Document Number: P3844R1
Date: 2025-11-07
Reply-to: Matthias Kretz <m.kretz@gsi.de>
Audience: LEWG
Target: C++26

Restore simd::vec broadcast from int

Abstract

The broadcast constructor in the Parallelism 2 TS allowed construction from (`unsigned`) `int`, allowing e.g. `vec<float>() + 1`, which is ill-formed in the CD. This breaks existing code that gets ported from the TS to `std::simd`. The design intent behind `std::simd` was for this to work. However, the understanding in LEWG appeared to be that we can't get this right without constexpr function arguments getting added to the language. This paper shows that a `consteval` constructor overload together with `constexpr` exceptions can resolve the issue for C++26 and is a better solution than constexpr function arguments would be.

Contents

1 CHANGELOG

1.1 CHANGES FROM REVISION 0

Previous revision: P3844R0

- Constrain the `consteval` ctor to arithmetic types that naturally convert to the `basic_vec`'s value-type (checked via `common_type`).
- Discuss consequence on `common_type` (besides `convertible_to`).
- Discuss consequence on [simd.math].
- Discuss `consteval` ctor as immediate-escalating expression.
- Discuss potential consequences for [simd.math].

2 STRAW POLLS

(placeholder)

3 MOTIVATION

It is very common in floating-point code to simply write e.g. `* 2` rather than `* 2.f` when multiplying a `float` with a constant:

```
float f(float x) { return x * 2; } // converts 2 to float (at compile time)

float g(float x) { return x * 2.; } // converts x to double (at run time)

float h(float x) { return x * 2.f; } // no conversions
```

More importantly, using `* 2` works reliably in generic code, where the type of `x` could be any arithmetic type.

Since this is so common, `std::experimental::simd<T>` made an exception for `int` in the broadcast constructor to not require value-preserving conversions. Consequently, the TS behavior is:

```
using floatv = std::experimental::native_simd<float>;

floatv f(floatv x) { return x * 2; } // converts 2 to float and broadcasts (at
                                     //                                 compile time)
floatv g(floatv x) { return x * 2.; } // ill-formed

floatv h(floatv x) { return x * 2.f; } // broadcasts 2.f to floatv
```

When porting existing code written against the TS to C++26, the first step is to adjust the types:

```
using floatv = std::~~experimental::native_simd~~simd::vec<float>;
```

Except for uses of `std::experimental::where`, which need to be refactored to use `simd::select`, the remaining code should work. The one place where it doesn't work is code such as in function `f`, where 2 needs to be replaced:

```
floatv f(floatv x) { return x * ~~2~~std::cw<2>; }
```

Since we don't have `constexpr` function arguments in the language, `std::simd` works around it by recognizing *`integral-constant-like`* / `constant-wrapper-like` types, that encode a *value* into a type. This, however, comes at a compile-time cost. Every different value leads to a template specialization of both `constant_wrapper` and a `basic_vec` broadcast constructor (with it's helper types/concepts to determine whether the specialization is allowed). Consequently, for `vec<float>`, I would recommend to always use an `f` suffix rather than `std::cw`.

But that solution is fairly limited, since we don't have literals for 8-bit and 16-bit integers in the language. A function template like

```
template <simd_integral V>
V f(V x) {
  return x + 1; // ill-formed for V::value_type = (u)int8_t, (un)int16_t, and uint32_t
}
```

needs to use `x + V(1)`[1]. A clever user might write `x + '\1'` instead. But that fails for the `char` type with different signedness.

Consequently, users would need to get used to writing explicit conversions for the constants they use in `std::simd` code. That's not only verbose and ugly, it is also error-prone. Whenever we coerce our users into writing explicit conversions, then value-changing conversions cannot be diagnosed as erroneous anymore. An explicit `static_cast<uint64_t>(-1)` means `0xffff'ffff'ffff'ffff`,

1 explicit conversion to `basic_vec` allows conversions that are not value-preserving

whereas `uint64_t x = -1` could have been intended to mean `0x0000'0000'ffff'ffff` or is a result of a logic flaw in the code. E.g., GCC's `-Wsign-conversion` diagnoses the latter, but not the former[2].

