More simd<> Operations

Document Number	P0918R0
Date	2018-02-08

Reply-to	Tim Shen <timshen91@gmail.com>

Audience	SG1, LEWG

Abstract

P0214 [1] defines portable types for SIMD, as well as a set of common SIMD operations. However, the set of operations are not sufficient to expose some of the hardware functionality.

Specifically, some SIMD operations that are heavily used in practice, but delivered by hardware with rather subtle differences, causing importability. This proposal allows these many variations of hardware features to be abstracted into a consistent, portable API.

This proposal also depends on P0820 [2] for `rebind_abi_t` and `split_by`.

Proposed Functions

shuffle

```
template <size_t... indices, typename T, typename Abi>
simd<T, rebind_abi_t<T, sizeof...(indices), Abi>>
shuffle(const simd<T, Abi>& v);

template <size_t... indices, typename T, typename Abi>
simd_mask<T, rebind_abi_t<T, sizeof...(indices), Abi>>
shuffle(const simd_mask<T, Abi>& v);
```

Remarks: These functions shall not participate overloading resolution unless `((indices < simd_size_v<T, Abi>) && ...)`.

Returns: A new simd/simd_mask object `r`, where `r[i] = v[indices[i]]` and `indices[i]` is the ith element in `indices`.

The shuffle operation permutes the input SIMD elements arbitrarily, allowing omission and repetition of these elements. The permutation needs to be specified at compile-time. The case where the permutation is only known at runtime is out of scope.

Note that hardware often provide interfaces that take two SIMD values, not one. For the proposed portable interface, this can be achieved by composing with `concat()`, e.g. `` `shuffle<7, 6, 5, 4, 3, 2, 1, 0>(concat(a, b))` ``, where `a` and `b` are with sizes of 4. With compiler optimizations, this comes to no performance penalty[1]. The single-argument shuffle is easier to learn and result in more explicit call sites.

Note that for variadic number of elements, users can use `std::index_sequence`:

```
template <size_t... indices>
simd<int> ElementsWithOddIndices(simd<int> a, simd<int> b,
                                 std::index_sequence<indices...>) {
  static_assert(sizeof...(indices) == a.size(), "");
  // Returns all elements with odd indices in concatenated a and b.
  return shuffle<(2 * indices + 1)...>(concat(a, b));
}
```

interleave

```
template <typename T, typename Abi>
simd<T, rebind_abi_t<T, simd_size_v<T, Abi> * 2, Abi>>
interleave(const simd<T, Abi>& u, const simd<T, Abi>& v);

template <typename T, typename Abi>
simd_mask<T, rebind_abi_t<T, simd_size_v<T, Abi> * 2, Abi>>
interleave(const simd_mask<T, Abi>& u, const simd_mask<T, Abi>& v);
```

Returns: `shuffle<(i / 2 + (i % 2) * simd_size_v<T, Abi>)...>(concat(u, v))`, where `i` is a variadic pack of `size_t` in `[0, simd_size_v<T, Abi> * 2)`.

`interleave()` shuffles the given two SIMD objects in a specific way, interleaving the input values into a single `simd<>` object. Hardware instructions like `punpcklwd` on x86 can be achieved by combining `split()` and `interleave()`, `` `interleave(split_by<2>(a)[0], split_by<2>(b)[0])` `` with the assist of proper optimizations[2]; vice versa, `interleave()` itself can be implemented in terms of instructions like `punpcklwd`.

sum_to

```
template <typename AccType, typename T, typename Abi>
AccType sum_to(const simd<T, Abi>& v, const AccType& acc);

template <typename AccType, typename T, typename Abi>
```

