
xsimd

Release

Jan 17, 2018

INSTALLATION

1	Licensing	3
1.1	Installation	3
1.2	Basic usage	3
1.3	Writing vectorized code	4
1.4	Instruction set macros	7
1.5	Wrapper types	9
1.6	Data transfer	24
1.7	Mathematical functions	27
1.8	Aligned memory allocator	38

C++ wrappers for SIMD intrinsics.

SIMD (Single Instruction, Multiple Data) is a feature of microprocessors that has been available for many years. SIMD instructions perform a single operation on a batch of values at once, and thus provide a way to significantly accelerate code execution. However, these instructions differ between microprocessor vendors and compilers.

xsimd provides a unified means for using these features for library authors. Namely, it enables manipulation of batches of numbers with the same arithmetic operators as for single values. It also provides accelerated implementation of common mathematical functions operating on batches.

You can find out more about this implementation of C++ wrappers for SIMD intrinsics at the [The C++ Scientist](#). The mathematical functions are a lightweight implementation of the algorithms also used in [boost.SIMD](#).

xsimd requires a C++14 compliant compiler. The following C++ compilers are supported:

Compiler	Version
Microsoft Visual Studio	MSVC 2015 update 2 and above
g++	4.9 and above
clang	3.7 and above

The following SIMD instruction set extensions are supported:

Architecture	Instruction set extensions
x86	SSE2, SSE3, SSSE3, SSE4.1, SSE4.2, AVX, FMA3, AVX2
x86 AMD	same as above + SSE4A, FMA4, XOP

We use a shared copyright model that enables all contributors to maintain the copyright on their contributions. This software is licensed under the BSD-3-Clause license. See the LICENSE file for details.

1.1 Installation

xsimd is a header-only library. We provide a package for the conda package manager.

```
conda install -c conda-forge xsimd
```

Or you can directly install it from the sources:

```
cmake -D CMAKE_INSTALL_PREFIX=your_install_prefix  
make install
```

1.2 Basic usage

1.2.1 Explicit use of an instruction set extension

Here is an example that computes the mean of two sets of 4 double floating point values, assuming AVX extension is supported:

```
#include <iostream>  
#include "xsimd/xsimd.hpp"  
  
namespace xs = xsimd;  
  
int main(int argc, char* argv[])  
{  
    xs::batch<double, 4> a(1.5, 2.5, 3.5, 4.5);
```

```

xs::batch<double, 4> b(2.5, 3.5, 4.5, 5.5);
auto mean = (a + b) / 2;
std::cout << mean << std::endl;
return 0;
}

```

This example outputs:

```
(2.0, 3.0, 4.0, 5.0)
```

1.2.2 Auto detection of the instruction set extension to be used

The same computation operating on vectors and using the most performant instruction set available:

```

#include <cstdlib>
#include <vector>
#include "xsimd/xsimd.hpp"

namespace xs = xsimd;
using vector_type = std::vector<double, xsimd::aligned_allocator<double, XSIMD_
↳DEFAULT_ALIGNMENT>>;

void mean(const vector_type& a, const vector_type& b, vector_type& res)
{
    std::size_t size = a.size();
    constexpr std::size_t simd_size = xsimd::simd_type<double>::size;
    std::size_t vec_size = size - size % simd_size;

    for(std::size_t i = 0; i < vec_size; i += simd_size)
    {
        auto ba = xs::load_aligned(&a[i]);
        auto bb = xs::load_aligned(&b[i]);
        auto bres = (ba + bb) / 2;
        bres.store_aligned(&res[i]);
    }
    for(std::size_t i = vec_size; i < size; ++i)
    {
        res[i] = (a[i] + b[i]) / 2;
    }
}

```

1.3 Writing vectorized code

Assume that we have a simple function that computes the mean of two vectors, something like:

```

#include <cstdlib>
#include <vector>

void mean(const std::vector<double>& a, const std::vector<double>& b, std::vector
↳<double>& res)
{
    std::size_t size = res.size();
    for(std::size_t i = 0; i < size; ++i)
    {

```



```

    res[i] = (a[i] + b[i]) / 2;
}
}

```

How can we use *xsimd* to take advantage of vectorization ?

1.3.1 Explicit use of an instruction set

xsimd provides the template class `batch<T, N>` where `N` is the number of scalar values of type `T` involved in SIMD instructions. If you know which instruction set is available on your machine, you can directly use the corresponding specialization of `batch`. For instance, assuming the AVX instruction set is available, the previous code can be vectorized the following way:

```

#include <cstdint>
#include <vector>
#include "xsimd/xsimd.hpp"

void mean(const std::vector<double>& a, const std::vector<double>& b, std::vector
↳<double>& res)
{
    using b_type = xsimd::batch<double, 4>;
    std::size_t inc = b_type::size;
    std::size_t size = res.size();
    // size for which the vectorization is possible
    std::size_t vec_size = size - size % inc;
    for(std::size_t i = 0; i < vec_size; i +=inc)
    {
        b_type avec(&a[i]);
        b_type bvec(&b[i]);
        b_type rvec = (avec + bvec) / 2;
        rvec.store_unaligned(&res[i]);
    }
    // Remaining part that cannot be vectorize
    for(std::size_t i = vec_size; i < size; ++i)
    {
        res[i] = (a[i] + b[i]) / 2;
    }
}

```

However, if you want to write code that is portable, you cannot rely on the use of `batch<double, 4>`. Indeed this won't compile on a CPU where only SSE2 instruction set is available for instance. To solve this, *xsimd* provides an auto-detection mechanism so you can use the most performant SIMD instruction set available on your hardware.

1.3.2 Auto detecting the instruction set

Using the auto detection mechanism does not require a lot of change:

```

#include <cstdint>
#include <vector>
#include "xsimd/xsimd.hpp"

void mean(const std::vector<double>& a, const std::vector<double>& b, std::vector
↳<double>& res)
{
    using b_type = xsimd::simd_type<double>;

```

```

std::size_t inc = b_type::size;
std::size_t size = res.size();
// size for which the vectorization is possible
std::size_t vec_size = size - size % inc;
for(std::size_t i = 0; i < vec_size; i += inc)
{
    b_type avec = xsimd::load_unaligned(&a[i]);
    b_type bvec = xsimd::load_unaligned(&b[i]);
    b_type rvec = (avec + bvec) / 2;
    xsimd::store_unaligned(&res[i], rvec);
    // or rvec.store_unaligned(&res[i]);
}
// Remaining part that cannot be vectorize
for(std::size_t i = vec_size; i < size; ++i)
{
    res[i] = (a[i] + b[i]) / 2;
}
}

```

1.3.3 Aligned vs unaligned memory

In the previous example, you may have noticed the `load_unaligned/store_unaligned` functions. These are meant for loading values from contiguous dynamically allocated memory into SIMD registers and reciprocally. When dealing with memory transfer operations, some instructions sets required the memory to be aligned by a given amount, others can handle both aligned and unaligned modes. In that latter case, operating on aligned memory is always faster than operating on unaligned memory.

`xsimd` provides an aligned memory allocator which follows the standard requirements, so it can be used with STL containers. Let's change the previous code so it can take advantage of this allocator:

```

#include <cstddef>
#include <vector>
#include "xsimd/xsimd.hpp"

using vector_type = std::vector<double, XSIMD_DEFAULT_ALLOCATOR(double)>;
void mean(const vector_type& a, const vector_type& b, vector_type& res)
{
    using b_type = xsimd::simd_type<double>;
    std::size_t inc = b_type::size;
    std::size_t size = res.size();
    // size for which the vectorization is possible
    std::size_t vec_size = size - size % inc;
    for(std::size_t i = 0; i < vec_size; i += inc)
    {
        b_type avec = xsimd::load_aligned(&a[i]);
        b_type bvec = xsimd::load_aligned(&b[i]);
        b_type rvec = (avec + bvec) / 2;
        xsimd::store_unaligned(&res[i], rvec);
        // or rvec.store_unaligned(&res[i]);
    }
    // Remaining part that cannot be vectorize
    for(std::size_t i = vec_size; i < size; ++i)
    {
        res[i] = (a[i] + b[i]) / 2;
    }
}

```

1.3.4 Memory alignment and tag dispatching

You may need to write code that can operate on any type of vectors or arrays, not only the STL ones. In that case, you cannot make assumption on the memory alignment of the container. *xsimd* provides a tag dispatching mechanism that allows you to easily write such a generic code:

```
#include <cstdint>
#include <vector>
#include "xsimd/xsimd.hpp"

template <class C, class Tag>
void mean(const C& a, const C& b, C& res)
{
    using b_type = xsimd::simd_type<double>;
    std::size_t inc = b_type::size;
    std::size_t size = res.size();
    // size for which the vectorization is possible
    std::size_t vec_size = size - size % inc;
    for(std::size_t i = 0; i < vec_size; i += inc)
    {
        b_type avec = xsimd::load_simd(&a[i], Tag());
        b_type bvec = xsimd::load_simd(&b[i], Tag());
        b_type rvec = (avec + bvec) / 2;
        xsimd::store_simd(&res[i], rvec, Tag());
    }
    // Remaining part that cannot be vectorize
    for(std::size_t i = vec_size; i < size; ++i)
    {
        res[i] = (a[i] + b[i]) / 2;
    }
}
```

Here, the Tag template parameter can be `xsimd::aligned_mode` or `xsimd::unaligned_mode`. Assuming the existence of a `get_alignment_tag` metafunction in the code, the previous code can be invoked this way:

```
mean<get_alignment_tag<decltype(a)>>(a, b, res);
```

1.4 Instruction set macros

xsimd defines different macros depending on the symbols defined by the compiler options.