If, with C++26, our users are starting to explicitly convert their `int` constants to `basic_vec`, then the interface of `basic_vec` is at least in part guilty for introducing harder to find bugs.

Tony Table 1 presents an example of the solution[3]. Note that the code on the left will never warn about the value-changing conversion, even with all conversion related warnings enabled. This is due to the explicit conversion, which is telling the compiler "I know what I'm doing; no need to warn me about it".

before	with P3844R1
`template <simd_floating_point V>`	`template <simd_floating_point V>`
`V f(V x) {`	`V f(V x) {`
`  return x + V(0x5EAF00D);`	`  return x + 0x5EAF00D;`
`}`	`}`
`f(vec<double>()); // OK`	`f(vec<double>()); // OK`
`// compiles but adds 99282960 instead of 99282957`	`// ill-formed: value-changing conversion`
`f(vec<float>());`	`f(vec<float>());`
`// compiles but adds infinity instead of 99282957`	`// ill-formed: value-changing conversion`
`f(vec<std::float16_t>());`	`f(vec<std::float16_t>());`

~~Tony~~Before/After Table 1: Add an offset

A safer implementation of the code on the left side of Tony Table 1 (without this paper) would have been to write `x + std::cw<0x5EAF00D>` instead. Then the value-changing conversion would have resulted in a constraint failure on the broadcast constructor. However, `V(0x5EAF00D)` is shorter and needs fewer template instantations. I expect most users (including myself) will/do not use `std::cw` all over the place.

4 DESIGN SPACE

In the design review of P1928 of this issue of the broadcast constructor it was overlooked (and never discussed) that a `consteval` overload of the broadcast constructor could solve this problem. Before `constexpr` exceptions, we would have worded it to be ill-formed (by unspecified means) if the value changes on conversion to the `basic_vec`'s value-type. Now that we have `constexpr` exceptions, we can specify a `consteval` broadcast overload that throws on value-changing conversion. If the caller cares, the exception can even be handled at compile time. (I believe it should not throw in C++26, for a minimal change to the WD this late in the C++26 cycle.)

Ordering the overloads for overload resolution is tricky, which is another reason why we should consider this issue before C++26 ships and potentially take action even if we don't add a `consteval` overload. Overload resolution does not take `consteval` into account. The process of finding candidate functions ([over.match.funcs.general]), however, does remove explicit constructors from the candidate set if the context does not allow the explicit constructor to be called.

2 And that's useful, because the former says "I'm intentionally doing this conversion, no need to warn."
3 I got bitten by this in my `std::simd` unit tests

4.1 potentially-convertible-to

R0 proposed to allow any conversion from arithmetic type `U`, that satisfies `convertible_to<value_type>` and does not satisfy *`value-preserving-convertible-to`*`<value_type>` via the `consteval` broadcast constructor. This was too broad, since it would lead to `vec<float>() + 1.5` being valid (of type `vec<float>`). While technically not wrong (no loss on conversion from `1.5`), it is too surprising that an operation involving a `double` operand is evaluated in single precision.

Therefore, R1 of this paper tightens the constraints for the `consteval` broadcast to producing a less surprising common type. If the given constant is of arithmetic type `T`, then we now require `common_type_t<T, value_type>` to be `value_type`. Since `common_type_t<double, float>` is `double`, the expression `vec<float>() + 1.5` becomes ill-formed. However, this rule alone still breaks the case of `vec<short>() + 1`, which the user cannot changed to use a `short` literal (because we don't have one). The TS made an explicit exception for `int` and `unsigned int`[4], which is what we still need for integer types (with lower rank than `int`).

