Data-Parallel Vector Types & Operations

ABSTRACT

This paper describes class templates for portable data-parallel (e.g. SIMD) programming via vector types.
Remarks

- This document talks about “vector” types/objects. In general this will not refer to the `std::vector` class template. References to the container type will explicitly call out the `std` prefix to avoid confusion.

- In the following, $W_T$ denotes the number of scalar values (width) in a vector of type $T$ (sometimes also called the number of SIMD lanes).


- This paper is not supposed to specify a complete API for data-parallel types and operations. It is meant as a useful starting point. Once the foundation is settled on, higher level APIs will be proposed.

Changelog

1.1 Changes from r0

Previous revision: [P0214R0].

- Extended the `datapar_abi` tag types with a `fixed_size<N>` tag to handle arbitrarily sized vectors (4.1.1.1).

- Converted `memory_alignment` into a non-member trait (4.1.1.2).

- Extended implicit conversions to handle `datapar_abi::fixed_size<N>` (4.1.2.2).

- Extended binary operators to convert correctly with `datapar_abi::fixed_size<N>` (4.1.3.1).

- Dropped the section on “`datapar logical operators`”. Added a note that the omission is deliberate (4.1.3.3).

- Added logical and bitwise operators to `mask` (4.1.5.1).

- Modified `mask` compares to work better with implicit conversions (4.1.5.3).

- Modified `where` to support different Abi tags on the `mask` and `datapar` arguments (4.1.5.5).
• Converted the load functions to non-member functions. SG1 asked for guidance from LEWG whether a load-expression or a template parameter to load is more appropriate.

• Converted the store functions to non-member functions to be consistent with the load functions.

• Added a note about masked stores not invoking out-of-bounds accesses for masked-off elements of the vector.

• Converted the return type of `datapar::operator[]` to return a smart reference instead of an lvalue reference.

• Modified the wording of `mask::operator[]` to match the reference type returned from `datapar::operator[]`.

• Added non-trig/pow/exp/log math functions on `datapar`.

• Added discussion on defaulting load/store flags.

• Added sum, product, min, and max reductions for `datapar`.

• Added load constructor.

• Modified the wording of `native_handle()` to make the existence of the functions implementation-defined, instead of only the return type. Added a section in the discussion (cf. Section 5.8).

• Fixed missing flag objects.

1.2 changes from r1

Previous revision: [P0214R1].

• Fixed converting constructor synopsis of `datapar` and `mask` to also allow varying Abi types.

• Modified the wording of `mask::native_handle()` to make the existence of the functions implementation-defined.

• Updated the discussion of member types to reflect the changes in R1.

• Added all previous SG1 straw poll results.

• Fixed `commonabi` to not invent native Abi that makes the operator ill-formed.
• Dropped table of math functions.
• Be more explicit about the implementation-defined Abi types.
• Discussed resolution of the `datapar_abi::fixed_size<N>` design (5.7.4).
• Made the `compatible` and `native` ABI aliases depend on `T` (4.1.1.1).
• Added `max_fixed_size` constant (4.1.1.1 p.4).
• Added masked loads.
• Added rationale for return type of `datapar::operator-()` (5.10).

— SG1 guidance:
• Dropped the default load / store flags.
• Renamed the (un)aligned flags to `element_aligned` and `vector_aligned`.
• Added an `overaligned<N>` load / store flag.
• Dropped the ampersand on `native_handle` (no strong preference).
• Completed the set of math functions (i.e. add trig, log, and exp).

— LEWG (small group) guidance:
• Dropped `native_handle` and add non-normative wording for supporting `static_cast` to implementation-defined SIMD extensions.
• Dropped non-member load and store functions. Instead have `copy_from` and `copy_to` member functions for loads and stores. (4.1.2.3, 4.1.2.4, 4.1.4.3, 4.1.4.4) (Did not use the load and store names because of the unfortunate inconsistency with `std::atomic`.)
• Added algorithm overloads for `datapar` reductions. Integrate with `where` to enable masked reductions. (4.1.3.5) This made it necessary to spell out the class `where_expression`. 
1.3 Changes from r2

Previous revision: [P0214R2].

✓ Fixed return type of masked reduce (4.1.3.5).
✓ Added binary min, max, minmax, and clamp (4.1.3.7).
✓ Moved member min and max to non-member hmin and hmax, which cannot easily be optimized from reduce, since no function object such as std::plus exists (4.1.3.5).
✓ Fixed neutral element of masked hmin/hmax and drop UB (4.1.3.5).
✓ Removed remaining reduction member functions in favor of non-member reduce (as requested by LEWG).
✓ Replaced init parameter of masked reduce with neutral_element (4.1.3.5).
✓ Extend where_expression to support const datapar objects (4.1.5.5).
✓ Fixed missing explicit keyword on mask(bool) constructor (4.1.4.2).
✓ Made binary operators for datapar and mask friend functions of datapar and mask, simplifying the SFINAE requirements considerably (4.1.3.1, 4.1.5.1).
✓ Restricted broadcasts to only allow non-narrowing conversions (4.1.2.2).
✓ Restricted datapar to datapar conversions to only allow non-narrowing conversions with fixed_size ABI (4.1.2.2).
✓ Added generator constructor (as discussed in LEWG in Issaquah) (4.1.2.2).
✓ Renamed copy_from to memload and copy_to to memstore. (4.1.2.3, 4.1.2.4, 4.1.4.3, 4.1.4.4)
✓ Documented effect of overaligned_tag<N> as Flags parameter to load/store. (4.1.2.3, 4.1.2.4, 4.1.4.3, 4.1.4.4)
✓ Clarified cv requirements on T parameter of datapar and mask.
✓ Allowed all implicit mask conversions with fixed_size ABI and equal size (4.1.4.2).
✓ Made increment and decrement of where_expression return void.
✓ Added static_datapar_cast for simple casts (4.1.3.6).
✓ Clarified default constructor (4.1.2.1, 4.1.2.1).

✓ Clarified datapar and mask with invalid template parameters to be complete types with deleted constructors, destructor, and assignment (4.1.2.1, 4.1.2.1).

✓ Wrote a new subsection for a detailed description of where_expression (4.1.1.3).

✓ Moved masked loads and stores from datapar and mask to where_expression (4.1.1.3). This required two more overloads of where to support value objects of type mask (4.1.5.5).

✓ Removed where_expression::operator! (4.1.1.3).

✓ Added aliases native_datapar, native_mask, fixed_size_datapar, fixed_size_mask (4.1.1).

✓ Removed bool overloads of mask reductions awaiting a better solution (4.1.5.4).

✓ Removed special math functions with f and l suffix and l and ll prefix (4.1.3.8).

✓ Modified special math functions with mixed types to use fixed_size instead of abi_for_size (4.1.3.8).

✓ Added simple ABI cast functions to_fixed_size, to_native, and to_compatible (4.1.3.6).

2 STRAW POLLS

2.1 sg1 at chicago 2013

Poll: Pursue SIMD/data parallel programming via types?

<table>
<thead>
<tr>
<th>SF</th>
<th>F</th>
<th>N</th>
<th>A</th>
<th>SA</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>8</td>
<td>5</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

2.2 sg1 at urbana 2014

Poll: SF = ABI via namespace, SA = ABI as template parameter

<table>
<thead>
<tr>
<th>SF</th>
<th>F</th>
<th>N</th>
<th>A</th>
<th>SA</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>6</td>
<td>11</td>
<td>2</td>
</tr>
</tbody>
</table>

Poll: Apply size promotion to vector operations? SF = shortv + shortv = intv

<table>
<thead>
<tr>
<th>SF</th>
<th>F</th>
<th>N</th>
<th>A</th>
<th>SA</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2</td>
<td>0</td>
<td>6</td>
<td>11</td>
</tr>
</tbody>
</table>
Poll: Apply “sign promotion” to vector operations? SF = ushortv + shortv = ushortv;
SA = no mixed signed/unsigned arithmetic

<table>
<thead>
<tr>
<th></th>
<th>F</th>
<th>N</th>
<th>A</th>
<th>SA</th>
</tr>
</thead>
<tbody>
<tr>
<td>SF</td>
<td>1</td>
<td>5</td>
<td>5</td>
<td>7</td>
</tr>
</tbody>
</table>

2.3          SG1 AT LENEXA 2015

Poll: Make vector types ready for LEWG with arithmetic, compares, write-masking, and math?

<table>
<thead>
<tr>
<th></th>
<th>F</th>
<th>N</th>
<th>A</th>
<th>SA</th>
</tr>
</thead>
<tbody>
<tr>
<td>SF</td>
<td>10</td>
<td>6</td>
<td>2</td>
<td>0</td>
</tr>
</tbody>
</table>

2.4          SG1 AT JAX 2016

Poll: Should subscript operator return an lvalue reference?

<table>
<thead>
<tr>
<th></th>
<th>F</th>
<th>N</th>
<th>A</th>
<th>SA</th>
</tr>
</thead>
<tbody>
<tr>
<td>SF</td>
<td>0</td>
<td>6</td>
<td>10</td>
<td>2</td>
</tr>
</tbody>
</table>

Poll: Should subscript operator return a “smart reference”?

<table>
<thead>
<tr>
<th></th>
<th>F</th>
<th>N</th>
<th>A</th>
<th>SA</th>
</tr>
</thead>
<tbody>
<tr>
<td>SF</td>
<td>1</td>
<td>7</td>
<td>10</td>
<td>0</td>
</tr>
</tbody>
</table>

Poll: Specify datapar width using ABI tag, with a special template tag for fixed size.

<table>
<thead>
<tr>
<th></th>
<th>F</th>
<th>N</th>
<th>A</th>
<th>SA</th>
</tr>
</thead>
<tbody>
<tr>
<td>SF</td>
<td>3</td>
<td>7</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

Poll: Specify datapar width using <T, N, abi>, where abi is not specified by the user.

<table>
<thead>
<tr>
<th></th>
<th>F</th>
<th>N</th>
<th>A</th>
<th>SA</th>
</tr>
</thead>
<tbody>
<tr>
<td>SF</td>
<td>1</td>
<td>2</td>
<td>5</td>
<td>2</td>
</tr>
</tbody>
</table>

2.5          SG1 AT OLU 2016

Poll: Keep native_handle in the wording (dropping the ampersand in the return type)?

<table>
<thead>
<tr>
<th></th>
<th>F</th>
<th>N</th>
<th>A</th>
<th>SA</th>
</tr>
</thead>
<tbody>
<tr>
<td>SF</td>
<td>0</td>
<td>6</td>
<td>3</td>
<td>3</td>
</tr>
</tbody>
</table>

Poll: Should the interface provide a way to specify a number for over-alignment?

<table>
<thead>
<tr>
<th></th>
<th>F</th>
<th>N</th>
<th>A</th>
<th>SA</th>
</tr>
</thead>
<tbody>
<tr>
<td>SF</td>
<td>2</td>
<td>6</td>
<td>5</td>
<td>0</td>
</tr>
</tbody>
</table>
3 Introduction

3.1 SIMD registers and operations

Since many years the number of SIMD instructions and the size of SIMD registers have been growing. Newer microarchitectures introduce new operations for optimizing certain (common or specialized) operations. Additionally, the size of SIMD registers has increased and may increase further in the future.