1.4.1 x86 architecture

If one of the following symbols is detected, `XSIMD_X86_INSTR_SET` is set to the corresponding version and `XSIMD_X86_INSTR_SET_AVAILABLE` is defined.

Symbol	Version
__SSE__	XSIMD_X86_SSE_VERSION
__M_IX86_FP >= 1	XSIMD_X86_SSE_VERSION
__SSE2__	XSIMD_X86_SSE2_VERSION
__M_X64	XSIMD_X86_SSE2_VERSION
__M_IX86_FP >= 2	XSIMD_X86_SSE2_VERSION
__SSE3__	XSIMD_X86_SSE3_VERSION
__SSSE3__	XSIMD_X86_SSSE3_VERSION
__SSE4_1__	XSIMD_X86_SSE4_1_VERSION
__SSE4_2__	XSIMD_X86_SSE4_2_VERSION
__AVX__	XSIMD_X86_AVX_VERSION
__FMA__	XSIMD_X86_FMA3_VERSION
__AVX2__	XSIMD_X86_AVX2_VERSION

1.4.2 x86_AMD architecture

If one of the following symbols is detected, XSIMD_X86_AMD_INSTR_SET is set to the corresponding version and XSIMD_X86_AMD_SET_AVAILABLE is defined.

Symbol	Version
__SSE4A__	XSIMD_X86_AMD_SSE4A_VERSION
__FMA__	XSIMD_X86_AMD_FMA4_VERSION
__XOP__	XSIMD_X86_AMD_XOP_VERSION

If one of the previous symbol is defined, other x86 instruction sets not specific to AMD should be available too; thus XSIMD_X86_INSTR_SET and XSIMD_X86_INSTR_SET_AVAILABLE should be defined. In that case, XSIMD_X86_AMD_INSTR_SET is set to the maximum of XSIMD_X86_INSTR_SET and the current value of XSIMD_X86_AMD_INSTR_SET.

1.4.3 PPC architecture

If one of the following symbols is detected, XSIMD_PPC_INSTR_SET is set to the corresponding version and XSIMD_PPC_INSTR_AVAILABLE is defined.

Symbol	Version
__ALTIVEC__	XSIMD_PPC_VMX_VERSION
__VEC__	XSIMD_PPC_VMX_VERSION
__VSX__	XSIMD_PPC_VSX_VERSION
__VECTOR4DOUBLE__	XSIMD_PPC_QPX_VERSION

1.4.4 ARM architecture

If one of the following symbols is detected, XSIMD_ARM_INSTR_SET is set to the corresponding version and XSIMD_ARM_INSTR_AVAILABLE is defined.

Symbol	Version
__ARM_NEON__	XSIMD_ARM_NEON_VERSION
__aarch64__	XSIMD_ARM_NEON_VERSION
__M_ARM	XSIMD_ARM_NEON_VERSION

1.4.5 Generic instruction set

If `XSIMD_*_INSTR_SET_AVAILABLE` has been defined as explained above, `XSIMD_INSTR_SET` is set to `XSIMD_*_INSTR_SET` and `XSIMD_INSTR_SET_AVAILABLE` is defined.

1.5 Wrapper types

1.5.1 `simd_batch_bool`

```
template <class X>
```

```
class xsimd::simd_batch_bool
```

Base class for batch of boolean values.

The `simd_batch_bool` class is the base class for all classes representing a batch of boolean values. Batch of boolean values is meant for operations that may involve batches of integer or floating point values. Thus, the boolean values are stored as integer or floating point values, and each type of batch has its dedicated type of boolean batch.

See `simd_batch`

Template Parameters

- `X`: The derived type

Bitwise computed assignement

```
X &operator&= (const X &rhs)
```

Assigns the bitwise and of `rhs` and `this`.

Return a reference to `this`.

Parameters

- `rhs`: the batch involved in the operation.

```
X &operator |= (const X &rhs)
```

Assigns the bitwise or of `rhs` and `this`.

Return a reference to `this`.

Parameters

- `rhs`: the batch involved in the operation.

```
X &operator^= (const X &rhs)
```

Assigns the bitwise xor of `rhs` and `this`.

Return a reference to `this`.

Parameters

- `rhs`: the batch involved in the operation.

Static downcast functions

`X &operator () ()`

Returns a reference to the actual derived type of the `simd_batch_bool`.

`const X &operator () () const`

Returns a constant reference to the actual derived type of the `simd_batch_bool`.

Logical operators

`template <class X>`

`X xsimd::operator&& (const simd_batch_bool<X> &lhs, const simd_batch_bool<X> &rhs)`

Computes the logical and of batches `lhs` and `rhs`.

Return the result of the logical and.

Template Parameters

- `X`: the actual type of boolean batch.

Parameters

- `lhs`: batch involved in the operation.
- `rhs`: batch involved in the operation.

`template <class X>`

`X xsimd::operator&& (const simd_batch_bool<X> &lhs, bool rhs)`

Computes the logical and of the batch `lhs` and the scalar `rhs`.

Equivalent to the logical and of two boolean batches, where all the values of the second one are initialized to `rhs`.

Return the result of the logical and.

Template Parameters

- `X`: the actual type of boolean batch.

Parameters

- `lhs`: batch involved in the operation.
- `rhs`: boolean involved in the operation.

`template <class X>`

`X xsimd::operator&& (bool lhs, const simd_batch_bool<X> &rhs)`

Computes the logical and of the scalar `lhs` and the batch `rhs`.

Equivalent to the logical and of two boolean batches, where all the values of the first one are initialized to `lhs`.

Return the result of the logical and.

Template Parameters

- `X`: the actual type of boolean batch.

Parameters

- `lhs`: boolean involved in the operation.
- `rhs`: batch involved in the operation.

`template <class X>`

`X xsimd::operator || (const simd_batch_bool<X> &lhs, const simd_batch_bool<X> &rhs)`
 Computes the logical or of batches `lhs` and `rhs`.

Return the result of the logical or.

Template Parameters

- `X`: the actual type of boolean batch.

Parameters

- `lhs`: batch involved in the operation.
- `rhs`: batch involved in the operation.

`template <class X>`

`X xsimd::operator || (const simd_batch_bool<X> &lhs, bool rhs)`
 Computes the logical or of the batch `lhs` and the scalar `rhs`.

Equivalent to the logical or of two boolean batches, where all the values of the second one are initialized to `rhs`.

Return the result of the logical or.

Template Parameters

- `X`: the actual type of boolean batch.

Parameters

- `lhs`: batch involved in the operation.
- `rhs`: boolean involved in the operation.

`template <class X>`

`X xsimd::operator || (bool lhs, const simd_batch_bool<X> &rhs)`
 Computes the logical or of the scalar `lhs` and the batch `rhs`.

Equivalent to the logical or of two boolean batches, where all the values of the first one are initialized to `lhs`.

Return the result of the logical or.

Template Parameters

- `X`: the actual type of boolean batch.

Parameters

- `lhs`: boolean involved in the operation.
- `rhs`: batch involved in the operation.

1.5.2 simd_batch

`template <class X>`

`class xsimd::simd_batch`

Base class for batch of integer or floating point values.

The `simd_batch` class is the base class for all classes representing a batch of integer or floating point values. Each type of batch (i.e. a class inheriting from `simd_batch`) has its dedicated type of boolean batch (i.e. a class inheriting from `simd_batch_bool`) for logical operations.