So the final *`potentially-convertible-to`* constraint looks like this:

```
template <typename From, typename To>
  concept potentially-convertible-to = is_arithmetic_v<From>
    && convertible_to<From, To> && !value-preserving-convertible-to<From, To>
    && (is_same_v<common_type_t<From, To>, To>
      || (is_same_v<From, int> && is_integral_v<To>)
      || (is_same_v<From, unsigned> && unsigned_integral<To>));
```

4.2 status quo

The following code shows the properties of the current broadcast constructor. See Appendix A for the definition of the `really_convertible_to` concept.

```
using V = simd::vec<float>;

template <typename T> struct X { explicit operator T() const; };

template <typename... Ts>
concept has_common_type = requires { typename std::common_type_t<Ts...>; };

static_assert(not   std::convertible_to<X<float>, V>);
static_assert(      std::convertible_to<float, V>);
static_assert(      std::convertible_to<short, V>);
static_assert(    really_convertible_to<short, V>);
static_assert(not   std::convertible_to<int, V>);
static_assert(not really_convertible_to<int, V>);

static_assert(    std::constructible_from<V, X<float>>);
static_assert(not std::constructible_from<V, X<short>>);
static_assert(    std::constructible_from<V, double>);
static_assert(    std::constructible_from<V, float>);
static_assert(    std::constructible_from<V, short>);
static_assert(    std::constructible_from<V, int>);

static_assert(not has_common_type<V, double>);
```

4 The TS broadcast constructor has a constraint "[...], or `From` is `int`, or `From` is `unsigned int` and `value_type` is an unsigned integral type."

```
static_assert(not has_common_type<V, int>);

V f(int n, short m, std::reference_wrapper<int> l, std::reference_wrapper<float> f)
{
  V x = '\1'; // OK
  x = 1;      // ill-formed
  x = 0x5EAF00D; // ill-formed
  x = V(n);   // OK
  x = m;      // OK
  x = l;      // OK (because convertible_to<decltype(l), float> is true)
  x = f;      // OK
  x = 1.1;    // ill-formed
  x = V(1.1); // OK
  x = X<float>(); // ill-formed: no match for operator= (no known conversion …[])
  x = float(X<float>()); // OK (obvious)
  x = V(X<float>()); // OK
}
```

4.3 more constrained constexpr overload

A possible solution selects the existing (`constexpr`) broadcast constructor for everything but the cases where the value of the argument needs to be checked. Thus, we need the existing constructor to always be *more constrained* ([temp.constr.order]) than the `consteval` constructor. The `consteval` constructor can then only be selected if the other constructor is not part of the candidate set at all (via `explicit`).

Sketch:

```
template <class From, class To>
  concept simd-consteval-broadcast-arg = explicitly-convertible-to<From, To>;

template <class From, class To>
  concept simd-broadcast-arg = simd-consteval-broadcast-arg<From, To> and true;

template <class T>
class basic_vec
{
public:
  template <simd-broadcast-arg<T> U>
    constexpr explicit(see below) basic_vec(U&&); // #1
  template <simd-consteval-broadcast-arg<T> U>
    consteval basic_vec(U&&);                     // #2
    // Mandates: potentially-convertible-to<remove_cvref_t<U>, value_type>
};
```

Now every explicit call to the broadcast constructor will always select `#1`. Implicit calls to the broadcast constructor will select `#1` if the condition in the `explicit` specifier is `false`. Otherwise, `#1` is not part of the candidate set and `#2` is called. Thus, the condition on the `explicit` specifier determines whether the `consteval` overload is chosen or not.

```
static_assert(      std::convertible_to<X<float>, V>); // different to status quo
static_assert(      std::convertible_to<float, V>);
static_assert(      std::convertible_to<short, V>);
static_assert(    really_convertible_to<short, V>);
static_assert(      std::convertible_to<int, V>);      // different to status quo
static_assert(not really_convertible_to<int, V>);

static_assert(    std::constructible_from<V, X<float>>);
static_assert(not std::constructible_from<V, X<short>>);
static_assert(    std::constructible_from<V, double>);
static_assert(    std::constructible_from<V, float>);
static_assert(    std::constructible_from<V, short>);
static_assert(    std::constructible_from<V, int>);

static_assert(not has_common_type<V, double>);
static_assert(    has_common_type<V, int>); // it's vec<float>

V f(int n, short m, std::reference_wrapper<int> l, std::reference_wrapper<float> f)
{
  V x = '\1'; // OK
  x = 1;       // OK (different to status quo)
  x = 0x5EAF00D; // ill-formed
  x = V(n);    // OK
  x = m;       // OK
  x = V(l);    // OK
  x = f;       // OK
  x = 1.1;     // ill-formed
  x = V(1.1); // OK
  x = X<float>(); // ill-formed: static_assert failed (different reason to status quo)
  x = float(X<float>()); // OK (obvious)
  x = V(X<float>()); // OK
}
```

