Document Number: P0214R2
Date: 2016-10-17
Reply-to: Matthias Kretz <m.kretz@gsi.de>
Audience: LEWG

Data-Parallel Vector Types & Operations

ABSTRACT

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

CONTENTS

0 REMARKS

- This documents 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, $\mathcal{W}_{\mathrm{T}}$ denotes the number of scalar values (width) in a vector of type `T` (sometimes also called the number of SIMD lanes)
- [N4184], [N4185], and [N4395] provide more information on the rationale and design decisions. [N4454] discusses a matrix multiplication example. My PhD thesis [1] contains a very thorough discussion of the topic.
- 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.

1 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 (4.1.5.2).
- Modified `where` to support different Abi tags on the `mask` and `datapar` arguments (4.1.5.4).

- 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 5.10).
- 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 (5.9.4).
- 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-()` (5.12).

— 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`.

2 STRAW POLLS

2.1 SG1 AT CHICAGO 2013

Poll: Pursue SIMD/data parallel programming via types?

SF	F	N	A	SA
1	8	5	0	0

2.2 SG1 AT URBANA 2014

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

SF	F	N	A	SA
0	0	6	11	2

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

SF	F	N	A	SA
1	2	0	6	11

Poll: Apply "sign promotion" to vector operations? SF = `ushortv` + `shortv` = `ushortv`; SA = no mixed signed/unsigned arithmetic

SF	F	N	A	SA
1	5	5	7	2

2.3 SG1 AT LENEXA 2015

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

SF	F	N	A	SA
10	6	2	0	0

2.4 SG1 AT JAX 2016

Poll: Should subscript operator return an lvalue reference?

SF	F	N	A	SA
0	6	10	2	1

Poll: Should subscript operator return a "smart reference"?

SF	F	N	A	SA
1	7	10	0	0

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

SF	F	N	A	SA
3	7	0	0	1

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

SF	F	N	A	SA
1	2	5	2	1

2.5 SG1 at Oulu 2016

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

SF	F	N	A	SA
0	6	3	3	0

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

SF	F	N	A	SA
2	6	5	0	0

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

SF	F	N	A	SA
0	0	7	4	1

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 $\mathcal{W}_\mathrm{T}$ successive scalar values starting from a given address into a SIMD register.
- Store instructions: store from a SIMD register to $\mathcal{W}_\mathrm{T}$ 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 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 [datapar.types]

4.1.1 Header `<datapar>` synopsis [datapar.syn]

```
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> {};
template <class T, class Abi = datapar_abi::compatible<T>>
constexpr size_t datapar_size_v = datapar_size<T, Abi>::value;

template <class T, class U = typename T::value_type>
constexpr size_t memory_alignment = implementation_defined;

// class template datapar [datapar]
template <class T, class Abi = datapar_abi::compatible<T>> class datapar;

// class template mask [mask]
template <class T, class Abi = datapar_abi::compatible<T>> class mask;

// compound assignment [datapar.cassign]
template <class T, class Abi, class U> datapar<T, Abi> &operator+= (datapar<T, Abi> &, const U &);
template <class T, class Abi, class U> datapar<T, Abi> &operator-= (datapar<T, Abi> &, const U &);
template <class T, class Abi, class U> datapar<T, Abi> &operator*= (datapar<T, Abi> &, const U &);
template <class T, class Abi, class U> datapar<T, Abi> &operator/= (datapar<T, Abi> &, const U &);
template <class T, class Abi, class U> datapar<T, Abi> &operator%= (datapar<T, Abi> &, const U &);
template <class T, class Abi, class U> datapar<T, Abi> &operator&= (datapar<T, Abi> &, const U &);
template <class T, class Abi, class U> datapar<T, Abi> &operator|= (datapar<T, Abi> &, const U &);
template <class T, class Abi, class U> datapar<T, Abi> &operator^= (datapar<T, Abi> &, const U &);
template <class T, class Abi, class U> datapar<T, Abi> &operator<<=(datapar<T, Abi> &, const U &);
template <class T, class Abi, class U> datapar<T, Abi> &operator>>=(datapar<T, Abi> &, const U &);

// binary operators [datapar.binary]
template <class L, class R> using datapar_return_type = ...;  // exposition only

template <class T, class Abi, class U>
datapar_return_type<datapar<T, Abi>, U> operator+ (const datapar<T, Abi> &, const U &);
template <class T, class Abi, class U>
datapar_return_type<datapar<T, Abi>, U> operator- (const datapar<T, Abi> &, const U &);
template <class T, class Abi, class U>
datapar_return_type<datapar<T, Abi>, U> operator* (const datapar<T, Abi> &, const U &);
template <class T, class Abi, class U>
datapar_return_type<datapar<T, Abi>, U> operator/ (const datapar<T, Abi> &, const U &);
template <class T, class Abi, class U>
datapar_return_type<datapar<T, Abi>, U> operator% (const datapar<T, Abi> &, const U &);
template <class T, class Abi, class U>
datapar_return_type<datapar<T, Abi>, U> operator& (const datapar<T, Abi> &, const U &);
template <class T, class Abi, class U>
datapar_return_type<datapar<T, Abi>, U> operator| (const datapar<T, Abi> &, const U &);
template <class T, class Abi, class U>
datapar_return_type<datapar<T, Abi>, U> operator^ (const datapar<T, Abi> &, const U &);
template <class T, class Abi, class U>
datapar_return_type<datapar<T, Abi>, U> operator<<(const datapar<T, Abi> &, const U &);
template <class T, class Abi, class U>
```

```
datapar_return_type<datapar<T, Abi>, U> operator>>(const datapar<T, Abi> &, const U &);
template <class T, class Abi, class U>
datapar_return_type<datapar<T, Abi>, U> operator+ (const U &, const datapar<T, Abi> &);
template <class T, class Abi, class U>
datapar_return_type<datapar<T, Abi>, U> operator- (const U &, const datapar<T, Abi> &);
template <class T, class Abi, class U>
datapar_return_type<datapar<T, Abi>, U> operator* (const U &, const datapar<T, Abi> &);
template <class T, class Abi, class U>
datapar_return_type<datapar<T, Abi>, U> operator/ (const U &, const datapar<T, Abi> &);
template <class T, class Abi, class U>
datapar_return_type<datapar<T, Abi>, U> operator% (const U &, const datapar<T, Abi> &);
template <class T, class Abi, class U>
datapar_return_type<datapar<T, Abi>, U> operator& (const U &, const datapar<T, Abi> &);
template <class T, class Abi, class U>
datapar_return_type<datapar<T, Abi>, U> operator| (const U &, const datapar<T, Abi> &);
template <class T, class Abi, class U>
datapar_return_type<datapar<T, Abi>, U> operator^ (const U &, const datapar<T, Abi> &);
template <class T, class Abi, class U>
datapar_return_type<datapar<T, Abi>, U> operator<<(const U &, const datapar<T, Abi> &);
template <class T, class Abi, class U>
datapar_return_type<datapar<T, Abi>, U> operator>>(const U &, const datapar<T, Abi> &);

// compares [datapar.comparison]
template <class T, class Abi, class U>
typename datapar_return_type<datapar<T, Abi>, U>::mask_type operator==(const datapar<T, Abi> &, const U &);
template <class T, class Abi, class U>
typename datapar_return_type<datapar<T, Abi>, U>::mask_type operator!=(const datapar<T, Abi> &, const U &);
template <class T, class Abi, class U>
typename datapar_return_type<datapar<T, Abi>, U>::mask_type operator>=(const datapar<T, Abi> &, const U &);
template <class T, class Abi, class U>
typename datapar_return_type<datapar<T, Abi>, U>::mask_type operator<=(const datapar<T, Abi> &, const U &);
template <class T, class Abi, class U>
typename datapar_return_type<datapar<T, Abi>, U>::mask_type operator> (const datapar<T, Abi> &, const U &);
template <class T, class Abi, class U>
typename datapar_return_type<datapar<T, Abi>, U>::mask_type operator< (const datapar<T, Abi> &, const U &);
template <class T, class Abi, class U>
typename datapar_return_type<datapar<T, Abi>, U>::mask_type operator==(const U &, const datapar<T, Abi> &);
template <class T, class Abi, class U>
typename datapar_return_type<datapar<T, Abi>, U>::mask_type operator!=(const U &, const datapar<T, Abi> &);
template <class T, class Abi, class U>
typename datapar_return_type<datapar<T, Abi>, U>::mask_type operator>=(const U &, const datapar<T, Abi> &);
template <class T, class Abi, class U>
typename datapar_return_type<datapar<T, Abi>, U>::mask_type operator<=(const U &, const datapar<T, Abi> &);
template <class T, class Abi, class U>
typename datapar_return_type<datapar<T, Abi>, U>::mask_type operator> (const U &, const datapar<T, Abi> &);
template <class T, class Abi, class U>
typename datapar_return_type<datapar<T, Abi>, U>::mask_type operator< (const U &, const datapar<T, Abi> &);

// casts [datapar.casts]
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...);

// mask binary operators [mask.binary]
template <class T0, class A0, class T1, class A1> using mask_return_type = ...   // exposition only
template <class T0, class A0, class T1, class A1>
mask_return_type<T0, A0, T1, A1> operator&&(const mask<T0, A0> &, const mask<T1, A1> &);
template <class T0, class A0, class T1, class A1>
```

```
mask_return_type<T0, A0, T1, A1> operator||(const mask<T0, A0> &, const mask<T1, A1> &);
template <class T0, class A0, class T1, class A1>
mask_return_type<T0, A0, T1, A1> operator& (const mask<T0, A0> &, const mask<T1, A1> &);
template <class T0, class A0, class T1, class A1>
mask_return_type<T0, A0, T1, A1> operator| (const mask<T0, A0> &, const mask<T1, A1> &);
template <class T0, class A0, class T1, class A1>
mask_return_type<T0, A0, T1, A1> operator^ (const mask<T0, A0> &, const mask<T1, A1> &);

// mask compares [mask.comparison]
template <class T0, class A0, class T1, class A1>
bool operator==(const mask<T0, A0> &, const mask<T1, A1> &);
template <class T0, class A0, class T1, class A1>
bool operator!=(const mask<T0, A0> &, const mask<T1, A1> &);

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

// masked assignment [mask.where]
template <class M, class T> class where_expression {
public:
  where_expression(const where_expression &) = delete;
  where_expression &operator=(const where_expression &) = delete;
  where_expression(const M &k, T &d);
  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);
  T &operator++();
  T operator++(int);
  T &operator--();
  T operator--(int);
  T operator-() const;
  auto operator!() const;

private:
  const M &mask;   // exposition only
  T &data;         // exposition only
};
```

```
    template <class T0, class A0, class T1, class A1>
    where_expression<mask<T1, A1>, datapar<T1, A1>> where(const mask<T0, A0> &, datapar<T1, A1> &);
    template <class T> where_expression<bool, T> where(bool, T &);

    // reductions [datapar.reductions]
    template <class BinaryOperation = std::plus<>, class T, class Abi>
    T reduce(const datapar<T, Abi> &, BinaryOperation = BinaryOperation());
    template <class BinaryOperation = std::plus<>, class M, class T, class Abi>
    U reduce(const where_expression<M, datapar<T, Abi>> &x, T init,
             BinaryOperation binary_op = BinaryOperation());
  }
}
```