See `simd_batch_bool`

Template Parameters

- `X`: The derived type

Static downcast functions

X &operator () ()

Returns a reference to the actual derived type of the *simd_batch_bool*.

const X &operator () () const

Returns a constant reference to the actual derived type of the *simd_batch_bool*.

Arithmetic computed assignment

X &operator += (const X &rhs)

Adds the batch *rhs* to *this*.

Return a reference to *this*.

Parameters

- *rhs*: the batch to add.

X &operator += (const value_type &rhs)

Adds the scalar *rhs* to each value contained in *this*.

Return a reference to *this*.

Parameters

- *rhs*: the scalar to add.

X &operator -= (const X &rhs)

Subtracts the batch *rhs* to *this*.

Return a reference to *this*.

Parameters

- *rhs*: the batch to subtract.

X &operator -= (const value_type &rhs)

Subtracts the scalar *rhs* to each value contained in *this*.

Return a reference to *this*.

Parameters

- *rhs*: the scalar to subtract.

X &operator *= (const X &rhs)

Multiplies *this* with the batch *rhs*.

Return a reference to *this*.

Parameters

- *rhs*: the batch involved in the multiplication.

`X &operator*=(const value_type &rhs)`

Multiplies each scalar contained in `this` with the scalar `rhs`.

Return a reference to `this`.

Parameters

- `rhs`: the scalar involved in the multiplication.

`X &operator/=(const X &rhs)`

Divides `this` by the batch `rhs`.

Return a reference to `this`.

Parameters

- `rhs`: the batch involved in the division.

`X &operator/=(const value_type &rhs)`

Divides each scalar contained in `this` by the scalar `rhs`.

Return a reference to `this`.

Parameters

- `rhs`: the scalar involved in the division.

Bitwise computed assignment

`X &operator&=(const X &rhs)`

Assigns the bitwise and of `rhs` and `this`.

Return a reference to `this`.

Parameters

- `rhs`: the batch involved in the operation.

`X &operator|=(const X &rhs)`

Assigns the bitwise or of `rhs` and `this`.

Return a reference to `this`.

Parameters

- `rhs`: the batch involved in the operation.

`X &operator^=(const X &rhs)`

Assigns the bitwise xor of `rhs` and `this`.

Return a reference to `this`.

Parameters

- `rhs`: the batch involved in the operation.

Increment and decrement operators

`X &operator++ ()`

Pre-increment operator.

Return a reference to `this`.

`X &operator++ (int)`

Post-increment operator.

Return a reference to `this`.

`X &operator-- ()`

Pre-decrement operator.

Return a reference to `this`.

`X &operator-- (int)`

Post-decrement operator.

Return a reference to `this`.

Arithmetic operators

`template <class X>`

`X xsimd::operator+ (const simd_batch<X> &lhs, const typename simd_batch_traits<X>::value_type &rhs)`

Computes the sum of the batch `lhs` and the scalar `rhs`.

Equivalent to the sum of two batches where all the values of the second one are initialized to `rhs`.

Return the result of the addition.

Template Parameters

- `X`: the actual type of batch.

Parameters

- `lhs`: batch involved in the addition.
- `rhs`: scalar involved in the addition.

`template <class X>`

`X xsimd::operator+ (const typename simd_batch<X>::value_type &lhs, const simd_batch<X> &rhs)`

Computes the sum of the scalar `lhs` and the batch `rhs`.

Equivalent to the sum of two batches where all the values of the first one are initialized to `rhs`.

Return the result of the addition.

Template Parameters

- `X`: the actual type of batch.

Parameters

- `lhs`: scalar involved in the addition.
- `rhs`: batch involved in the addition.

```
template <class X>
X xsimd::operator- (const simd_batch<X> &lhs, const typename
                    simd_batch_traits<X>::value_type &rhs)
```

Computes the difference of the batch `lhs` and the scalar `rhs`.

Equivalent to the difference of two batches where all the values of the second one are initialized to `rhs`.

Return the result of the difference.

Template Parameters

- `X`: the actual type of batch.

Parameters

- `lhs`: batch involved in the difference.
- `rhs`: scalar involved in the difference.

```
template <class X>
X xsimd::operator- (const typename simd_batch<X>::value_type &lhs, const simd_batch<X>
                    &rhs)
```

Computes the difference of the scalar `lhs` and the batch `rhs`.

Equivalent to the difference of two batches where all the values of the first one are initialized to `rhs`.

Return the result of the difference.

Template Parameters

- `X`: the actual type of batch.

Parameters

- `lhs`: scalar involved in the difference.
- `rhs`: batch involved in the difference.

```
template <class X>
X xsimd::operator* (const simd_batch<X> &lhs, const typename
                    simd_batch_traits<X>::value_type &rhs)
```

Computes the product of the batch `lhs` and the scalar `rhs`.

Equivalent to the product of two batches where all the values of the second one are initialized to `rhs`.

Return the result of the product.

Template Parameters

- `X`: the actual type of batch.

Parameters

- `lhs`: batch involved in the product.
- `rhs`: scalar involved in the product.

```
template <class X>
X xsimd::operator* (const typename simd_batch<X>::value_type &lhs, const simd_batch<X>
                    &rhs)
```

Computes the product of the scalar `lhs` and the batch `rhs`.

Equivalent to the difference of two batches where all the values of the first one are initialized to `rhs`.

Return the result of the product.

Template Parameters

- `X`: the actual type of batch.

Parameters

- lhs: scalar involved in the product.
- rhs: batch involved in the product.

```
template <class X>
X xsimd::operator/ (const simd_batch<X> &lhs, const typename
                    simd_batch_traits<X>::value_type &rhs)
```

Computes the division of the batch lhs by the scalar rhs.

Equivalent to the division of two batches where all the values of the second one are initialized to rhs.

Return the result of the division.

Template Parameters

- X: the actual type of batch.

Parameters

- lhs: batch involved in the division.
- rhs: scalar involved in the division.

```
template <class X>
X xsimd::operator/ (const typename simd_batch<X>::value_type &lhs, const simd_batch<X>
                    &rhs)
```

Computes the division of the scalar lhs and the batch rhs.

Equivalent to the difference of two batches where all the values of the first one are initialized to rhs.

Return the result of the division.

Template Parameters

- X: the actual type of batch.

Parameters

- lhs: scalar involved in the division.
- rhs: batch involved in the division.

Comparison operators

```
template <class X>
simd_batch_traits<X>::batch_bool_type xsimd::operator> (const simd_batch<X> &lhs, const
                                                         simd_batch<X> &rhs)
```

Element-wise greater than comparison of batches lhs and rhs.

Return a boolean batch.

Template Parameters

- X: the actual type of batch.

Parameters

- lhs: batch involved in the comparison.
- rhs: batch involved in the comparison.

```
template <class X>
simd_batch_traits<X>::batch_bool_type xsimd::operator>= (const simd_batch<X> &lhs, const
                                                          simd_batch<X> &rhs)
```

Element-wise greater or equal comparison of batches lhs and rhs.

Return a boolean batch.

Template Parameters

- X: the actual type of batch.

Parameters

- lhs: batch involved in the comparison.
- rhs: batch involved in the comparison.

Other operators

```
template <class X>
simd_batch_traits<X>::batch_bool_type xsimd::operator! (const simd_batch<X> &rhs)
    Element-wise not of rhs.
```

Return boolean batch.

Template Parameters

- X: the actual type of batch.

Parameters

- rhs: batch involved in the logical not operation.

```
template <class X>
std::ostream &xsimd::operator<< (std::ostream &out, const simd_batch<X> &rhs)
    Insert the batch rhs into the stream out.
```

Return the output stream.

Template Parameters

- X: the actual type of batch.

Parameters

- out: the output stream.
- rhs: the batch to output.

1.5.3 batch_bool

```
template <class T, std::size_t N>
class xsimd::batch_bool
    Batch of boolean values.
```

The *batch_bool* class represents a batch of boolean values, that can be used in operations involving batches of integer or floating point values. The boolean values are stored as integer or floating point values, depending on the type of batch they are dedicated to.

Template Parameters

- T: The value type used for encoding boolean.
- N: The number of scalar in the batch.

