ABSTRACT

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

CONTENTS

0 Remarks
1 Introduction
  1.1 SIMD Registers and Operations
  1.2 Motivation for Data-Parallel Types
  1.3 Problem
2 Wording
Data-Parallel Types
  Header <datapar> synopsis
  Class template datapar
  datapar non-member operations
  Class template mask
  mask non-member operations
3 Discussion

3.1 Member Types .................................................. 21
3.2 Conversions ..................................................... 22
3.3 Broadcast Constructor ......................................... 22
3.4 Aliasing of Subscript Operators .............................. 22
3.5 Parameters of Binary and Compare Operators ............... 23
3.6 Compound Assignment .......................................... 23
3.7 Return Type of Masked Assignment Operators ............... 23
3.8 Fundamental SIMD Type or Not? ............................ 24
   3.8.1 The Issue .................................................. 24
   3.8.2 Standpoints ............................................... 24
   3.8.3 Issues ...................................................... 25

A Acknowledgements .............................................. 26

B Bibliography ..................................................... 26
0 REMARKS

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

• In the following $\mathcal{W}_T$ 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 starting point. Once the foundation is settled on the API will be completed.

1 INTRODUCTION

(Contains minor improvements and clarifications compared to the earlier revisions.)

1.1 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 $\mathcal{W}_T$ successive scalar values starting from a given address into a SIMD register.

• Store instructions: store from a SIMD register to $\mathcal{W}_T$ successive scalar values at a given address.

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

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

• Bitwise instructions: bitwise operations on SIMD registers.
1 Introduction

- 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.

1.2 Motivation for data-parallel types

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

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

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

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

The challenge remains in providing portable SIMD types and operations.
1.3 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.)

2 Wording

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

2.1 Data-Parallel Types

2.1.1 Header

```cpp
namespace std {
    namespace experimental {
        namespace datapar_abi {
            struct scalar {}; // always present
            typedef implementation_defined compatible; // always present
            typedef implementation_defined native; // always present
        }
        struct unaligned_tag {};
        struct aligned_tag {};
        // traits
        template <class T> struct is_datapar {
            static bool value = false;
        }
        template <class T> struct is_mask {
            static bool value = false;
        }
        template <class T, size_t N> struct abi_for_size {
            typedef implementation_defined type;
        }
        template <size_t N> using abi_for_size_t = typename abi_for_size<N>::type;
        template <class T, class Abi = implementation_defined> struct datapar_size : public integral_constant<
            size_t, implementation_defined> {};
        template <class T, class Abi = implementation_defined> constexpr size_t datapar_size_v = datapar_size<T, Abi>::value;
    }
}
```

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

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

// compound assignment [dataparassign]
template <class T, class Abi, class U> datapar<T, Abi> & operator= (datapar<T, Abi> & T, Abi> & U, const U &);  // exposition only
template <class T, class Abi, class U> datapar<T, Abi> & operator+= (datapar<T, Abi> & T, Abi> & U, const U &);  
template <class T, class Abi, class U> datapar<T, Abi> & operator-= (datapar<T, Abi> & T, Abi> & U, const U &);

template <class T, class Abi, class U> operator= (datapar<T, Abi> & T, Abi> & U, const U &);  

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

template <class T, class Abi, class U> datapar_return_type<T, Abi>, U > operator+ (datapar<T, Abi>, T, Abi> & U, 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>);

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

// 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 compares [mask.comparison]
template <class T, class Abi, class U> bool operator==(mask<T, Abi>, const U &);
template <class T, class Abi, class U> bool operator!=(mask<T, Abi>, const U &);
template <class T, class Abi, class U> bool operator==(const U &, mask<T, Abi>);
template <class T, class Abi, class U> bool operator!=(const U &, mask<T, Abi>);

// reductions [mask.reductions]
template <class T, class Abi> bool all_of(mask<T, Abi>);
constexpr bool any_of(bool);
template <class T, class Abi> bool none_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);

// masked assignment [mask.where]
template <class T, class U, class Abi> implementation_defined where(mask<U, Abi>, datapar<T, Abi> &)
template <class T> implementation_defined where(bool, T &);
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.

```cpp
namespace datapar_abi {
    struct scalar {}; // implementation-defined tag types, e.g. sse, avx, avx512, neon, ...
    typedef implementation_defined compatible;
    typedef implementation_defined native;
}
```

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

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).