1 The header `<datapar>` defines two class templates (`datapar`, and `mask`), 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 [datapar.abi]

```
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
}
```

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

2 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 1 `scalar` could be an alias for `fixed_size<1>`. However, I think it is better to have a different ABI tag to make implicit conversions behave as for all the other non-fixed-size ABIs.

3 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 $N \in [1 \dots 32]$. Additionally, an implementation must support any $N \in \{$`datapar<U>::size()`$, \forall U \in \{\text{arithmetic types}\}\}$. [*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 compatibilty between differently compiled translation units, the efficiency of `datapar<T, Abi>` is likely to be better than for `datapar<T, fixed_size<datapar_size_v<T, Abi>>>` (with `Abi` not a instance of `datapar_abi::fixed_size`). — *end note*]

4 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*]

Note 2 I'm afraid this makes changes to the maximum size an ABI break, no? Unfounded fear?

5 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.

6 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.

7 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 support 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 [datapar.traits]

```
template <class T> struct is_datapar;
```

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

```
template <class T> struct is_mask;
```

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

```
template <class T, size_t N> struct abi_for_size { typedef implementation_defined type; };
```

3 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

- `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 `datapar_abi::fixed_size<N>`.

4 `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]).

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

5 The `datapar_size` class template inherits from `integral_constant` with a value that equals `datapar<T, Abi>::size()`.

6 `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;
```

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

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

9 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`.

4.1.2 Class template `datapar` [datapar]

4.1.2.1 Class template `datapar` overview [datapar.overview]

```
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
      datapar(value_type);

      // implicit type conversion constructor
      template <class U, class Abi2> datapar(datapar<U, Abi2>);

      // 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 copy_from(const U *mem, Flags);
      template <class U, class Flags> void copy_from(const U *mem, mask_type k, Flags);

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

      // scalar access:
      reference operator[](size_type);
      value_type operator[](size_type) const;

      // increment and decrement:
      datapar &operator++();
      datapar operator++(int);
      datapar &operator--();
      datapar operator--(int);

      // unary operators (for integral T)
      mask_type operator!() const;
      datapar operator~() const;

      // unary operators (for any T)
      datapar operator+() const;
```

```
    datapar operator-() const;

    // reductions
    value_type sum() const;
    value_type sum(mask_type) const;
    value_type product() const;
    value_type product(mask_type) const;
    value_type min() const;
    value_type min(mask_type) const;
    value_type max() const;
    value_type max(mask_type) const;
  };
 }
}
```

1 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.

2 The first template argument `T` must be an integral or floating-point fundamental type. The type `bool` is not allowed.

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

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

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

5 *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);
```

4.1.2.2 `datapar` constructors [datapar.ctor]

```
datapar() = default;
```

1 *Effects:* Constructs an object with all elements initialized to `T()`. [*Note:* This zero-initializes the object. — *end note*]

```
datapar(value_type);
```

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

Note 3 Should I add a generator ctor, taking a lambda to initialize the elements?

```
template <class U, class Abi2> datapar(datapar<U, Abi2> x);
```

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

- `Abi` and `Abi2` are equal and `U` and `T` are different integral types and `make_signed<U>::type` equals `make_signed<T>::type`, or
- at least one of `Abi` or `Abi2` is an instantiation of `datapar_abi::fixed_size` and `size() == x.size()` and `U` is implicitly convertible to `T`.