The typical minimal set of SIMD instructions for a given scalar data type comes down to the following:

- Load instructions: load $W_T$ successive scalar values starting from a given address into a SIMD register.

- Store instructions: store from a SIMD register to $W_T$ successive scalar values at a given address.

- Arithmetic instructions: apply the arithmetic operation to each pair of scalar values in the two SIMD registers and store the results back to a SIMD register.

- Compare instructions: apply the compare operation to each pair of scalar values in the two SIMD registers and store the results back to a SIMD mask register.

- Bitwise instructions: bitwise operations on SIMD registers.
• Shuffle instructions: permutation and/or blending of scalars in (a) SIMD register(s).

The set of available instructions may differ considerably between different microarchitectures of the same CPU family. Furthermore there are different SIMD register sizes. Future extensions will certainly add more instructions and larger SIMD registers.

3.2 Motivation for data-parallel types

SIMD registers and operations are the low-level ingredients to efficient programming for SIMD CPUs. At a more abstract level this is not only about SIMD CPUs, but efficient data-parallel execution (CPUs, GPUs, possibly FPGAs and classical vector supercomputers). Operations on fundamental types in C++ form the abstraction for CPU registers and instructions. Thus, a data-parallel type (SIMD type) can provide the necessary interface for writing software that can utilize data-parallel hardware efficiently. Higher-level abstractions can be built on top of these types. Note that if a low-level access to SIMD is not provided, users of C++ are either constrained to work within the limits of the provided abstraction or resort to non-portable extensions, such as SIMD intrinsics.

In some cases the compiler might generate better code if only the intent is stated instead of an exact sequence of operations. Therefore, higher-level abstractions might seem preferable to low-level SIMD types. In my experience this is a non-issue because programming with SIMD types makes intent very clear and compilers can optimize sequences of SIMD operations just like they can for scalar operations. SIMD types do not lead to an easy and obvious answer for efficient and easily usable data structures, though. But, in contrast to vector loops, SIMD types make unsuitable data structures glaringly obvious and can significantly support the developer in creating more suitable data layouts.

One major benefit from SIMD types is that the programmer can gain an intuition for SIMD. This subsequently influences further design of data structures and algorithms to better suit SIMD architectures.

There are already many users of SIMD intrinsics (and thus a primitive form of SIMD types). Providing a cleaner and portable SIMD API would provide many of them with a better alternative. Thus, SIMD types in C++ would capture and improve on widespread existing practice.

The challenge remains in providing portable SIMD types and operations.
**3.3 Problem**

C++ has no means to use SIMD operations directly. There are indirect uses through automatic loop vectorization or optimized algorithms (that use extensions to C/C++ or assembly for their implementation).

All compiler vendors (that I worked with) add intrinsics support to their compiler products to make SIMD operations accessible from C. These intrinsics are inherently not portable and most of the time very directly bound to a specific instruction. (Compilers are able to statically evaluate and optimize SIMD code written via intrinsics, though.)

**4 Wording**

The following is a draft of possible wording that defines a basic set of data-parallel types and operations.

**4.1 Data-Parallel Types**

```cpp
namespace std {
    namespace experimental {
        namespace datapar_abi {
            struct scalar {}; // always present
            template <int N> struct fixed_size {}; // always present
            constexpr int max_fixed_size = implementation_defined;
            // implementation-defined tag types, e.g. sse, avx, neon, altivec, ...
            template <typename T> using compatible = implementation_defined; // always present
            template <typename T> using native = implementation_defined; // always present
        }
        namespace flags {
            struct element_aligned_tag {};
            struct vector_aligned_tag {};
            template <std::align_val_t> struct overaligned_tag {};
            constexpr element_aligned_tag element_aligned{};
            constexpr vector_aligned_tag vector_aligned{};
            template <std::align_val_t N> constexpr overaligned_tag<N> overaligned = {};
        }
    }
}

// traits [datapar traits]
template <class T> struct is_datapar;
template <class T> constexpr bool is_datapar_v = is_datapar<T>::value;

template <class T> struct is_mask;
template <class T> constexpr bool is_mask_v = is_mask<T>::value;
```
template<class T, size_t N> struct abi_for_size {
    typedef implementation_defined type;
};
template<class T, size_t N> using abi_for_size_t = typename abi_for_size<T, N>::type;

template<class T, class Abi = datapar_abi::compatible<T>> struct datapar_size :
    public integral_constant<size_t, implementation_defined> {};
constexpr size_t datapar_size_v = datapar_size<T, Abi>::value;

// class template datapar [datapar]
template<class T, class Abi = datapar_abi::compatible<T>> class datapar;
template<class T> using native_datapar = datapar<T, datapar_abi::native<T>>;
template<class T, int N> using fixed_size_datapar = datapar<T, datapar_abi::fixed_size<N>>;

// class template mask [mask]
template<class T, class Abi = datapar_abi::compatible<T>> class mask;
template<class T> using native_mask = mask<T, datapar_abi::native<T>>;
template<class T, int N> using fixed_size_mask = mask<T, datapar_abi::fixed_size<N>>;

// casts [datapar.casts]
template<class T, class U, class A> datapar<T, /* see below */> static_datapar_cast(const datapar<U, A>&);
template<class T, class A> datapar<T, datapar_abi::fixed_size<datapar_size_v<T, A>>> to_fixed_size(
    const datapar<T, A>&);
template<class T, class A> mask<T, datapar_abi::fixed_size<datapar_size_v<T, A>>> to_fixed_size(
    const mask<T, A>&);
template<class T, size_t N> datapar<T, datapar_abi::native<T>> to_native(
    const datapar<T, datapar_abi::fixed_size<N>>&);
template<class T, size_t N> mask<T, datapar_abi::native<T>> to_native(
    const mask<T, datapar_abi::fixed_size<N>>&);
template<class T, size_t N> datapar<T, datapar_abi::compatible<T>> to_compatible(
    const datapar<T, datapar_abi::fixed_size<N>>&);
template<class T, size_t N> mask<T, datapar_abi::compatible<T>> to_compatible(
    const mask<T, datapar_abi::fixed_size<N>>&);

template<class T, class U, class... Us>
    conditional_t<(T::size() == (U::size() + Us::size()...)), T, array<T, (U::size() + Us::size()...)/T::size()>> datapar_cast(U, Us...);

// reductions [mask.reductions]
template<class T, class Abi> bool all_of(mask<T, Abi>);
template<class T, class Abi> bool any_of(mask<T, Abi>);
template<class T, class Abi> bool none_of(mask<T, Abi>);
template<class T, class Abi> bool some_of(mask<T, Abi>);
template<class T, class Abi> int popcount(mask<T, Abi>);
template<class T, class Abi> int find_first_set(mask<T, Abi>);
template <class T, class Abi> int find_last_set(mask<T, Abi>);

// masked assignment [mask.where]

template <class M, class T> class where_expression {
    public:
        const M &mask; // exposition only
        T &data; // exposition only
        where_expression(const M & m, T & t); // exposition only

    where_expression(const where_expression & w) = delete;
    where_expression & operator=(const where_expression & w) = delete;

    template <class U> void operator=(U &&x);
    template <class U> void operator+=(U &&x);
    template <class U> void operator-=(U &&x);
    template <class U> void operator*=(U &&x);
    template <class U> void operator/=(U &&x);
    template <class U> void operator%=(U &&x);
    template <class U> void operator&=(U &&x);
    template <class U> void operator|=(U &&x);
    template <class U> void operator^=(U &&x);
    template <class U> void operator<<=( U &&x);
    template <class U> void operator>>=( U &&x);
    void operator++();
    void operator++(int);
    void operator--();
    void operator--(int);
    remove_const_t <T> operator~() const;

    template <class U, class Flags> void memload(const U *mem, Flags);
    template <class U, class Flags> void memstore(U *mem, Flags) const;
    
    template <class T, class A>
    where_expression=mask<T, A>, datapar<T, A>> where{
        const typename datapar<T, A>::mask_type & mask<T, A> & w);
        where_expression(const mask<T, A> & mask<T, A> & w);

    template <class T, class A>
    const where_expression=mask<T, A>, const datapar<T, A> & where{
        const typename datapar<T, A>::mask_type & mask<T, A> & w);

    template <class T, class A>
    const where_expression=mask<T, A>, mask<T, A>> where(const remove_const_t<mask<T, A>> & mask<T, A> & w);

    template <class T, class A>
    const where_expression=mask<T, A>, const mask<T, A> & where{
        const remove_const_t<mask<T, A>> & mask<T, A> & w);

    template <class T> where_expression=boolean, T> where( boolean k, T & d);

    // reductions [datapar.reductions]

    template <class BinaryOperation = std::plus<> , class T, class Abi>
    T reduce(const datapar<T, Abi> & m, BinaryOperation binary_op = BinaryOperation());
    template <class BinaryOperation = std::plus<> , class M, class V>
    typename V::value_type reduce(const where_expression<M, V> & w,
        BinaryOperation binary_op = BinaryOperation());

11
The header `<datapar>` defines the class templates (datapar, mask, and where_expression), several tag types, and a series of related function templates for concurrent manipulation of the values in datapar and mask objects.

4.1.1.1 datapar ABI tags

```cpp
namespace datapar_abi {
    struct scalar {}; // always present
    template <int N> struct fixed_size {}; // always present
    constexpr int max_fixed_size = implementation_defined;
    // implementation-defined tag types, e.g. sse, avx, neon, altivec, ...
    template <typename T> using compatible = implementation_defined; // always present
    template <typename T> using native = implementation_defined; // always present
}
```

The ABI types are tag types to be used as the second template argument to datapar and mask.

1. The scalar tag is present in all implementations and forces datapar and mask to store a single component (i.e. `datapar<T, datapar_abi::scalar>::size()` returns 1). [Note: scalar shall not be an alias for `fixed_size<1>`. — end note]

2. The fixed_size tag is present in all implementations. Use of `datapar_abi::fixed_size<N>` forces datapar and mask to store and manipulate N components (i.e. `datapar<T, datapar_abi::fixed_size<N>>::size()` returns N). An implementation must support at least any `N ∈ [1 ... 32]`. Additionally, an implementation must support any `N ∈ {datapar<U>::size(), ∀U ∈ {arithmetic types}}`. [Note: An implementation may choose to not ensure ABI compatibility for datapar and mask instantiations using the same `datapar_abi::fixed_size<N>` tag. In case of ABI compatibility between differently compiled translation units, the efficiency of `datapar<T, Abi>` is likely to be better than for `datapar<T, fixed_size<datatype_size_v<T, Abi>>>` (with Abi not a instance of `datapar_abi::fixed_size`). — end note]

3. The value of `max_fixed_size` declares that an instance of `datapar<T, fixed_size<N>>` with `N ≤ max_fixed_size` is supported by the implementation. [Note: It is still possible for an implementation to support `datapar<U, fixed_size<K>>` with `K > max_fixed_size`. — end note]

An implementation may choose to implement data-parallel execution for many different targets. An additional implementation-defined tag type should be added to the `datapar_abi` namespace, for each target.
the implementation supports. [ Note: There can certainly be
more than one tag type per (micro-)architecture, e.g. to
support different vector lengths or partial register
usage. — end note ] All tag types an
implementation supports shall be present independent of the
target architecture determined at invocation of the
compiler.