Inherits from *xsimd::simd_batch_bool<batch_bool<T, N>>*

Public Functions

batch_bool ()

Builds an uninitialized batch of boolean values.

batch_bool (bool *b*)

Initializes all the values of the batch to *b*.

xsimd::batch_bool::batch_bool (bool *b0*, ..., bool *bn*)

Initializes a batch of booleans with the specified boolean values.

batch_bool (const simd_data &*rhs*)

Initializes a batch of boolean with the specified SIMD value.

batch_bool &operator= (const simd_data &*rhs*)

Assigns the specified SIMD value.

operator simd_data () const

Converts *this* to a SIMD value.

Logical operators

template <class T, std::size_t N>

batch_bool<T, N> xsimd::operator& (const *batch_bool*<T, N> &*lhs*, const *batch_bool*<T, N> &*rhs*)

Computes the bitwise and of the batches *lhs* and *rhs*.

Return the result of the bitwise and.

Parameters

- *lhs*: batch involved in the operation.
- *rhs*: batch involved in the operation.

template <class T, std::size_t N>

batch_bool<T, N> xsimd::operator| (const *batch_bool*<T, N> &*lhs*, const *batch_bool*<T, N> &*rhs*)

Computes the bitwise or of the batches *lhs* and *rhs*.

Return the result of the bitwise or.

Parameters

- *lhs*: batch involved in the operation.
- *rhs*: batch involved in the operation.

template <class T, std::size_t N>

batch_bool<T, N> xsimd::operator^ (const *batch_bool*<T, N> &*lhs*, const *batch_bool*<T, N> &*rhs*)

Computes the bitwise xor of the batches *lhs* and *rhs*.

Return the result of the bitwise xor.

Parameters

- *lhs*: batch involved in the operation.
- *rhs*: batch involved in the operation.

template <class T, std::size_t N>

batch_bool<T, N> xsimd::operator~ (const *batch_bool*<T, N> &*rhs*)

Computes the bitwise not of the batch *rhs*.

Return the result of the bitwise not.

Parameters

- rhs: batch involved in the operation.

```
template <class T, std::size_t N>
batch_bool<T, N> xsimd::bitwise_andnot (const batch_bool<T, N> &lhs, const batch_bool<T, N>
                                     &rhs)
```

Computes the bitwise and not of the batches lhs and rhs.

Equivalent to

```
lhs & ~rhs
```

Return the result of the bitwise not.

Parameters

- lhs: batch involved in the operation.
- rhs: batch involved in the operation.

Comparison operators

```
template <class T, std::size_t N>
batch_bool<T, N> xsimd::operator==(const batch_bool<T, N> &lhs, const batch_bool<T, N>
                                  &rhs)
```

Element-wise equality of batches lhs and rhs.

Return the result of the equality comparison.

Parameters

- lhs: batch involved in the comparison.
- rhs: batch involved in the comparison.

```
template <class T, std::size_t N>
batch_bool<T, N> xsimd::operator!=(const batch_bool<T, N> &lhs, const batch_bool<T, N>
                                  &rhs)
```

Element-wise inequality of batches lhs and rhs.

Return the result of the inequality comparison.

Parameters

- lhs: batch involved in the comparison.
- rhs: batch involved in the comparison.

Reducers

```
template <class T, std::size_t N>
bool xsimd::all (const batch_bool<T, N> &rhs)
```

Returns true if all the boolean values in the batch are true, false otherwise.

Return a boolean scalar.

Parameters

- `rhs`: the batch to reduce.

```
template <class T, std::size_t N>
```

```
bool xsimd::any(const batch_bool<T, N> &rhs)
```

Return true if any of the boolean values in the batch is true, false otherwise.

Return a boolean scalar.

Parameters

- `rhs`: the batch to reduce.

1.5.4 batch

```
template <class T, std::size_t N>
```

```
class xsimd::batch
```

Batch of integer or floating point values.

The batch class represents a batch of integer or floating point values. Types supported are `int32_t`, `int64_t`, `float` and `double`.

Template Parameters

- `T`: The value type.
- `N`: The number of scalar in the batch.

Inherits from `xsimd::simd_batch<batch<T, N>>`

Public Functions

```
batch()
```

Builds an uninitialized batch.

```
batch(T f)
```

Initializes all the values of the batch to `b`.

```
xsimd::batch::batch(T f0, ..., T f3)
```

Initializes a batch with the specified boolean values.

```
batch(const T *src, aligned_mode)
```

Initializes a batch to the `N` contiguous values pointed by `src`; `src` is not required to be aligned.

```
batch(const T *src, unaligned_mode)
```

Initializes a batch to the values pointed by `src`; `src` must be aligned.

```
batch(const simd_data &rhs)
```

Initializes a batch with the specified SIMD value.

```
batch &operator=(const simd_data &rhs)
```

Assigns the specified SIMD value to the batch.

```
operator simd_data() const
```

Converts `this` to a SIMD value.

```
batch &load_aligned(const T *src)
```

Loads the `N` contiguous values pointed by `src` into the batch.

`src` must be aligned.

batch & **load_unaligned** (const T *src)

Loads the N contiguous values pointed by *src* into the batch.

src is not required to be aligned.

void **store_aligned** (T *dst) const

Stores the N values of the batch into a contiguous array pointed by *dst*.

dst must be aligned.

void **store_unaligned** (T *dst) const

Stores the N values of the batch into a contiguous array pointed by *dst*.

dst is not required to be aligned.

T **operator** [] (std::size_t i) const

Return the i-th scalar in the batch.

Arithmetic operators

template <class T, std::size_t N>

batch<T, N> xsimd::operator- (const *batch*<T, N> &rhs)

Computes the opposite of the batch *rhs*.

Return the opposite of *rhs*.

Parameters

- *rhs*: batch involved in the operation.

template <class T, std::size_t N>

batch<T, N> xsimd::operator+ (const *batch*<T, N> &lhs, const *batch*<T, N> &rhs)

Computes the sum of the batches *lhs* and *rhs*.

Return the result of the addition.

Parameters

- *lhs*: batch involved in the addition.
- *rhs*: batch involved in the addition.

template <class T, std::size_t N>

batch<T, N> xsimd::operator- (const *batch*<T, N> &lhs, const *batch*<T, N> &rhs)

Computes the difference of the batches *lhs* and *rhs*.

Return the result of the difference.

Parameters

- *lhs*: batch involved in the difference.
- *rhs*: batch involved in the difference.

template <class T, std::size_t N>

batch<T, N> xsimd::operator* (const *batch*<T, N> &lhs, const *batch*<T, N> &rhs)

Computes the product of the batches *lhs* and *rhs*.

Return the result of the product.

Parameters

- *lhs*: batch involved in the product.

- rhs: batch involved in the product.

template <class T, std::size_t N>

batch<T, N> xsimd::operator/ (const *batch*<T, N> &lhs, const *batch*<T, N> &rhs)

Computes the division of the batch lhs by the batch rhs.

Return the result of the division.

Parameters

- lhs: batch involved in the division.
- rhs: batch involved in the division.

Comparison operators

template <class T, std::size_t N>

batch_bool<T, 4> xsimd::operator==(const *batch*<T, N> &lhs, const *batch*<T, N> &rhs)

Element-wise equality comparison of batches lhs and rhs.

Return a boolean batch.

Parameters

- lhs: batch involved in the comparison.
- rhs: batch involved in the comparison.

template <class T, std::size_t N>

batch_bool<T, 4> xsimd::operator!=(const *batch*<T, N> &lhs, const *batch*<T, N> &rhs)

Element-wise inequality comparison of batches lhs and rhs.

Return a boolean batch.

Parameters

- lhs: batch involved in the comparison.
- rhs: batch involved in the comparison.

template <class T, std::size_t N>

batch_bool<T, 4> xsimd::operator<(const *batch*<T, N> &lhs, const *batch*<T, N> &rhs)

Element-wise lesser than comparison of batches lhs and rhs.

Return a boolean batch.

Parameters

- lhs: batch involved in the comparison.
- rhs: batch involved in the comparison.

template <class T, std::size_t N>

batch_bool<T, 4> xsimd::operator<=(const *batch*<T, N> &lhs, const *batch*<T, N> &rhs)

Element-wise lesser or equal to comparison of batches lhs and rhs.

Return a boolean batch.

Parameters

- lhs: batch involved in the comparison.
- rhs: batch involved in the comparison.

Logical operators

template <class T, std::size_t N>
batch<T, N> xsimd::operator& (const *batch*<T, N> &lhs, const *batch*<T, N> &rhs)
 Computes the bitwise and of the batches lhs and rhs.

Return the result of the bitwise and.