4.4 More constrained consteval overload

A viable alternative involves the removal of explicit conversions from arithmetic types to `basic_vec`. The `consteval` constructor is declared with additional constraints over the existing constructor (satisfies `convertible_to`, `is_arithmetic_v`, and not value-preserving conversion). This way the `consteval` constructor is always chosen if the conversion of the given (arithmetic) type to `value_type` is *`potentially-convertible-to`*. Otherwise, the `constexpr` overload is used.

Sketch:

```
template <class From, class To>
  concept simd-broadcast-arg = explicitly-convertible-to<From, To>;

template <class From, class To>
  concept simd-consteval-broadcast-arg =
    simd-broadcast-arg<From, To> && potentially-convertible-to<remove_cvref_t<From>, To>;

template <class T>
class basic_vec
{
public:
  template <simd-broadcast-arg<T> U>
    constexpr explicit(see below) basic_vec(U&&); // #1
  template <simd-consteval-broadcast-arg<T> U>
    consteval basic_vec(U&&); // #2
};
```

Here, every explicit call to the broadcast constructor with a type `U` that satisfies *`potentially-convertible-to`*<T> is equivalent to an implicit conversion, since the `consteval` overload is viable and more constrained. Every type with a value-preserving conversion to `T` will select `#1` (because of the constraint on `#2`). Every non-arithmetic type (notably, user-defined types with conversion operator to some arithmetic type) will continue to work as today, since `#2` is not viable.

```
static_assert(not   std::convertible_to<X<float>, V>); // equal to status quo / different to above
static_assert(      std::convertible_to<float, V>);
static_assert(      std::convertible_to<short, V>);
static_assert(    really_convertible_to<short, V>);
static_assert(      std::convertible_to<int, V>);      // different to status quo / equal to above
static_assert(not really_convertible_to<int, V>);

static_assert(    std::constructible_from<V, X<float>>);
static_assert(not std::constructible_from<V, X<short>>);
static_assert(    std::constructible_from<V, double>);
static_assert(    std::constructible_from<V, float>);
static_assert(    std::constructible_from<V, short>);
static_assert(    std::constructible_from<V, int>);

static_assert(not has_common_type<V, double>);
static_assert(    has_common_type<V, int>); // it's vec<float>

V f(int n, short m, std::reference_wrapper<int> l, std::reference_wrapper<float> f)
{
  V x = '\1'; // OK
  x = 1;      // OK (different to status quo / equal to above)
  x = 0x5EAF00D; // ill-formed
  x = V(n);   // ill-formed (different to both)
  x = m;      // OK
  x = V(l);   // OK
  x = f;      // OK
  x = 1.1;    // ill-formed
  x = V(1.1); // OK
  x = X<float>(); // ill-formed: no match for operator= (no known conversion …[])
  x = float(X<float>()); // OK (obvious)
  x = V(X<float>()); // OK
}
```

4.5 How to handle bad value-preserving casts

The `consteval` broadcast overload needs to be ill-formed if the argument value cannot be converted to the value type without changing the value. This can be achieved via the mechanism used in ([simd.bit]) for `bit_ceil`. The constructor would spell out a precondition followed by *Remarks*: An expression that violates the precondition in the *Preconditions*: element is not a core constant expression ([expr.const]).