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

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

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

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

7 *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.

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

8 *Effects:* Constructs an object where the i-th element is initialized to `k[i] ? static_cast<T>(mem[i]) : 0` for all `i` ∈ `[0, size())`.

9 *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.

10 *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.

4.1.2.3 `datapar` load functions [datapar.load]

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

1 *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())`.

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

3 *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.

```
template <class U, class Flags> void copy_from(const U *mem, mask_type k, Flags);
```

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

5 *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. [*Note:* Masked loads only access the bytes in memory selected by the `k` argument. This prohibits implementations that load the complete vector before blending with the previous values. — *end note*]

6 *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.

4.1.2.4 `datapar` store functions [datapar.store]

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

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

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

3 *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.

```
template <class U, class Flags> void copy_to(U *mem, mask_type k, Flags);
```

4 *Effects:* Copies each `datapar` element i where `k[i]` is `true` as if `mem[i] = static_cast<U>(operator[](i))` for all `i` ∈ `[0, size())`.

5 *Remarks:* If the largest i where `k[i]` is `true` is greater than the number of values pointed to by `mem`, the behavior is undefined. [*Note:* Masked stores only write to the bytes in memory selected by the `k` argument. This prohibits implementations that load, blend, and store the complete vector. — *end note*]

6 *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.

4.1.2.5 `datapar` subscript operators [datapar.subscr]

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

1 *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;
```

2 *Returns:* A copy of the i-th element.

4.1.2.6 `datapar` unary operators [datapar.unary]

```
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;
```

14 *Requires:* The first template argument `T` to `datapar` must be an integral type.

15 *Returns:* A `datapar` object where each bit is the inverse of the corresponding bit in `*this`.

16 *Remarks:* `datapar::operator~()` shall not participate in overload resolution if `T` is a floating-point type.

```
datapar operator+() const;
```

17 *Returns:* A copy of `*this`

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

18 *Returns:* A `datapar` object where the i-th element is initialized to `-operator[](i)` for all `i` ∈ `[0, size())`.

4.1.2.7 `datapar` reductions [datapar.redu]

Note 4 These functions are now redundant because of non-member `reduce`. I prefer to keep these as shorthands, though.

```
value_type sum() const;
```

1 *Returns:* The sum of all the elements stored in the `datapar` object. The order of summation is arbitrary.

```
value_type sum(mask_type) const;
```

2 *Returns:* The sum of all the elements stored in the `datapar` object where the corresponding element in the first argument is `true`. The order of summation is arbitrary. If all elements in the first argument are `false`, then the return value is 0.

```
value_type product() const;
```

3 *Returns:* The product of all the elements stored in the `datapar` object. The order of multiplication is arbitrary.

```
value_type product(mask_type) const;
```

4 *Returns:* The product of all the elements stored in the `datapar` object where the corresponding element in the first argument is `true`. The order of multiplication is arbitrary. If all elements in the first argument are `false`, then the return value is 1.

```
value_type min() const;
```

5 *Returns:* The value of an element j for which `operator[](j) <= operator[](i)` for all `i` ∈ `[0, size())`.

```
value_type min(mask_type k) const;
```

6 *Returns:* The value of an element j for which `k[j] == true` and `operator[](j) <= operator[](i) || !k[i]` for all `i` ∈ `[0, size())`.

7 *Remarks:* If all elements in `k` are `false`, the return value is undefined.

Note 5 Alternatively, it could return `numeric_limits<value_type>::min()`.

```
value_type max() const;
```

8 *Returns:* The value of an element j for which `operator[](j) >= operator[](i)` for all `i` ∈ `[0, size())`.

```
value_type max(mask_type k) const;
```

9 *Returns:* The value of an element j for which `k[j] == true` and `operator[](j) >= operator[](i) || !k[i]` for all `i` ∈ `[0, size())`.

10 *Remarks:* If all elements in `k` are `false`, the return value is undefined.

Note 6 Alternatively, it could return `numeric_limits<value_type>::max()`.

4.1.3 `datapar` non-member operations [datapar.nonmembers]

4.1.3.1 `datapar` binary operators [datapar.binary]

```
template <class L, class R> using datapar_return_type = ...;  // exposition only
template <class T, class Abi, class U>
datapar_return_type<datapar<T, Abi>, U> operator+ (datapar<T, Abi>, const U &);
template <class T, class Abi, class U>
datapar_return_type<datapar<T, Abi>, U> operator- (datapar<T, Abi>, const U &);
template <class T, class Abi, class U>
datapar_return_type<datapar<T, Abi>, U> operator* (datapar<T, Abi>, const U &);
template <class T, class Abi, class U>
datapar_return_type<datapar<T, Abi>, U> operator/ (datapar<T, Abi>, const U &);
template <class T, class Abi, class U>
datapar_return_type<datapar<T, Abi>, U> operator% (datapar<T, Abi>, const U &);
```