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

The `datapar_abi::compatible` tag is defined by the implementation to alias the tag type with the most efficient data parallel execution that ensures the highest compatibility on the target architecture.

The `datapar_abi::native` tag is defined by the implementation to alias the tag type with the most efficient data parallel execution that is supported on the target system.

### 2.1.1.1 datapar type traits

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

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

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

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

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

The `abi_for_size` class template defines the member type `type` to one of the tag types in `datapar_abi` or not at all, depending on the value of the template parameters.

`datapar<T, abi_for_size_t<N>::size>()` shall return `N` or result in a substitution failure.

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

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

`datapar_size<T, Abi>::value` shall result in a substitution failure if any of the template arguments `T` or `Abi` are invalid template arguments to `datapar`. 

---

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).
3 An implementation may choose to implement data-parallel execution for many different targets. [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.
4 The `datapar_abi::compatible` tag is defined by the implementation to alias the tag type with the most efficient data parallel execution that ensures the highest compatibility on the target architecture.
5 The `datapar_abi::native` tag is defined by the implementation to alias the tag type with the most efficient data parallel execution that is supported on the target system.

---

2 The ABI types are tag types to be used as the second template argument to `datapar` and `mask`.
3 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).
4 An implementation may choose to implement data-parallel execution for many different targets. [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.
5 The `datapar_abi::compatible` tag is defined by the implementation to alias the tag type with the most efficient data parallel execution that ensures the highest compatibility on the target architecture.
6 The `datapar_abi::native` tag is defined by the implementation to alias the tag type with the most efficient data parallel execution that is supported on the target system.
2.1.2 Class template `datapar`

2.1.2.2 Class template `datapar` overview

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

                template <class U = T> static constexpr size_t memory_alignment = implementation_defined;

                static constexpr size_type size() = default;

                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> datapar(datapar<U, Abi>);

                // loads:
                static datapar load(const value_type *);
                template <class Flags> static datapar load(const value_type *, Flags);
                template <class U, class Flags = unaligned_tag> static datapar load(const U *, Flags = Flags());

                // stores:
                void store(value_type *);
                template <class Flags> void store(value_type *, Flags);
                template <class U, class Flags = unaligned_tag> void store(U *, Flags = Flags());

                // masked stores:
                void store(value_type *, mask_type);
                template <class Flags> void store(value_type *, mask_type, Flags);
                template <class U, class Flags = unaligned_tag> void store(U *, mask_type, Flags = Flags());

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

                // increment and decrement:
                datapar & operator++();
                datapar operator++(int);
                datapar & operator--();
            }
        }
    }
}
```
The class template `datapar<T, Abi>` is a one-dimensional smart array. In contrast to `valarray` (26.6), the number of elements in the array is determined at compile time, according to the `Abi` template parameter.

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

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

typedef implementation_defined native_handle_type;

The `native_handle_type` member type is an alias for the `native_handle()` member function return type. It is used to expose an implementation-defined handle for implementation- and target-specific extensions.

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

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

### 2.1.2.3 `datapar` constructors

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

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

**`datapar(value_type);`**

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

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

```
Remarks: This constructor shall not participate in overload resolution unless `U` and `T` are different integral types and `make_signed<U>::type equals make_signed<T>::type`. 
Effects: Constructs an object of type `datapar`. 
Postcondition: The `i`-th element equals `static_cast<T>(x[i])` for all elements.
```
2.1.2.4 datapar load functions

```
static datapar load(const value_type *x);
```

1 Effects: Constructs an object with each element \(i\) initialized to \(x[i]\) for all elements.
2 Remarks: If `datapar::size()` is greater than the number of values pointed to by the argument, the behavior is undefined.

```
template <class Flags> static datapar load(const value_type *x, Flags);
```

4 Effects: Constructs an object with each element \(i\) initialized to \(x[i]\).
5 Remarks: If `datapar::size()` is greater than the number of values pointed to by the first argument, the behavior is undefined.
6 Remarks: If the template parameter is of type `aligned_tag` and the pointer value is not a multiple of `memory_alignment<T>`, the behavior is undefined.

