Feedback on P0214R5

Document Number	P0820R0
Date	2017-10-13
Reply-to	Tim Shen <timshen91@gmail.com>
Audience	SG1, LEWG

Abstract

We investigated some of our SIMD applications and have some feedback on p0214r5.

This proposal does not intend to slow down p0214r5 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.

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()`.

Proposed Change

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

template <size_t... Sizes, class T, class A>
tuple<simd<T, ~~abi_for_size_t<Sizes>~~/* implementation defined */>...>
    split(const simd<T, A>&);

template <size_t... Sizes, class T, class A>
tuple<simd_mask<T, ~~abi_for_size_t<Sizes>~~/* implementation defined */>...>
    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. The pack expansions in the returned type are on Sizes.... The returned type contains in total (Sizes + …) number of elements.

```
template <class T, class... As>
simd<T, ~~abi_for_size_t<T, (simd_size_v<T, As> + ...)>~~/* implementation defined */>
concat(const simd<T, As>&...);
template <class T, class... As>
simd<T, ~~abi_for_size_t<T, (simd_size_v<T, As> + ...)>~~/* implementation defined */>
concat(const simd_mask<T, As>&...);
```

Returns: A simd/simd_mask object initialized with the concatenated values in the xs pack of simd/simd_- mask objects. The *i*-th simd/simd_mask element of the *j*-th parameter in the xs pack is copied to the return value's element with index *i* + partial sum of the size() of the first *j* parameters in the xs pack. The returned type contains `(simd_size_v<T, As> + …)` number of elements.

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, /* implementation defined */> concat(const std::array<simd<T, Abi>, N>&);

template <class T, class Abi, size_t N>
simd_mask<T, /* implementation defined */> 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. `n` is defaulted to 2.

Consequently, `split_by()::abi_type` may be an ABI that users can't spell out.

Proposed Change

```
template <size_t n = 2, class T, class A>
array<simd<T, /* implementation defined */>, n> split_by(
    const simd<T, A>& x);
template <size_t n = 2, class T, class A>
array<simd_mask<T, /* implementation defined */>, 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.

Relation operators don't return bool

Currently relation operations return simd_mask<>, which can't be converted to bools. This is inconsistent with what other algorithms expect. The proposed change is to rename the operators to normal free functions, but also add `operator==()` and `operator!=()` for returning bools. Alternatively, the operators can also be deleted.

It's unclear to us whether it's useful to add lexicographical, bool-returning `operator<(), operator>(), operator<=()`, and `operator>=()`. Avoid them for now.

Name candidates for component-wise ==, !=, >=, <=, >, <, respectively:

- eq, ne, ge, le, gt, lt
- cmpeq, cmpne, cmpge, cmple, cmpgt, cmplt
- cmp_eq, cmp_ne, cmp_ge, cmp_le, cmp_gt, cmp_lt
- equal_to, not_equal_to, greater_equal, less_equal, greater, less
- simd_equal_to, simd_not_equal_to, simd_greater_equal, simd_less_equal, simd_greater, simd_less

Proposed Change

friend mask_type ~~operator==~~eq(const simd&, const simd&);
friend mask_type ~~operator!=~~ne(const simd&, const simd&);
friend mask_type ~~operator>=~~ge(const simd&, const simd&);
friend mask_type ~~operator<=~~le(const simd&, const simd&);
friend mask_type ~~operator>~~gt(const simd&, const simd&);
friend mask_type ~~operator<~~lt(const simd&, const simd&);

Returns: A simd_mask object initialized with the results of the component-wise application of the indicated ~~operator~~operation.
Throws: Nothing.

friend simd_mask ~~operator==~~eq(const simd_mask&, const simd_mask&) noexcept;
friend simd_mask ~~operator!=~~ne(const simd_mask&, const simd_mask&) noexcept;

Returns: A simd_mask object initialized with the results of the component-wise application of the indicated ~~operator~~operation.

```
friend bool operator==(const simd&, const simd&);
friend bool operator!=(const simd&, const simd&);

friend bool operator==(const simd_mask&, const simd_mask&);
```

`friend bool operator!=(const simd_mask&, const simd_mask&);`

Returns: The result of performing lexicographical operation of the arguments. The type of operation is indicated by the operators.
Throws: Nothing.

Alternative Proposal

Same to proposed change, but remove the definitions of operators.

fixed_size<N> might be hard to implement

In our implementation (without `fixed_size<N>` yet), all ABIs are defined in terms of (storage policy, total bytes), e.g.

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

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

This implementation enables the opportunity to experiment with ABIs like `Abi<StoragePolicy::kXmm, 32>` (use two XMM registers). All algorithms are implemented and specialized for `StoragePolicy` and `num_bytes`.

We think that the implementation above is a reasonable implementation, and should be allowed. The fundamental reason is that ABI is a "low-level" term, defined at **binary** level, as well as "number of bytes". Meanwhile, "element type" and "number of element" are high-level terms. To not mix low-level and high-level terms in a same place is good.

However, to support `fixed_size<N>`, the implementation needs to specialize again for many operations, as fixed_size is its own struct, not an alias. Also, `fixed_size<N>` can't be represented by `struct Abi`.

This feedback doesn't have a proposed change, as several options we thought about are too complicated. That said, the following are the considered changes.

(1) remove `fixed_size<N>`, and change `fixed_size_simd<T, N>` to be implementation defined. Example implementation:

```
template <typename T, size_t N>
using fixed_size_simd =
    simd<T, Abi<StoragePolicy::kXmm, sizeof(T) * N>>;
```

The problem is that `fixed_size_simd<>` may introduce a non-deduced context.

(2) change `fixed_size<N>` to `fixed_size<T, N>`:

```
template <typename T, size_t N>
using fixed_size = Abi<StoragePolicy::kXmm, sizeof(T) * N>;
```

Problem is that it complicates the ABI. For example, is `fixed_size<int32_t, N>` the same as `fixed_size<float, N>`? It may also introduce a non-deduced context.