The datapar_abi::compatible<T> tag is defined by the
implementation to alias the tag type with the
most efficient data parallel execution for the element type T that
ensures the highest compatibility on the

target architecture.

The datapar_abi::native<T> tag is defined by the
implementation to alias the tag type with the
most efficient data parallel execution for the element type T that is
supported on the target system. [ Example:
Consider a target with the implementation-defined ABI tags simd128
and simd256 where hardware sup-
port for simd256 only exists for floating-point types. In this case the
native<T> alias equals simd256 if
T is a floating-point type and simd128 otherwise. — end example ]

4.1.1.2 datapar type traits

\[ \text{template } \langle \text{class } T \rangle \text{ struct is_datapar; } \]

The is_datapar type derives from true_type if \( T \) is an instance of the
datapar class template. Otherwise it derives from false_type.

\[ \text{template } \langle \text{class } T \rangle \text{ struct is_mask; } \]

The is_mask type derives from true_type if \( T \) is an instance of the
mask class template. Otherwise it derives from false_type.

\[ \text{template } \langle \text{class } T, \text{size_t } N \rangle \text{ struct abi_for_size } \{ \text{typedef implementation_defined type; } \}; \]

The abi_for_size class template defines the member type type to one of the
tag types in datapar_abi. If a tag type A exists that satisfies

- \( \text{datapar_size_v<T, A> == N,} \)
- A is a supported Abi parameter to datapar<T, Abi> for the current compilation target, and
- A is not datapar_abi::fixed_size<N>,

then the member type type is an alias for A. Otherwise type is an alias for
\( \text{datapar_abi::fixed_size<N>}. \)

abi_for_size<T, N>::type shall result in a substitution failure if \( T \) is not supported by datapar or if
\( N \) is not supported by the implementation (cf. [4.1.1.1 p.3]).

\[ \text{template } \langle \text{class } T, \text{class Abi = datapar_abi::compatible<T>> } \]

struct datapar_size : public integral_constant<size_t, implementation_defined> {};

The datapar_size class template inherits from integral_constant with a value that equals data-

\[ \text{datapar_size<T, Abi>::size()}. \]

datapar_size<T, Abi>::value shall result in a substitution failure if any of the template arguments T
or Abi are invalid template arguments to datapar.
template <class T, class U = typename T::value_type>
constexpr size_t memory_alignment = implementation_defined;

Requires: The template parameter T must be a valid instantiation of either the datapar or the mask class template.

Requires: The template parameter U must be a type supported by the load and store functions for T.

The value of memory_alignment<T, U> identifies the alignment restrictions on pointers used for (converting) loads and stores for the given type T on arrays of type U.

4.1.1.3 Class template where_expression

The class template where_expression<M, T> combines a predicate and a value object to implement an interface that restricts assignments and/or operations on the value object to the elements selected via the predicate.

The first template argument M must be cv-unqualified bool or a cv-unqualified mask instantiation.

The second template argument T must be a cv-unqualified or const qualified type T'. If M is bool, T' must be an arithmetic type. Otherwise, T' must either be M or M::datapar_type.

const M *mask; // exposition only
T &data; // exposition only
where_expression(const M &m, T &t); // exposition only

Effects: The implementation initializes a where_expression<M, T> object with a predicate of type M and a reference to a value object of type T.

Note: The predicate object may be copied by the constructor implementation. If T is const qualified the constructor may copy the value object.

Note: The following declarations refer to the predicate as data member mask and to the value reference as data member data.

template <class U> void operator=(U &&x);
template <class U> void operator+=(U &&x);
template <class U> void operator-=(U &&x);
template <class U> void operator*=(U &&x);
template <class U> void operator/=(U &&x);
template <class U> void operator%=(U &&x);
template <class U> void operator&=(U &&x);
template <class U> void operator|=(U &&x);
template <class U> void operator^=(U &&x);
template <class U> void operator<<=(U &&x);
template <class U> void operator>>=(U &&x);

Remarks: Each of these operators only participate in overload resolution if the indicated operator can be applied to objects of type T.

Effects: If M is bool, applies the indicated operator on data and forward<U>(x) unless mask is false. If M is not bool, applies the indicated operator on data and forward<U>(x) without modifying the elements data[i] where mask[i] is false for all i ∈ [0, M::size()).

Remarks: It is unspecified whether the arithmetic/bitwise operation, which is implied by a compound assignment operator, is executed on all elements or only on the ones written back.
**void operator++();**

**void operator++(int);**

**void operator--();**

**void operator--(int);**

10 **Remarks:** Each of these operators only participate in overload resolution if the indicated operator can be applied to objects of type `T`.

11 **Effects:** If `M` is `bool`, applies the indicated operator on `data` unless `mask` is `false`. If `M` is not `bool`, applies the indicated operator on `data` without modifying the elements `data[i]` where `mask[i]` is false for all `i ∈ [0, M::size())`.

12 **Note:** It is unspecified whether the inc-/decrement operation is executed on all elements or only on the ones written back.

**remove_const_t <T> operator()-();**

13 **Remarks:** This operator only participates in overload resolution if the indicated operator can be applied to objects of type `T`.

14 **Returns:** If `M` is `bool`, `-data` if `mask` is `true`, `data` otherwise. If `M` is not `bool`, returns an object with the `i`-th element initialized to `-data[i]` if `mask[i]` is `true` and `data[i]` otherwise for all `i ∈ [0, M::size())`.

**template <class U, class Flags> void memload(const U *mem, Flags);**

15 **Remarks:** If `M` is `bool`, the function only participates in overload resolution if `U` is `bool`.

16 **Remarks:** If `T` is a mask instantiation, the function only participates in overload resolution if `U` is `bool`.

17 **Effects:** If `M` is `bool`, assign `mem[0]` to `data` unless `mask` is `false`. If `M` is not `bool`, replace the elements of `data` where `mask[i]` is `true` such that the `i`-th element is assigned with `static_cast<typename T::value_type>{mem[i]}` for all `i ∈ [0, M::size())`.

18 **Remarks:** If `M` is not `bool` and if the largest `i` where `mask[i]` is `true` is greater than the number of values pointed to by `mem`, the behavior is undefined.

19 **Remarks:** If the `Flags` template parameter is of type `flags::vector_aligned_tag` and the pointer value is not a multiple of `memory_alignment<T, U>`, the behavior is undefined. If the `Flags` template parameter is of type `flags::overaligned_tag<N>` and the pointer value is not a multiple of `N`, the behavior is undefined.

**template <class U, class Flags> void memstore(U *mem, Flags) const;**

20 **Effects:** If `M` is `bool`, assign `data` to `mem[0]` unless `mask` is `false`.

21 **Remarks:** If `T` is a (const qualified) mask instantiation, the function only participates in overload resolution if `U` is `bool`. If `M` is not `bool`, copies the elements `data[i]` where `mask[i]` is `true` as if `mem[i] = static_cast<U>{data[i]}` for all `i ∈ [0, M::size())`.

22 **Remarks:** If `M` is not `bool` and if the largest `i` where `mask[i]` is `true` is greater than the number of values pointed to by `mem`, the behavior is undefined.

23 **Remarks:** If the `Flags` template parameter is of type `flags::vector_aligned_tag` and the pointer value is not a multiple of `memory_alignment<remove_const_t<T>, U>`, the behavior is undefined. If the `Flags` template parameter is of type `flags::overaligned_tag<N>` and the pointer value is not a multiple of `N`, the behavior is undefined.
4.1.2 Class template `datapar`

4.1.2.1 Class template `datapar` overview

```cpp
namespace std {
    namespace experimental {
        template <class T, class Abi> class datapar {
            public:
                typedef T value_type;
                typedef implementation_defined reference;
                typedef mask<T, Abi> mask_type;
                typedef size_t size_type;
                typedef Abi abi_type;

                static constexpr size_type size();

                datapar() = default;
                datapar(const datapar &) = default;
                datapar(datapar &&) = default;
                datapar & operator=(const datapar &) = default;
                datapar & operator=(datapar &&) = default;

                // implicit broadcast constructor
                template <class U> datapar(const U &);

                // implicit type conversion constructor
                template <class U> datapar(const datapar<U, datapar_abi::fixed_size<size()>> &);

                // generator constructor
                template <class G> datapar(G &&gen);

                // load constructor
                template <class U, class Flags> datapar(const U *mem, Flags);
                template <class U, class Flags> datapar(const U *mem, mask_type k, Flags);

                // loads [datapar.load]
                template <class U, class Flags> void memload(const U *mem, Flags);

                // stores [datapar.store]
                template <class U, class Flags> void memstore(U *mem, Flags) const;

                // scalar access [datapar.subscr]
                reference operator[](size_type);
                value_type operator[](size_type) const;

                // unary operators [datapar.unary]
                datapar &operator++();
                datapar &operator++(int);
                datapar &operator--();
                datapar &operator--(int);
                mask_type operator!() const;
                datapar operator~() const;
                datapar operator+(int) const;
                datapar operator-(int) const;

                // binary operators [datapar.binary]
        }
    }
}
```
The class template `datapar<T, Abi>` is a one-dimensional smart array. In contrast to `valarray` (26.6), the number of elements in the array is determined at compile time, according to the `Abi` template parameter.

The first template argument `T` must be a cv-unqualified integral or floating-point type (3.9.1 [basic.fundamental]). The type `bool` is not allowed.

The second template argument `Abi` must be a tag type from the `datapar_abi` namespace.

If any of the template arguments does not conform to the above requirements, the resulting class shall be a complete type with deleted default constructor, deleted destructor, deleted copy constructor, and deleted copy assignment.

Default initialization performs no initialization of the elements; value-initialization initializes each element with `T()`. [Note: Thus, default initialization leaves the elements in an indeterminate state. — end note]

```
static constexpr size_type size();
```

Returns: the number of elements stored in objects of the given `datapar<T, Abi>` type.

Note: Implementations are encouraged to enable `static_cast`ing from/to (an) implementation-defined SIMD type(s). This would add one or more of the following declarations to class `datapar`:

```
explicit operator implementation_defined() const;
explicit datapar<const implementation_defined &init>;
```
template <class U> datapar(const U &);

Remarks: This constructor shall not participate in overload resolution unless either:

- \( U \) is a fundamental arithmetic type and every possible value of type \( U \) can be represented with type value_type,
- or \( U \) is not a fundamental arithmetic type and is implicitly convertible to value_type,
- or \( U \) is int,
- or \( U \) is unsigned int and value_type is an unsigned integral type.

Effects: Constructs an object with each element initialized to the value of the argument.

template <class U> datapar(const datapar<U, datapar_abi::fixed_size<size()>>& x);

Remarks: This constructor shall not participate in overload resolution unless

- abi_type equals datapar_abi::fixed_size<size()>,
- and every possible value of \( U \) can be represented with type value_type,
- and, if both \( U \) and value_type are integral, the integer conversion rank \( [N4618, (4.15)] \) of value_type is greater than the integer conversion rank of \( U \).