```
template <class U, class Flags = unaligned_tag> static datapar load(const U *x, Flags);
```

8 Effects: Constructs an object with each element \(i\) initialized to `static_cast<T>(x[i])`.
9 Remarks: If `datapar::size()` is greater than the number of values pointed to by the first argument, the behavior is undefined.
10 Remarks: If the second template parameter is of type `aligned_tag` and the pointer value is not a multiple of `memory_alignment<U>`, the behavior is undefined.

2.1.2.5 datapar store functions

```
void store(value_type *x);
```

1 Effects: Copies each element such that the \(i\)-th element is stored to \(x[i]\).
2 Remarks: If `datapar::size()` is greater than the number of values pointed to by the first argument, the behavior is undefined.

```
template <class Flags> void store(value_type *x, Flags);
```

3 Effects: Copies each element such that the \(i\)-th element is stored to \(x[i]\).
4 Remarks: If `datapar::size()` is greater than the number of values pointed to by the first argument, the behavior is undefined.
5 Remarks: If the template parameter is of type `aligned_tag` and the pointer value is not a multiple of `memory_alignment<T>`, the behavior is undefined.

```
template <class U, class Flags = unaligned_tag> void store(U *x, Flags = Flags());
```

2.1.2.6 \texttt{datapar} subscript operators

\texttt{reference} \texttt{operator}[](size\_type \texttt{i});

\textbf{Returns:} An lvalue reference to the $i$-th element.

\textbf{Postconditions:} Assignment of objects of type \texttt{T} modify the $i$-th element without aliasing violations.

Modification of \texttt{*this} does not invalidate references held to the return value. Subsequent reads from such references yield the new value of the $i$-th element.

\texttt{const\_reference} \texttt{operator}[](size\_type) \texttt{const};

\textbf{Returns:} A \texttt{const} lvalue reference to the $i$-th element.

\textbf{Postconditions:} Modification of \texttt{*this} does not invalidate references held to the return value. Subsequent reads from such references yield the new value of the $i$-th element.
2.1.2.7 `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 elements.

`datapar operator~() const;`

14. **Requires:** The first template argument T to `datapar` must be an integral type.
15. **Effects:** Constructs an object where each bit of *this is inverted.
16. **Returns:** The new object.
17. **Remarks:** `datapar::operator~()` shall not participate in overload resolution if T is a floating-point type.

`datapar operator+() const;`

18. **Returns:** A copy of *this

`datapar operator-() const;`

19. **Effects:** Constructs an object where the \( i \)-th element is initialized to \(-operator[](i)\) for all elements.
20. **Returns:** The new object.
2.1.2.8 datapar native handles

native_handle_type &native_handle();

1

Returns: An lvalue reference to the implementation-specific object implementing the data-parallel semantics.

const native_handle_type &native_handle() const;

2

Returns: A const lvalue reference to the implementation-specific object implementing the data-parallel semantics.

2.1.3 datapar non-member operations

2.1.3.9 datapar binary operators

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 &);
Remarks: The return type of these operators shall be deduced according to the following rules:

- If `is_datapar_v<U> == true` then the return type shall be determined from `T` and `U::value_type` according to the following paragraph.
- 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 or `U` is convertible to `int` then the return type shall be determined from `T` and `U` according to the following paragraph.
- Otherwise, if `U` is implicitly convertible to `datapar<V, Abi>`, where `V` is determined according to standard template type deduction, then the return type shall be determined from `T` and `V` according to the following paragraph.
- Otherwise, if `U` is implicitly convertible to `datapar<T, Abi>`, the return type shall be `datapar<T, Abi>`.
- Otherwise no return type is defined (SFINAE).

Remarks: Given the types `T` and `Abi` from the class template argument list and a third type `U` determined by the rules of the previous paragraph a return type is deduced according to the following rules:

- If `U` is not a fundamental arithmetic type then the return type shall be `datapar<T, Abi>`.
- Otherwise, if at least one of the types `T` and `U` is a floating-point type the return type shall be `datapar<decltype(T() + U()), Abi>`.
- Otherwise, if `sizeof(T) < sizeof(U)` the return type shall be `datapar<U, Abi>`.
- Otherwise, if `sizeof(T) > sizeof(U)` the return type shall be `datapar<T, Abi>`.
- Otherwise, the type `T` or `U` that is farthest back in the list of standard integer types (cf. [basic.fundamental]) is used as type `V` and the return type shall be `datapar<V, Abi>` if both types `T` and `U` are signed, otherwise the return type shall be `datapar<make_unsigned_t<V>, Abi>`.

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 `R::value_type`, with `R` the return type.
- `datapar<T, Abi>` is implicitly convertible to the return type.
- `U` is implicitly convertible to the return type.

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

Effects: Both arguments are first converted to the return type. Each of these operators subsequently performs the indicated operation component-wise on each of the elements of the first argument and the corresponding element of the second argument.