[1] https://godbolt.org/g/BEXRmZ
[2] https://godbolt.org/g/svsyfh

`AccType sum_to(const simd<T, Abi>& v);`

`Let U be typename AccType::value_type.`

Remarks: This function shall not participate overloading resolution unless

- `is_simd_v<AccType>`, and
- `simd<T, Abi>::size() % AccType::size() == 0`, and
- `is_integral_v<T> && is_integral_v<U>`, and
- `is_signed_v<T> == is_signed_v<U>`, and
- `sizeof(U) >= sizeof(T)`

Returns: For the first overloading, `r + acc`, where `r[i]` is `GENERALIZED_SUM(std::plus<>, static_cast<U>(v[S*i]), static_cast<U>(v[S*i+1]), ..., static_cast<U>(v[(S+1)*i - 1]))`, and `S` is `v.size() / AccType::size()`. For all `i`, `r[i]` has an unspecified value if the corresponding `GENERALIZED_SUM` overflows. For the second overloading, `sum_to(v, AccType(0))`.

This function partially sums up every `M` adjacent elements in the input simd<> object, where `M` is `simd<T, Abi>::size() / AccType::size()`.

On some architectures - x86 for example - this can be used to implement an efficient[3] full summation over a large buffer of integers:

```
// Returns the sum of all uint8_ts in the buffer.
int64_t Sum(uint8_t* buf, int n) {
  constexpr size_t stride = native_simd<uint8_t>::size();
  native_simd<int64_t> acc(0);
  int i;
  for (i = 0; n - i >= stride; i += stride) {
    acc = sum_to(native_simd<uint8_t>(buf + i), acc);
  }
  // handle leftovers in [i, n)
  return reduce(acc);
}
```

In practice, summation usage does not always fit in one or more calls to `Sum()`, e.g. multiple summations with their loops fused. Therefore, it makes sense to let the accumulator `acc` and the loop exposed in the user code.

[3] `_mm_sad_epu8` is the fastest approach in the benchmark:
https://gist.github.com/timshen91/0f321fe2c5cfb04015917c0529052158

This provides a simple and consistent interface for various flavors of hardware summation instructions:

- Elements are not widened, and total number of bytes is changed: `phaddd` on x86, `VPADD` on ARM.
- Elements are widened, but total number of bytes isn't changed: `psadbw`, `pmaddwd` on x86, `vmsumshm` on PowerPC.
- Full sum, e.g. `ADDV` on ARMv8.

Note that the efficiency of `sum_to()` is architecture-specific for a given `(T, Abi, AccType)` combination. Users do need architectural knowledge to pick the most efficient `AccType` on that architecture, as well as using `sum_to()` or not. Implementations are suggested to document which instruction is generated by which instantiation, and warn about uses of inefficient ones.

multiply_sum_to

```
template <typename AccType, typename T, typename Abi>
AccType multiply_sum_to(
    const simd<T, Abi>& v, const simd<T, Abi>& u, const AccType& acc);

template <typename AccType, typename T, typename Abi>
AccType multiply_sum_to(
    const simd<T, Abi>& v, const simd<T, Abi>& u);
```

Let `U` be `typename AccType::value_type`.

Remarks: This function shall not participate overloading resolution unless

- `is_simd_v<AccType>`, and
- `simd<T, Abi>::size() % AccType::size() == 0`, and
- `is_integral_v<T> && is_integral_v<U>`, and
- `is_signed_v<T> == is_signed_v<U>`, and
- `sizeof(U) == 2 * sizeof(T)`

Returns: For the first overloading, `sum_to<AccType>(static_simd_cast<U>(v) * static_simd_cast<U>(u), acc)`. For the second overloading, `multiply_sum_to(v, AccType(0))`.

This function provides integral "element-wise multiply + partial sum" functionality on various architectures, for example

- `pmaddwd` on x86
- `vmsumshm` on PowerPC
- `VMLAL` on ARM

In practice, this is often used for implementing integral dot product. It makes sense to expose the accumulator to the users for the same reason as `sum_to()` does.

saturated_simd_cast

```
template <typename U, typename T, typename Abi>
simd<U, rebind_abi_t<U, simd_size_v<T, Abi>, Abi>>
saturated_simd_cast(const simd<T, Abi>& v);
```

If `is_integral_v<U>`, then let `L` be `numeric_limits<U>::min()` and `R` be `numeric_limits<U>::max()`.

