P2638R0: Intel's response to P1915R0 for std::simd parallelism in TS 2

Authors

Daniel Towner

Sergey Maslov

Hideki Saito

Ilya Burylov

Introduction

ISO/IEC 19570:2018 (1) introduced data-parallel types to the standard library. P1915R0 (2) outlined the feedback request for std::simd.

This paper gives our feedback based on our experience with enabling vectorization across multiple Intel products, developing our own simd class libraries, and developing a reference implementation of std::simd using LLVM and Intel's OneAPI compiler. We have also looked at the reference implementation developed by Matthias Kretz which is part of the experimental libstdc++ in this repo: https://gcc.gnu.org/git/.

Overall we believe that std::simd strives to fill a long standing gap of a standard interface to a range of intrinsic-level APIs provided by several vendors. The goal is honourable, and we would welcome such an addition to the C++ standard.

In this document we also propose some additional features and capabilities that we think should be considered for std::simd, including ideas for closer alignment with C++20 features.

Table of Contents

Questions raised in P1915R0

In this section we shall respond to each of the questions raised in P1915R0.

1.1 name and ub of popcount, find_first_set, find_last_set

std::simd (1) introduced a set of functions for counting the number of entries in a simd_mask<T>. C++20 introduced scalar [bit.count] functions for the same purpose and slightly different semantics. The definition of count[lr]_(zero|one) functions in C++20 has an advantage over find_(first|last)_set from std::simd by not introducing the UB for zero input.

There is no significant advantage of existing std::simd definition, because HW instructions, which mimic this behavior are scalar (see BSF/BSR instruction (3)). At the same time similar vector instructions (see VPLZCNTD instruction (3)) are defined without introducing UB and closer to C++20 functions definition.

Alternatively, it may be useful to allow simd_mask to be convertible to and from a std::bitset, rather than inventing new names and operations on std:simd_mask. The various bit operations proposed for simd_mask can then be accessed as following instead:

```
auto numBits = simd_mask.to_bitset().count();
```

Recommendation: align with C++20 in function names and definitions.

Alternative: Provide a way to convert a simd_mask to a std::bitset and use its functions instead.

1.2 names copy_from, copy_to

```
template<class U, class Flags> copy_from(const U* mem, Flags f);
```

```
template<class U, class Flags> copy_to(U* mem, Flags f);
```

The existing names are clear, and the Flags indicate the alignment of the provided pointer. If the list of valid flags be extended with conversions (see Should converting loads/stores be safer wrt. conversions question below), the name will become less accurate. Load/store pair will be more extendable in this case since the sense of exact copy is a little bit less strict there.

Recommendation: depends on Section 1.9

1.3 name of where function

"where" function syntax is counter intuitive to use and hard to comprehend what is meant to happen:.

```
        std::where(a > 0, b) = c;
```

operator?: overload, as proposed by P0917R3 (4) is a more familiar syntax and thus easier to read.

```
        b = (a > 0) ? c : b;
```

If that proposal is rejected or deferred then a similar behaviour can be implemented using a named function:

```
        x = std::select(a>0, valueIfTrue, valueIfFalse);
```

The false value can be value-initialised if not provided:

```
        y = std::select(a>0, valueIfTrue); // Use value-initialised otherwise
```

Or keep the proposed name but adjust it to be used as follows:

```
        x = std::where(a>0, valueIfTrue);
```

Recommendation: replace the where function with easier to read alternative, such as overloaded operator?:, or a new select/where function.

1.4 name of simd misleading

While "simd" name is somewhat more specific, as compared to what std::simd is capable to represent, at the same time it depicts the intended usage model in a very clear way.

Recommendation: simd is a good name

1.5 fixed_size vs. deduce_t

There is a wider question about the names used in the std::simd proposal which goes beyond whether some uses should be fixed_size or deduce_t. There are basically 5 types of ABI in the current std::simd paper and most of their description is not normative.

1.5.1 Scalar

```
stdx::simd<Type, stdx::simd_abi::scalar>;
```

Represents a single element.

Ideally the interface to std::simd should be such that it behaves in the same way as a scalar. It should be possible to write a templated algorithm which can be given a real scalar or a std::simd value and the algorithm will work unchanged between the two. It is understood that this may not be currently possible with the definition of std::simd (e.g., the behaviour of operator? breaks this), but it may be better to strive to fix that than introduce a scalar type.