The alternative that was mentioned before is to throw an exception (at compile time). Since in basically all cases such an exception would not be caught at compile time, the program becomes ill-formed. The ability to catch the exception allowed me to hack up a `really_convertible_to` concept. But otherwise, the utility of using an exception here seems fairly limited. The main reason for using an exception is better diagnostics on ill-formed programs. If we decide to add the `consteval` constructor for C++26, then we might want to delay the new exception type for C++29, though.

5 DIFFERENCES

Differences between the status quo and the two alternatives above:

	status quo	Section 4.3	Section 4.4
`convertible_to<X<float>, V>`	`false`	`true`	`false`
`convertible_to<int, V>`	`false`	`true`	`true`
`common_type_t<V, int>`	✘	`V`	`V`
`x = 1;`	✘	✔	✔
`x = V(n);`	✔	✔	✘

Note that `X<float>` is never implicitly convertible to `vec<float>`, so the solution in Section 4.3 lies about that. Also while some values of constant expressions of type `int` are convertible to `vec<float>`, it is not true in general that `int` is convertible to `vec<float>`.

6 SHOULD COMMON_TYPE REALLY CHANGE? WHAT IF IT DOES?

After `convertible_to` changes, consequently also conditional expressions such as `false ? vec<float>() : 1` become valid. `common_type_t<vec<float>, int>` simply reflects that. The surprising aspect, similar to `convertible_to`, is that this isn't true in general.

If we accept that `common_type` changes (we don't have to, as we can specialize `common_type`), this has consquences on [simd.math]. Multi-argument math functions use `common_type` in it's specification to spell out the design intent to match `<cmath>` overloads of these functions. For some background, let's use 2-arg `std::hypot` as an example. C defines three functions `hypot`, `hypotf`, and `hypotl`. C++ then overloads `hypot(double, double)` with `hypot(float, float)`, and `hypot(long double, long double)` (and since C++23 also `std::floatN_t`). It is not correct to implement this as `template <class T> hypot(T, T)`, since a call to `hypot(1., 1)` would then be ill-formed. With explicit overloads `hypot(1., 1)` calls `hypot(double, double)`.

The std::simd (and TS) math overloads were designed to match that behavior. The mechanism for this was spelled out after the design went into LWG wording review for C++26. For std::simd it's not as simple as spelling out all overloads. Consider an implementation that supports up to 256 elements in a `basic_vec`. The float overloads would need a minimum of 256 overloads (`hypot(vec<float, N>, vec<float, N>)`). But actually more are needed because of differences in ABI tags (vector mask vs. bit mask; different register widths for different targets). That explosion in function overloads just isn't reasonable to spell out (neither in the standard, nor in an implementation).

Therefore, std::simd uses function templates. The 2-arg `hypot` function is declared as:

```
template <class V0, class V1>
  math-common-simd-t<V0, V1> hypot(const V0& x, const V1& y);
```

The return type is constrained such that at least one of `V0` and `V1` is a `basic_vec` of floating-point type[5]. In the common cases the *`math-common-simd-t`* alias is simply `common_type_t<V0, V1>`. If `common_type` has no type member, then the overload is removed from the overload set (SFINAE).

5 or is like a `reference_wrapper`

The status-quo is that for `simd::hypot(vec<float>(), 1)` *`math-common-simd-t`*`<vec<float>, int>` is not valid and thus no viable `simd::hypot` overload exists.

The above proposal makes *`math-common-simd-t`*`<vec<float>, int>` a valid type. In principle, that's correct, because `1` can be represented without loss of value as a `float`.

The question is what should happen for `simd::hypot(vec<float>(), 0x5EAF00D)`, which would need to convert `0x5EAF00D` to `float` inside the `hypot` implementation and thereby change its value. Currently, the *`make-compatible-simd-t`* trait converts `0x5EAF00D` from `int` to `vec<int, vec<float>::size()>`. If the *`make-compatible-simd-t<V, T>`* trait is changed to instead be an alias for `V`[6] rather than `vec<T, V::size()>`, then the as-if implementation requires a conversion to `vec<float>`, and thus implying the broadcast constructor behavior.