Effects: Constructs an object where the \( i \)-th element equals static_cast<T>(x[i]) for all \( i \in [0, size()) \).

template <class G> datapar(G &gen);

Remarks: This constructor shall not participate in overload resolution unless \( \text{gen}(\text{declval<integral_constant<size_t, 0>>()}) \) is well-formed with a return type of value_type.

Effects: Constructs an object where the \( i \)-th element is initialized to \( \text{gen}(\text{integral_constant<size_t, i>())} \).

Remarks: The order of calls to \( \text{gen} \) is unspecified.

template <class U, class Flags> datapar(const U *mem, Flags);

Effects: Constructs an object where the \( i \)-th element is initialized to static_cast<T>(mem[i]) for all \( i \in [0, size()) \).

Remarks: If size() returns a value greater than the number of values pointed to by the first argument, the behavior is undefined.

Remarks: If the Flags template parameter is of type flags::vector_aligned_tag and the pointer value is not a multiple of memory_alignment<datapar, U>, the behavior is undefined. If the Flags template parameter is of type flags::overaligned_tag<N> and the pointer value is not a multiple of N, the behavior is undefined.

template <class U, class Flags> datapar(const U *mem, mask_type k, Flags);
### 4.1.2.3 datapar load function

```
template <class U, class Flags> void memload(const U *mem, Flags);
```

**Effects:** Replaces the elements of the `datapar` object such that the `i`-th element is assigned with `static_cast<T>(mem[i])` for all `i ∈ [0, size())`.

**Remarks:** If `size()` returns a value greater than the number of values pointed to by the first argument, the behavior is undefined.

**Remarks:** If the `Flags` template parameter is of type `flags::vector_aligned_tag` and the pointer value is not a multiple of `memory_alignment<datapar, U>`, the behavior is undefined. If the `Flags` template parameter is of type `flags::overaligned_tag<N>` and the pointer value is not a multiple of `N`, the behavior is undefined.

### 4.1.2.4 datapar store function

```
template <class U, class Flags> void memstore(U *mem, Flags);
```

**Effects:** Copies all `datapar` elements as if `mem[i] = static_cast<U>(operator[](i))` for all `i ∈ [0, size())`.

**Remarks:** If `size()` returns a value greater than the number of values pointed to by `mem`, the behavior is undefined.

**Remarks:** If the `Flags` template parameter is of type `flags::vector_aligned_tag` and the pointer value is not a multiple of `memory_alignment<datapar, U>`, the behavior is undefined. If the `Flags` template parameter is of type `flags::overaligned_tag<N>` and the pointer value is not a multiple of `N`, the behavior is undefined.

### 4.1.2.5 datapar subscript operators

```
reference operator[](size_type i);
```

**Returns:** A temporary object with the following properties:

- **Remarks:** The object is neither `DefaultConstructible`, `CopyConstructible`, `MoveConstructible`, `CopyAssignable`, nor `MoveAssignable`.
• Remarks: Assignment, compound assignment, increment, and decrement operators only participate in overload resolution if called in rvalue context and the corresponding operator of type `value_-
type` is usable.

• Effects: The assignment, compound assignment, increment, and decrement operators execute the indicated operation on the i-th element in the `datapar` object.

• Effects: Conversion to `value_type` returns a copy of the i-th element.

```
value_type operator[](size_type) const;

Returns: A copy of the i-th element.
```

4.1.2.6 `datapar` unary operators

```
datapar &operator++();
```

1 Effects: Increments every element of `*this` by one.

2 Returns: An lvalue reference to `*this` after incrementing.

3 Remarks: Overflow semantics follow the same semantics as for T.

```
datapar operator++(int);
```

4 Effects: Increments every element of `*this` by one.

5 Returns: A copy of `*this` before incrementing.

6 Remarks: Overflow semantics follow the same semantics as for T.

```
datapar &operator--();
```

7 Effects: Decrements every element of `*this` by one.

8 Returns: An lvalue reference to `*this` after decrementing.

9 Remarks: Underflow semantics follow the same semantics as for T.

```
datapar operator--(int);
```

10 Effects: Decrements every element of `*this` by one.

11 Returns: A copy of `*this` before decrementing.

12 Remarks: Underflow semantics follow the same semantics as for T.

```
mask_type operator!() const;
```

13 Returns: A mask object with the i-th element set to !operator[](i) for all i ∈ [0, size()).

```
datapar operator-() const;
```

20
14 Requires: The first template argument \( T \) to \texttt{datapar} must be an integral type.

15 Returns: A \texttt{datapar} object where each bit is the inverse of the corresponding bit in \texttt{*this}.

16 Remarks: \texttt{datapar::operator\~()} shall not participate in overload resolution if \( T \) is a floating-point type.

\begin{verbatim}
  datapar operator\~{}() const;
\end{verbatim}

17 Returns: A copy of \texttt{*this}

\begin{verbatim}
  datapar operator\-{}() const;
\end{verbatim}

18 Returns: A \texttt{datapar} object where the \( i \)-th element is initialized to \(-\texttt{operator\[]()}(i)\) for all \( i \in [0, \texttt{size()}] \).

4.1.3 \texttt{datapar} non-member operations

4.1.3.1 \texttt{datapar} binary operators

\begin{verbatim}
friend datapar operator\+ (const datapar &\texttt{a}, const datapar &\texttt{b});
friend datapar operator\- (const datapar &\texttt{a}, const datapar &\texttt{b});
friend datapar operator\* (const datapar &\texttt{a}, const datapar &\texttt{b});
friend datapar operator/ (const datapar &\texttt{a}, const datapar &\texttt{b});
friend datapar operator\% (const datapar &\texttt{a}, const datapar &\texttt{b});
friend datapar operator& (const datapar &\texttt{a}, const datapar &\texttt{b});
friend datapar operator| (const datapar &\texttt{a}, const datapar &\texttt{b});
friend datapar operator^ (const datapar &\texttt{a}, const datapar &\texttt{b});
friend datapar operator<< (const datapar &\texttt{a}, const datapar &\texttt{b});
friend datapar operator>>(const datapar &\texttt{a}, const datapar &\texttt{b});
\end{verbatim}

1 Remarks: Each of these operators only participate in overload resolution if the indicated operator can be applied to objects of type \texttt{value_type}.

2 Returns: A \texttt{datapar} object initialized with the results of the component-wise application of the indicated operator.

4.1.3.2 \texttt{datapar} compound assignment

\begin{verbatim}
friend datapar \&operator\+= (datapar &\texttt{a}, const datapar &\texttt{b});
friend datapar \&operator\-= (datapar &\texttt{a}, const datapar &\texttt{b});
friend datapar \&operator\*= (datapar &\texttt{a}, const datapar &\texttt{b});
friend datapar \&operator\/= (datapar &\texttt{a}, const datapar &\texttt{b});
friend datapar \&operator\%= (datapar &\texttt{a}, const datapar &\texttt{b});
friend datapar \&operator\&= (datapar &\texttt{a}, const datapar &\texttt{b});
friend datapar \&operator\|= (datapar &\texttt{a}, const datapar &\texttt{b});
friend datapar \&operator\^= (datapar &\texttt{a}, const datapar &\texttt{b});
friend datapar \&operator<<= (datapar &\texttt{a}, const datapar &\texttt{b});
friend datapar \&operator>>= (datapar &\texttt{a}, const datapar &\texttt{b});
\end{verbatim}
Remarks: Each of these operators only participates in overload resolution if the indicated operator can be applied to objects of type `value_type`.

Effects: Each of these operators performs the indicated operator component-wise on each of the corresponding elements of the arguments.

Returns: A reference to the first argument.

4.1.3.3 datapar logical operators

Note: The omission of logical operators is deliberate.

4.1.3.4 datapar compare operators


friend mask_type operator||(const datapar &, const datapar &);
fraction mask_type operator!=(const datapar &, const datapar &);
fraction mask_type operator||(const datapar &, const datapar &);
fraction mask_type operator||(const datapar &, const datapar &);
fraction mask_type operator||(const datapar &, const datapar &);
fraction mask_type operator||(const datapar &, const datapar &);

Returns: A mask object initialized with the results of the component-wise application of the indicated operator.

4.1.3.5 datapar reductions

template <class BinaryOperation = std::plus<>, class T, class Abi>
T reduce(const datapar<T, Abi> &x, BinaryOperation binary_op = BinaryOperation());

Returns: \(\text{GENERALIZED}\_\text{SUM}(\text{binary}_\text{op}, \text{x.data}[i], \ldots)\) for all \(i \in [0, \text{size()}]\).

Requires: \(\text{binary}_\text{op}\) shall be callable with two arguments of type \(T\) or two arguments of type \(\text{datapar}<T, A1>\), where \(A1\) may be different to \(Abi\).

Note: This overload of \reduce\ requires a neutral value to enable a parallelized implementation: A temporary \(\text{datapar}\) object initialized with \(\text{neutral}_\text{element}\) is conditionally assigned from \(\text{x.data}\) using \(\text{x.mask}\). Subsequently, the parallelized reduction (without mask) is applied to the temporary object.

template <class T, class A> T hmin(const datapar<T, A> &x);

Returns: The value of an element \(x[j]\) for which \(x[j] \leq x[i]\) for all \(i \in [0, \text{size()}]\).
template <class M, class V> T hmin (const where_expression<M, V> &x);  

8 Returns: The value of an element x.data[j] for which x.mask[j] == true and x.data[j] <= x.data[i] for all i ∈ [0, size()).

9 Remarks: If all elements in x.mask are false, the return value is numeric_limits<V::value_type>::max().

template <class T, class A> T hmax (const datapar<T, A> &x);  

10 Returns: The value of an element x[j] for which x[j] >= x[i] for all i ∈ [0, size()).

template <class M, class V> T hmax (const where_expression<M, V> &x);  

11 Returns: The value of an element x.data[j] for which x.mask[j] == true and x.data[j] >= x.data[i] for all i ∈ [0, size()).

12 Remarks: If all elements in x.mask are false, the return value is numeric_limits<V::value_type>::min().

4.1.3.6 datapar casts

template <class T, class U, class A>
datapar<T, /* see below */ static_datapar_cast (const datapar<U, A> &x);  

1 Remarks: The return type is datapar<T, A> if either U and T are equal or U and T are integral types that only differ in signedness. Otherwise, the return type is datapar<T, datapar_abi::fixed_size<datapar<U, A>::size()>>.