1.5.2 Native

```
        stdx::simd<Type, stdx::simd_abi::native<Type>>;
```

Native should match the target architecture's default. For example, native<float> could be 8xFP32 on AVX2, or 16xFP32 on AVX-512.

1.5.3 Compatible

```
        stdx::simd<Type, stdx::simd_abi::compatible<Type>>;
```

The most efficient data-parallel execution for the element type T that ensures ABI compatibility between translation units on the target architecture.

In the case of Intel Architecture, the most compatible type would be equivalent to an SSE register, which is very out-of-date with current hardware. It would appear best to avoid this type entirely if high performance is required, which runs counter to the aims of std::simd.

1.5.4 Fixed size

```
    stdx::simd<Type, stdx::simd_abi::fixed_size<Type, N>>;
```

The intention of this type is to provide a safe way to move values across translation unit boundaries, but it is inefficient when passing it across translation unit boundaries, or to inlined functions. Its name is the most logical to use for a SIMD value of arbitrary size and it even has a convenient alias - `std::fixed_size_simd<T>` to make it easy to use, which is unfortunate since the most obvious and useful name for a variable sized SIMD is deliberately inefficient.

1.5.5 Deduce

```
    stdx::simd<Type, stdx::simd_abi::deduce_t<Type, N>>;
```

The most efficient available ABI for any given vector size, but less safe to pass between translation units. Here, less safe means it can be implemented in a way, that if it is passed to another translation unit, which does not support a given architecture, there will be a linkage error, but it is non-normative from spec perspective.

For performance code this type is the most obvious to use, especially when an application that uses it has control of its translation-unit borders. However, its name is not obvious, and it doesn't have an alias to make it easy to use.

1.5.6 Alternative naming

The ABIs do not match their names in a clear way, especially given some ABI aspects are hard to represent in spec, due to scope limitations. Another naming schema can be considered for better matching intended use cases for the ABIs:

TS name	New name	Reason
simd_abi::fixed_size	**simd_abi::compatible**	The main essence of the type is its compatibility on different sides of translation unit borders.
simd_abi::deduce_t	**simd_abi::fixed_size**	The main essence of the type is the best possible representation for the given size of elements.
simd_abi::compatible	Removed	The main essence of the type is the maximum width of vectorization, which is guaranteed to be supported for the given architecture. X86_64 started from SSE2, but trying to standardize the anchor to a 20-year-old target prevents code modernization, where it is possible. It is better to be represented with other means when needed, e.g. **simd_abi::deduce_t** (TS name) with some fixed size + proper compilation flags.
simd_abi::native	**simd_abi::native**	No change.
simd_abi::scalar	**simd_abi::scalar**	No change.

Recommendation:

- ABIs can be renamed to better match their essence
- deduce_t (TS name) shall be consistently used, including simd_cast

1.6 allowed template arguments for (static_) simd_cast

simd_cast allows both scalar T and vector std::simd<T,Abi> to be a template argument for casting.

```
std::fixed_size_simd<char, 32> a{};
auto b1 = std::simd_cast< std::fixed_size_simd<int, 32> >( a );
auto b2 = std::simd_cast< int >( a );
```

While using vector type looks more pedantic, it is also significantly more verbose.

It is also worth noting, that simd_cast is a more strict version of static_simd_cast:

simd_cast is allowed only when "every possible value of type U can be represented with type To" , but it is unclear how this difference is connected with "static_" prefix.

It should also be possible to convert a simd_mask from one type to another. After conversion the new mask has the same number of bits and each bit has the same sense as the original, but it is in a form which can be applied to the new type.

Recommendation: keep both versions of arguments

Recommendation: make it clear that simd_mask also allows simd_cast if that is the intention, or add it if not.

1.7 consider more implicit conversions

The status quo of the TS:

- simd_mask is implicitly convertible to simd_mask if
  - both A0 and A1 are fixed_size (same N).
- simd is implicitly convertible to simd if
  - both A0 and A1 are fixed_size (same N), and
  - converting T0 to T1 preserves the value of all possible values of T0.

Portability is the problem, which this decision is targeted to address:

- std::fixed_size_simd<int, 4> x = std::simd(17);

Right-hand side has default value of abi - simd_abi::compatible, thus this code will work only on platforms, where compatible abi results in 4-elements simd (if conversion with equal N size be allowed).

