Feedback on P0214

Document Number	P0820R2
Date	2018-02-06
Reply-to	Tim Shen <timshen91@gmail.com>
Audience	SG1, LEWG

Abstract

We investigated some of our SIMD applications and have some feedback on P0214R7.

This proposal does not intend to slow down P0214R7 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

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.

Is abi_for_size_t the right way to specify the ABIs for split() and concat()?

Currently, the return types of `split()` and `concat()` don't depend on the input ABI(s) other than for calculating sizes. This limits the implementation by enforcing the following expressions to produce the same type of objects:

- `concat(native_simd<int32>())`
- `concat(compatible_simd<int32>(), compatible_simd<int32>())`

Suppose that `compatible_simd<int32>` is implemented by 16-bytes, XMM registers on x86; and `native_simd<int32>` is implemented by 32-bytes, YMM registers on x86. Ideally, we'd like both concat()s to be no-ops, if they are allowed to return different types: in the first case the return value stays in the same YMM register; in the second case, the returned values still stay in the same XMM registers.

To make both calls no-ops, the return types of those two need to be different.

That said, it may not practically matter **in the function body**, if the optimizer is smart enough. It always affects **function call boundaries**, though. Example of a function call boundary: https://godbolt.org/g/6EEE8H.

The fundamental issue is that `abi_for_size` only depends on the element type and the size. Since it is only used by `concat()` and `split()`, we propose to drop `abi_for_size` and `abi_for_size_t`, and let the implementation pick the returned ABI(s) for `concat()` and `split()`.

Besides the performance benefits, `rebind_abi_t` also allows `static_simd_cast`, `simd_cast`, `to_compatible`, `to_native` to extend naturally.

Proposed Change

```
template <class T, size_t N, typename... As>
    struct ~~abi_for_size~~rebind_abi { using type = implementation-defined; };

template <class T, size_t N, typename... As>
    using ~~abi_for_size_t~~rebind_abi_t =
        typename ~~abi_for_size~~rebind_abi<T, N, As...>::type;

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

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

Returns: A tuple of simd/simd_mask objects with the *i*-th simd/simd_mask element of the *j*-th tuple element initialized to the value of the element in x with index *i*+ partial sum of the first *j* values in the Sizes pack.

`template <class T, class... As>`
`simd<T, `~~`abi_for_size_t`~~`rebind_abi_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`~~`rebind_abi_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.

If `is_simd_v<T>` is true, the return type is `T`. Otherwise, ~~if `U` is `T`, the return type is `simd<T, Abi>`. Otherwise, the return type is `simd<T, simd_abi::fixed_size<simd<U, Abi>::size()>>`~~the return type is `simd<To, rebind_abi_t<To, simd<U, Abi>::size(), 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. If `is_simd_v<T>` is true, the return type is `T`. Otherwise, ~~if either `U` is `T` or `U` and `T` are integral types that only differ in signedness, the return type is simd. Otherwise, the return type is `simd<T, simd_abi::fixed_size<simd<U, Abi>::size()>>`~~the return type is `simd<To, rebind_abi_t<To, simd<U, Abi>::size(), 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>`
`simd<T, rebind_abi_t<T, N, Abi>> concat(const std::array<simd<T, Abi>, N>&);`

```
template <class T, class Abi, size_t N>
simd_mask<T, rebind_abi_t<T, N, 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<simd<T, rebind_abi_t<T, simd_size_v<T, A> / n, A>>, n> split_by(
    const simd<T, A>& x);
template <size_t n, class T, class A>
array<simd_mask<T, rebind_abi_t<T, simd_size_v<T, A> / n, 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, size_t 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` ~~`{}`~~`= /* implementation defined */`;

Remark: scalar shall not introduce a non-deduced context.

`template <int N>` ~~`struct`~~`using` `fixed_size` ~~`{}`~~`= /* implementation defined */`;

Remark: fixed_size shall not introduce a non-deduced context.

Reference

- The original paper: [P0214R7]
- Experimental implementation: https://github.com/google/dimsum