If `is_floating_point_v<U>`, then `L` is implementation-defined among `-numeric_limits<U>::max()` and `-numeric_limits<U>::inf()`, depending on the rounding mode; `R` is implementation-defined among `numeric_limits<U>::max()` and `numeric_limits<U>::inf()`, depending on the rounding mode.

[Note: `L` may not equal to `-R` even with `is_floating_point_v<U>` --end note]

Remarks: This function shall not participate overloading resolution unless `U` is a vectorizable type

Returns: A simd object `r`, where `r[i]` is

- `L`, if `v[i]` is less than `L`, or
- `R`, if `v[i]` is greater than `R`, or
- `static_cast<U>(v[i])`.

This function is similar to `simd_cast()`, but clamps the result when overflow happens. This captures many of the uses of "saturated pack" integral operation, which effectively narrows down each element by half of its size, and clamps each narrowed value.

It also provides floating point -> integer saturated conversion.

Hardware instruction examples include:

- `packsswb`, `packuswb` on x86
- `vpkswss`, `vpkswus`, `vctsxs` on PowerPC
- `VQMOVN`, `VQMOVUN` on ARM

Optional Designs

Integer-only vs Integer and Floating Point

`sum_to()`, `multiply_sum_to()` are actually well defined on `simd<A, Abi> -> simd<B, Abi>`, where `A` and `B` are a floating point types, but the proposal requires `A` and `B` to be integral types. This is because the floating point version is not seen heavily used in practice, nor efficiently

supported by hardware. The integral type constraints complicates the interface specification and learning experience. On the other hand, to allow floating points may create performance pitfalls, e.g. using `sum_to()` to sum up a large buffer of floats is not the fastest way.

Alternative: we could open them up to floating points only for simpler interface and completeness. Implementations are suggested to warn about inefficient uses.

Provide a trait for `AccType`

In generic code, when using `sum_to()` or `multiple_sum_to()`, if `AccType` is not deduced from the function parameter `acc`, it can be hard to specify by the users generically, e.g. the input element type T and output element type U, where `sizeof(U) = 2 * sizeof(T)`. The standard library may provide a type trait that takes `T` and produces a `(u)intN_t`, where N is 2^K `* sizeof(T)`.

This is optional, as we don't see a lot of generic SIMD programming today.

For example[4]:

```
template <typename T, size_t numerator, size_t denominator = 1>
struct scale_width_by {
  // a type with width sizeof(T) * numerator / denominator,
  // otherwise similar to T in terms of is_integral, and
  // is_signed/is_unsigned.
  using type = ...;
};

template <typename T, size_t numerator, size_t denominator = 1>
using scale_width_by_t =
    typename scale_width_by<T, numerator, denominator>::type;

// Examples:
// scale_width_by_t<int8_t, 4> => int32_t
// scale_width_by_t<int64_t, 1, 2> => int32_t
// scale_width_by_t<int, 2> => int64_t, if int is int32_t
// scale_width_by_t<float, 2> => double, given proper sizes of them
```

Alternatives:

- In addition, provide `scale_width_by`.
- In addition, provide `scale_width_by`, and make `U` in `multiply_sum_to()` default to `scale_width_by_t<T, 2>`.

[4] `scale_width_by` can be used for generic purposes, so maybe it should be in a separate proposal.

Prototype

Dimsum [3] implements variations of `shuffle()`, `interleave()` (with the name `zip`), `sum_to()` (with the name `reduce_add`), and `multiply_sum_to()` (with the name `mul_sum`).

Reference

[1] P0214, the SIMD proposal
[2] P0820, the supplemental proposal on the top of P0214
[3] Dimsum, the prototype