At the same time, this line is portable:

- std::fixed_size_simd<int, 4> x = std::simd<int, std::deduce_t<int, 4>>(17);

But it is forbidden as well, because deduce_t is an alias and is indistinguishable from the previous example.

If the result of deduce_t, was an implementation defined type and not an alias, the problem will be solved.

Recommendation: consider making deduce_t a dedicated implementation defined type, and not an alias.

1.8 no default for the load/store flags

Current load/store operations require always passing alignment flag and have no default, which makes it verbose.

TS syntax:

std::simd<float> v(addr, std::vector_aligned);

v.copy_from(addr + 1, std::element_aligned);

v.copy_to(dest, std::element_aligned);

At the same time, C++20 defined another approach:

~~std::simd v(std::assume_aligned<std::memory_alignment_v<std::simd>>(addr));~~

std::simd<float> v(std::assume_aligned< 64 >(addr));

```
v.copy_from(addr + 1);

v.copy_to(dest);
```

C++20 syntax is misused in the first line (taken from P1915R0) – alignment of `addr` is an input property of memory and should not be taken from std::simd desired alignment. When we get alignment from somewhere else as a value, this syntax looks reasonable.

Given this C++20 feature was designed to cover exactly such cases, it is better apply it here as well.

Recommendation: apply C++20 approach and drop alignment flag.

1.9 should converting loads/stores be safer wrt. conversions

Implicit conversions on broadcast and memory loads stores are inconsistent in allowed conversions.

Broadcasting constructor has multiple restrictions for possible input type:

```
simd<float> {(double)(v)}; // shall be error, double -> float conversion is not value-preserving

simd<float> {(short)(v)}; // shall be good, short -> float is value-preserving, if sizeof(short)==16
```

This is based on the following restrictions for From type (From = remove_cv_t<remove_reference_t<U>>):

- From is a vectorizable type and every possibly value of From can be represented with type value_type, or
- From is not an arithmetic type and is implicitly convertible to value_type, or
- From is int, or
- From is unsigned int and value_type is an unsigned integral type.

Memory source constructor has simpler restriction for possible input type:

```
simd<float> {(double*)mem, flags::element_aligned}; // converting load, compiles (inconsistent)

simd<float> {(short*)mem, flags::element_aligned}; // value-preserving conversion
```

The only restriction on memory pointer type:

- U is a vectorizable type.

Both are inconsistent with narrowing conversion definition:

```
float f {(short)(v)}; // error, integer type to float is narrowing, unless v is constant expression
```

Narrowing conversion definition:

- from a floating-point type to an integer type, or
- from long double to double or float, or from double to float
  - except where the source is a constant expression and the actual value after conversion is within the range of values that can be represented (even if it cannot be represented exactly), or
- from an integer type or unscoped enumeration type to a floating-point type
  - except where the source is a constant expression and the actual value after conversion will fit into the target type and will produce the original value when converted back to the original type, or
- from an integer type or unscoped enumeration type to an integer type that cannot represent all the values of the original type
  - except where the source is a constant expression whose value after integral promotions will fit into the target type.

Let us not introduce one more definition with the same purpose to narrowing conversion definition for broadcasting constructor. Restriction for broadcasting constructor may look similar to this one:

- From and value_type are arithmetic types, and conversion from From to value_type is not a narrowing conversion, or
- From or value_type is not an arithmetic type and From is implicitly convertible to value_type.

In alignment with the direction to make default behavior safer, memory source constructor wording can have same restrictions.

This would disallow direct representation of such instructions like VCVTPS2PH or VCVTPD2PS, which allow memory source operand and are narrowing conversion by definition.

Recommendation: align broadcasting and memory source constructors type restrictions with narrowing conversion definition.

1.10 relation operators

Tim Shen proposed to replace relation operators with free functions (5), and optionally rework operators to return bool.

```
array<simd<float>, 16> arr;

std::sort(arr.begin(), arr.end());
```

This code will fail at compile time because comparison operators do not return bool and any attempt to use comparison result as a bool will be doomed.

At the same time this use case is much more readable, as compared to free function alternative:

```
where(a>0, b) = 0;
```

Alternative:

```
where(gt(a, 0), b) = 0;
```

The value of std::simd_mask result returned from relation operators overweight some inconsistencies, given misuse will not be silently taken but fail during compilation.