```
template <class T, class Abi, class U>
datapar_return_type<datapar<T, Abi>, U> operator& (datapar<T, Abi>, const U &);
template <class T, class Abi, class U>
datapar_return_type<datapar<T, Abi>, U> operator| (datapar<T, Abi>, const U &);
template <class T, class Abi, class U>
datapar_return_type<datapar<T, Abi>, U> operator^ (datapar<T, Abi>, const U &);
template <class T, class Abi, class U>
datapar_return_type<datapar<T, Abi>, U> operator<<(datapar<T, Abi>, const U &);
template <class T, class Abi, class U>
datapar_return_type<datapar<T, Abi>, U> operator>>(datapar<T, Abi>, const U &);
template <class T, class Abi, class U>
datapar_return_type<datapar<T, Abi>, U> operator+ (const U &, datapar<T, Abi>);
template <class T, class Abi, class U>
datapar_return_type<datapar<T, Abi>, U> operator- (const U &, datapar<T, Abi>);
template <class T, class Abi, class U>
datapar_return_type<datapar<T, Abi>, U> operator* (const U &, datapar<T, Abi>);
template <class T, class Abi, class U>
datapar_return_type<datapar<T, Abi>, U> operator/ (const U &, datapar<T, Abi>);
template <class T, class Abi, class U>
datapar_return_type<datapar<T, Abi>, U> operator% (const U &, datapar<T, Abi>);
template <class T, class Abi, class U>
datapar_return_type<datapar<T, Abi>, U> operator& (const U &, datapar<T, Abi>);
template <class T, class Abi, class U>
datapar_return_type<datapar<T, Abi>, U> operator| (const U &, datapar<T, Abi>);
template <class T, class Abi, class U>
datapar_return_type<datapar<T, Abi>, U> operator^ (const U &, datapar<T, Abi>);
template <class T, class Abi, class U>
datapar_return_type<datapar<T, Abi>, U> operator<<(const U &, datapar<T, Abi>);
template <class T, class Abi, class U>
datapar_return_type<datapar<T, Abi>, U> operator>>(const U &, datapar<T, Abi>);
```

1 *Remarks:* The return type of these operators (`datapar_return_type<datapar<T, Abi>, U>`) shall be deduced according to the following rules:

- Let *common*`(A, B)` identify the type:
  - `A` if `A` equals `B`.
  - Otherwise, `A` if `B` is not a fundamental arithmetic type.
  - Otherwise, `B` if `A` is not a fundamental arithmetic type.
  - Otherwise, `decltype(A() + B())` if any of the types `A` or `B` is a floating-point type.
  - Otherwise, `A` if `sizeof(A) > sizeof(B)`.
  - Otherwise, `B` if `sizeof(A) < sizeof(B)`.
  - Otherwise, `C` shall identify the type with greater integer conversion rank of the types `A` and `B` and:
    * `C` is used if `is_signed_v<A> == is_signed_v<B>`, and
    * `make_unsigned_t<C>` otherwise.
- Let *commonabi*`(V0, V1, T)` identify the type:
  - `V0::abi_type` if `V0::abi_type` equals `V1::abi_type`.
  - Otherwise, `abi_for_size_t<T, V0::size()>` if both `V0` and `V1` are implicitly convertible to `datapar<T, abi_for_size_t<T, V0::size()>>`.
  - Otherwise, `datapar_abi::fixed_size<V0::size()>`.

Note 7 See `https://github.com/VcDevel/Vc/blob/191e13b2268c630d8e14 tests/datapar.cpp#L435` for a test of an implementation of these rules.

Note 8 *unusual arithmetic conversions ;-)*

- If is_datapar_v<U> == true then the return type is datapar<*common*(T, U::value_type), *commonabi*(datapar<T, Abi>, U, *common*(T, U::value_type))>. [*Note:* This rule also matches if datapar_size_v<T, Abi> != U::size(). The overload resolution participation condition in the next paragraph discards the operator. — *end note*]
- Otherwise, if T is integral and U is int the return type shall be datapar<T, Abi>.
- Otherwise, if T is integral and U is unsigned int the return type shall be datapar<make_unsigned_t<T>, Abi>.
- Otherwise, if U is a fundamental arithmetic type then the return type shall be datapar<*common*(T, U), *commonabi*(datapar<T, Abi>, datapar<U, datapar_abi::fixed_size<datapar_size_v<T, Abi>>>, *common*(T, U))>.
- Otherwise, if U is convertible to int then the return type shall be datapar<*common*(T, U), *commonabi*(datapar<T, Abi>, datapar<int, datapar_abi::fixed_size<datapar_size_v<T, Abi>>>, *common*(T, U))>.
- Otherwise, if U is implicitly convertible to datapar<V, A>, where V and A are determined according to standard template type deduction, then the return type shall be datapar<*common*(T, V), *commonabi*(datapar<T, Abi>, datapar<V, A>, *common*(T, V))>.
- Otherwise, if U is implicitly convertible to datapar<T, Abi>, the return type shall be datapar<T, Abi>.
- Otherwise the operator does not participate in overload resolution.

Note 9 i.e. datapar<signed char>() + int() -> datapar<signed char>, however datapar<signed char>() + short() -> datapar<short, fixed_size<datapar_size_v<signed char>>>

Note 10 i.e. datapar<signed char> () + uint() -> datapar<unsigned char>, however datapar<signed char> () + ushort() -> datapar<ushort, fixed_size<datapar_size_v<signed char>>>

2 *Remarks:* Each of these operators only participate in overload resolution if all of the following hold:

- The indicated operator can be applied to objects of type datapar_return_type<datapar<T, Abi>, U>::value_type.
- datapar<T, Abi> is implicitly convertible to datapar_return_type<datapar<T, Abi>, U>.
- U is implicitly convertible to datapar_return_type<datapar<T, Abi>, U>.

Note 11 This seems a strange place to put this. Alternatively, modify the above rule to unconditionally use datapar<T, Abi>. The paragraph below would lead to the same effect.

3 *Remarks:* The operators with const U & as first parameter shall not participate in overload resolution if is_datapar_v<U> == true.

4 *Returns:* A datapar object initialized with the results of the component-wise application of the indicated operator after both operands have been converted to the return type.

Note 12 I think this allows datapar<float, fixed_size<4>>() + double() and returns datapar<double, fixed_size<4>>. I also think that's what we want.

4.1.3.2 datapar compound assignment [datapar.cassign]

```
template <class T, class Abi, class U> datapar<T, Abi> &operator+= (datapar<T, Abi> &, const U &);
template <class T, class Abi, class U> datapar<T, Abi> &operator-= (datapar<T, Abi> &, const U &);
template <class T, class Abi, class U> datapar<T, Abi> &operator*= (datapar<T, Abi> &, const U &);
template <class T, class Abi, class U> datapar<T, Abi> &operator/= (datapar<T, Abi> &, const U &);
template <class T, class Abi, class U> datapar<T, Abi> &operator%= (datapar<T, Abi> &, const U &);
template <class T, class Abi, class U> datapar<T, Abi> &operator&= (datapar<T, Abi> &, const U &);
template <class T, class Abi, class U> datapar<T, Abi> &operator|= (datapar<T, Abi> &, const U &);
template <class T, class Abi, class U> datapar<T, Abi> &operator^= (datapar<T, Abi> &, const U &);
template <class T, class Abi, class U> datapar<T, Abi> &operator<<=(datapar<T, Abi> &, const U &);
template <class T, class Abi, class U> datapar<T, Abi> &operator>>=(datapar<T, Abi> &, const U &);
```

1 *Remarks:* Each of these operators only participates in overload resolution if all of the following hold:

- The indicated operator can be applied to objects of type `datapar_return_type<datapar<T, Abi>, U>::value_type`.
- `datapar<T, Abi>` is implicitly convertible to `datapar_return_type<datapar<T, Abi>, U>`.
- `U` is implicitly convertible to `datapar_return_type<datapar<T, Abi>, U>`.
- `datapar_return_type<datapar<T, Abi>, U>` is implicitly convertible to `datapar<T, Abi>`.

2 *Effects:* Each of these operators performs the indicated operation component-wise on each of the elements of the first argument and the corresponding element of the second argument after conversion to `datapar<T, Abi>`.

3 *Returns:* A reference to the first argument.

4.1.3.3 `datapar` logical operators [datapar.logical]

Note: The omission of logical operators is deliberate.

4.1.3.4 `datapar` compare operators [datapar.comparison]