2 Returns: A datapar object with the i-th element initialized to static_cast<T>(x[i]).

template <class T, class A>
 datapar<T, datapar_abi::fixed_size<datapar_size_v<T, A> N>> to_fixed_size (const datapar<T, A> &x);  

template <class T, class A>
 mask<T, datapar_abi::fixed_size<datapar_size_v<T, A> N>> to_fixed_size (const mask<T, A> &x);  

Returns: An object with the i-th element initialized to x[i].

template <class T, size_t N>
 datapar<T, datapar_abi::native<T>> to_native (const datapar<T, datapar_abi::fixed_size<N>> &x);  

template <class T, size_t N>
 mask<T, datapar_abi::native<T>> to_native (const mask<T, datapar_abi::fixed_size<N>> &x);  

Remarks: These functions only participate in overload resolution if datapar_size_v<T, datapar_abi::native<T>> is equal to N. Returns: An object with the i-th element initialized to x[i].

template <class T, size_t N>
 datapar<T, datapar_abi::compatible<T>> to_compatible (const datapar<T, datapar_abi::fixed_size<N>> &x);  

template <class T, size_t N>
 mask<T, datapar_abi::compatible<T>> to_compatible (const mask<T, datapar_abi::fixed_size<N>> &x);
**Remarks:** These functions only participate in overload resolution if `datapar_size_v<T, datapar_abi::compatible<T>>` is equal to N. **Returns:** An object with the i-th element initialized to x[i].

```cpp
template <class T, class U, class... Us>
conditional_t<((T::size() == (U::size() + Us::size())), T,
array<T, (U::size() + Us::size()) / T::size()>) > datapar_cast(U, Us...);
```

**Remarks:** The `datapar_cast` function only participates in overload resolution if all of the following hold:

- `is_datapar_v<T>`
- `is_datapar_v<U>`
- All types in the template parameter pack `Us` are equal to `U`.
- `U::size() + Us::size()...` is an integral multiple of `T::size()`.

**Returns:** A `datapar` object initialized with the converted values as one object of `T` or an array of `T`. All scalar elements `x_i` of the function argument(s) are converted as if `y_i = static_cast<typename T::value_type>(x_i)` is executed. The resulting `y_i` initialize the return object(s) of type `T`. [Note: For `T::size() == 2 * U::size()` the following holds: `datapar_cast<T>(x0, x1)[i] = static_cast<typename T::value_type>(array<U, 2>{x0, x1}[i / U::size()][i % U::size()])`. For `2 * T::size() == U::size()` the following holds: `datapar_cast<T>(x)[i][j] = static_cast<type-name T::value_type>(x[i * T::size() + j])`. — end note]

### 4.1.3.7 `datapar` algorithms

**Remarks:** An object with the i-th element initialized with the smaller value of `a[i]` and `b[i]` for all `i ∈ [0, size())`.

```cpp
template <class T, class A>
datapar<T, A> min(const datapar<T, A> &a, const datapar<T, A> &b);
```

**Returns:** An object with the i-th element initialized with the larger value of `a[i]` and `b[i]` for all `i ∈ [0, size())`.

```cpp
template <class T, class A>
datapar<T, A> max(const datapar<T, A> &a, const datapar<T, A> &b);
```

**Returns:** An object with the i-th element in the first pair member initialized with the smaller value of `a[i]` and `b[i]` for all `i ∈ [0, size())`. The i-th element in the second pair member is initialized with the larger value of `a[i]` and `b[i]` for all `i ∈ [0, size())`.

```cpp
template <class T, class A>
datapar<T, A> clamp(const datapar<T, A> &v, const datapar<T, A> &lo, const datapar<T, A> &hi);
```
Requires: No element in $lo$ shall be greater than the corresponding element in $hi$.

Returns: An object with the $i$-th element initialized with $lo[i]$ if $v[i]$ is smaller than $lo[i], hi[i]$ if $v[i]$ is larger than $hi[i], otherwise $v[i]$ for all $i \in \{0, \text{size}()\}$.

4.1.3.8 *datapar* math library

```cpp
namespace std {
    namespace experimental {
      template <class Abi>
        using intv = datapar<int, Abi>; // exposition only
      template <class Abi>
        using longv = datapar<long int, Abi>; // exposition only
      template <class Abi>
        using longlongv = datapar<long long int, Abi>; // exposition only
      template <class Abi>
        using floatv = datapar<float, Abi>; // exposition only
      template <class Abi>
        using doublev = datapar<double, Abi>; // exposition only
      template <class Abi>
        using ldoublev = datapar<long double, Abi>; // exposition only
      template <class T, class V>
        using samesize = fixed_size_datapar<T, V::size()>; // exposition only
      template <class Abi>
        floatv<Abi> acos(floatv<Abi> x);
      template <class Abi>
        doublev<Abi> acos(doublev<Abi> x);
      template <class Abi>
        ldoublev<Abi> acos(ldoublev<Abi> x);
      template <class Abi>
        floatv<Abi> asin(floatv<Abi> x);
      template <class Abi>
        doublev<Abi> asin(doublev<Abi> x);
      template <class Abi>
        ldoublev<Abi> asin(ldoublev<Abi> x);
      template <class Abi>
        floatv<Abi> atan(floatv<Abi> x);
      template <class Abi>
        doublev<Abi> atan(doublev<Abi> x);
      template <class Abi>
        ldoublev<Abi> atan(ldoublev<Abi> x);
      template <class Abi>
        floatv<Abi> atan2(floatv<Abi> y, floatv<Abi> x);
      template <class Abi>
        doublev<Abi> atan2(doublev<Abi> y, doublev<Abi> x);
      template <class Abi>
        ldoublev<Abi> atan2(ldoublev<Abi> y, ldoublev<Abi> x);
      template <class Abi>
        floatv<Abi> cos(floatv<Abi> x);
      template <class Abi>
        doublev<Abi> cos(doublev<Abi> x);
      template <class Abi>
        ldoublev<Abi> cos(ldoublev<Abi> x);
      template <class Abi>
        floatv<Abi> sin(floatv<Abi> x);
      template <class Abi>
        doublev<Abi> sin(doublev<Abi> x);
      template <class Abi>
        ldoublev<Abi> sin(ldoublev<Abi> x);
      template <class Abi>
        floatv<Abi> tan(floatv<Abi> x);
      template <class Abi>
        doublev<Abi> tan(doublev<Abi> x);
      template <class Abi>
        ldoublev<Abi> tan(ldoublev<Abi> x);
      template <class Abi>
        floatv<Abi> acosh(floatv<Abi> x);
      template <class Abi>
        doublev<Abi> acosh(doublev<Abi> x);
      template <class Abi>
        ldoublev<Abi> acosh(ldoublev<Abi> x);
      template <class Abi>
        floatv<Abi> asinh(floatv<Abi> x);
      template <class Abi>
        doublev<Abi> asinh(doublev<Abi> x);
      template <class Abi>
        ldoublev<Abi> asinh(ldoublev<Abi> x);
      template <class Abi>
        floatv<Abi> atanh(floatv<Abi> x);
      template <class Abi>
        doublev<Abi> atanh(doublev<Abi> x);
      template <class Abi>
        ldoublev<Abi> atanh(ldoublev<Abi> x);
    }
}
```
template <class Abi> floatv<Abi> cosh(floatv<Abi> x);
template <class Abi> doublev<Abi> cosh(doublev<Abi> x);
template <class Abi> ldoublev<Abi> cosh(ldoublev<Abi> x);

template <class Abi> floatv<Abi> sinh(floatv<Abi> x);
template <class Abi> doublev<Abi> sinh(doublev<Abi> x);
template <class Abi> ldoublev<Abi> sinh(ldoublev<Abi> x);

template <class Abi> floatv<Abi> tanh(floatv<Abi> x);
template <class Abi> doublev<Abi> tanh(doublev<Abi> x);
template <class Abi> ldoublev<Abi> tanh(ldoublev<Abi> x);

template <class Abi> floatv<Abi> exp(floatv<Abi> x);
template <class Abi> doublev<Abi> exp(doublev<Abi> x);
template <class Abi> ldoublev<Abi> exp(ldoublev<Abi> x);

template <class Abi> floatv<Abi> exp2(floatv<Abi> x);
template <class Abi> doublev<Abi> exp2(doublev<Abi> x);
template <class Abi> ldoublev<Abi> exp2(ldoublev<Abi> x);

template <class Abi> floatv<Abi> expm1(floatv<Abi> x);
template <class Abi> doublev<Abi> expm1(doublev<Abi> x);
template <class Abi> ldoublev<Abi> expm1(ldoublev<Abi> x);

template <class Abi> floatv<Abi> frexp(floatv<Abi> value, samesize<int, floatv<Abi>>* exp);
template <class Abi> doublev<Abi> frexp(doublev<Abi> value, samesize<int, doublev<Abi>>* exp);
template <class Abi> ldoublev<Abi> frexp(ldoublev<Abi> value, samesize<int, ldoublev<Abi>>* exp);

template <class Abi> floatv<Abi> ldexp(floatv<Abi> x, samesize<int, floatv<Abi>> exp);
template <class Abi> doublev<Abi> ldexp(doublev<Abi> x, samesize<int, doublev<Abi>> exp);
template <class Abi> ldoublev<Abi> ldexp(ldoublev<Abi> x, samesize<int, ldoublev<Abi>> exp);

template <class Abi> floatv<Abi> ilogb(floatv<Abi> x);
template <class Abi> doublev<Abi> ilogb(doublev<Abi> x);
template <class Abi> ldoublev<Abi> ilogb(ldoublev<Abi> x);

template <class Abi> floatv<Abi> log(floatv<Abi> x);
template <class Abi> doublev<Abi> log(doublev<Abi> x);
template <class Abi> ldoublev<Abi> log(ldoublev<Abi> x);

template <class Abi> floatv<Abi> log10(floatv<Abi> x);
template <class Abi> doublev<Abi> log10(doublev<Abi> x);
template <class Abi> ldoublev<Abi> log10(ldoublev<Abi> x);

template <class Abi> floatv<Abi> log1p(floatv<Abi> x);
template <class Abi> doublev<Abi> log1p(doublev<Abi> x);
template <class Abi> ldoublev<Abi> log1p(ldoublev<Abi> x);

template <class Abi> floatv<Abi> log2(floatv<Abi> x);
template <class Abi> doublev<Abi> log2(doublev<Abi> x);
template <class Abi> ldoublev<Abi> log2(ldoublev<Abi> x);

template <class Abi> floatv<Abi> logb(floatv<Abi> x);
template <class Abi> doublev<Abi> logb(doublev<Abi> x);
template <class Abi> ldoublev<Abi> logb(ldoublev<Abi> x);

template <class Abi> floatv<Abi> modf(floatv<Abi> value, floatv<Abi>* iptr);
template <class Abi> doublev<Abi> modf(doublev<Abi> value, doublev<Abi>* iptr);
template <class Abi> ldoublev<Abi> modf(ldoublev<Abi> value, ldoublev<Abi>* iptr);

template <class Abi> floatv<Abi> scalbn(floatv<Abi> x, samesize<int, floatv<Abi>> n);
template <class Abi> doublev<Abi> scalbn(doublev<Abi> x, samesize<int, doublev<Abi>> n);
template <class Abi> ldoublev<Abi> scalbn(ldoublev<Abi> x, samesize<int, ldoublev<Abi>> n);

template <class Abi> floatv<Abi> scalbln(floatv<Abi> x, samesize<long int, floatv<Abi>> n);
template <class Abi> doublev<Abi> scalbln(doublev<Abi> x, samesize<long int, doublev<Abi>> n);
template <class Abi> ldoublev<Abi> scalbln(ldoublev<Abi> x, samesize<long int, ldoublev<Abi>> n);