Parameters

- lhs: batch involved in the operation.
- rhs: batch involved in the operation.

template <class T, std::size_t N>
batch<T, N> xsimd::operator| (const *batch*<T, N> &lhs, const *batch*<T, N> &rhs)
 Computes the bitwise or of the batches lhs and rhs.

Return the result of the bitwise or.

Parameters

- lhs: batch involved in the operation.
- rhs: batch involved in the operation.

template <class T, std::size_t N>
batch<T, N> xsimd::operator^ (const *batch*<T, N> &lhs, const *batch*<T, N> &rhs)
 Computes the bitwise xor of the batches lhs and rhs.

Return the result of the bitwise xor.

Parameters

- lhs: batch involved in the operation.
- rhs: batch involved in the operation.

template <class T, std::size_t N>
batch<T, N> xsimd::operator~ (const *batch*<T, N> &rhs)
 Computes the bitwise not of the batches lhs and rhs.

Return the result of the bitwise not.

Parameters

- rhs: batch involved in the operation.

template <class T, std::size_t N>
batch<T, N> xsimd::bitwise_andnot (const *batch*<T, N> &lhs, const *batch*<T, N> &rhs)
 Computes the bitwise andnot of the batches lhs and rhs.

Return the result of the bitwise andnot.

Parameters

- lhs: batch involved in the operation.
- rhs: batch involved in the operation.

Miscellaneous

```
template <class T, std::size_t N>
batch<T, N> xsimd::select (const batch_bool<T, N> &cond, const batch<T, N> &a, const
                        batch<T, N> &b)
```

Ternary operator for batches: selects values from the batches a or b depending on the boolean values in cond.

Equivalent to

```
for (std::size_t i = 0; i < N; ++i)
    res[i] = cond[i] ? a[i] : b[i];
```

Return the result of the selection.

Parameters

- cond: batch condition.
- a: batch values for truthy condition.
- b: batch value for falsy condition.

1.5.5 Available wrappers

The *batch* and *batch_bool* generic template classes are not implemented, only full specializations of these templates are available depending on the instruction set macros defined according to the instruction sets provided by the compiler.

x86 architecture

Depending on the value of `XSIMD_X86_INSTR_SET`, the following wrappers are available:

- `XSIMD_X86_INSTR_SET >= XSIMD_X86_SSE2_VERSION`

batch	batch_bool
batch<float, 4>	batch_bool<float, 4>
batch<int32_t, 4>	batch_bool<int32_t, 4>
batch<double, 2>	batch_bool<double, 2>
batch<int64_t, 2>	batch_bool<int64_t, 2>

- `XSIMD_X86_INSTR_SET >= XSIMD_X86_AVX_VERSION`

In addition to the wrappers defined above, the following wrappers are available

batch	batch_bool
batch<float, 8>	batch_bool<float, 8>
batch<int32_t, 8>	batch_bool<int32_t, 8>
batch<double, 4>	batch_bool<double, 4>
batch<int64_t, 4>	batch_bool<int64_t, 4>

1.6 Data transfer

1.6.1 Data transfer instructions

```
template <class T1, class T2 = T1>
```

```
simd_type<T2> xsimd::set_simd(const T1 &value)
```

Returns a batch with all values initialized to `value`.

Return the batch wrapping the highest available instruction set.

Parameters

- `value`: the scalar used to initialize the batch.

```
template <class T1, class T2 = T1>
```

```
simd_type<T2> xsimd::load_aligned(const T1 *src)
```

Loads the memory array pointed to by `src` into a batch and returns it.

`src` is required to be aligned.

Return the batch wrapping the highest available instruction set.

Parameters

- `src`: the pointer to the memory array to load.

```
template <class T1, class T2 = T1>
```

```
void xsimd::load_aligned(const T1 *src, simd_type<T2> &dst)
```

Loads the memory array pointed to by `src` into the batch `dst`.

`src` is required to be aligned.

Parameters

- `src`: the pointer to the memory array to load.
- `dst`: the destination batch.

```
template <class T1, class T2 = T1>
```

```
simd_type<T2> xsimd::load_unaligned(const T1 *src)
```

Loads the memory array pointed to by `src` into a batch and returns it.

`src` is not required to be aligned.

Return the batch wrapping the highest available instruction set.

Parameters

- `src`: the pointer to the memory array to load.

```
template <class T1, class T2 = T1>
```

```
void xsimd::load_unaligned(const T1 *src, simd_type<T2> &dst)
```

Loads the memory array pointed to by `src` into the batch `dst`.

`src` is not required to be aligned.

Parameters

- `src`: the pointer to the memory array to load.
- `dst`: the destination batch.

```
template <class T1, class T2 = T1>
```

```
void xsimd::store_aligned(T1 *dst, const simd_type<T2> &src)
```

Stores the batch `src` into the memory array pointed to by `dst`.

`dst` is required to be aligned.

Parameters

- `dst`: the pointer to the memory array.
- `src`: the batch to store.

```
template <class T1, class T2 = T1>
```

void xsimd::store_unaligned(T1 *dst, const simd_type<T2> &src)
Stores the batch src into the memory array pointed to by dst.

dst is not required to be aligned.

Parameters

- dst: the pointer to the memory array.
- src: the batch to store.

1.6.2 Generic load and store

template <class T1, class T2 = T1>
simd_type<T2> xsimd::load_simd(const T1 *src, aligned_mode)
Loads the memory array pointed to by src into a batch and returns it.
src is required to be aligned.

Return the batch wrapping the highest available instruction set.

Parameters

- src: the pointer to the memory array to load.

template <class T1, class T2 = T1>
void xsimd::load_simd(const T1 *src, simd_type<T2> &dst, aligned_mode)
Loads the memory array pointed to by src into the batch dst.
src is required to be aligned.

Parameters

- src: the pointer to the memory array to load.
- dst: the destination batch.

template <class T1, class T2 = T1>
simd_type<T2> xsimd::load_simd(const T1 *src, unaligned_mode)
Loads the memory array pointed to by src into a batch and returns it.
src is not required to be aligned.

Return the batch wrapping the highest available instruction set.

Parameters

- src: the pointer to the memory array to load.

template <class T1, class T2 = T1>
void xsimd::load_simd(const T1 *src, simd_type<T2> &dst, unaligned_mode)
Loads the memory array pointed to by src into the batch dst.
src is not required to be aligned.

Parameters

- src: the pointer to the memory array to load.
- dst: the destination batch.

template <class T1, class T2 = T1>
void xsimd::store_simd(T1 *dst, const simd_type<T2> &src, aligned_mode)
Stores the batch src into the memory array pointed to by dst.
dst is required to be aligned.

Parameters

- `dst`: the pointer to the memory array.
- `src`: the batch to store.

```
template <class T1, class T2 = T1>
```

```
void xsimd::store_simd(T1 *dst, const simd_type<T2> &src, unaligned_mode)
```

Stores the batch `src` into the memory array pointed to by `dst`.

`dst` is not required to be aligned.

Parameters

- `dst`: the pointer to the memory array.
- `src`: the batch to store.

1.7 Mathematical functions

1.7.1 Basic functions

```
template <class T, std::size_t N>
```

```
batch<T, N> xsimd::abs(const batch<T, N> &x)
```

Computes the absolute values of each scalar in the batch `x`.

Return the absolute values of `x`.

Parameters

- `x`: batch of integer or floating point values.

```
template <class T, std::size_t N>
```

```
batch<T, N> xsimd::fabs(const batch<T, N> &x)
```

Computes the absolute values of each scalar in the batch `x`.

Return the absolute values of `x`.

Parameters

- `x`: batch floating point values.

```
template <class T, std::size_t N>
```

```
batch<T, N> xsimd::fmod(const batch<T, N> &x, const batch<T, N> &y)
```

Computes the floating-point remainder of the division operation x/y .

The floating-point remainder of the division operation x/y calculated by this function is exactly the value $x - n*y$, where n is x/y with its fractional part truncated. The returned value has the same sign as `x` and is less than `y` in magnitude.

Return the floating-point remainder of the division.

Parameters

- `x`: batch of floating point values.
- `y`: batch of floating point values.

```
template <class T, std::size_t N>
```

```
batch<T, N> xsimd::remainder(const batch<T, N> &x, const batch<T, N> &y)
```

Computes the IEEE remainder of the floating point division operation x/y .

The IEEE floating-point remainder of the division operation x/y calculated by this function is exactly the value $x - n*y$, where the value n is the integral value nearest the exact value x/y . When $|n-x/y| = 0.5$, the

value n is chosen to be even. In contrast to `fmod`, the returned value is not guaranteed to have the same sign as x . If the returned value is 0, it will have the same sign as x .