```
template <class T, class Abi, class U>
typename datapar_return_type<datapar<T, Abi>, U>::mask_type operator==(datapar<T, Abi>, const U &);
template <class T, class Abi, class U>
typename datapar_return_type<datapar<T, Abi>, U>::mask_type operator!=(datapar<T, Abi>, const U &);
template <class T, class Abi, class U>
typename datapar_return_type<datapar<T, Abi>, U>::mask_type operator>=(datapar<T, Abi>, const U &);
template <class T, class Abi, class U>
typename datapar_return_type<datapar<T, Abi>, U>::mask_type operator<=(datapar<T, Abi>, const U &);
template <class T, class Abi, class U>
typename datapar_return_type<datapar<T, Abi>, U>::mask_type operator> (datapar<T, Abi>, const U &);
template <class T, class Abi, class U>
typename datapar_return_type<datapar<T, Abi>, U>::mask_type operator< (datapar<T, Abi>, const U &);
template <class T, class Abi, class U>
typename datapar_return_type<datapar<T, Abi>, U>::mask_type operator==(const U &, datapar<T, Abi>);
template <class T, class Abi, class U>
typename datapar_return_type<datapar<T, Abi>, U>::mask_type operator!=(const U &, datapar<T, Abi>);
template <class T, class Abi, class U>
typename datapar_return_type<datapar<T, Abi>, U>::mask_type operator>=(const U &, datapar<T, Abi>);
template <class T, class Abi, class U>
typename datapar_return_type<datapar<T, Abi>, U>::mask_type operator<=(const U &, datapar<T, Abi>);
template <class T, class Abi, class U>
typename datapar_return_type<datapar<T, Abi>, U>::mask_type operator> (const U &, datapar<T, Abi>);
template <class T, class Abi, class U>
typename datapar_return_type<datapar<T, Abi>, U>::mask_type operator< (const U &, datapar<T, Abi>);
```

1 *Remarks:* The return type of these operators shall be the `mask_type` member type of the type deduced according to the rules defined in [datapar.binary].

2 *Remarks:* Each of these operators only participates in overload resolution if all of the following hold:

- `datapar<T, Abi>` is implicitly convertible to `datapar_return_type<datapar<T, Abi>, U>`.
- `U` is implicitly convertible to `datapar_return_type<datapar<T, Abi>, U>`.

3 *Remarks:* The operators with `const U &` as first parameter shall not participate in overload resolution if `is_datapar_v<U> == true`.

4 *Returns:* A `mask` object initialized with the results of the component-wise application of the indicated operator after both operands have been converted to `datapar_return_type<datapar<T, Abi>, U>`.

4.1.3.5 `datapar` non-member reductions [datapar.reductions]

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

1 *Returns:* `GENERALIZED_SUM(binary_op, x.data[i], …)` for all `i` $\in$ `[0, size())`.

2 *Requires:* `binary_op` shall be callable on arguments of type `T` and arguments of type `datapar<T, A1>`, where `A1` may be different to `Abi`.

3 *Note:* This overload of `reduce` does not require an initial value because `x` is guaranteed to be non-empty.

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

4 *Returns:* `GENERALIZED_SUM(binary_op, init, x.data[i], …)` for all `i` $\in \{j \in \mathbb{N}_0 | j <$ `size()` $\bigwedge$ `x.mask[`j`]`$\}$.

5 *Requires:* `binary_op` shall be callable on arguments of type `T` and arguments of type `datapar<T, A1>`, where `A1` may be different to `Abi`.

6 *Note:* This overload of `reduce` requires an initial value because `x` may be empty.

4.1.3.6 `datapar` casts [datapar.casts]

```
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...);
```

1 *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()`.

2 *Returns:* A `datapar` object initialized with the converted values as one object of `T` or an array of `T`. All scalar elements $\mathtt{x}_\mathtt{i}$ of the function argument(s) are converted as if $\mathtt{y}_\mathtt{i}$ `= static_cast<typename T::value_type>(`$\mathtt{x}_\mathtt{i}$`)` is executed. The resulting $\mathtt{y}_\mathtt{i}$ intialize 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<typename T::value_type>(x[i * T::size() + j])`. — *end note*]

4.1.3.7 `datapar` math library [datapar.math]

```
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 llongv = 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 = datapar<T, abi_for_size_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> acosf(floatv<Abi> x);
template <class Abi> ldoublev<Abi> acosl(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> asinf(floatv<Abi> x);
template <class Abi> ldoublev<Abi> asinl(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> atanf(floatv<Abi> x);
template <class Abi> ldoublev<Abi> atanl(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> atan2f(floatv<Abi> y, floatv<Abi> x);
template <class Abi> ldoublev<Abi> atan2l(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> cosf(floatv<Abi> x);
template <class Abi> ldoublev<Abi> cosl(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> sinf(floatv<Abi> x);
template <class Abi> ldoublev<Abi> sinl(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> tanf(floatv<Abi> x);
template <class Abi> ldoublev<Abi> tanl(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> acoshf(floatv<Abi> x);
template <class Abi> ldoublev<Abi> acoshl(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> asinhf(floatv<Abi> x);
template <class Abi> ldoublev<Abi> asinhl(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> atanhf(floatv<Abi> x);
template <class Abi> ldoublev<Abi> atanhl(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> coshf(floatv<Abi> x);
template <class Abi> ldoublev<Abi> coshl(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> sinhf(floatv<Abi> x);
template <class Abi> ldoublev<Abi> sinhl(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> tanhf(floatv<Abi> x);
template <class Abi> ldoublev<Abi> tanhl(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> expf(floatv<Abi> x);
template <class Abi> ldoublev<Abi> expl(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> exp2f(floatv<Abi> x);
template <class Abi> ldoublev<Abi> exp2l(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> expm1f(floatv<Abi> x);
template <class Abi> ldoublev<Abi> expm1l(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> frexpf(floatv<Abi> value, samesize<int, floatv<Abi>>* exp);
template <class Abi> ldoublev<Abi> frexpl(ldoublev<Abi> value, samesize<int, ldoublev<Abi>>* exp);

template <class Abi> samesize<int, floatv<Abi>> ilogb(floatv<Abi> x);
template <class Abi> samesize<int, doublev<Abi>> ilogb(doublev<Abi> x);
template <class Abi> samesize<int, ldoublev<Abi>> ilogb(ldoublev<Abi> x);
template <class Abi> samesize<int, floatv<Abi>> ilogbf(floatv<Abi> x);
template <class Abi> samesize<int, ldoublev<Abi>> ilogbl(ldoublev<Abi> x);

