P2664R1:
Proposal to extend std::simd with permutation API

Authors

Daniel Towner
(daniel.towner@intel.com)

Introduction

ISO/IEC 19570:2018 introduced data-parallel types to the standard library. Intel supports the concept of a standard interface to SIMD instruction capabilities and have made extra suggestions for other APIs and facilities for std::simd in document P2638R0 (1).

One of the extra features we suggested was the ability to be able to permute elements across or within simd values. These operations are critical for supporting formatting of data in preparation for other operations (e.g., transpose, strided operations, interleaving, and many more), but are relatively poorly supported in the current std::simd proposal. Many modern SIMD processors include sophisticated support for permutation instructions so in this document I shall propose a set of API functions which make those underlying instructions accessible in a generic way.

In this document I shall start with a little background into the current std::simd proposal and extensions, and then describe a proposal for some new API functions.

Revisions

R1 – Added outcome of straw polls from Kona '22

Straw Polls from SG1 at Kona '22

Poll (to related paper P1928R1): After significant experience with the TS, we recommend that the next version (the TS version with improvements) of std::simd target the IS (C++26)

SF	F	N	A	AS
10	8	0	0	0

Unanimous consent

Poll: Future papers and future revisions of existing papers that target std::simd should go directly to LEWG. (We do not believe there are SG1 issues with std::simd today.)

SF	F	N	A	AS
9	8	0	0	0

Support for permutation in existing std::simd proposal

In the original std::simd TS (2) there were only a few functions which could be used to achieve any sort of element permutation across or within simd<> values. They were relatively modest in scope and they built upon the ability to split and concatenate simd values at compile-time. For example, a simd value containing 5 elements could be split into two smaller pieces of sizes 2 and 3 respectively:

```
fixed_size_simd<float, 5> x;
…
const auto [piece0, piece1] = split<2, 3>(x);
```

Those smaller pieces could be acted on using any of the std::simd features, and then later recombined using concat, possibly in a different order or with duplication:

```
const auto output = concat(piece1, piece1, piece0);
```

The basic split and concat functions can be used to achieve several permutation effects, but they are a little unwieldy for anything complicated, and since they are also compile-time they also preclude any dynamic runtime permutation.

In Intel's response to the original TS document P2638R0 (1) we suggested a couple of extra operations to insert and extract simd values to and from other simd containers:

```
const auto t = extract<2, 5>(v); // Read out 3 simd values, starting from position 2
insert<5>(v, t); // Put the 3 previously read values back into the container at position 5
```

These are convenient, but still don't allow arbitrary or expressive permute operations.

A couple of suggestions were made in P0918R1 (3) to add more permutation features for an arbitrary compile-time shuffle, and for a permutation targeted at a specific opertion called an interleave. The compile-time shuffle can be used as follows:

```
const auto p = shuffle<3, 4, 1, 1, 2, 3>(x);
```

Note that arbitrary duplication and reordering of elements can be achieved with the shuffle function. The output simd has its size specified from the number of supplied compile-time indexes. It is noted also in the shuffle API that this function can be applied to simd_mask values too, and that shuffling across multiple simd values can be achieved by first concatenating the values into a single larger simd:

```
const auto p = shuffle<10, 9, 0, 1>(concat(x, y));
```

In addition to the shuffle function, P0918R1 also proposes a function called interleave which accepts two simd values and interleaves respective elements into a new simd value which is twice the size. For example, given inputs [a0 a1 a2] and [b0 b1 b2], the output would be [a0 b0 a1 b1 a2 b2].

Interleaving is a common operation and is often directly supported by processors with explicit instructions (e.g., punpcklwd), but there are frequently other common permutation operations which will have specialist hardware support. Some SIMD libraries try to expose those underlying instructions by providing a richer API which maps to operations with known hardware support. For example, Google's Highway (4) provides DupOdd, DupEven, ReverseBlocks, Shuffle0231[etc], which map efficiently to underlying instructions.

Extending std::simd to support generic permutations

There are three classes of permutation which could be supported by std::simd –using compile-time computed indexes, using another simd as the index source, and using a bit-mask for selection. Each of these classes of permute will be examined separately in the following sections. We also consider what it means to permute memory.

Note that we assume that these permutes can be applied to any simd-like value, including simd_mask values.

1.1 Using compile-time computed indexes

The first proposal is to provide a permute function which accepts a compile-time function which is used to generate the indexes to use in the permutation. This is a very powerful concept: firstly, it allows the task of computing the indexes to be offloaded to the compiler using a pattern generating function, and secondly it works on simd<> values of arbitrary size. For example, the DupEven function could be implemented as:

```
const auto dupEvenIndex = [](size_t idx) -> size_t { return (idx / 2) * 2; };
const auto de = permute(values, dupEvenIndex);
```

Note that the permutation function maps from the index of each output element to the index which should be used as the source for that position. So, for example, the dupEvenIndex function would map the output indexes [0, 1, 2…) to the source indexes [0 0 2 2 4 4…). It is an error to generate indexes which are negative or bigger than the size of the input simd.