template <class Abi> floatv<Abi> cbrt(floatv<Abi> x);
template <class Abi> doublev<Abi> cbrt(doublev<Abi> x);
template <class Abi> ldoublev<Abi> cbrt(ldoublev<Abi> x);

template <class Abi> intv<Abi> abs(intv<Abi> j);
template <class Abi> longv<Abi> abs(longv<Abi> j);
template <class Abi> llongv<Abi> abs(llongv<Abi> j);
template <class Abi> floatv<Abi> abs(floatv<Abi> j);
template <class Abi> doublev<Abi> abs(doublev<Abi> j);
template <class Abi> ldoublev<Abi> abs(ldoublev<Abi> j);

template <class Abi> floatv<Abi> hypot(floatv<Abi> x, floatv<Abi> y);
template <class Abi> doublev<Abi> hypot(doublev<Abi> x, doublev<Abi> y);
template <class Abi> ldoublev<Abi> hypot(ldoublev<Abi> x, ldoublev<Abi> y);

template <class Abi> floatv<Abi> hypot(floatv<Abi> x, floatv<Abi> y, floatv<Abi> z);
template <class Abi> doublev<Abi> hypot(doublev<Abi> x, doublev<Abi> y, doublev<Abi> z);
template <class Abi> ldoublev<Abi> hypot(ldoublev<Abi> x, ldoublev<Abi> y, ldoublev<Abi> z);

template <class Abi> floatv<Abi> pow(floatv<Abi> x, floatv<Abi> y);
template <class Abi> doublev<Abi> pow(doublev<Abi> x, doublev<Abi> y);
template <class Abi> ldoublev<Abi> pow(ldoublev<Abi> x, ldoublev<Abi> y);

template <class Abi> floatv<Abi> sqrt(floatv<Abi> x);
template <class Abi> doublev<Abi> sqrt(doublev<Abi> x);
template <class Abi> ldoublev<Abi> sqrt(ldoublev<Abi> x);

template <class Abi> floatv<Abi> erf(floatv<Abi> x);
template <class Abi> doublev<Abi> erf(doublev<Abi> x);
template <class Abi> ldoublev<Abi> erf(ldoublev<Abi> x);

template <class Abi> floatv<Abi> erfc(floatv<Abi> x);
template <class Abi> doublev<Abi> erfc(doublev<Abi> x);
template <class Abi> ldoublev<Abi> erfc(ldoublev<Abi> x);

template <class Abi> floatv<Abi> lgamma(floatv<Abi> x);
template <class Abi> doublev<Abi> lgamma(doublev<Abi> x);
template <class Abi> ldoublev<Abi> lgamma(ldoublev<Abi> x);

template <class Abi> floatv<Abi> tgamma(floatv<Abi> x);
template <class Abi> doublev<Abi> tgamma(doublev<Abi> x);
template <class Abi> ldoublev<Abi> tgamma(ldoublev<Abi> x);

template <class Abi> floatv<Abi> ceil(floatv<Abi> x);
template <class Abi> doublev<Abi> ceil(doublev<Abi> x);
template <class Abi> ldoublev<Abi> ceil(ldoublev<Abi> x);
template <class Abi> doublev<Abi> floor(doublev<Abi> x);
template <class Abi> doublev<Abi> floor(ldoublev<Abi> x);
template <class Abi> floatv<Abi> nearbyint(floatv<Abi> x);
template <class Abi> doublev<Abi> nearbyint(doublev<Abi> x);
template <class Abi> ldoublev<Abi> nearbyint(ldoublev<Abi> x);
template <class Abi> doublev<Abi> rint(doublev<Abi> x);
template <class Abi> ldoublev<Abi> rint(ldoublev<Abi> x);
template <class Abi> floatv<Abi> rint(floatv<Abi> x);
template <class Abi> doublev<Abi> rint(doublev<Abi> x);
template <class Abi> ldoublev<Abi> rint(ldoublev<Abi> x);
template <class Abi> floatv<Abi> round(floatv<Abi> x);
template <class Abi> doublev<Abi> round(doublev<Abi> x);
template <class Abi> ldoublev<Abi> round(ldoublev<Abi> x);
template <class Abi> floatv<Abi> trunc(floatv<Abi> x);
template <class Abi> doublev<Abi> trunc(doublev<Abi> x);
template <class Abi> ldoublev<Abi> trunc(ldoublev<Abi> x);
template <class Abi> floatv<Abi> fmod(floatv<Abi> x, floatv<Abi> y);
template <class Abi> doublev<Abi> fmod(doublev<Abi> x, doublev<Abi> y);
template <class Abi> ldoublev<Abi> fmod(ldoublev<Abi> x, ldoublev<Abi> y);
template <class Abi> floatv<Abi> remainder(floatv<Abi> x, floatv<Abi> y);
template <class Abi> doublev<Abi> remainder(doublev<Abi> x, doublev<Abi> y);
template <class Abi> ldoublev<Abi> remainder(ldoublev<Abi> x, ldoublev<Abi> y);
template <class Abi> floatv<Abi> remquo(floatv<Abi> x, floatv<Abi> y, samesize<int, floatv<Abi>>* quo);
template <class Abi> doublev<Abi> remquo(doublev<Abi> x, doublev<Abi> y, samesize<int, doublev<Abi>>* quo);
template <class Abi> ldoublev<Abi> remquo(ldoublev<Abi> x, ldoublev<Abi> y, samesize<int, ldoublev<Abi>>* quo);
template <class Abi> floatv<Abi> copysign(floatv<Abi> x, floatv<Abi> y);
template <class Abi> doublev<Abi> copysign(doublev<Abi> x, doublev<Abi> y);
template <class Abi> ldoublev<Abi> copysign(ldoublev<Abi> x, ldoublev<Abi> y);
template <class Abi> doublev<Abi> nan(const char* tagp);
template <class Abi> floatv<Abi> nanf(const char* tagp);
template <class Abi> ldoublev<Abi> nanl(const char* tagp);
template <class Abi> floatv<Abi> nextafter(floatv<Abi> x, floatv<Abi> y);
template <class Abi> doublev<Abi> nextafter(doublev<Abi> x, doublev<Abi> y);

template <class Abi> doublev<Abi> nextafter(ldoublev<Abi> x, ldoublev<Abi> y);

template <class Abi> floatv<Abi> nexttoward(floatv<Abi> x, ldoublev<Abi> y);

template <class Abi> doublev<Abi> nexttoward(doublev<Abi> x, ldoublev<Abi> y);

template <class Abi> floatv<Abi> fdim(floatv<Abi> x, floatv<Abi> y);

template <class Abi> doublev<Abi> fdim(doublev<Abi> x, doublev<Abi> y);

template <class Abi> ldoublev<Abi> nexttoward(ldoublev<Abi> x, ldoublev<Abi> y);

template <class Abi> floatv<Abi> fmax(floatv<Abi> x, floatv<Abi> y);

template <class Abi> doublev<Abi> fmax(doublev<Abi> x, doublev<Abi> y);

template <class Abi> ldoublev<Abi> fmax(ldoublev<Abi> x, ldoublev<Abi> y);

template <class Abi> floatv<Abi> fmin(floatv<Abi> x, floatv<Abi> y);

template <class Abi> doublev<Abi> fmin(doublev<Abi> x, doublev<Abi> y);

template <class Abi> ldoublev<Abi> fmin(ldoublev<Abi> x, ldoublev<Abi> y);

template <class Abi> floatv<Abi> fma(floatv<Abi> x, floatv<Abi> y, floatv<Abi> z);

template <class Abi> doublev<Abi> fma(doublev<Abi> x, doublev<Abi> y, doublev<Abi> z);

template <class Abi> ldoublev<Abi> fma(ldoublev<Abi> x, ldoublev<Abi> y, ldoublev<Abi> z);

template <class Abi> samesize<int, floatv<Abi>> fpclassify(floatv<Abi> x);

template <class Abi> samesize<int, doublev<Abi>> fpclassify(doublev<Abi> x);

template <class Abi> samesize<int, ldoublev<Abi>> fpclassify(ldoublev<Abi> x);

template <class Abi> mask<float, Abi> isfinite(floatv<Abi> x);

template <class Abi> mask<double, Abi> isfinite(doublev<Abi> x);

template <class Abi> mask<long double, Abi> isfinite(ldoublev<Abi> x);

template <class Abi> samesize<int, floatv<Abi>> isinf(floatv<Abi> x);

template <class Abi> samesize<int, doublev<Abi>> isinf(doublev<Abi> x);

template <class Abi> samesize<int, ldoublev<Abi>> isinf(ldoublev<Abi> x);

template <class Abi> mask<float, Abi> isnan(floatv<Abi> x);

template <class Abi> mask<double, Abi> isnan(doublev<Abi> x);

template <class Abi> mask<long double, Abi> isnan(ldoublev<Abi> x);

template <class Abi> samesize<int, floatv<Abi>> isnormal(floatv<Abi> x);

template <class Abi> samesize<int, doublev<Abi>> isnormal(doublev<Abi> x);

template <class Abi> samesize<int, ldoublev<Abi>> isnormal(ldoublev<Abi> x);

template <class Abi> mask<float, Abi> signbit(floatv<Abi> x);

template <class Abi> mask<double, Abi> signbit(doublev<Abi> x);

template <class Abi> mask<long double, Abi> signbit(ldoublev<Abi> x);

template <class Abi> mask<float, Abi> isgreater(floatv<Abi> x, floatv<Abi> y);

template <class Abi> mask<double, Abi> isgreater(doublev<Abi> x, doublev<Abi> y);

template <class Abi> mask<long double, Abi> isgreater(ldoublev<Abi> x, ldoublev<Abi> y);

template <class Abi> mask<float, Abi> isgreaterequal(floatv<Abi> x, floatv<Abi> y);

template <class Abi> mask<double, Abi> isgreaterequal(doublev<Abi> x, doublev<Abi> y);

template <class Abi> mask<long double, Abi> isgreaterequal(ldoublev<Abi> x, ldoublev<Abi> y);

template <class Abi> mask<float, Abi> isless(floatv<Abi> x, floatv<Abi> y);

template <class Abi> mask<double, Abi> isless(doublev<Abi> x, doublev<Abi> y);

template <class Abi> mask<long double, Abi> isless(ldoublev<Abi> x, ldoublev<Abi> y);
template <class Abi> mask <float, Abi> islessequal (floatv <Abi> x, floatv <Abi> y);
//
template <class Abi> mask <double, Abi> islessequal (doublev <Abi> x, doublev <Abi> y);
//
template <class Abi> mask <long double, Abi> islessequal (ldoublev <Abi> x, ldoublev <Abi> y);
//
template <class Abi> mask <float, Abi> islessgreater (floatv <Abi> x, floatv <Abi> y);
//
template <class Abi> mask <double, Abi> islessgreater (doublev <Abi> x, doublev <Abi> y);
//
template <class Abi> mask <long double, Abi> islessgreater (ldoublev <Abi> x, ldoublev <Abi> y);
//
template <class Abi> mask <float, Abi> isunordered (floatv <Abi> x, floatv <Abi> y);
//
template <class Abi> mask <double, Abi> isunordered (doublev <Abi> x, doublev <Abi> y);
//
template <class Abi> mask <long double, Abi> isunordered (ldoublev <Abi> x, ldoublev <Abi> y);
/// broadcast constructor
explicit mask (value_type);
//
/// implicit type conversion constructor
template <class U> mask (const mask <U, datapar_abi::fixed_size <size ()>> &);
//
// load constructor

Each listed function concurrently applies the indicated mathematical function component-wise. The results per component are not required to be binary equal to the application of the function which is overloaded for the element type.

