intel

P2663R0:
Proposal to support interleaved complex values in std::simd

Authors

Daniel Towner
(daniel.towner@intel.com)

Introduction

ISO/IEC 19570:2018 (1) 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. One of the extra features we suggested was the ability to work with interleaved complex values since it is now becoming common for processor instruction sets to include native support for these operations (e.g., Intel AVX512-FP16, ARM Neon) and we think that allowing std::simd to directly access these features is advantageous.

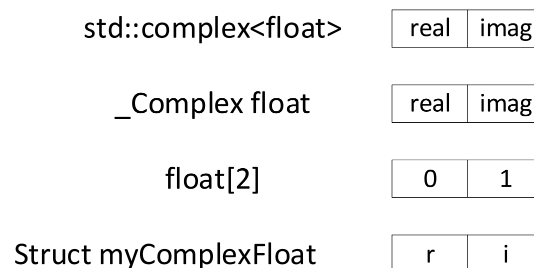
This document gives more details about the motivation for supporting complex-values, the different storage and implementation alternatives, and some suggestions for a suitable API.

Introduction to complex-values

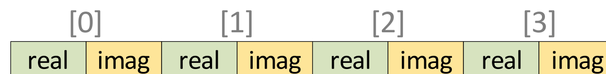
Complex-valued mathematics is widely used in scientific computing and many types of engineering, with applications including physical simulations, wireless signal processing, media processing, game physics, and much more besides. C++ already supports complex-valued operations using the `std::complex` template available in the `<complex>` header file and C supports complex values by using the `_Complex` keyword. Both C and C++ formally support only complex values which are of type float, double or long double, though in practice compilers may support integer or half-precision complex values too. Such is the importance of these interleaved complex values that some mainstream processor vendors now provide native hardware support for complex-value SIMD instructions (e.g., Intel AVX512-FP16, ARM Neon v8.3).

Complex-valued data layout

Any complex value is represented as a pair of values, one for the real component and one for the imaginary component. In C and C++ a complex value is a pair of two values of the appropriate data type, allocated to a single array-like unit. This is illustrated in the figure below. This storage layout is also used in other languages too, allowing for easy interchange between software written in different languages, or with comparable user-defined structures:



When many complex values are stored together, such as in a C-style array, a C++ std::array or a C++ std::vector, then the real and imaginary elements end up being interleaved in memory:



Many applications, libraries, programming languages and interfaces between diverse software components will store data in this interleaved format, and it is widely regarded as the default layout for blocks of complex-valued data. To improve the compute efficiency of this data format both Intel and ARM have introduced native hardware support to allow efficient SIMD operations to be performed in this data format without having to resort to reformatting the data into a more SIMD-amenable form. Where the underlying target hardware does not have native support for interleaved complex values it is straight-forward to synthesize a sequence of operations which mimic the effect.

Given the commonality of interleaved complex-values as a data exchange and compute format, we propose that std::simd should be able to directly represent this simd format in a form such as:

```
std::simd<std::complex<float>, ABI> myIntrlvValue
```

Such a simd format can trivially move interleaved complex data to and from memory represented as C, or C++ style arrays without requiring any special operations, and operator overloads can map operations such as multiply directly to the underlying hardware support where it is available, or to use synthesized code sequences where it is not.

Note that there is an alternative format for complex data called separated storage. In this form the real and imaginary data blocks are stored in entirely different regions of memory or registers. Under some conditions this format can be more efficient, but it requires separate operations to convert to and from the more common interleaved storage format. In future it may be desirable to allow std::simd to support such a format, but it will not be considered further in this document.

Proposal to allow std::complex as a std::simd element type

We propose that std::simd should allow an interleaved data-parallel complex storage format to be modelled by allowing the base element in std::simd to be a complex type, as exemplified by the following:

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

std::fixed_size_simd<std::complex<double>, 9>

std::fixed_size_simd<_Complex float, 9>   // Note: C-style _Complex extension could be valid too?
```

In all these cases the fundamental element type will be a suitable complex value, and the individual components of each complex element will be worked on as a single unit, according to the rules of complex arithmetic where appropriate. This includes:

- All numeric operators (e.g., +/-*) and their derivatives (e.g., assignment operators) will operate as per their standard type definition. For example, the multiply operator will invoke a complex multiply for each complex element.
- Any numeric operation will apply the operation on a per-element basis, generating a simd object of the same size as the input, but modifying the element type to be the appropriate return type. For example, when applied to a simd<complex<T>>:
  - exp, sin, log, tan, cos and so on will generate a simd<complex<T>>
  - abs or norm will generate a simd<T> (i.e., real-valued simd equivalent).

  Note that the mathematical behaviour will also be applied appropriately (e.g., if a real or imaginary component is a NaN, then the complete operation for a complex operation will also be NaN).
- The simd_mask associated with a simd<complex> will have each individual mask bit refer to a complete complex element, not to sub-components of the complex elements. When the simd_mask is backed by a compact representation (e.g., AVX-512), then each individual bit will refer to a complete complex element. When the simd_mask is backed by an element mask (i.e., multiple bits filling a container of the same size as the simd element) then the size of each bit element will be twice the size of the underlying complex value type (e.g., a simd_mask of complex<_Float16> would be equivalent to something like simd<uint32_t>, since sizeof(uint32_t) == 2 * sizeof(_Float16).
- Operations or functions which are invalid for complex values will be removed. For example, this includes relational operators (i.e., <, <=, etc are not defined for complex values) and those derived from them such, as min or max.
- Indexing a simd<complex> results in a std::complex<> being accessed.
- Any sort of permute, resize, split, concat, insert, or extract operation works on complete complex elements as though they are atomic units.
- Reduction operations apply at the level of complex values (this follows from the previous bullet), provided the mathematical operation is valid (e.g., reduce-max would not be available).

There are some special cases of the above operations which can be made more efficient, and we should consider providing additional overloads to handle them. To begin with we should follow the convention of std::complex and provide overloads for binary operators which take a real-value. For example, the multiply operator would come in three forms:

```
simd<std::complex<T>> operator*( simd<std::complex<T>>, simd<std::complex<T>>); // Complex * complex
simd<std::complex<T>> operator*( simd<std::complex<T>>, simd<T>);              // Complex * real
simd<std::complex<T>> operator*( simd<T>, simd<std::complex<T>>);              // Real * complex
```

We propose that it should also be permitted to overload by a plain T since that can generate more efficient code. As an example, consider the code generated for multiplying a simd<complex<_Float16>> by a plain _Float16 real-valued scalar. With the real-valued overloads above, the real-valued scalar would be turned into a simd<complex<_Float16>> through the broadcast constructor (i.e., a simd value where the real elements are the duplicated scalar, and the imaginary elements are zero). That value is then multiplied by the other complex value. The generated code (with oneAPI 2022.0) looks like this:

```
vmovsh  %xmm1, %xmm2, %xmm1             # Move the 16-bit real into a 32-bit element with a
                                        # zero in the imaginary position
vbroadcastss    %xmm1, %zmm2            # Broadcast the now-complex scalar across the simd
vfmulcph        %zmm2, %zmm0, %zmm1     # Perform a complex multiply
```

The issue here is that although multiplying by a real-valued scalar is equivalent to simply multiplying every numeric sub-element (i.e., both reals and imaginaries) by the same scalar value, the code above must convert the real value into a complex value and invoke a more expensive complex multiply. Instead, consider what happens when real-valued overloads are provided:

```
simd<std::complex<T>> operator*( simd<std::complex<T>>, simd<std::complex<T>>); // Complex * complex
simd<std::complex<T>> operator*( simd<std::complex<T>>, T);           // Complex * real-scalar
simd<std::complex<T>> operator*( T, simd<std::complex<T>>);           // Real-scalar * complex
```

Recompiling the code from above with these operators enabled generates the following code:

```
vbroadcastw    %xmm1, %zmm1            # Broadcast 16-bits to every simd element
vmulph  %zmm0, %zmm1, %zmm0            # Perform a real-valued multiply
```

This code sequence will have approximately half the cycle latencyt as the other sequence, partly because it has fewer instructions in a dependent chain, but primarily because it allows a real-valued multiply to be used instead of the more expensive complex-valued multiply. Where appropriate, overloads by real-valued-scalar should be allowed to improve performance in these special cases.

One of the aims of std::simd is to make it possible to write generic or templated code which can use either a scalar type or a data-parallel type interchangeably. To this end, std::simd provides overloads of many of the standard functions which operate in data-parallel mode. The same should apply to a simd of complex values; it should be possible to write an algorithm which uses either std::complex<T> or simd<std::complex<T>> with only a type replacement, not with major API changes.

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 be provided for std::simd<std::complex> values to match the behavior of std::complex values. For example, given a std::simd<std::complex<T>,ABI>:

> x.real(), x.imag() would return a std::simd<T, ABI> (i.e., real-valued simd with the same number of elements).
>
> s.imag(v), s.real(x) would accept a simd<T,ABI> (i.e., real-valued components) and set the real or imaginary elements appropriately.
>
> conj() would return the conjugate of the complex simd (negated imaginary)

In C the _Complex keyword is allowed if <complex.h> is included. This feature is not permitted in C++, but some compilers do choose to allow this extension anyway. Those compilers may choose to overload the functions from <complex.h> to work on std::simd<complex>types too. One advantage of <complex.h> is that other types (e.g., integer, _Float16) are also allowed.

On processors with native support for complex values, any of the operations described in this document can be replaced by appropriate hardware instructions. On machines without native support an equivalent result should be generated through the appropriate synthesized instruction sequence.

Performance varies by use, configuration and other factors. Learn more at www.Intel.com/PerformanceIndex.

Performance results are based on testing as of dates shown in configurations and may not reflect all publicly available updates. See backup for configuration details. No product or component can be absolutely secure.

Intel disclaims all express and implied warranties, including without limitation, the implied warranties of merchantability, fitness for a particular purpose, and non-infringement, as well as any warranty arising from course of performance, course of dealing, or usage in trade.

Intel technologies may require enabled hardware, software or service activation.

Intel does not control or audit third-party data. You should consult other sources to evaluate accuracy.

The products described may contain design defects or errors known as errata which may cause the product to deviate from published specifications. Current characterized errata are available on request.

© Intel Corporation. Intel, the Intel logo, and other Intel marks are trademarks of Intel Corporation or its subsidiaries. Other names and brands may be claimed as the property of others.