Document Number:	P3690R0
Date:	2025-05-13
Reply-to:	Olaf Krzikalla <olaf.krzikalla@dlr.de>, Matthias Kretz <m.kretz@gsi.de>
Audience:	LEWG
Target:	C++26

Consistency fix: Make simd reductions SIMD-generic

ABSTRACT

One design goal of the `simd` interface is to enable SIMD-generic code. This is why all arithmetic operators and functions have corresponding overloads. However, arithmetic reductions are still missing overloads for non-`simd`, vectorizable types.

CONTENTS

1 CHANGELOG

(placeholder)

2 STRAW POLLS

(placeholder)

3 INTRODUCTION

[P1928R15] introduced `std::simd` and related types and functions. It enables the programmer to write *SIMD-generic* code, i.e. function templates instantiable with scalar types as well as vector types.

```
template<class T>
auto f(const T& x, const T& y, const T& z)
{
  return x + std::sqrt(y) * std::pow(z);
}
```

This function template can be instantiated with scalar floating types, with complex types, or with their vectorized counterparts (e.g. `std::simd<double>` or `std::simd<std::complex<double>>`).

SIMD-generic code is also possible for boolean reductions.

```
template<class T>
bool all_lt(const T& x, const T& y)
{
  return std::datapar::all_of(x < y);
}
```

Such code is possible because [P1928R15] explicitly includes overloads for `all_of(bool)` aso.

On the other hand, we believe, that [P1928R15] just forgot to provide scalar overloads for arithmetic reductions.

```
template<class T>
auto calc_contribution(const T& x, const T& y)
{
  return std::datapar::reduce(x * y);
}
```

With just [P1928R15] this code is not SIMD-generic yet. It cannot be called with scalar types, as there is no scalar overload for `std::datapar::reduce` yet. That makes this part of the simd interface incoherent with the rest that does work.

4 Proposal

We propose an introduction of scalar overloads for all arithmetic reduce functions introduced in [P1928R15]. This applies to the functions in 29.10.7.5: `reduce`, `reduce_min`, and `reduce_max`. The semantic of the functions is mostly trivial: they just return the passed argument. The masked functions take a scalar `bool` as mask argument. If the value of that argument is `false`, then the functions behave like their vectorized counterparts if `none_of(mask) == true` applies to them.

5 Discussion

When the function was still called `std::reduce` there was some doubt whether such an overload of the name could be too much. But now that reduce moved into the subnamespace `std::datapar` there is no apparent reason to avoid a scalar overload of reduce.

6 Wording

The proposed changes are relative to the current working draft [N5008].

6.1 Feature test macro

In [version.syn] bump the `__cpp_lib_simd` version.

6.2 Changes to [simd]

Add the following to ([simd.syn]):

[simd.syn]

```
// ([simd.reductions]), basic_simd reductions
template<class T, class Abi, class BinaryOperation = plus<>>
  constexpr T reduce(const basic_simd<T, Abi>&, BinaryOperation = {});
template<class T, class Abi, class BinaryOperation = plus<>>
  constexpr T reduce(
    const basic_simd<T, Abi>& x, const typename basic_simd<T, Abi>::mask_type& mask,
    BinaryOperation binary_op = {}, type_identity_t<T> identity_element = see below);

template<class T, class BinaryOperation = plus<>>
  constexpr T reduce(const T&, BinaryOperation = {});
template<class T, class BinaryOperation = plus<>>
  constexpr T reduce(const T& x, same_as<bool> mask, BinaryOperation binary_op = {},
                     type_identity_t<T> identity_element = see below);

template<class T, class Abi>
  constexpr T reduce_min(const basic_simd<T, Abi>&) noexcept;
```

```
template<class T, class Abi>
  constexpr T reduce_min(const basic_simd<T, Abi>&,
                         const typename basic_simd<T, Abi>::mask_type&) noexcept;
template<class T, class Abi>
  constexpr T reduce_max(const basic_simd<T, Abi>&) noexcept;
template<class T, class Abi>
  constexpr T reduce_max(const basic_simd<T, Abi>&,
                         const typename basic_simd<T, Abi>::mask_type&) noexcept;

template<class T> constexpr T reduce_min(const T&) noexcept;
template<class T> constexpr T reduce_min(const T&, same_as<bool>) noexcept;
template<class T> constexpr T reduce_max(const T&) noexcept;
template<class T> constexpr T reduce_max(const T&, same_as<bool>) noexcept;

// ([simd.alg]), Algorithms
template<class T, class Abi>
  constexpr basic_simd<T, Abi>
    min(const basic_simd<T, Abi>& a, const basic_simd<T, Abi>& b) noexcept;
```

Add the following to ([simd.reductions]):

(6.2.0.1) **29.10.7.6 `basic_simd` reductions [simd.reductions]**

```
template<class T, class Abi, class BinaryOperation = plus<>>
  constexpr T reduce(const basic_simd<T, Abi>& x, BinaryOperation binary_op = {});
```

1 *Constraints*: `BinaryOperation` models *`reduction-binary-operation`*`<T>`.

2 *Preconditions*: `binary_op` does not modify `x`.

3 *Returns:* *`GENERALIZED_SUM`*`(binary_op, simd<T, 1>(x[0]), ..., simd<T, 1>(x[x.size() - 1]))[0]` ([numerics.defns]).