Recommendation: vote for keeping current status quo.

Additional questions

1.11 C++20 support

There are many features in C++20 which could be used in xvec. The main one is concepts, which could be used to simplify and clarify the implementation behaviour (instead of enable_if), and to handle the checking of various flags/tags (e.g., add concepts to specify memory alignment flags).

Other features which could be supported include modules, consteval, constinit, and so on.

1.12 Is the default ABI tag the correct default?

simd<T> uses the simd_abi::compatible<T> ABI tag but native_simd<T> is another possible option.

simd_abi::compatible<T> will become the least supported architecture by chosen compiler and/or standard library. While for the given project, one would need to choose the least supported architecture within the project support matrix.

Using simd_abi::compatible<T> default can make the default choice of architecture overly pessimistic. Using simd_abi::native<T> should safe enough as the default option, because:

- It is supported by this translation unit, because of chosen compiler flags;
- It may have linker error, when passing argument to another translation unit,

But the solution to that is using fixed_size abi, and not the compatible abi.

See fixed_size vs. deduce_t question above, for more details on ABI difference description.

Recommendation: default into simd_abi::native<T> as a better default option.

1.13 Use constexpr everywhere possible

After adding std::is_constant_evaluated() to C++20, the necessary facilities to be certain that a constexpr implementation is possible are there.

Recommendation: Allowing as much compile-time value generation as possible would be a good thing.

1.14 Specify triviality of simd and simd_mask

Whether simd<T> or simd_mask<T> is trivial is currently a QoI issue.

There is a common practice of reinterpreting of contents of a register into a different type. Guaranteeing triviality enables reliable use of std::bit_cast for this purpose.

Current simd spec restricts T to only trivial types:

- all cv-unqualified arithmetic types other than bool.

This should allow making simd and simd_mask trivial as well.

Additional potential use case for std::simd is std::complex, which is not allowed so far, but is described as a proposal in the next major section.

Recommendation: prescribe simd and simd_mask to be trivial in all cases.

Proposals for additional API functions and features

Intel has long experience of using building hardware with SIMD instruction sets, developing its own libraries to use those instructions and writing SIMD-parallel software both with those libraries and with explicit intrinsics. We would like to propose a number of extra features and APIs which would enhance std::simd and make certain important tasks and operations easier.

We have our own reference implementation of std::simd which includes the features described here. All the extra features have generic implementations or could map to target-specific intrinsics/instructions where a vendor makes them available.

1.15 iota

The C++ iota function generates a sequence of ascending values and writes them into a container. The Intel implementation of std::simd provides a way of achieving the same effect for a simd value:

```
static constexpr simd<_T, _Abi> simd::iota()
```

The iota function provides an alternative way to building known values without having to use the generator described in the previous sub-section. For example:

```
constexpr multipleOf3 = native_simd<float>::iota() * 3;
```

This mechanism allows the simd implementation itself to manipulate the values in simd form, instead of having to manipulate them in scalar form using an index. It would work best if constexpr were provided throughout the simd implementation so that there is no restriction on creating compile-time simd constants.

1.16 Direct resizing

Given a simd/simd_mask value it can be useful to directly resize it:

```
template<size_t N, typename T, typename V> resize_simd_t<N, V> resize(V v);
```

This returns a simd/simd_mask value of the new size. When the new simd has fewer elements than the original it will be truncated. When the new simd has more elements the new elements will be value-initialized. Resizing to 0 is not permitted and resizing to the same size returns a copy of the original.

(side-note: Intel's implementation also provides a variant of this which resizes to the smallest default target type which is big enough to store the simd value, which makes interfacing with intrinsics easier).

1.17 Insert and extract sub-simd values

std::simd provides ways to decompose a simd value into a set of smaller components using split, and also to glue small simds together to form bigger simd values using concat. However, it can often be useful to directly insert and extract smaller simd values into and out of larger simd values without requiring every component of such an operation to be explicitly managed. For example, given a small simd value, it can be inserted into a larger simd value as follows:

```
const auto n = insert<4>(parentSimd, childSimd);
```

This creates a new simd value which is a copy of `parent', but with the values from child inserted at positions [4, 4 + child.size()).

