Feedback on P0214

Document Number	P0820R3
Date	2018-03-29
Reply-to	Tim Shen <timshen91@gmail.com>
Audience	LEWG

Abstract

We investigated some of our SIMD applications and have some feedback on P0214R9.

This proposal does not intend to slow down P0214R9 from getting into the TS, but points out the flaws that are likely to encounter sooner or later. Fixing these flaws now is supposed to save time for the future.

Revision History

P0820R2 to P0820R3

- Rebase onto P0214R9.
- Adapt to P0964R1.
- Changed wording for alias `scalar` and `fixed_size`.

P0820R1 to P0820R2

- Rebased onto P0214R7.
- Extended `static_simd_cast` and `simd_cast` to use `rebind_abi_t`.
- Change `simd_abi::scalar` to an alias.

P0820R0 to P0820R1

- Rebased onto P0214R6.
- Added reference implementation link.
- For concat() and split(), instead of making them return simd types with implementation defined ABIs, make them return `rebind_abi_t<...>`, which is an extension and replacement of original `abi_for_size_t`.
- Removed the default value of `n` in `split_by()`.
- Removed discussion on relational operators. Opened an issue for it (https://issues.isocpp.org/show_bug.cgi?id=401).
- Proposed change to fixed_size from a struct to an alias, as well as guaranteeing the alias to have deduced-context.

Adapt functions onto P0964

P0964R1 proposes to extend `abi_for_size_t` to accept input ABIs to serve as hints. The following proposes to propagate the input ABI(s) to the output.

Proposed Change

```
template <size_t... Sizes, class T, class A>
tuple<~~simd<T, abi_for_size_t<T, Sizes>>~~resize_simd<Sizes, simd<T, A>>...>
    split(const simd<T, A>&);

template <size_t... Sizes, class T, class A>
tuple<simd_mask<T, ~~abi_for_size_t<T, Sizes>>~~resize_simd<Sizes, simd<T, A>>...>
    split(const simd_mask<T, A>&);

template <class T, class... As>
simd<T, ~~abi_for_size_t~~simd_abi::deduce_t<T, (simd_size_v<T, As> + ...), As...>>
concat(const simd<T, As>&...);
template <class T, class... As>
simd_mask<T, ~~abi_for_size_t~~simd_abi::deduce_t<T, (simd_size_v<T, As> + ...), As...>>
concat(const simd_mask<T, As>&...);


template <class T, class U, class Abi> see below simd_cast(const simd<U, Abi>& x);
```

Remarks: The function shall not participate in overload resolution unless

- every possible value of type U can be represented with type `To`, and
- either `is_simd_v<T>` is false, or `T::size() == simd<U, Abi>::size()` is true.

~~The return type is~~

- ~~`T` if `is_simd_v<T>` is true, otherwise~~
- ~~`simd<T, Abi>` if `U` is `T`, otherwise~~
- ~~`simd<T, simd_abi::fixed_size<simd<U, Abi>::size()>>.`~~

The return type is `rebind_simd<To, simd<U, Abi>>`.

```
template <class T, class U, class Abi> see below static_simd_cast(const simd& x);
```

Remarks: The function shall not participate in overload resolution unless either `is_simd_v<T>` is false or `T::size() == simd<U, Abi>::size()` is true.

~~The return type is~~

- ~~`T` if `is_simd_v<T>` is true, otherwise~~

- ~~simd<T, Abi> if U is T, otherwise~~
- ~~simd<T, simd_abi::fixed_size<simd<U, Abi>::size()>>.~~

The return type is `rebind_simd<To, simd<U, Abi>>`.

concat() doesn't support std::array

We propose it for being consistent with `split()`. Users may take the array from split(), do some operations, and concat back the array. It'd be hard for them to use the existing variadic parameter `concat()`.

Proposed Change

```
template <class T, class Abi, size_t N>
resize_simd<simd_size_v<T, Abi> * N, simd<T, Abi>>
concat(const std::array<simd<T, Abi>, N>&);

template <class T, class Abi, size_t N>
resize_simd<simd_size_v<T, Abi> * N, simd_mask<T, Abi>>
concat(const std::array<simd_mask<T, Abi>, N>&);
```

Returns: A `simd/simd_mask` object, the i-th element of which is initialized by the input element, indexed by `i / simd_size_v<T, Abi>` as the array index, and `i % simd_size_v<T, Abi>` as the simd/simd_mask array element index. The returned type contains `(simd_size_v<T, Abi> * N)` number of elements.

split() is sometimes verbose to use

It is sometimes verbose and not intuitive to use the array version of `split()`, e.g.

```
template <typename T, typename Abi>
void Foo(simd<T, Abi> a) {
  auto arr = split<simd<T, fixed_size<a.size() / 4>>>(a);
  // auto arr = split_by<4>(a) is much better.
  /* … */
}
```

and it's even more verbose for non-fixed_size types. We propose to add `split_by()` that splits the input by an `n` parameter.

Proposed Change

```
template <size_t n, class T, class A>
array<resize_simd<simd_size_v<T, A> / n, simd<T, A>>, n>
```

`split_by(const simd<T, A>& x);`

`template <size_t n, class T, class A>`
`array<resize_simd<simd_size_v<T, A> / n, simd_mask<T, A>>, n>`
`split_by(const simd_mask<T, A>& x);`

Remarks: The calls to the functions are ill-formed unless simd_size_v<T, A> is a multiple of n.

Returns: An array of simd/simd_mask objects with the *i*-th simd/simd_mask element of the *j*-th array element initialized to the value of the element in x with index *i* + *j* *(simd_size_v<T, A> / n). Each element in the returned array has `size_simd_size_v<T, A>::size() / n` elements.

simd_abi::scalar and fixed_size<N> are not an aliases

One possible implementation of ABI is to create a centralized ABI struct, and specialize around it:

```
enum class StoragePolicy { kXmm, kYmm, /* ... */ };
template <StoragePolicy policy, int N> struct Abi {};

template <typename T> using native = Abi<kYmm, 32 / sizeof(T)>;
template <typename T> using compatible = Abi<kXmm, 16 / sizeof(T)>;
```

Then every operation is implemented and specialized around the centralized `struct Abi`.

Unlike `native` and `compatible`, `scalar` and `fixed_size` is not an alias. Currently they require extra specializations other than the ones on `struct Abi`.

Proposed Change

~~`struct`~~`using scalar ~~{}~~= /* see below */;`
`template <int N> ~~struct~~using fixed_size ~~{}~~= /* see below */;`

[simd.abi]
`scalar` aliases to an implementation-defined ABI tag that is different from `fixed_size<1>`. Use of the scalar tag type requires data-parallel types to store a single element (i.e., simd::size() returns 1). ~~[Note: `scalar` shall not be an alias for `fixed_size<1>`. — end note]~~`scalar` shall not introduce a non-deduced context.

`fixed_size<N>` aliases to an implementation-defined ABI tag. Use of the simd_abi::fixed_size tag type requires data-parallel types to store N elements (i.e. simd>::size() returns N). simd> and simd_- mask> with N > 0 and N <= max_fixed_size shall be supported. Additionally, for every supported simd (see [simd.overview]), where Abi is an ABI tag that is not a specialization

of simd_abi::fixed_size, N == simd::size() shall be supported. `fixed_size` shall not introduce a non-deduced context.

Reference

- The original paper: P0214R9
- Experimental implementation: https://github.com/google/dimsum