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> ldexpf(floatv<Abi> x, samesize<int, floatv<Abi>> exp);
template <class Abi> ldoublev<Abi> ldexpl(ldoublev<Abi> x, samesize<int, ldoublev<Abi>> exp);

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> logf(floatv<Abi> x);
template <class Abi> ldoublev<Abi> logl(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> log10f(floatv<Abi> x);
template <class Abi> ldoublev<Abi> log10l(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> log1pf(floatv<Abi> x);
template <class Abi> ldoublev<Abi> log1pl(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> log2f(floatv<Abi> x);
template <class Abi> ldoublev<Abi> log2l(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> logbf(floatv<Abi> x);
template <class Abi> ldoublev<Abi> logbl(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> modff(floatv<Abi> value, floatv<Abi>* iptr);
template <class Abi> ldoublev<Abi> modfl(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> scalbnf(floatv<Abi> x, samesize<int, floatv<Abi>> n);
template <class Abi> ldoublev<Abi> scalbnl(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> scalblnf(floatv<Abi> x, samesize<long int, floatv<Abi>> n);
template <class Abi> ldoublev<Abi> scalblnl(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> floatv<Abi> cbrtf(floatv<Abi> x);
template <class Abi> ldoublev<Abi> cbrtl(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> longv<Abi> labs(longv<Abi> j);
template <class Abi> llongv<Abi> llabs(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> fabs(floatv<Abi> x);
template <class Abi> doublev<Abi> fabs(doublev<Abi> x);
template <class Abi> ldoublev<Abi> fabs(ldoublev<Abi> x);
template <class Abi> floatv<Abi> fabsf(floatv<Abi> x);
template <class Abi> ldoublev<Abi> fabsl(ldoublev<Abi> x);

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(doublev<Abi> x, doublev<Abi> y);
template <class Abi> floatv<Abi> hypotf(floatv<Abi> x, floatv<Abi> y);
template <class Abi> ldoublev<Abi> hypotl(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> powf(floatv<Abi> x, floatv<Abi> y);
template <class Abi> ldoublev<Abi> powl(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> sqrtf(floatv<Abi> x);
template <class Abi> ldoublev<Abi> sqrtl(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> erff(floatv<Abi> x);
template <class Abi> ldoublev<Abi> erfl(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> erfcf(floatv<Abi> x);
template <class Abi> ldoublev<Abi> erfcl(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> lgammaf(floatv<Abi> x);
template <class Abi> ldoublev<Abi> lgammal(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> tgammaf(floatv<Abi> x);
template <class Abi> ldoublev<Abi> tgammal(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> floatv<Abi> ceilf(floatv<Abi> x);
template <class Abi> ldoublev<Abi> ceill(ldoublev<Abi> x);

template <class Abi> floatv<Abi> floor(floatv<Abi> x);
template <class Abi> doublev<Abi> floor(doublev<Abi> x);
template <class Abi> ldoublev<Abi> floor(ldoublev<Abi> x);
template <class Abi> floatv<Abi> floorf(floatv<Abi> x);
template <class Abi> ldoublev<Abi> floorl(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> floatv<Abi> nearbyintf(floatv<Abi> x);
template <class Abi> ldoublev<Abi> nearbyintl(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> rintf(floatv<Abi> x);
template <class Abi> ldoublev<Abi> rintl(ldoublev<Abi> x);

template <class Abi> samesize<long int, floatv<Abi>> lrint(floatv<Abi> x);
template <class Abi> samesize<long int, doublev<Abi>> lrint(doublev<Abi> x);
template <class Abi> samesize<long int, ldoublev<Abi>> lrint(ldoublev<Abi> x);
template <class Abi> samesize<long int, floatv<Abi>> lrintf(floatv<Abi> x);
template <class Abi> samesize<long int, ldoublev<Abi>> lrintl(ldoublev<Abi> x);

template <class Abi> samesize<long long int, floatv<Abi>> llrint(floatv<Abi> x);
template <class Abi> samesize<long long int, doublev<Abi>> llrint(doublev<Abi> x);
template <class Abi> samesize<long long int, ldoublev<Abi>> llrint(ldoublev<Abi> x);
template <class Abi> samesize<long long int, floatv<Abi>> llrintf(floatv<Abi> x);
template <class Abi> samesize<long long int, ldoublev<Abi>> llrintl(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> roundf(floatv<Abi> x);
template <class Abi> ldoublev<Abi> roundl(ldoublev<Abi> x);

template <class Abi> samesize<long int, floatv<Abi>> lround(floatv<Abi> x);
template <class Abi> samesize<long int, doublev<Abi>> lround(doublev<Abi> x);
template <class Abi> samesize<long int, ldoublev<Abi>> lround(ldoublev<Abi> x);
template <class Abi> samesize<long int, floatv<Abi>> lroundf(floatv<Abi> x);
template <class Abi> samesize<long int, ldoublev<Abi>> lroundl(ldoublev<Abi> x);

template <class Abi> samesize<long long int, floatv<Abi>> llround(floatv<Abi> x);
template <class Abi> samesize<long long int, doublev<Abi>> llround(doublev<Abi> x);
template <class Abi> samesize<long long int, ldoublev<Abi>> llround(ldoublev<Abi> x);
template <class Abi> samesize<long long int, floatv<Abi>> llroundf(floatv<Abi> x);
template <class Abi> samesize<long long int, ldoublev<Abi>> llroundl(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> truncf(floatv<Abi> x);
template <class Abi> ldoublev<Abi> truncl(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> fmodf(floatv<Abi> x, floatv<Abi> y);
template <class Abi> ldoublev<Abi> fmodl(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> remainderf(floatv<Abi> x, floatv<Abi> y);
template <class Abi> ldoublev<Abi> remainderl(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> remquof(floatv<Abi> x, floatv<Abi> y, samesize<int, floatv<Abi>>* quo);
template <class Abi>
ldoublev<Abi> remquol(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> floatv<Abi> copysignf(floatv<Abi> x, floatv<Abi> y);
template <class Abi> ldoublev<Abi> copysignl(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> ldoublev<Abi> nextafter(ldoublev<Abi> x, ldoublev<Abi> y);
template <class Abi> floatv<Abi> nextafterf(floatv<Abi> x, floatv<Abi> y);
template <class Abi> ldoublev<Abi> nextafterl(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> ldoublev<Abi> nexttoward(ldoublev<Abi> x, ldoublev<Abi> y);
template <class Abi> floatv<Abi> nexttowardf(floatv<Abi> x, ldoublev<Abi> y);
template <class Abi> ldoublev<Abi> nexttowardl(ldoublev<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> fdim(ldoublev<Abi> x, ldoublev<Abi> y);
template <class Abi> floatv<Abi> fdimf(floatv<Abi> x, floatv<Abi> y);
template <class Abi> ldoublev<Abi> fdiml(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> fmaxf(floatv<Abi> x, floatv<Abi> y);
template <class Abi> ldoublev<Abi> fmaxl(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> fminf(floatv<Abi> x, floatv<Abi> y);
template <class Abi> ldoublev<Abi> fminl(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> floatv<Abi> fmaf(floatv<Abi> x, floatv<Abi> y, floatv<Abi> z);
template <class Abi> ldoublev<Abi> fmal(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> mask<float, Abi> isnormal(floatv<Abi> x);
template <class Abi> mask<double, Abi> isnormal(doublev<Abi> x);
template <class Abi> mask<long double, 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);