Return the IEEE remainder remainder of the floating point division.

Parameters

- x : batch of floating point values.
- y : batch of floating point values.

```
template <class T, std::size_t N>
```

```
batch<T, N> xsimd::fma(const batch<T, N> &x, const batch<T, N> &y, const batch<T, N> &z)
```

Computes $(x*y) + z$ in a single instruction when possible.

Return the result of the fused multiply-add operation.

Parameters

- x : a batch of integer or floating point values.
- y : a batch of integer or floating point values.
- z : a batch of integer or floating point values.

```
template <class T, std::size_t N>
```

```
batch<T, N> xsimd::fms(const batch<T, N> &x, const batch<T, N> &y, const batch<T, N> &z)
```

Computes $(x*y) - z$ in a single instruction when possible.

Return the result of the fused multiply-sub operation.

Parameters

- x : a batch of integer or floating point values.
- y : a batch of integer or floating point values.
- z : a batch of integer or floating point values.

```
template <class T, std::size_t N>
```

```
batch<T, N> xsimd::fnma(const batch<T, N> &x, const batch<T, N> &y, const batch<T, N> &z)
```

Computes $-(x*y) + z$ in a single instruction when possible.

Return the result of the fused negated multiply-add operation.

Parameters

- x : a batch of integer or floating point values.
- y : a batch of integer or floating point values.
- z : a batch of integer or floating point values.

```
template <class T, std::size_t N>
```

```
batch<T, N> xsimd::fnms(const batch<T, N> &x, const batch<T, N> &y, const batch<T, N> &z)
```

Computes $-(x*y) - z$ in a single instruction when possible.

Return the result of the fused negated multiply-sub operation.

Parameters

- x : a batch of integer or floating point values.
- y : a batch of integer or floating point values.
- z : a batch of integer or floating point values.


```
template <class T, std::size_t N>
batch<T, N> xsimd::min(const batch<T, N> &x, const batch<T, N> &y)
    Returns the smaller values of the batches x and y.
```

Return a batch of the smaller values.

Parameters

- x: a batch of integer or floating point values.
- y: a batch of integer or floating point values.

```
template <class T, std::size_t N>
batch<T, N> xsimd::max(const batch<T, N> &x, const batch<T, N> &y)
    Returns the larger values of the batches x and y.
```

Return a batch of the larger values.

Parameters

- x: a batch of integer or floating point values.
- y: a batch of integer or floating point values.

```
template <class T, std::size_t N>
batch<T, N> xsimd::fmin(const batch<T, N> &x, const batch<T, N> &y)
    Returns the smaller values of the batches x and y.
```

Return a batch of the smaller values.

Parameters

- x: a batch of floating point values.
- y: a batch of floating point values.

```
template <class T, std::size_t N>
batch<T, N> xsimd::fmax(const batch<T, N> &x, const batch<T, N> &y)
    Returns the larger values of the batches x and y.
```

Return a batch of the larger values.

Parameters

- x: a batch of floating point values.
- y: a batch of floating point values.

```
template <class T, std::size_t N>
batch<T, N> xsimd::fdim(const batch<T, N> &x, const batch<T, N> &y)
    Computes the positive difference between x and y, that is,  $\max(0, x-y)$ .
```

Return the positive difference.

Parameters

- x: batch of floating point values.
- y: batch of floating point values.

```
template <class T, std::size_t N>
batch<T, N> xsimd::clip(const batch<T, N> &x, const batch<T, N> &lo, const batch<T, N> &hi)
    Clips the values of the batch x between those of the batches lo and hi.
```

Return the result of the clipping.

Parameters

- *x*: batch of floating point values.
- *lo*: batch of floating point values.
- *hi*: batch of floating point values.

<i>abs</i>	absolute value
<i>fabs</i>	absolute value
<i>fmod</i>	remainder of the floating point division operation
<i>remainder</i>	signed remainder of the division operation
<i>fma</i>	fused multiply-add operation
<i>fms</i>	fused multiply-sub operation
<i>fnma</i>	fused negated multiply-add operation
<i>fnms</i>	fused negated multiply-sub operation
<i>min</i>	smaller of two batches
<i>max</i>	larger of two batches
<i>fmin</i>	smaller of two batches of floating point values
<i>fmax</i>	larger of two batches of floating point values
<i>fdim</i>	positive difference
<i>clip</i>	clipping operation

1.7.2 Exponential functions

```
template <class T, std::size_t N>
batch<T, N> xsimd::exp(const batch<T, N> &x)
    Computes the natural exponential of the batch x.
```

Return the natural exponential of *x*.

Parameters

- *x*: batch of floating point values.

```
template <class T, std::size_t N>
batch<T, N> xsimd::exp2(const batch<T, N> &x)
    Computes the base 2 exponential of the batch x.
```

Return the base 2 exponential of *x*.

Parameters

- *x*: batch of floating point values.

```
template <class T, std::size_t N>
batch<T, N> xsimd::exp10(const batch<T, N> &x)
    Computes the base 10 exponential of the batch x.
```

Return the base 10 exponential of *x*.

Parameters

- *x*: batch of floating point values.

```
template <class T, std::size_t N>
```

`batch<T, N> xsimd::expm1 (const batch<T, N> &x)`
 Computes the natural exponential of the batch `x`, minus one.

Return the natural exponential of `x`, minus one.

Parameters

- `x`: batch of floating point values.

`template <class T, std::size_t N>`
`batch<T, N> xsimd::log (const batch<T, N> &x)`
 Computes the natural logarithm of the batch `x`.

Return the natural logarithm of `x`.

Parameters

- `x`: batch of floating point values.

`template <class T, std::size_t N>`
`batch<T, N> xsimd::log2 (const batch<T, N> &x)`
 Computes the base 2 logarithm of the batch `x`.

Return the base 2 logarithm of `x`.

Parameters

- `x`: batch of floating point values.

`template <class T, std::size_t N>`
`batch<T, N> xsimd::log10 (const batch<T, N> &x)`
 Computes the base 10 logarithm of the batch `x`.

Return the base 10 logarithm of `x`.

Parameters

- `x`: batch of floating point values.

`template <class T, std::size_t N>`
`batch<T, N> xsimd::log1p (const batch<T, N> &x)`
 Computes the natural logarithm of one plus the batch `x`.

Return the natural logarithm of one plus `x`.

Parameters

- `x`: batch of floating point values.

<code>exp</code>	natural exponential function
<code>exp2</code>	base 2 exponential function
<code>exp10</code>	base 10 exponential function
<code>expm1</code>	natural exponential function, minus one
<code>log</code>	natural logarithm function
<code>log2</code>	base 2 logarithm function
<code>log10</code>	base 10 logarithm function
<code>log1p</code>	natural logarithm of one plus function

1.7.3 Power functions

```
template <class T, std::size_t N>
batch<T, N> xsimd::pow (const batch<T, N> &x, const batch<T, N> &y)
    Computes the value of the batch  $x$  raised to the power  $y$ .
```

Return x raised to the power y .

Parameters

- x : batch of floating point values.
- y : batch of floating point values.

```
template <class T, std::size_t N>
batch<T, N> xsimd::sqrt (const batch<T, N> &x)
    Computes the square root of the batch  $x$ .
```

Return the square root of x .

Parameters

- x : batch of floating point values.

```
template <class T, std::size_t N>
batch<T, N> xsimd::cbrt (const batch<T, N> &x)
    Computes the cubic root of the batch  $x$ .
```

Return the cubic root of x .

Parameters

- x : batch of floating point values.

```
template <class T, std::size_t N>
batch<T, N> xsimd::hypot (const batch<T, N> &x, const batch<T, N> &y)
    Computes the square root of the sum of the squares of the batches  $x$ , and  $y$ .
```

Return the square root of the sum of the squares of x and y .

Parameters

- x : batch of floating point values.
- y : batch of floating point values.

<i>pow</i>	power function
<i>sqrt</i>	square root function
<i>cbrt</i>	cubic root function
<i>hypot</i>	hypotenuse function

1.7.4 Trigonometric functions

```
template <class T, std::size_t N>
batch<T, N> xsimd::sin (const batch<T, N> &x)
    Computes the sine of the batch  $x$ .
```

Return the sine of x .

Parameters

- x : batch of floating point values.

```
template <class T, std::size_t N>
batch<T, N> xsimd::cos(const batch<T, N> &x)
    Computes the cosine of the batch  $x$ .
```

Return the cosine of x .