4 *Throws:* Any exception thrown from `binary_op`.

```
template<class T, class Abi, class BinaryOperation = plus<>>
  constexpr T reduce(
    const basic_simd<T, Abi>& x, const typename basic_simd<T, Abi>::mask_type& mask,
    BinaryOperation binary_op = {}, type_identity_t<T> identity_element = see below);
```

5 *Constraints*:

- `BinaryOperation` models *`reduction-binary-operation`*`<T>`.
- An argument for `identity_element` is provided for the invocation, unless `BinaryOperation` is one of `plus<>`, `multiplies<>`, `bit_and<>`, `bit_or<>`, or `bit_xor<>`.

6 *Preconditions*:

- `binary_op` does not modify `x`.
- For all finite values `y` representable by `T`, the results of `y == binary_op(simd<T, 1>(identity_element), simd<T, 1>(y))[0]` and `y == binary_op(simd<T, 1>(y), simd<T, 1>(identity_element))[0]` are `true`.

7 *Returns:* If `none_of(mask)` is `true`, returns `identity_element`. Otherwise, returns *`GENERALIZED_SUM`*`(binary_op, simd<T, 1>(x[`k_0`]), ..., simd<T, 1>(x[`k_n`]))[0]` where $k_0, \ldots, k_n$ are the selected indices of `mask`.

8 *Throws:* Any exception thrown from `binary_op`.

9 *Remarks:* The default argument for `identity_element` is equal to

- `T()` if `BinaryOperation` is `plus<>`,
- `T(1)` if `BinaryOperation` is `multiplies<>`,
- `T(~T())` if `BinaryOperation` is `bit_and<>`,
- `T()` if `BinaryOperation` is `bit_or<>`, or
- `T()` if `BinaryOperation` is `bit_xor<>`.

```
template<class T, class BinaryOperation = plus<>>
  constexpr T reduce(const T& x, BinaryOperation binary_op = {});
template<class T, class BinaryOperation = plus<>>
  constexpr T reduce(const T& x, same_as<bool> mask, BinaryOperation binary_op = {},
                     type_identity_t<T> identity_element = see below);
```

10 Let `mask` be `true` for the overload with no `mask` parameter.

11 *Constraints*:

- `T` is vectorizable.
- `BinaryOperation` models *`reduction-binary-operation`*`<T>`.
- An argument for `identity_element` is provided for the invocation, unless `BinaryOperation` is one of `plus<>`, `multiplies<>`, `bit_and<>`, `bit_or<>`, or `bit_xor<>`.

12 *Returns:* If `mask` is `false`, returns `identity_element`. Otherwise, returns `x`.

13 *Throws:* Nothing.

14 *Remarks:* The default argument for `identity_element` is equal to

- `T()` if `BinaryOperation` is `plus<>`,
- `T(1)` if `BinaryOperation` is `multiplies<>`,
- `T(~T())` if `BinaryOperation` is `bit_and<>`,
- `T()` if `BinaryOperation` is `bit_or<>`, or
- `T()` if `BinaryOperation` is `bit_xor<>`.

```
template<class T, class Abi> constexpr T reduce_min(const basic_simd<T, Abi>& x) noexcept;
```

15 *Constraints*: `T` models `totally_ordered`.

16 *Returns:* The value of an element `x[`j`]` for which `x[`i`] < x[`j`]` is `false` for all i in the range of $[0,$ `basic_simd<T, Abi>::size()`$)$.

```
template<class T, class Abi>
  constexpr T reduce_min(
    const basic_simd<T, Abi>&, const typename basic_simd<T, Abi>::mask_type&) noexcept;
```

17 *Constraints*: `T` models `totally_ordered`.

18 *Returns:* If `none_of(mask)` is `true`, returns `numeric_limits<T>::max()`. Otherwise, returns the value of a selected element `x[`j`]` for which `x[`i`] < x[`j`]` is `false` for all selected indices i of `mask`.

```
template<class T, class Abi> constexpr T reduce_max(const basic_simd<T, Abi>& x) noexcept;
```

19 *Constraints*: `T` models `totally_ordered`.

20 *Returns:* The value of an element `x[`j`]` for which `x[`j`] < x[`i`]` is `false` for all i in the range of $[0,$ `basic_simd<T, Abi>::size()`$)$.

```
template<class T, class Abi>
  constexpr T reduce_max(
    const basic_simd<T, Abi>&, const typename basic_simd<T, Abi>::mask_type&) noexcept;
```

21 *Constraints*: `T` models `totally_ordered`.

22 *Returns:* If `none_of(mask)` is `true`, returns `numeric_limits<V::value_type>::lowest()`. Otherwise, returns the value of a selected element `x[`j`]` for which `x[`j`] < x[`i`]` is `false` for all selected indices i of `mask`.

```
template<class T> constexpr T reduce_min(const T& x) noexcept;
template<class T> constexpr T reduce_min(const T& x, same_as<bool> mask) noexcept;
template<class T> constexpr T reduce_max(const T& x) noexcept;
template<class T> constexpr T reduce_max(const T& x, same_as<bool> mask) noexcept;
```

23 Let `mask` be `true` for the overloads with no `mask` parameter.

24 *Constraints*:

- `T` is vectorizable.
- `T` models `totally_ordered`.

25 *Returns:* If `mask` is `false`, returns `numeric_limits<T>::max()` for `reduce_min` and `numeric_limits<T>::lowest()` for `reduce_max`. Otherwise, returns `x`.

A BIBLIOGRAPHY

[N5008] Thomas Köppe, ed. *Working Draft, Programming Languages — C++*. ISO/IEC JTC1/SC22/WG21, 2025. URL: https://wg21.link/n5008.

[P1928R15] Matthias Kretz. *std::simd — merge data-parallel types from the Parallelism TS 2*. ISO/IEC C++ Standards Committee Paper. 2024. URL: https://wg21.link/p1928r15.