `B = 16` and `B = 256`. This avoids the common false positive in which an implementation gets block-aligned prefixes right but fails to attach the completed-block carry correctly to a partial local prefix: block-aligned positions evaluate the

§6.4 expression with an empty partial-peer term `P_b(j)` and reduce to `op(init, D_b)`, which a wide class of incorrect implementations would still get right.

The probes here therefore force the peer-attached partial-block-prefix rule to be exercised at every reported position, not merely root agreement or block-boundary agreement. The full-array FNV-1a hash separately covers the block-aligned positions plus all other outputs.

The 24 probe positions are the union of `{1/3, 2/3}`, `{k/9 : 1 ≤ k ≤ 8}`, and `{k/27 : 1 ≤ k ≤ 26, k mod 3 ≠ 0}` fractions of `N`, after applying the probe-position filter above.

Each emitted value is compared on-line against a hard-coded expected bit pattern, with a `[PASS]` / `[FAIL]` verdict per row and a `SUMMARY` line at the end. The expected values were recorded from an authoritative first run on x86-64 with `-O3 -std=c++20 -ffp-contract=off` and IEEE 754 round-to-nearest-even. Each platform's witness returns 0 on full agreement and a non-zero exit code on any mismatch. The build flags required for the witness to PASS are stated in each artifact: `-ffp-contract=off` on host, `--fmad=false` on CUDA, no `-ffast-math`, no `-mfma`.

The three artifacts report `84 PASS, 0 FAIL` each, with bit-identical outputs across x86-64 (GCC 16.1), AArch64 (Clang), and CUDA (NVCC, single-thread kernel on a server-class GPU). Identical means: the three full-array FNV-1a 64-bit scan hashes per expression agree across all three platforms, the 24 mid-block probe values per expression agree, the reduction value agrees, and the back element of the scan agrees. The three expressions produce three different hashes on each platform — same input, different bracketings, distinguishable outputs — and each expression's hash is invariant across the three platforms.

The standardization-relevant observation is that the cross-ISA scope of determinism §2 distinguishes — agreement of independent conforming implementations of the same named expression on the same input — is achievable in practice, given a floating-point environment that is constrained at the build level rather than left to the implementation. A matching FNV-1a 64-bit hash plus 24 matching mid-block probes is necessary but not sufficient evidence that two implementations evaluate the same expression tree at every node; it is evidence that the observed outputs are bit-identical at every position, which is the property the paper claims under "expression state composes, returned values are observed" (§5).

References

- [Blelloch1990] Guy E. Blelloch. *Prefix Sums and Their Applications.* Technical Report CMU-CS-90-190, School of Computer Science, Carnegie Mellon University, November 1990.
- [P4016R0] Andrew Drakeford. *Canonical Parallel Reduction: A Fixed Expression Structure for Run-To-Run Consistency.* WG21 working paper P4016R0, ISO/IEC JTC1/SC22/WG21, pre-Croydon mailing, February 2026.

- [P3375R3] Guy Davidson. *Reproducible Floating-Point Results.* WG21 working paper P3375R3, ISO/IEC JTC1/SC22/WG21, 2025.
- [Higham2002] Nicholas J. Higham. *Accuracy and Stability of Numerical Algorithms*, 2nd edition. SIAM, 2002.
- [Dalton2014] Barnaby Dalton, Amy Wang, and Bob Blainey. *SIMDizing Pairwise Sums: A Summation Algorithm Balancing Accuracy with Throughput.* In Proceedings of the 2014 Workshop on Programming Models for SIMD/Vector Processing (WPMVP '14), pp. 65–70, ACM, 2014.
- [Merrill2016] Duane Merrill and Michael Garland. *Single-pass Parallel Prefix Scan with Decoupled Look-back*. NVIDIA Technical Report NVR-2016-002, NVIDIA Corporation, March 2016.