Similarly:

```
        const auto child = extract<2, 5>(parentSimd);
```

will extract a new simd of size 3 (i.e., 5 – 2), starting from position 2 in the parent simd

Both insert and extract require the child simd to be a proper sub-component of the parent (i.e., it must be smaller, and completely contained with the parent's elements).

Note that the presence of insert and extract functions makes other operations like concat and split easy to implement, as they are effectively repeated applications of these functions, and they also allow easier interfacing to intrinsics which are normally some fixed size.

1.18 <bit> support

C++20 includes support for a number of extra bit-wise operations, including byteswap, bit_[ceil/floor], rot[lr], and so on. We propose that those operations should also be valid in simd where the type is appropriate.

1.19 Floating-point fused-multiply-add

The std::simd proposal requires that all overloads in cmath should be supported by std::simd. This will ensure that the fma function must be implemented, which means that the programmer can access the higher precision guaranteed by IEEE in not quantizing the result of the multiply before being added.

In simd it can often be useful to perform alternate add/subtract or subtract/add on adjacent simd elements, and there is no easy way to specify this using other constructs in std::simd, so we also propose that the following functions are provided:

`fmaddsub`	Add and subtract alternate elements
`fmsubadd`	Subtract and add alternate elements

1.20 Bit bashing operations

The following bit-bashing types of operations have been found to be frequently useful on unsigned integer simd values and are not easily expressed using any of the existing std::simd API features:

`bitreverse`	Bit-reverse each element
`bitblend`	Blend two inputs together on a bit-by-bit basis, using a third simd object as a selector (i.e., for each bit, if set(bit) use srcA(bit) else srcB(bit))

Note that byteswap (which reverse the order of all bytes in an element) would be provided by default if <bit> is supported as described in Section 1.18.

1.21 bitset representation of simd_mask

A simd_mask is designed to be an efficient abstraction of a set of boolean predicate values, where each mask element can be used to control a property of the equivalent element in one or more other simd values. A simd_mask value could also be represented as a std::bitset, and we propose that conversions between the two should be possible:

`simd_mask_value.to_bitset()`	Return a std::bitset with the same number of elements as the simd_mask and where each element has the same sense as its equivalent mask element.
`simd_mask(someBitset)`	Construct a simd_mask with the same number of elements as the given bitset, where each bit in the mask has the same sense as its equivalent bitset element.

Note also that since std::bitset already has an API which allows query operations like count, any, none, or all it might not be necessary to support these directly in the simd_mask itself. They could be handled by, for example:

```
        simd_mask_value.to_bitset().count()
```

instead of:

```
        simd_mask.popcount()
```

1.22 Support for interleaved complex values

Scientific, engineering, media and signal processing workloads frequently require the complex-valued operations. A common format for such data is where the real and imaginary data of the complex values are interleaved together. For example, a 2-element array of complex values could be expressed in C++ as:

```
std::array<std::complex<float>, 3>
```

and this would be laid out in its storage as

```
real(0), imag(0), real(1), imag(1), real(2), imag(2)
```

Such is the value of interleaved complex value that Intel's most recent instruction set has native hardware support for both scalar and simd values (6). Other processor vendors also provide native support for complex interleaved simd values.

We propose that std::simd should allow complex data simd values to be created, as exemplified by the following:

```
        std::simd<std::complex<float>, std::fixed_size<5>>

        std::fixed_size_simd<std::complex<_Float16>, 9>
```

The interface should allow all standard simd API functions to work as expected:

- All operators (+-/*) and their derivations work on the numeric values as expected. For multiply and division operations a suitable complex equivalent is used.
- All appropriate numeric functions would work on a per-complex element basis. For numeric functions whose definition would be different for complex (e.g., abs, sin, cos), the equivalent complex-valued operation would be performed.
- Elements in a simd_mask refer to complex valued elements. Indexing the simd leads to complex values (e.g., cmplx_simd[2] would give a std::complex<_T> value).
- Resizing, casting, converting, splitting and concatenating all work on complex elements
- Etc.