    template <class V> struct datapar_div_t { V quot, rem; };
    template <class Abi> datapar_div_t<intv<Abi>> div(intv<Abi> numer, intv<Abi> denom);
    template <class Abi> datapar_div_t<longv<Abi>> div(longv<Abi> numer, longv<Abi> denom);
    template <class Abi> datapar_div_t<llongv<Abi>> div(llongv<Abi> numer, llongv<Abi> denom);
    template <class Abi> datapar_div_t<longv<Abi>> ldiv(longv<Abi> numer, longv<Abi> denom);
    template <class Abi> datapar_div_t<llongv<Abi>> lldiv(llongv<Abi> numer, llongv<Abi> denom);
  }
}
```

1 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.

Note 13 Neither the C nor the C++ standard say anything about expected error/precision. It seems returning 0 from all functions is a conforming implementation — just bad QoI.

2 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.

4.1.4 Class template `mask` [mask]

4.1.4.1 Class template `mask` overview [mask.overview]

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

      static constexpr size_type size();

      mask() = default;

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

      // implicit broadcast constructor
      mask(value_type);

      // implicit type conversion constructor
      template <class U, class Abi2> mask(mask<U, Abi2>);

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

      // loads [mask.load]
      template <class Flags> void copy_from(const value_type *mem, Flags);
      template <class Flags> void copy_from(const value_type *mem, mask k, Flags);

      // stores [mask.store]
      template <class Flags> void copy_to(value_type *mem, Flags) const;
```

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

      // scalar access:
      reference operator[](size_type);
      value_type operator[](size_type) const;

      // negation:
      mask operator!() const;
    };
  }
}
```

1 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>`.

2 The first template argument `T` must be an integral or floating-point fundamental type. The type `bool` is not allowed.

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

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

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

5 *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`:

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

4.1.4.2 `mask` constructors [mask.ctor]

```
mask() = default;
```

1 *Effects:* Constructs an object with all elements initialized to `bool()`. [*Note:* This zero-initializes the object. — *end note*]

```
mask(value_type);
```

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

```
template <class U, class Abi2> mask(mask<U, Abi2> x);
```

3 *Remarks:* This constructor shall not participate in overload resolution unless `datapar<U, Abi2>` is implicitly convertible to `datapar<T, Abi>`.

4 *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);
```

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

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

7 *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.

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

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

9 *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.

10 *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.

4.1.4.3 `mask` load function [mask.load]

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

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

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

3 *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.

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

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

5 *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.

6 *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.

4.1.4.4 `mask` store functions [mask.store]

```
template <class Flags> void copy_to(value_type *mem, Flags);
```

1 *Effects:* Copies all `mask` elements as if `mem[i] = operator[](i)` for all `i` ∈ `[0, size())`.

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

3 *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.

```
template <class Flags> void copy_to(value_type *mem, mask k, Flags);
```

4 *Effects:* Copies each `mask` element i where `k[i]` is `true` as if `mem[i] = operator[](i)` for all `i` ∈ `[0, size())`.

5 *Remarks:* If the largest i where `k[i]` is `true` is greater than the number of values pointed to by `mem`, the behavior is undefined. [*Note:* Masked stores only write to the bytes in memory selected by the `k` argument. This prohibits implementations that load, blend, and store the complete vector. — *end note*]

6 *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.

4.1.4.5 `mask` subscript operators [mask.subscr]

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

1 *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;
```

2 *Returns:* A copy of the i-th element.

4.1.4.6 `mask` unary operators [mask.unary]

```
mask operator!() const;
```

1 *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 [mask.nonmembers]

4.1.5.1 `mask` binary operators [mask.binary]

```
template <class T0, class A0, class T1, class A1> using mask_return_type = ...   // exposition only
template <class T0, class A0, class T1, class A1>
mask_return_type<T0, A0, T1, A1> operator&&(const mask<T0, A0> &, const mask<T1, A1> &);
template <class T0, class A0, class T1, class A1>
mask_return_type<T0, A0, T1, A1> operator||(const mask<T0, A0> &, const mask<T1, A1> &);
template <class T0, class A0, class T1, class A1>
mask_return_type<T0, A0, T1, A1> operator& (const mask<T0, A0> &, const mask<T1, A1> &);
template <class T0, class A0, class T1, class A1>
mask_return_type<T0, A0, T1, A1> operator| (const mask<T0, A0> &, const mask<T1, A1> &);
template <class T0, class A0, class T1, class A1>
mask_return_type<T0, A0, T1, A1> operator^ (const mask<T0, A0> &, const mask<T1, A1> &);
```

Note 14 I think we need an additional `operator@(const mask<T, A> &, const mask<T, A> &)` overload to enable use of objects that are implicitly convertible to more than one `mask` instantiation with equal priority.

1 *Remarks:* The return type, `mask_return_type<T, Abi, U>`, shall be `mask<`*`common`*`(T0, T1),` *`commonabi`*`(datapar<T0, A0>, datapar<T1, A1>,` *`common`*`(T0, T1))>`. The functions *`common`* and *`commonabi`* identify the functions described in 4.1.3.1 p.1.

2 *Remarks:* Each of these operators only participate in overload resolution if both arguments are implicitly convertible to `mask_return_type<T, Abi, U>`.

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

4.1.5.2 `mask` compares [mask.comparison]

```
template <class T0, class A0, class T1, class A1>
bool operator==(const mask<T0, A0> &, const mask<T1, A1> &);
```

Note 15 ditto.

1 *Remarks:* This operator only participates in overload resolution if `mask<T0, A0>` is implicitly convertible to `mask<T1, A1>` or `mask<T1, A1>` is implicitly convertible to `mask<T0, A0>`.

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

```
template <class T0, class A0, class T1, class A1>
bool operator!=(const mask<T0, A0> &a, const mask<T1, A1> &b);
```

Note 16 ditto.

3 *Remarks:* This operator only participates in overload resolution if `mask<T0, A0>` is implicitly convertible to `mask<T1, A1>` or `mask<T1, A1>` is implicitly convertible to `mask<T0, A0>`.

4 *Returns:* `!operator==(a, b)`.

4.1.5.3 `mask` reductions [mask.reductions]

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

1 *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>);
constexpr bool  any_of(bool);
```

2 *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>);
constexpr bool none_of(bool);
```

3 *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>);
constexpr bool some_of(bool);
```

4 *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.

5 *Note:* `some_of(bool)` unconditionally returns `false`.