Parameters

- x : batch of floating point values.

```
template <class T, std::size_t N>
batch<T, N> xsimd::tan(const batch<T, N> &x)
    Computes the tangent of the batch  $x$ .
```

Return the tangent of x .

Parameters

- x : batch of floating point values.

```
template <class T, std::size_t N>
batch<T, N> xsimd::asin(const batch<T, N> &x)
    Computes the arc sine of the batch  $x$ .
```

Return the arc sine of x .

Parameters

- x : batch of floating point values.

```
template <class T, std::size_t N>
batch<T, N> xsimd::acos(const batch<T, N> &x)
    Computes the arc cosine of the batch  $x$ .
```

Return the arc cosine of x .

Parameters

- x : batch of floating point values.

```
template <class T, std::size_t N>
batch<T, N> xsimd::atan(const batch<T, N> &x)
    Computes the arc tangent of the batch  $x$ .
```

Return the arc tangent of x .

Parameters

- x : batch of floating point values.

```
template <class T, std::size_t N>
batch<T, N> xsimd::atan2(const batch<T, N> &y, const batch<T, N> &x)
    Computes the arc tangent of the batch  $x/y$ , using the signs of the arguments to determine the correct quadrant.
```

Return the arc tangent of x/y .

Parameters

- x : batch of floating point values.
- y : batch of floating point values.

<i>sin</i>	sine function
<i>cos</i>	cosine function
<i>tan</i>	tangent function
<i>asin</i>	arc sine function
<i>acos</i>	arc cosine function
<i>atan</i>	arc tangent function
<i>atan2</i>	arc tangent function, determining quadrants

1.7.5 Hyperbolic functions

template <class T, std::size_t N>
batch<T, N> xsimd::sinh(const *batch*<T, N> &x)
Computes the hyperbolic sine of the batch x.

Return the hyperbolic sine of x.

Parameters

- x: batch of floating point values.

template <class T, std::size_t N>
batch<T, N> xsimd::cosh(const *batch*<T, N> &x)
Computes the hyperbolic cosine of the batch x.

Return the hyperbolic cosine of x.

Parameters

- x: batch of floating point values.

template <class T, std::size_t N>
batch<T, N> xsimd::tanh(const *batch*<T, N> &x)
Computes the hyperbolic tangent of the batch x.

Return the hyperbolic tangent of x.

Parameters

- x: batch of floating point values.

template <class T, std::size_t N>
batch<T, N> xsimd::asinh(const *batch*<T, N> &x)
Computes the inverse hyperbolic sine of the batch x.

Return the inverse hyperbolic sine of x.

Parameters

- x: batch of floating point values.

template <class T, std::size_t N>
batch<T, N> xsimd::acosh(const *batch*<T, N> &x)
Computes the inverse hyperbolic cosine of the batch x.

Return the inverse hyperbolic cosine of x.

Parameters

- x: batch of floating point values.

```
template <class T, std::size_t N>
batch<T, N> xsimd::atanh(const batch<T, N> &x)
    Computes the inverse hyperbolic tangent of the batch x.
```

Return the inverse hyperbolic tangent of x .

Parameters

- x : batch of floating point values.

<i>sinh</i>	hyperbolic sine function
<i>cosh</i>	hyperbolic cosine function
<i>tanh</i>	hyperbolic tangent function
<i>asinh</i>	inverse hyperbolic sine function
<i>acosh</i>	inverse hyperbolic cosine function
<i>atanh</i>	inverse hyperbolic tangent function

1.7.6 Error and gamma functions

```
template <class T, std::size_t N>
batch<T, N> xsimd::erf(const batch<T, N> &x)
    Computes the error function of the batch x.
```

Return the error function of x .

Parameters

- x : batch of floating point values.

```
template <class T, std::size_t N>
batch<T, N> xsimd::erfc(const batch<T, N> &x)
    Computes the complementary error function of the batch x.
```

Return the error function of x .

Parameters

- x : batch of floating point values.

```
template <class T, std::size_t N>
batch<T, N> xsimd::tgamma(const batch<T, N> &x)
    Computes the gamma function of the batch x.
```

Return the gamma function of x .

Parameters

- x : batch of floating point values.

```
template <class T, std::size_t N>
batch<T, N> xsimd::lgamma(const batch<T, N> &x)
    Computes the natural logarithm of the gamma function of the batch x.
```

Return the natural logarithm of the gamma function of x .

Parameters

- x : batch of floating point values.

<i>erf</i>	error function
<i>erfc</i>	complementary error function
<i>tgamma</i>	gamma function
<i>lgamma</i>	natural logarithm of the gamma function

1.7.7 Nearest integer floating point operations

template <class T, std::size_t N>

batch<T, N> xsimd::ceil(const *batch*<T, N> &x)

Computes the batch of smallest integer values not less than scalars in *x*.

Return the batch of smallest integer values not less than *x*.

Parameters

- *x*: batch of floating point values.

template <class T, std::size_t N>

batch<T, N> xsimd::floor(const *batch*<T, N> &x)

Computes the batch of largest integer values not greater than scalars in *x*.

Return the batch of largest integer values not greater than *x*.

Parameters

- *x*: batch of floating point values.

template <class T, std::size_t N>

batch<T, N> xsimd::trunc(const *batch*<T, N> &x)

Computes the batch of nearest integer values not greater in magnitude than scalars in *x*.

Return the batch of nearest integer values not greater in magnitude than *x*.

Parameters

- *x*: batch of floating point values.

template <class T, std::size_t N>

batch<T, N> xsimd::round(const *batch*<T, N> &x)

Computes the batch of nearest integer values to scalars in *x* (in floating point format), rounding halfway cases away from zero, regardless of the current rounding mode.

Return the batch of nearest integer values.

Parameters

- *x*: batch of floating point values.

template <class T, std::size_t N>

batch<T, N> xsimd::nearbyint(const *batch*<T, N> &x)

Rounds the scalars in *x* to integer values (in floating point format), using the current rounding mode.

Return the batch of nearest integer values.

Parameters

- *x*: batch of floating point values.

template <class T, std::size_t N>

`batch<T, N> xsimd::rint (const batch<T, N> &x)`

Rounds the scalars in `x` to integer values (in floating point format), using the current rounding mode.

Return the batch of rounded values.

Parameters

- `x`: batch of floating point values.

<i>ceil</i>	nearest integers not less
<i>floor</i>	nearest integers not greater
<i>trunc</i>	nearest integers not greater in magnitude
<i>round</i>	nearest integers, rounding away from zero
<i>nearbyint</i>	nearest integers using current rounding mode
<i>rint</i>	nearest integers using current rounding mode

1.7.8 Classification functions

`template <class T, std::size_t N>`

`batch_bool<T, N> xsimd::isfinite (const batch<T, N> &x)`

Determines if the scalars in the given batch `x` have finite values, i.e.

they are different from infinite or NaN.

Return a batch of booleans.

Parameters

- `x`: batch of floating point values.

`template <class T, std::size_t N>`

`batch_bool<T, N> xsimd::isinf (const batch<T, N> &x)`

Determines if the scalars in the given batch `x` are positive or negative infinity.

Return a batch of booleans.

Parameters

- `x`: batch of floating point values.

`template <class T, std::size_t N>`

`batch_bool<T, 4> xsimd::isnan (const batch<T, N> &x)`

Determines if the scalars in the given batch `x` are NaN values.

Return a batch of booleans.

Parameters

- `x`: batch of floating point values.

<i>isfinite</i>	Checks for finite values
<i>isinf</i>	Checks for infinite values
<i>isnan</i>	Checks for NaN values

1.8 Aligned memory allocator

```
template <class T, size_t Align>
class xsimd::aligned_allocator
```

Allocator for aligned memory.

The *aligned_allocator* class template is an allocator that performs memory allocation aligned by the specified value.

Template Parameters

- *T*: type of objects to allocate.
- *Align*: alignment in bytes.

Public Functions

```
aligned_allocator ()
```

Default constructor.

```
aligned_allocator (const aligned_allocator &rhs)
```

Copy constructor.

```
template <class U>
```

```
aligned_allocator (const aligned_allocator<U, Align> &rhs)
```

Extended copy constructor.

```
~aligned_allocator ()
```

Destructor.

```
auto address (reference r)
```

Returns the actual address of *r* even in presence of overloaded operator&.

Return the actual address of *r*.

Parameters

- *r*: the object to acquire address of.

```
auto address (const_reference r) const
```

Returns the actual address of *r* even in presence of overloaded operator&.