There are a number of functions which operate on complex values but for which there is no equivalent operation in the existing std::simd proposal. We propose that overloads should be provided for std::simd complex values to match the behaviour of of std::complex scalar values. For example:

`s.real()`	Returns a new std::simd with the same number of real-valued elements
`s.imag(v)`	Set the imaginary values to those specified in v. v is the same size as the simd, but real-valued (i.e., T, not std::complex<T>).
`norm(s)`	Return a simd of real values giving the squared magnitude
`conj(s)`	Return the conjugate of the complex simd (negated imaginary)

Note that std::simd already requires that cmath overloads are provided for abs, sin, cos, exp, log and more, so these would already operate in their normal mathematical sense on complex numbers

1.23 Permutation

While the primary motivation for data-parallel libraries is vertical (map) operations, reordering data within or across simd values is common, and fast ways to achieve these operations are provided in many instruction sets. The current std::simd proposal has no API for permuting data. Document P0918 (7) does provide some proposals for permute operations but they are relatively limited in scope. After experimentation with our own LLVM implementation we have defined a permutation API that covers all the common use cases, while being easy to implement in compilers and in hardware.

A common use case is to generate a set of permutation indexes in one simd value, and then apply them to another simd value using the function similar to:

```
template<typename _V, typename _Idx>
constexpr simd<typename _V::value_type, _Idx::size()>
permute(const _V& values, const _Idx& indexes);
```

This generates a new simd which has the same element type as the input simd, and the same number of elements as the index simd. Each output element [0..idx) comes from input[idx[i]]. All index elements must be in the range [0..value::size()). This operation maps efficiently to the permute or swizzle instructions found in many ISAs.

Instructions sets may allow indexing from multiple sources (e.g., permutex2var in AVX-512). Rather than defining a new function to handle multi-source permutes, the same effect can be achieved as follows:

```
permute(concat(simd0, simd1, simd2), indexes);
```

The compiler will be able to elide the concatenation and convert this into an efficient instruction sequence using multi-source permute instructions.

As with the generate function already found in the std::simd proposal, it can be useful to specify a permutation function which can be applied to a simd of unknown length. For example, to extract every third element of a simd of indeterminate length the following is permitted:

```
const auto t = permute(simd_value, [=N](auto idx) { return i + 3 % N); });
```

This will generate a new simd of the same size, where every value x[i] is permuted to a new position x[fn(i)]. The permute function is constexpr so the required permutation can be completely determined at compiler time and the compiler will substitute the most efficient way to implement the desired effect.

It may be necessary to change the size of the output vector (and consequently how many times the index is computed), and this can be done using a template parameter specifying the new simd size:

```
auto even = [](size_t i) -> size_t { return i * 2; };

const auto t = permute<4>(simd_value, even);
```

this generates a simd containing the first 4 even values of the input simd.

The compile-time permutation function can be used to construct any permutation. There is no need to expose ISA-specific instructions (e.g., pack, unpack, interleave, align) since the compiler will be able to determine for itself which target instruction best matches the desired reordering.

The permutes described above should also apply to simd_masks, where individual bits can be reordered according to either a fixed index simd, or a permutation function.

In addition to permuting from a simd value source it should also be possible to permute from a memory region which is indicated by a pointer to an element. For example:

```
permute(ptr_to_memory,  simdIndexes);
```

This will return a simd which has the same type as *ptr_to_memory, and the same number of elements as simdIndexes. This can be converted into a gather instruction where the target supports it, or the compiler can synthesize it where such an instruction does not exist. A memory-based permute may also accept a generator function instead of an explicit index simd.

Question: Should a memory based permute be permitted for simd_mask?

Note that, for convenience, it may be desirable to provide some common generator aliases which pre-define operations such as strided reads, std::reverse (8), get_odd, get_even, and so on.

I. References

1. **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..**

2. Kretz, Matthias. P1915R0: Expected Feedback from simd in the Parallelism TS 2. ISO/IEC C++ Standards Committee Paper. [Online] 2019. http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2019/p1915r0.pdf.

3. Intel Corporation. Intel® 64 and IA-32 Architectures Software Developer's Manual. [Online] https://software.intel.com/content/www/us/en/develop/articles/intel-sdm.html.

4. Kretz, Matthias. P0917R3: Making operator?: overloadable, 2019. url:. [Online] http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2019/p0917r3.pdf.

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

6. Towner, Daniel. Intel® AVX-512 - FP16 Instruction Set for Intel® Xeon® Processor Based Products Technology Guide. [Online] April 2022.

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

8. https://en.cppreference.com/w/cpp/algorithm/rotate. [Online]