If abs() is called with an argument of type datapar<X, Abi> for which is_unsigned<X>::value is true, the program is ill-formed.
The class template `mask<T, Abi>` is a one-dimensional smart array of booleans. The number of elements in the array is determined at compile time, equal to the number of elements in `datapar<T, Abi>`.

The first template argument `T` must be a cv-unqualified integral or floating-point type (3.9.1 [basic.fundamental]). The type `bool` is not allowed.

The second template argument `Abi` must be a tag type from the `datapar_abi` namespace.

If any of the template arguments does not conform to the above requirements, the resulting class shall be a complete type with deleted default constructor, deleted destructor, deleted copy constructor, and deleted copy assignment.

Default initialization performs no initialization of the elements; value-initialization initializes each element with `bool()`. [Note: Thus, default initialization leaves the elements in an indeterminate state. — end note]

```cpp
static constexpr size_type size();
```

Returns: the number of boolean elements stored in objects of the given `mask<T, Abi>` type.

Note: Implementations are encouraged to enable `static_cast`ing from/to (an) implementation-defined SIMD mask type(s). This would add one or more of the following declarations to class `mask`:

```cpp
explicit operator implementation_defined() const;
explicit datapar(const implementation_defined &init);
```

4.1.4.2 `mask` constructors
explicit mask(value_type);

Effects: Constructs an object with each element initialized to the value of the argument.

template <class U> mask(const mask<U, datapar_abi::fixed_size<size()>> & x);

Remarks: This constructor shall not participate in overload resolution unless abi_type equals datapar_abi::fixed_size<size()>.

Effects: Constructs an object of type mask where the i-th element equals x[i] for all i ∈ [0, size()).

template <class Flags> mask(const value_type *mem, Flags);

Effects: Constructs an object where the i-th element is initialized to mem[i] for all i ∈ [0, size()).

Remarks: If size() returns a value greater than the number of values pointed to by the first argument, the behavior is undefined.

Remarks: If the Flags template parameter is of type flags::vector_aligned_tag and the pointer value is not a multiple of memory_alignment<mask>, the behavior is undefined. If the Flags template parameter is of type flags::overaligned_tag<N> and the pointer value is not a multiple of N, the behavior is undefined.

template <class Flags> mask(const value_type *mem, mask k, Flags);

Effects: Constructs an object where the i-th element is initialized to k[i] ? mem[i] : false for all i ∈ [0, size()).

Remarks: If the largest i where k[i] is true is greater than the number of values pointed to by the first argument, the behavior is undefined.

Remarks: If the Flags template parameter is of type flags::vector_aligned_tag and the pointer value is not a multiple of memory_alignment<mask>, the behavior is undefined. If the Flags template parameter is of type flags::overaligned_tag<N> and the pointer value is not a multiple of N, the behavior is undefined.

4.1.4.3 mask load function [mask.load]

template <class Flags> void memload(const value_type *mem, Flags);

Effects: Replaces the elements of the mask object such that the i-th element is assigned with mem[i] for all i ∈ [0, size()).

Remarks: If size() returns a value greater than the number of values pointed to by the first argument, the behavior is undefined.

Remarks: If the Flags template parameter is of type flags::vector_aligned_tag and the pointer value is not a multiple of memory_alignment<mask>, the behavior is undefined. If the Flags template parameter is of type flags::overaligned_tag<N> and the pointer value is not a multiple of N, the behavior is undefined.

4.1.4.4 mask store function [mask.store]
4.1.4.5 mask subscript operators

reference operator[](size_type i);

Returns: A temporary object with the following properties:

- Remarks: The object is neither DefaultConstructible, CopyConstructible, MoveConstructible, Copy.Assignable, nor MoveAssignable.
- Remarks: Assignment, compound assignment, increment, and decrement operators only participate in overload resolution if called in rvalue context and the corresponding operator of type value_type is usable.
- Effects: The assignment, compound assignment, increment, and decrement operators execute the indicated operation on the i-th element in the datapar object.
- Effects: Conversion to value_type returns a copy of the i-th element.

value_type operator[](size_type) const;

Returns: A copy of the i-th element.

4.1.4.6 mask unary operators

mask operator!() const;

Returns: A mask object with the i-th element set to the logical negation for all i ∈ [0, size()].

4.1.5 mask non-member operations

4.1.5.1 mask binary operators

friend mask operator&& (const mask &a, const mask &b);
friend mask operator||(const mask &a, const mask &b);
friend mask operator& (const mask &a, const mask &b);
friend mask operator| (const mask &a, const mask &b);
friend mask operator^ (const mask &a, const mask &b);
Returns: A mask object initialized with the results of the component-wise application of the indicated operator.

4.1.5.2 mask compound assignment

friend mask &operator=(mask &, const mask &);
friend mask &operator|=(mask &, const mask &);
friend mask &operator^=(mask &, const mask &);

Effects: Each of these operators performs the indicated operator component-wise on each of the corresponding elements of the arguments.

Returns: A reference to the first argument.

4.1.5.3 mask compares

friend bool operator==(const mask &, const mask &);
friend bool operator!=(const mask &, const mask &);

Returns: true if all boolean elements of the first argument equal the corresponding element of the second argument. It returns false otherwise.

4.1.5.4 mask reductions

template<class T, class Abi> bool all_of(mask<T, Abi>);

Returns: true if all boolean elements in the function argument equal true, false otherwise.

template<class T, class Abi> bool any_of(mask<T, Abi>);

Returns: true if at least one boolean element in the function argument equals true, false otherwise.

template<class T, class Abi> bool none_of(mask<T, Abi>);

Returns: true if none of the boolean element in the function argument equals true, false otherwise.

template<class T, class Abi> bool some_of(mask<T, Abi>);

Returns: true if at least one of the boolean elements in the function argument equals true and at least one of the boolean elements in the function argument equals false, false otherwise.

template<class T, class Abi> int popcount(mask<T, Abi>);

Returns: The number of boolean elements that are true.
5 Discussion

4.1.5.5 Masked assignment

Returns: An object of type where_expression (see 4.1.1.3) initialized with the predicate \( k \) and the value reference \( v \).

Remarks: The function only participates in overload resolution if \( T \) is neither a datapar nor a mask instantiation.

Returns: An object of type where_expression (see 4.1.1.3) initialized with the predicate \( k \) and the value reference \( v \).
5 Discussion

reference
   Used as the return type of the non-const scalar subscript operator.

mask_type
   The natural mask type for this datapar instantiation. This type is used as return type of compares and write-mask on assignments.

datapar_type
   The natural datapar type for this mask instantiation.

size_type
   Standard member type used for size() and operator[].

abi_type
   The Abi template parameter to datapar.

5.2 conversions

The datapar conversion constructor only allows implicit conversion from datapar template instantiations with the same Abi type and compatible value_type. Discussion in SG1 showed clear preference for only allowing implicit conversion between integral types that only differ in signedness. All other conversions could be implemented via an explicit conversion constructor. The alternative (preferred) is to use datapar_cast consistently for all other conversions.

After more discussion on the LEWG reflector, in Issaquah, and between me and Jens, we modified conversions to be even more conservative. No implicit conversion will ever allow a narrowing conversion of the element type (and signed - unsigned is narrowing in both directions).

5.3 broadcast constructor

The datapar broadcast constructor is not declared explicit to ease the use of scalar prvalues in expressions involving data-parallel operations. The operations where such a conversion should not be implicit consequently need to use SFINAE / concepts to inhibit the conversion.

Experience from Vc shows that the situation is different for mask, where an implicit conversion from bool typically hides an error. (Since there is little use for broadcasting true or false.)
5.4 aliasing of subscript operators

The subscript operators return an rvalue. The const overload returns a copy of the element. The non-const overload returns a smart reference. This reference behaves mostly like an lvalue reference, but without the requirement to implement assignment via type punning. At this point the specification of the smart reference is very conservative / restrictive: The reference type is neither copyable nor movable. The intention is to avoid users to program like the operator returned an lvalue reference. The return type is significantly larger than an lvalue reference and harder to optimize when passed around. The restriction thus forces users to do element modification directly on the `datapar/mask` objects.

Guidance from SG1 at JAX 2016:

Poll: Should subscript operator return an lvalue reference?

<table>
<thead>
<tr>
<th>SF</th>
<th>F</th>
<th>N</th>
<th>A</th>
<th>SA</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>6</td>
<td>10</td>
<td>2</td>
<td>1</td>
</tr>
</tbody>
</table>

Poll: Should subscript operator return a “smart reference”?

<table>
<thead>
<tr>
<th>SF</th>
<th>F</th>
<th>N</th>
<th>A</th>
<th>SA</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>7</td>
<td>10</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

5.5 compound assignment

The semantics of compound assignment would allow less strict implicit conversion rules. Consider `datapar<int>() *= double()`: the corresponding binary multiplication operator would not compile because the implicit conversion to `datapar<double>` is non-portable. Compound assignment, on the other hand, implies an implicit conversion back to the type of the expression on the left of the assignment operator. Thus, it is possible to define compound operators that execute the operation correctly on the promoted type without sacrificing portability. There are two arguments for not relaxing the rules for compound assignment, though:

1. Consistency: The conversion of an expression with compound assignment to a binary operator might make it ill-formed.

2. The implicit conversion in the `int * double` case could be expensive and unintended. This is already a problem for `builtin` types, where many developers multiply `float` variables with `double` prvalues, though.


Listing 1: Possible declaration of the class template parameters of a `datapar` class with arbitrary width.

5.6 Return Type of Masked Assignment Operators

The assignment operators of the type returned by `where(mask, datapar)` could return one of:

- A reference to the `datapar` object that was modified.
- A temporary `datapar` object that only contains the elements where the mask is true.
- A reference to the `where_expression` object.
- Nothing (i.e. `void`).

My first choice was a reference to the modified `datapar` object. However, then the statement `(where(x < 0, x) *= -1) += 2` may be surprising: it adds 2 to all vector entries, independent of the mask. Likewise, `y += (where(x < 0, x) *= -1)` has a possibly confusing interpretation because of the `mask` in the middle of the expression.

Consider that write-masked assignment is used as a replacement for `if`-statements. Using `void` as return type therefore is a more fitting choice because `if`-statements have no return value. By declaring the return type as `void` the above expressions become ill-formed, which seems to be the best solution for guiding users to write maintainable code and express intent clearly.