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

6 *Returns:* The number of boolean elements that are `true`.

```
template <class T, class Abi> int find_first_set(mask<T, Abi> m);
```

7 *Returns:* The lowest element index `i` where `m[i] == true`.

8 *Remarks:* If `none_of(m) == true` the behavior is undefined.

```
template <class T, class Abi> int find_last_set(mask<T, Abi> m);
```

9 *Returns:* The highest element index `i` where `m[i] == true`.

10 *Remarks:* If `none_of(m) == true` the behavior is undefined.

```
constexpr int find_first_set(bool);
constexpr int find_last_set(bool);
```

11 *Returns:* 0 if the argument is `true`.

4.1.5.4 Masked assignment [mask.where]

```
template <class T0, class A0, class T1, class A1>
implementation_defined where(const mask<T0, A0> &m, datapar<T1, A1> &v);
```

Note 17 Since we have class template deduction, should I drop the function and have only a `where` class instead.

1 *Remarks:* The function only participates in overload resolution if `mask<T0, A0>` is implicitly convertible to `mask<T1, A1>`.

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

- The object is not *CopyConstructible*.
- Assignment and compound assignment operators only participate in overload resolution if the corresponding operator for `datapar<T1, A1>` is usable.
- *Effects:* Assignment and compound assignment implement the same semantics as the corresponding operator for `datapar<T1, A1>` with the exception that elements of `v` stay unmodified if the corresponding boolean element in `m` is `false`.
- The assignment and compound assignment operators return `void`.

```
template <class T> implementation_defined where(bool, T &);
```

3 *Remarks:* The function only participates in overload resolution if `T` is a fundamental arithmetic type.

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

- The object is not *CopyConstructible*.

- Assignment and compound assignment operators only participate in overload resolution if the corresponding operator for `T` is usable.
- *Effects:* If the first argument is `false`, the assignment operators do nothing. If the first argument is `true`, the assignment operators forward to the corresponding builtin assignment operator.
- The assignment and compound assignment operators return `void`.

---

5 DISCUSSION

5.1 MEMBER TYPES

The member types may not seem obvious. Rationales:

`value_type`
: In the spirit of the `value_type` member of STL containers, this type denotes the *logical* type of the values in the vector.

`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 DEFAULT CONSTRUCTION

The default constructors of `datapar` and `mask` zero-initialize the object. This is important for compatibility with `T()`, which zero-initializes fundamental types. There may be a concern that unnecessary initialization could lead to unnecessary instructions. I consider this a QoI issue. Implementations are certainly able to recognize unnecessary initializations in many cases.

5.3 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.

5.4 BROADCAST CONSTRUCTOR

The broadcast constructor is not declared as `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.

5.5 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?

SF	F	N	A	SA
0	6	10	2	1

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

SF	F	N	A	SA
1	7	10	0	0

5.6 PARAMETERS OF BINARY AND COMPARE OPERATORS

It is easier to implement and possibly easier to specify these operators if the signature is specified as:

```
template <class T, class U>
datapar_return_type<T, U> operator+(const T &, const U &);
```

The motivation for using the variant where at least one function parameter is constrained to the `datapar` class template is compilation speed and less diagnostics noise in error cases. The compiler can drop the operator from the overload resolution set without having to go through a substitution failure. With concepts it might be worthwhile to revisit this decision.

5.7 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.

5.8 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`.
- The object returned from the `where` function.
- 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

```cpp
template <class T, size_t N = datapar_size_v<T, datapar_abi::compatible<T>>,
          class Abi = datapar_abi::compatible<T>>
class datapar;
```

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

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.

5.9 FUNDAMENTAL SIMD TYPE OR NOT?

5.9.1 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.

5.9.2 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 builtin 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).

5.9.3 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:

```
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:

```
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.

5.9.4 progress

SG1 Guidance at JAX 2016:
Poll: Specify datapar width using ABI tag, with a special template tag for fixed size.

SF	F	N	A	SA
3	7	0	0	1

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

SF	F	N	A	SA
1	2	5	2	1

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 `datapar` objects is equal or a builtin arithmetic type is broadcast).

Examples:

1. Mixed `int`−`float` operations

```
1 using floatv = datapar<float>;  // native ABI
2 using float_sized_abi = datapar_abi::fixed_size<floatv::size()>;
3 using intv = datapar<int, float_sized_abi>;
4
5 auto x = floatv() + intv();
6 intv y = floatv() + intv();
```

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

2. Native `int` vectors

```
1 using intv = datapar<int>;  // native ABI
2 using int_sized_abi = datapar_abi::fixed_size<intv::size()>;
3 using floatv = datapar<float, int_sized_abi>;
4
5 auto x = floatv() + intv();
6 intv y = floatv() + intv();
```

Line 5 is well-formed: It states that N (= `intv::size()`) additions shall be executed concurrently. The type of `x` is `datapar<float_v, int_sized_abi>` (i.e. `floatv`) and never `datapar<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, datapar_abi::fixed_size<N>>` to `datapar<U, Abi>` is allowed whenever `N == datapar<U, Abi>::size()`.

5.10 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?

SF	F	N	A	SA
0	6	3	3	0

5.11 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
- 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

1 Whether that's a reasonable fear is a different discussion.

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.[2] 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?

SF	F	N	A	SA
2	6	5	0	0

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

SF	F	N	A	SA
0	0	7	4	1

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

5.12 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`.

2 As I realized too late.

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`.

A 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).

B BIBLIOGRAPHY

[1] Matthias Kretz. "Extending C++ for Explicit Data-Parallel Programming via SIMD Vector Types." Frankfurt (Main), Univ. PhD thesis. 2015. DOI: 10.13140/RG.2.1.2355.4323. URL: http://publikationen.ub.uni-frankfurt.de/frontdoor/index/index/docId/38415.

[N4184] Matthias Kretz. *N4184: SIMD Types: The Vector Type & Operations*. ISO/IEC C++ Standards Committee Paper. 2014. URL: http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2014/n4184.pdf.

[N4185] Matthias Kretz. *N4185: SIMD Types: The Mask Type & Write-Masking*. ISO/IEC C++ Standards Committee Paper. 2014. URL: http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2014/n4185.pdf.

[N4395] Matthias Kretz. *N4395: SIMD Types: ABI Considerations*. ISO/IEC C++ Standards Committee Paper. 2015. URL: http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2015/n4395.pdf.

[N4454] Matthias Kretz. *N4454: SIMD Types Example: Matrix Multiplication*. ISO/IEC C++ Standards Committee Paper. 2015. URL: http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2015/n4454.pdf.

[P0214R0] Matthias Kretz. *P0214R0: Data-Parallel Vector Types & Operations*. ISO/IEC C++ Standards Committee Paper. 2016. URL: http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2016/p0214r0.pdf.

[P0214R1] Matthias Kretz. *P0214R1: Data-Parallel Vector Types & Operations*. ISO/IEC C++ Standards Committee Paper. 2016. URL: http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2016/p0214r1.pdf.