7 BROADCAST AS IMMEDIATE-ESCALATING EXPRESSION

Refresher (`https://compiler-explorer.com/z/qvdoxavMK`):

```
struct A
{ consteval A(int) {} };

constexpr A f(int, auto y) // f gets promoted to an immediate function
{ return y; } // immediate-escalating expression 'A(y)'

A test(int x)
{ return f(x, 1); } // Error: 'x' is not a constant expression
```

If we remove `constexpr` from `f`, the underlying issue becomes apparent: `f` calls a `consteval` constructor that uses `y` as it's argument. And that can't work, because `y` is not a constant expression. The magic of promotion to immediate function simply makes the compiler try harder to make it compile anyway.

7.1 ESCALATING [SIMD.MATH] FUNCTIONS

Any `simd::vec` broadcast expression that promotes the surrounding function to an immediate function becomes "interesting" if not surprising or problematic. The [simd.math] functions as discussed above are affected. While `simd::hypot(vec<float>(), 1)` is fine (because the first argument is a constant expression),

```
auto f(simd::vec<float> x) { return simd::hypot(x, 1); }
```

is not, even though `1` can convert to `float` without loss of value. That's because the `hypot` implementation needs to call a `consteval` function, which then promotes `hypot` itself to an immediate function, which in turn requires all arguments to `hypot` to be constant expressions. If immediate-escalation were not applied, then the conversion from `int` to `vec<float>` inside of `hypot` would be ill-formed. Either way, `simd::hypot(x, 1)` with not-constant `x` cannot be valid.

7.2 MOVE CONVERSIONS BEFORE MATH CALLS

We can respecify [simd.math] in such a way that conversions happen before the function is called, mirroring the actual behavior of `<cmath>` overloads. For 2-arg math functions we would have to change from one function template to three function templates:

6 Also the classification and comparison functions need to be split off in the wording to provide the correct template argument to *`make-compatible-simd-t`*.

```
template<class V>
  constexpr deduced-vec-t<V>
  hypot(const V&, const V&);

template<class V>
  constexpr deduced-vec-t<V>
  hypot(const V&, const deduced-vec-t<V>&);

template<class V>
  constexpr deduced-vec-t<V>
  hypot(const deduced-vec-t<V>&, const V&);
```

However, for 3-arg math functions we would have to change to seven function templates. I can report that this works for all my test cases. I believe I tested a representative set of argument types and permutations.[7]

```
template<class V>
  constexpr deduced-vec-t<V>
  hypot(const V&, const V&, const V&);

template<class V>
  constexpr deduced-vec-t<V>
  hypot(const V&, const deduced-vec-t<V>&, const deduced-vec-t<V>&);

template<class V>
  constexpr deduced-vec-t<V>
  hypot(const deduced-vec-t<V>&, const V&, const deduced-vec-t<V>&);

template<class V>
  constexpr deduced-vec-t<V>
  hypot(const deduced-vec-t<V>&, const deduced-vec-t<V>&, const V&);

template<class V>
  constexpr deduced-vec-t<V>
  hypot(const V&, const V&, const deduced-vec-t<V>&);

template<class V>
  constexpr deduced-vec-t<V>
  hypot(const V&, const deduced-vec-t<V>&, const V&);

template<class V>
  constexpr deduced-vec-t<V>
  hypot(const deduced-vec-t<V>&, const V&, const V&);
```

7.3 ALTERNATIVE VIA "EXPRESSION ALIAS" P2826

P2826, which is awaiting a revision for consideration for C++29, could solve this more elegantly. However, we don't have the feature available yet. Thus we would need to ship C++26 with a [simd.math] specification that is forward-compatible

7 I tested arguments of type `reference_wrapper<vec<float>>`, `reference_wrapper<float>`, `reference_wrapper<short>`, `vec<float>`, `float`, `short`, and immediate arguments with value `1` (consteval broadcast from `int`). I tested all permutations where at least one argument is either `vec<float>` or `reference_wrapper<vec<float>>`.