5.7 Fundamental SIMD Type or Not?

5.7.1 The Issue

There was substantial discussion on the reflectors and SG1 meetings over the question whether C++ should define a fundamental, native SIMD type (let us call it `fundamental<T>`) and additionally a generic data-parallel type which supports an arbitrary number of elements (call it `arbitrary<T, N>`). The alternative to defining both types is to only define `arbitrary<T, N = default_size<T>>`, since it encompasses the `fundamental<T>` type.

With regard to this proposal this second approach would add a third template parameter to `datapar` and `mask` as shown in Listing 1.
5.7.2 standpoints

The controversy is about how the flexibility of a type with arbitrary $N$ is presented to the users. Is there a (clear) distinction between a “fundamental” type with target-dependent (i.e. fixed) $N$ and a higher-level abstraction with arbitrary $N$ which can potentially compile to inefficient machine code? Or should the C++ standard only define arbitrary and set it to a default $N$ value that corresponds to the target-dependent $N$. Thus, the default $N$, of arbitrary would correspond to fundamental.

It is interesting to note that arbitrary$<$T, 1$>$ is the class variant of T. Consequently, if we say there is no need for a fundamental type then we could argue for the deprecation of the built-in arithmetic types, in favor of arbitrary$<$T, 1$>$. [Note: This is an academic discussion, of course. — end note]

The author has implemented a library where a clear distinction is made between fundamental$<$T, Abi$>$ and arbitrary$<$T, N$>$. The documentation and all teaching material says that the user should program with fundamental. The arbitrary type should be used in special circumstances, or wherever fundamental works with the arbitrary type in its interfaces (e.g. for gather & scatter or the ldexp & frexp functions).

5.7.3 issues

The definition of two separate class templates can alleviate some source compatibility issues resulting from different $N$ on different target systems. Consider the simplest example of a multiplication of an int vector with a float vector:

```cpp
arbitrary<float>() * arbitrary<int>(); // compiles for some targets, fails for others
fundamental<float>() * fundamental<int>(); // never compiles, requires explicit cast
```

The datapar<T> operators are specified in such a way that source compatibility is ensured. For a type with user definable $N$, the binary operators should work slightly different with regard to implicit conversions. Most importantly, arbitrary$<$T, N$>$ solves the issue of portable code containing mixed integral and floating-point values. A user would typically create aliases such as:

```cpp
using floatvec = datapar<float>;
using intvec = arbitrary<int, floatvec::size>();
using doublevec = arbitrary<int, floatvec::size>();
```

Objects of types floatvec, intvec, and doublevec will work together, independent of the target system.

Obviously, these type aliases are basically the same if the $N$ parameter of arbitrary has a default value:
using floatvec = arbitrary<float>;
using intvec = arbitrary<int, floatvec::size>();
using doublevec = arbitrary<int, floatvec::size>();

The ability to create these aliases is not the issue. Seeing the need for using such a pattern is the issue. Typically, a developer will think no more of it if his code compiles on his machine. If arbitrary<float>() * arbitrary<int>() just happens to compile (which is likely) then this is the code that will get checked in to the repository. Note that with the existence of the fundamental class template, the N parameter of the arbitrary class would not have a default value and thus force the user to think a second longer about portability.

5.7.4 progress

SGI Guidance at JAX 2016:
Poll: Specify datapar width using ABI tag, with a special template tag for fixed size.

<table>
<thead>
<tr>
<th>SF</th>
<th>F</th>
<th>N</th>
<th>A</th>
<th>SA</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>7</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>

Poll: Specify datapar width using <T, N, abi>, where abi is not specified by the user.

<table>
<thead>
<tr>
<th>SF</th>
<th>F</th>
<th>N</th>
<th>A</th>
<th>SA</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2</td>
<td>5</td>
<td>2</td>
<td>1</td>
</tr>
</tbody>
</table>

At the Jacksonville meeting, SG1 decided to continue with the datapar<T, Abi> class template, with the addition of a new Abi type that denotes a user-requested number of elements in the vector (datapar_abi::fixed_size<N>). This has the following implications:

- There is only one class template with a common interface for fundamental and arbitrary (fixed_size) vector types.

- There are slight differences in the conversion semantics for datapar types with the fixed_size Abi type. This may look like the vector<bool> mistake all over again. I’ll argue below why I believe this is not the case.

- The fundamental class instances could be implemented in such a way that they do not guarantee ABI compatibility on a given architecture where translation units are compiled with different compiler flags (for micro-architectural differences).

- The fixed_size class instances, on the other hand, could be implemented to be the ABI stable types (if an implementation thinks this is an important feature).
In implementation terms this means that fundamental types are allowed to be passed via registers on function calls. fixed_size types can be implemented in such a way that they are only passed via the stack, and thus an implementation only needs to ensure equal alignment and memory representation across TU borders for a given T, N.

The conversion differences between the fundamental and fixed_size class template instances are the main motivation for having a distinction (cf. discussion above). The differences are chosen such that, in general, fundamental types are more restrictive and do not turn into fixed_size types on any operation that involves no fixed_size types. Operations of fixed_size types allow easier use of mixed precision code as long as no elements need to be dropped / generated (i.e. the number of elements of all involved datar objects is equal or a builtin arithmetic type is broadcast).

Examples:

1. Mixed int–float operations

   ```
   using floatv = datar<float>; // native ABI
   using float_sized_abi = datar_abi::fixed_size<floatv::size>();
   using intv = datar<int, float_sized_abi>;
   auto x = floatv() + intv();
   intv y = floatv() + intv();
   ```

   Line 5 is well-formed: It states that $N$ ($= \text{floatv::size()}$) additions shall be executed concurrently. The type of $x$ is datar<float>, because it stores $N$ elements and both types intv and floatv are implicitly convertible to datar<float>. Line 6 is also well-formed because implicit conversion from datar<T, Abi> to datar<U, datar_abi::fixed_size<N>> is allowed whenever $N == \text{datar<T, Abi>::size()}$.

2. Native int vectors

   ```
   using intv = datar<int>; // native ABI
   using int_sized_abi = datar_abi::fixed_size<intv::size>();
   using floatv = datar<float, int_sized_abi>;
   auto x = floatv() + intv();
   intv y = floatv() + intv();
   ```

   Line 5 is well-formed: It states that $N$ ($= \text{intv::size()}$) additions shall be executed concurrently. The type of $x$ is datar<float_v, int_sized_abi> (i.e. floatv) and never datar<float>, because ...
... the Abi types of intv and floatv are not equal.

... either datapar<float>::size() != N or intv is not implicitly convertible to datapar<float>.

... the last rule for commonabi(V0, V1, T) sets the Abi type to int_sized_abi.

Line 6 is also well-formed because implicit conversion from datapar<T, data-par_abi::fixed_size<N>> to datapar<U, Abi> is allowed whenever N == datapar<U, Abi>::size().

5.8 native handle

The presence of a native_handle function for accessing an internal data member such as e.g. a vector builtin or SIMD intrinsic type is seen as an important feature for adoption in the target communities. Without such a handle the user is constrained to work within the (limited) API defined by the standard. Many SIMD instruction sets have domain-specific instructions that will not easily be usable (if at all) via the standardized interface. A user considering whether to use datapar or a SIMD extension such as vector builtins or SIMD intrinsics might decide against datapar just for fear of not being able to access all functionality.\(^1\)

I would be happy to settle on an alternative to exposing an lvalue reference to a data member. Consider implementation-defined support casting (static_cast?) between datapar and non-standard SIMD extension types. My understanding is that there could not be any normative wording about such a feature. However, I think it could be useful to add a non-normative note about making static_cast (?) able to convert between such non-standard extensions and datapar.

Guidance from SG1 at Oulu 2016:

Poll: Keep native_handle in the wording?

<table>
<thead>
<tr>
<th>SF</th>
<th>F</th>
<th>N</th>
<th>A</th>
<th>SA</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>6</td>
<td>3</td>
<td>3</td>
<td>0</td>
</tr>
</tbody>
</table>

5.9 load & store flags

SIMD loads and stores require at least an alignment option. This is in contrast to implicit loads and stores present in C++, where alignment is always assumed. Many SIMD instruction sets allow more options, though:

- Streaming, non-temporal loads and stores

\(^1\) Whether that’s a reasonable fear is a different discussion.
• Software prefetching

In the Vc library I have added these as options in the load store flag parameter of the load and store functions. However, non-temporal loads & stores and prefetching are also useful for the existing builtin types. I would like guidance on this question: should the general direction be to stick to only alignment options for datapar loads and stores?

The other question is on the default of the load and store flags. Some argue for setting the default to aligned, as that’s what the user should always aim for and is most efficient. Others argue for unaligned since this is safe per default. The Vc library before version 1.0 used aligned loads and stores per default. After the guidance from SG1 I changed the default to unaligned loads and stores with the Vc 1.0 release. Changing the default is probably the worst that could be done, though. For Vc 2.0 I will drop the default.

For datapar I prefer no default:

• This makes it obvious that the API has the alignment option. Users should not just take the default and think no more of it.

• If we decide to keep the load constructor, the alignment parameter (without default) nicely disambiguate the load from the broadcast.

• The right default would be application/domain/experience specific.

• Users can write their own load/store wrapper functions that implement their chosen default.

Guidance from SG1 at Oulu 2016:

Poll: Should the interface provide a way to specify a number for over-alignment?

<table>
<thead>
<tr>
<th>SF</th>
<th>F</th>
<th>N</th>
<th>A</th>
<th>SA</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>6</td>
<td>5</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

Poll: Should loads and stores have a default load/store flag?

<table>
<thead>
<tr>
<th>SF</th>
<th>F</th>
<th>N</th>
<th>A</th>
<th>SA</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>7</td>
<td>4</td>
<td>1</td>
</tr>
</tbody>
</table>

The discussion made it clear that we only want to support alignment flags in the load and store operations. The other functionality is orthogonal.

---

2 As I realized too late.
UNARY MINUS RETURN TYPE

The return type of `datapar<T, Abi>::operator-()` is `datapar<T, Abi>`. This is slightly different to the behavior of the underlying element type `T`, if `T` is an integral type of lower integer conversion rank than `int`. In this case integral promotion promotes the type to `int` before applying unary minus. Thus, the expression `-T()` is of type `int` for all `T` with lower integer conversion rank than `int`. This is widening of the element size is likely unintended for SIMD vector types.

Fundamental types with integer conversion rank greater than `int` are not promoted and thus a unary minus expression has unchanged type. This behavior is copied to element types of lower integer conversion rank for `datapar`.

There may be one interesting alternative to pursue here: We can make it ill-formed to apply unary minus to unsigned integral types. Anyone who wants to have the modulo behavior of a unary minus could still write `0u - x`.

ACKNOWLEDGEMENTS

- This work was supported by GSI Helmholtzzentrum für Schwerionenforschung and the Hessian LOEWE initiative through the Helmholtz International Center for FAIR (HIC for FAIR).

- Jens Maurer contributed important feedback and suggestions on the API. Thanks also for presenting the paper in Kona 2017.

- Thanks to Hartmut Kaiser for presenting in Issaquah 2016.

BIBLIOGRAPHY