Return the actual address of *r*.

Parameters

- *r*: the object to acquire address of.

```
auto allocate (size_type n, typename std::allocator<void>::const_pointer hint = 0)
```

Allocates $n * \text{sizeof}(T)$ bytes of uninitialized memory, aligned by *A*.

The alignment may require some extra memory allocation.

Return a pointer to the first byte of a memory block suitably aligned and sufficient to hold an array of *n* objects of type *T*.

Parameters

- *n*: the number of objects to allocate storage for.

- `hint`: unused parameter provided for standard compliance.

void **deallocate** (pointer *p*, size_type *n*)

Deallocates the storage referenced by the pointer *p*, which must be a pointer obtained by an earlier call to *allocate()*.

The argument *n* must be equal to the first argument of the call to *allocate()* that originally produced *p*; otherwise, the behavior is undefined.

Parameters

- *p*: pointer obtained from *allocate()*.
- *n*: number of objects earlier passed to *allocate()*.

auto **size_max** () const

Returns the maximum theoretically possible value of `\n`, for which the call `allocate(n, 0)` could succeed.

Return the maximum supported allocated size.

template <class U, class... Args>

void **construct** (U **p*, Args&&... *args*)

Constructs an object of type `\T` in allocated uninitialized memory pointed to by `\p`, using placement-new.

Parameters

- *p*: pointer to allocated uninitialized memory.
- *args*: the constructor arguments to use.

template <class U>

void **destroy** (U **p*)

Calls the destructor of the object pointed to by *p*.

Parameters

- *p*: pointer to the object that is going to be destroyed.

1.8.1 Comparison operators

template <class T1, size_t A1, class T2, size_t A2>

bool xsimd::operator==(const *aligned_allocator*<T1, A1> &*lhs*, const *aligned_allocator*<T2, A2> &*rhs*)

Compares two aligned memory allocator for equality.

Since allocators are stateless, return `true` iff `A1 == A2`.

Return true if the allocators have the same alignment.

Parameters

- *lhs*: *aligned_allocator* to compare.
- *rhs*: *aligned_allocator* to compare.

template <class T1, size_t A1, class T2, size_t A2>

bool xsimd::operator!=(const *aligned_allocator*<T1, A1> &*lhs*, const *aligned_allocator*<T2, A2> &*rhs*)

Compares two aligned memory allocator for inequality.

Since allocators are stateless, return `true` iff `A1 != A2`.

Return true if the allocators have different alignments.

Parameters

- lhs: *aligned_allocator* to compare.
- rhs: *aligned_allocator* to compare.

X

- xsimd::abs (C++ function), 27
- xsimd::acos (C++ function), 33
- xsimd::acosh (C++ function), 34
- xsimd::aligned_allocator (C++ class), 38
- xsimd::aligned_allocator::~~aligned_allocator (C++ function), 38
- xsimd::aligned_allocator::address (C++ function), 38
- xsimd::aligned_allocator::aligned_allocator (C++ function), 38
- xsimd::aligned_allocator::allocate (C++ function), 38
- xsimd::aligned_allocator::construct (C++ function), 39
- xsimd::aligned_allocator::deallocate (C++ function), 39
- xsimd::aligned_allocator::destroy (C++ function), 39
- xsimd::aligned_allocator::size_max (C++ function), 39
- xsimd::all (C++ function), 19
- xsimd::any (C++ function), 20
- xsimd::asin (C++ function), 33
- xsimd::asinh (C++ function), 34
- xsimd::atan (C++ function), 33
- xsimd::atan2 (C++ function), 33
- xsimd::atanh (C++ function), 34
- xsimd::batch (C++ class), 20
- xsimd::batch::batch (C++ function), 20
- xsimd::batch::load_aligned (C++ function), 20
- xsimd::batch::load_unaligned (C++ function), 21
- xsimd::batch::operator simd_data (C++ function), 20
- xsimd::batch::operator= (C++ function), 20
- xsimd::batch::operator[] (C++ function), 21
- xsimd::batch::store_aligned (C++ function), 21
- xsimd::batch::store_unaligned (C++ function), 21
- xsimd::batch_bool (C++ class), 17
- xsimd::batch_bool::batch_bool (C++ function), 18
- xsimd::batch_bool::operator simd_data (C++ function), 18
- xsimd::batch_bool::operator= (C++ function), 18
- xsimd::bitwise_andnot (C++ function), 19, 23
- xsimd::cbrt (C++ function), 32
- xsimd::ceil (C++ function), 36
- xsimd::clip (C++ function), 29
- xsimd::cos (C++ function), 33
- xsimd::cosh (C++ function), 34
- xsimd::erf (C++ function), 35
- xsimd::erfc (C++ function), 35
- xsimd::exp (C++ function), 30
- xsimd::exp10 (C++ function), 30
- xsimd::exp2 (C++ function), 30
- xsimd::expm1 (C++ function), 30
- xsimd::fabs (C++ function), 27
- xsimd::fdim (C++ function), 29
- xsimd::floor (C++ function), 36
- xsimd::fma (C++ function), 28
- xsimd::fmax (C++ function), 29
- xsimd::fmin (C++ function), 29
- xsimd::fmod (C++ function), 27
- xsimd::fms (C++ function), 28
- xsimd::fnma (C++ function), 28
- xsimd::fnms (C++ function), 28
- xsimd::hypot (C++ function), 32
- xsimd::isfinite (C++ function), 37
- xsimd::isinf (C++ function), 37
- xsimd::isnan (C++ function), 37
- xsimd::lgamma (C++ function), 35
- xsimd::load_aligned (C++ function), 25
- xsimd::load_simd (C++ function), 26
- xsimd::load_unaligned (C++ function), 25
- xsimd::log (C++ function), 31
- xsimd::log10 (C++ function), 31
- xsimd::log1p (C++ function), 31
- xsimd::log2 (C++ function), 31
- xsimd::max (C++ function), 29
- xsimd::min (C++ function), 28
- xsimd::nearbyint (C++ function), 36
- xsimd::operator
 - = (C++ function), 19, 22, 39
 - (C++ function), 17
- xsimd::operator* (C++ function), 15, 21
- xsimd::operator+ (C++ function), 14, 21
- xsimd::operator- (C++ function), 14, 15, 21

xsimd::operator/ (C++ function), 16, 22
xsimd::operator== (C++ function), 19, 22, 39
xsimd::operator& (C++ function), 18, 23
xsimd::operator&& (C++ function), 10
xsimd::operator^ (C++ function), 18, 23
xsimd::operator~ (C++ function), 18, 23
xsimd::operator| (C++ function), 18, 23
xsimd::operator|| (C++ function), 10, 11
xsimd::operator> (C++ function), 16
xsimd::operator>= (C++ function), 16
xsimd::operator< (C++ function), 22
xsimd::operator<= (C++ function), 22
xsimd::operator<< (C++ function), 17
xsimd::pow (C++ function), 32
xsimd::remainder (C++ function), 27
xsimd::rint (C++ function), 36
xsimd::round (C++ function), 36
xsimd::select (C++ function), 24
xsimd::set_simd (C++ function), 24
xsimd::simd_batch (C++ class), 11
xsimd::simd_batch::operator() (C++ function), 12
xsimd::simd_batch::operator*= (C++ function), 12
xsimd::simd_batch::operator++ (C++ function), 14
xsimd::simd_batch::operator+= (C++ function), 12
xsimd::simd_batch::operator- (C++ function), 14
xsimd::simd_batch::operator-= (C++ function), 12
xsimd::simd_batch::operator/= (C++ function), 13
xsimd::simd_batch::operator&= (C++ function), 13
xsimd::simd_batch::operator^= (C++ function), 13
xsimd::simd_batch::operator|= (C++ function), 13
xsimd::simd_batch_bool (C++ class), 9
xsimd::simd_batch_bool::operator() (C++ function), 10
xsimd::simd_batch_bool::operator&= (C++ function), 9
xsimd::simd_batch_bool::operator^= (C++ function), 9
xsimd::simd_batch_bool::operator|= (C++ function), 9
xsimd::sin (C++ function), 32
xsimd::sinh (C++ function), 34
xsimd::sqrt (C++ function), 32
xsimd::store_aligned (C++ function), 25
xsimd::store_simd (C++ function), 26, 27
xsimd::store_unaligned (C++ function), 25
xsimd::tan (C++ function), 33
xsimd::tanh (C++ function), 34
xsimd::tgamma (C++ function), 35
xsimd::trunc (C++ function), 36