7.4 THE IMPORTANTANCE OF CONVERSIONS

Consider the `pow` function. It is fairly common to call `pow` with an integral exponent, e.g.

```
std::pow(x, 3); // x³
```

This is well-formed if `x` is of floating-point type. It would be unfortunate if the same expression would not work for `x` of type `vec<`*`floating-point-type`*`>`.

8 IMPLEMENTATION EXPERIENCE

Both solutions (and a lot more variants that were discarded) have been implemented and tested in my implementation. Several days (if not weeks) of exploration and testing went into this paper. I implemented the `consteval` overloads for a complete set of vectorizable types with an ability to select between the different behaviors discussed in this paper.[8] A representative set of [simd.math] is implemented.

9 RECOMMENDATION

My recommendation is:

- Adopt with the solution presented in Section 4.4,
- without a new exception type (Section 4.5), and
- change [simd.math] to the overload sets presented in Section 7.2 for C++26.

This would roll back a small part of a recent change done by [P3430R3].

Hope for P2826 "Replacement function" — the paper is getting renamed to "expression alias" — to get into C++29 and provide a simple way to restore `<cmath>`-like overload resolution and conversions on [simd.math] functions.

Rationale for my preference:

1. The explicit conversion from arithmetic types via broadcast constructor is significantly less important after implicit conversion from constant expressions becomes possible.
2. The new `consteval` overload cannot be fully constrained in the solution presented in Section 4.3, leading to incorrect answers on traits or in requires expressions.
3. This should be part of C++26 because it helps avoiding bugs in user code.
4. A new exception type is not important enough to add it to C++26 and it can easily be added later.
5. [simd.math] was already complicated; making it more complicated is not warranted. Either we get a language feature to make it work or users have to be explicit about conversions.

If LEWG is uncomfortable with adding the `consteval` overload now I recommend to:

- remove explicit broadcasts and
- simplify [simd.math] to `template<class T> T fun(T, T)` for C++26.

8 The code is currently in patch review for libstdc++.

10 PROPOSED POLLS

Poll: Something needs to be done for C++26. (If we do nothing, the design space is constrained and Section 4.4 would be a breaking change.)

SF	F	N	A	SA

Poll: Add a `consteval` broadcast overload for value-preserving conversions to C++26.

SF	F	N	A	SA

If the vote for C++26 failed:
Poll: Add a `consteval` broadcast overload for value-preserving conversions to C++29.

SF	F	N	A	SA

Otherwise *maybe* poll:
Poll: It is better to add a `consteval` broadcast overload for C++29 rather than C++26.

SF	F	N	A	SA

Note: The next poll makes sense for C++26, even if we only intend to add the new overload for C++29.
Poll: The new constructor overload should be fully constrained, which requires the removal of `explicit` (not value-preserving conversions) from the existing broadcast constructor.

SF	F	N	A	SA

Poll: Add a new exception type to C++26 that is thrown for value-changing conversions from the new `consteval` broadcast constructor.

SF	F	N	A	SA

Poll: Encourage a paper targeting C++29 on a new exception type that is thrown from the new `consteval` broadcast constructor.

SF	F	N	A	SA

Poll: Modify [simd.math] functions to allow conversion in the caller on function arguments (as presented in Section 7.2) for C++26.

SF	F	N	A	SA

Poll: Simplify [simd.math] functions to use `template<`*`simd-floating-point`*` T> T fun(T, T)` signatures (for now/C++26), hoping we can restore `<cmath>`-like behavior for C++29.