By default, the permute function will generate as many elements in the output simd as there were input values, but the output size can be explicitly specified if it cannot be inferred or it needs to be modified. For example, the following permute generates 4 values with a stride of 3 (i.e., indexes [0, 3, 6, 9]).

```
const auto stride3 = permute<4>(values, [](size_t idx) -> size_t { return idx * 3; });
```

Inevitably programmers will want to use certain types of permutation more often than others – dupOdd, dupEven, strideRead, element rotation (e.g., `std::rotate`), interleave, and so on. It could be useful for std::simd to either define a set of useful lambda functions which can be used in a permutation:

```
const auto dupEven = [](size_t idx) -> size_t { return (idx / 2) * 2; }
const auto dupOdd = [](size_t idx) -> size_t { return idx | 1; }
const auto swapOddEven = [](size_t idx) -> size_t { return idx ^ 1; }
const auto even= [](size_t idx) -> size_t { return idx * 2; };
```

or define a set of functions which abstract the permute entirely:

```
auto dupEven(simdable v) { return permute(v, [](size_t idx) -> size_t { return (idx / 2) * 2; }; };
auto dupOdd(simdable v) { return permute(v, [](size_t idx) -> size_t { return idx | 1; }; };
auto swapOddEven(simdable auto v) { return permute(v, [](size_t idx) -> size_t { return idx ^1; }
auto even(simdable auto v) { return permute<v::size()/2>(v, [](size_t idx) -> size_t { return idx *
2; }; );
```

The advantage of the second form is that the function can also compensate for a change in size of the output simd in the call to the permute, whereas in the first form the user would have to handle the change in size in the call itself, rather than having the function take care of it automatically.

In other libraries which provide named specific permute functions, the reason often given is that it allows specific hardware instructions to be used to implement those permutes. For example, if dupLow were provided, then the Intel instruction called movldup could be invoked directly when compiling for that target. In practice however, modern compilers are easily capable of determining the best instruction sequence for an arbitrary set of compile-time indexes without help. For example, the following table shows some compile-time function permute calls and the code that the LLVM 14.0.0 compiler has generated for each:

permute call	Purpose	Output from clang 14.0.0

`permute (x, [](auto idx) {`
`  return idx & ~1;`
`});`
	Duplicate even elements	`vmovsldup zmm0, zmm0`

`permute (x, [](auto idx) {`
`  return idx ^ 1;`
`});`
	Swap even/odd elements in each pair (complex-valued IQ swap)	`vpermilps ymm0, ymm0, 177`

`permute<8>(x, [](auto idx) {`
`  return idx + 8;`
`});`
	Extract upper half of a 16-element vector. Note the instruction sequence accepts a zmm input and returns a ymm output.	`vextractf64x4 ymm0, zmm0, 1`

In each case the compiler has accepted the compile-time index constants and converted them into an instruction which efficiently implements the desired permutation. We can safely leave the compiler to determine the best instruction from an arbitrary compile-time function.[1]

1.2 Using another simd as the dynamic index

The second proposal for the permute API is to allow the required indexes to be passed in using a second simd value containing the run-time indexes:

```
fixed_size_simd<unsigned, 8> indexes = getIndexes();
fixed_size_simd<float, 5> values = getValues();
...
const auto result = permute(values, indexes);
```

The permute function will return a new simd value which has the same element type as the input value being permuted, and the same number of elements as the indexing simd (i.e., float and 5 in the example above). The permute may duplicate or arbitrarily reorder the values. The index values must be of unsigned integral type, with no implicit conversions from other types permitted. The behavior is undefined if any index is greater than the number of available elements. Dynamic permutes across multiple input values are handled by concatenating the input values together.