SF	F	N	A	SA

11 WORDING FOR SECTION 4.4

11.1 FEATURE TEST MACRO

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

11.2 MODIFY [SIMD.EXPOS]

In [simd.expos], insert:

[simd.expos]

```
template<class V, class T> using make-compatible-simd-t = see below; // exposition only

template<class From, class To>
  concept simd-broadcast-arg = explicitly-convertible-to<From, To>;  // exposition only

template<class From, class To>
  concept simd-consteval-broadcast-arg = see below;                  // exposition only

template<class V>
  concept simd-vec-type =                                            // exposition only
```

11.3 MODIFY [SIMD.EXPOS.DEFN]

In [simd.expos.defn], insert:

[simd.expos.defn]

```
template<class V, class T> using make-compatible-simd-t = see below;
```

1 Let `x` denote an lvalue of type `const T`.

2 *`make-compatible-simd-t`*`<V, T>` is an alias for

- *`deduced-vec-t`*`<T>`, if that type is not `void`, otherwise
- `vec<decltype(x + x), V::size()>`.

```
template<class From, class To> concept simd-consteval-broadcast-arg = see below;
```

3 *`simd-consteval-broadcast-arg`* subsumes *`simd-broadcast-arg`*.

4 `From` satisfies *`simd-consteval-broadcast-arg`*`<To>` only if

- `remove_cvref_t<From>` is an arithmetic type,
- `From` satisfies `convertible_to<To>`,
- the conversion from `remove_cvref_t<From>` to `To` is not value-preserving, and
- either
  - `common_type_t<From, To>` is `To`,
  - `To` is integral and `remove_cvref_t<From>` is `int`, or
  - `To` satisfies `unsigned_integral` and `remove_cvref_t<From>` is `unsigned int`.

11.4 Modify [simd.overview]

In [simd.overview], insert:

[simd.overview]

```
// ([simd.ctor]), basic_vec constructors
template<~~class~~simd-broadcast-arg U>
  constexpr explicit(see below) basic_vec(U&& value) noexcept;
template<simd-consteval-broadcast-arg U>
  consteval basic_vec(U&& x)
template<class U, class UAbi>
  constexpr explicit(see below) basic_vec(const basic_vec<U, UAbi>&) noexcept;
```

11.5 Modify [simd.ctor]

[simd.ctor]

`template<~~class~~simd-broadcast-arg U> constexpr explicit(see below) basic_vec(U&& value) noexcept;`

5 Let `From` denote the type `remove_cvref_t<U>`.

6 ~~*Constraints*: `value_type` satisfies `constructible_from<U>`.~~

6 *Effects*: Initializes each element to the value of the argument after conversion to `value_type`.

7 *Remarks:* The expression inside `explicit` evaluates to `false` if and only if `U` satisfies `convertible_-to<value_type>`, and either

- `From` is not an arithmetic type and does not satisfy *`constexpr-wrapper-like`*,
- `From` is an arithmetic type and the conversion from `From` to `value_type` is value-preserving ([simd.general]), or
- `From` satisfies *`constexpr-wrapper-like`*, `remove_const_t<decltype(From::value)>` is an arithmetic type, and `From::value` is representable by `value_type`.

`template<simd-consteval-broadcast-arg U> consteval basic_vec(U&& x)`

8 *Preconditions*: The value of `x` is equal to the value of `x` after conversion to `value_type`.

9 *Effects*: Initializes each element to the value of the argument after conversion to `value_type`.

10 *Remarks:* An expression that violates the precondition in the *Preconditions*: element is not a core constant expression ([expr.const]).

12 Wording changes for [simd.math]

A set of fairly mechanical changes. TBD.

A REALLY_CONVERTIBLE_TO DEFINITION

```
template <typename To, typename From>
  consteval bool converting_limits_throws()
  {
    try {
      using L = std::numeric_limits<From>;
      [[maybe_unused]] To x = L::max();
      x = L::min();
      x = L::lowest();
    } catch(...) {
      return true;
    }
    return false;
  }

template <typename From, typename To>
  concept really_convertible_to = std::convertible_to<From, To>
                                    and not converting_limits_throws<To, From>();
```

B BIBLIOGRAPHY

[P3430R3] Matthias Kretz. *simd issues: explicit, unsequenced, identity-element position, and members of disabled simd*. ISO/IEC C++ Standards Committee Paper. 2025. URL: https://wg21.link/p3430r3.