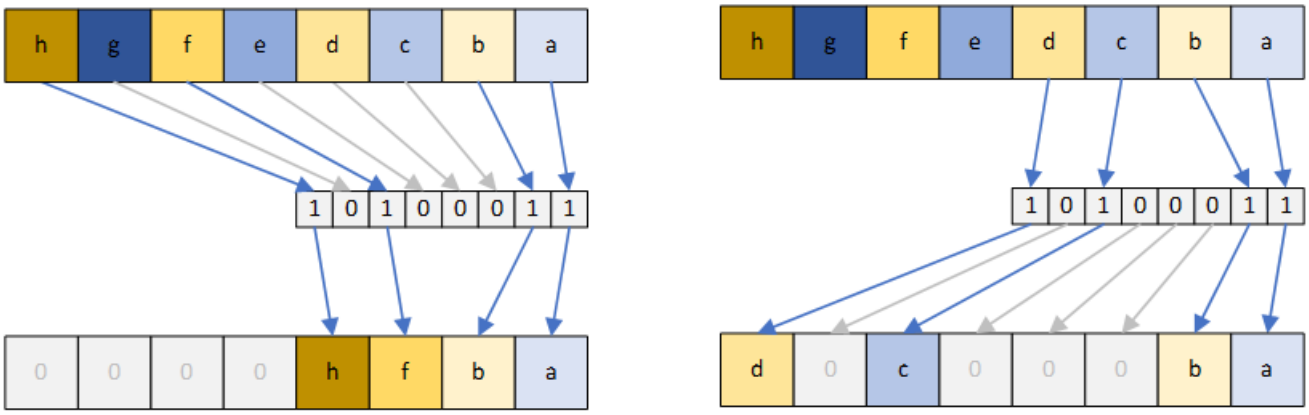
1.3 Permutation using simd masks

A third variant of permutation is where a simd_mask is used as a proxy for an index sequence instead. The presence or absence of an element in the simd_mask will cause the element to be used or not. The following diagrams illustrate this more clearly:

Compress

Expand

[1] When testing Intel's example implementation of the std::simd permute function one example code fragment ran 15% faster because the compiler noticed a better way to handle a particular permute pattern when compared to the programmer's explicit use of what they thought was a good intrinsic!



On the left the values in the simd are compressed; only those elements which have their respective simd_mask element set will be copied into the next available space in the output simd. Their relative order will be maintained, and the selected elements will be stored contiguously in the output. The output value will have the same static number of elements as the input value, but unused values will be zero initialized. The behavior of the function is similar to std::copy_if, although in many simd libraries this is called compression instead. The expansion function on the left has the opposite effect, copying values from a contiguous region to potentially non-contiguous positions indicated by a mask bit. The two functions have prototypes as follows:

```
template<typename _T, typename _Abi>
simd<_T, _Abi> compress(const simd<_T, _Abi>& value, const simd_mask<_T, _Abi>& mask);
template<typename _T, typename _Abi>
simd<_T, _Abi> expand(const simd<_T, _Abi>& value, const simd_mask<_T, _Abi>& mask);
```

Some target processors already have support for these operations (e.g., `_mm512_mask_compress_epiX` in AVX-512) and this should be used for implementation where possible, as the native instructions will be much faster than any equivalent software sequence. When the target doesn't support these operations directly the necessary implementation will be non-trivial. It is suggested that these operations should be part of std::simd so that the library and compiler can take care of implementing these tricky but useful operations.

Open questions:

- Should compress be called copy_if instead to match C++ std::copy_if? If so, what should expand be called?
- Return number of bits too or rely on popcount being used on the mask to get this if needed?
- Provide a partition-like function too (i.e., like std::stable_partition). It can either be implemented using the above or be a separate implementation if that is more efficient.
- Should an addition parameter be provided to use for unused output positions rather than value initialising?

1.4 Memory-based permutes

When permuting values which are stored in memory, the operations are normally called gather (reading data from memory), or scatter (writing values to memory). The existing proposals for permute above could be overloaded to also accept a memory pointer:

```
int array[1024];
…
const auto r0 = permute<6>(array, dupEven);
const auto r1 = permute(array, indexes);
```

Alternatively, these functions could be called gather and scatter to make it explicitly clear that they are operating on memory.

Questions:

- Permute of simd_mask stored in memory may be impossible in the case of compact bit masks, since individual bits are not addressable. In such case, naming the operations gather/scatter would make it cleaner to disallow simd_mask as a source.
- It is easy to allow a pointer as a permute source for a gather, but representing a scatter is harder. Then it makes sense to call these operations gather/scatter instead.

Summary

In this document we have described three styles of interface for handling permutes: compile-time generated, simd-indexed, and mask-indexed. In combination, these three interfaces allow all types of common permute to be expressed in a way which is clean, concise, and efficient for the compiler to implement.

Bibliography

1. **Intel Corporation.** P2638R0. [Online] https://www.open-std.org/jtc1/sc22/wg21/docs/papers/2022/p2638r0.pdf.

2. **ISO/IEC 19570:2018. Programming Languages — Technical Specification for C++ Extensions for Parallelism. Standard. ISO/IEC JTC 1/SC 22. [Online] 2018. https://www.iso.org/standard/70588.html..**

3. Shen, Tim. P0820R: Feedback on P0214R5. [Online] 2017. http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2017/p0820r0.pdf.

4. Google. Highway. [Online] https://github.com/google/highway/.

5. Shen, Tim. https://open-std.org/JTC1/SC22/WG21/docs/papers/2018/p0918r1.pdf. [Online]