Returns: An object containing the results of the component-wise operator application.
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>`.

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>`.

Returns: A reference to the first argument.

2.1.3.11 `datapar` logical operators

2.1.3.12 `datapar` compare operators
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].

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>`.

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

Effects: Both arguments are first converted to `datapar_return_type<datapar<T, Abi>, U>` if each of the elements of the first argument and the corresponding element of the second argument.

Returns: An object containing the results of the component-wise operator application.

2.1.3.13 `datapar` casts

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

Remarks: The `datapar_cast` function only participates in overload resolution if all of the following hold:
- `is_datapar_v<T>`
- `is_datapar_v<U>`
- All types in the template parameter pack `Us` are equal to `U`.
- `U::size() + Us::size()...` is an integral multiple of `T::size()`.

Effects: All scalar elements `x_i` of the function argument(s) are converted as if `y_i = static_cast<typename T::value_type>(x_i)` is executed. The resulting `y_i` are stored into 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]

Returns: The converted values as one object of `T` or an array of `T`.

2.1.3.14 `datapar` transcendentals

[datapar.transcend]

TODO

2.1.4 Class template `mask`

[mask]

2.1.4.15 Class template `mask` overview

[mask.overview]
typedef implementation_defined reference;
typedef implementation_defined const_reference;
typedef datapar<T, Abi> datapar_type;
typedef size_t size_type;
typedef Abi abi_type;

template <class U = T> static constexpr size_t memory_alignment = implementation_defined;

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> mask(mask<U, Abi>);

// loads:
static mask load(const value_type *);
template <class Flags> static mask load(const value_type *, Flags);

// stores:
void store(value_type *);
template <class Flags> void store(value_type *, Flags);

// masked stores:
void store(value_type *, mask);
template <class Flags> void store(value_type *, mask, Flags);

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

// negation:
mask operator!() const;

// access to the internals for implementation-specific extensions
native_handle_type &native_handle();
const native_handle_type &native_handle() const;

};

};

template <class T, class Abi>
template <class U>
constexpr size_t mask<T, Abi>::memory_alignment<U>;

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.
typedef implementation_defined native_handle_type;

The native_handle_type member type is an alias for the native_handle() member function return type. It is used to expose an implementation-defined handle for implementation- and target-specific extensions.

static constexpr size_type size();

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

2.1.4.16 mask constructors

mask() = default;

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

mask(value_type);

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

template <class U> mask(mask<U, Abi> x);

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

Effects: Constructs an object of type mask.

Postcondition: The i-th element equals x[i] for all elements.

2.1.4.17 mask load functions

static mask load(const value_type *x);

Effects: Constructs an object with each element i initialized to x[i] for all elements.

Returns: The constructed object.

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

template <class Flags> static mask load(const value_type *x, Flags);

Effects: Constructs an object with each element i initialized to x[i] for all elements.

Returns: The constructed object.

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

Remarks: If the template parameter is of type aligned_tag and the pointer value is not a multiple of memory_alignment<T>, the behavior is undefined.
2.1.4.18 **mask store functions**

```cpp
void store(value_type *x);
```

**Effects:** Copies each element such that the \( i \)-th element is stored to \( x[i] \).

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

```cpp
template <class Flags> void store(value_type *x, Flags);
```

**Effects:** Copies each element such that the \( i \)-th element is stored to \( x[i] \).

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

**Remarks:** If the template parameter is of type `aligned_tag` and the pointer value is not a multiple of memory_alignment\(<T>\), the behavior is undefined.

```cpp
void store(value_type *x, mask);
```

**Effects:** Copies each element where the corresponding element in the second argument is `true` such that the \( i \)-th element is stored to \( x[i] \).

**Remarks:** If the largest \( i \) where the second argument is `true` is greater than the number of values pointed to by the first argument, the behavior is undefined.

```cpp
template <class Flags> void store(value_type *x, mask, Flags);
```

**Effects:** Copies each element where the corresponding element in the second argument is `true` such that the \( i \)-th element is stored to \( x[i] \).

**Remarks:** If the largest \( i \) where the second argument is `true` is greater than the number of values pointed to by the first argument, the behavior is undefined.

**Remarks:** If the template parameter is of type `aligned_tag` and the pointer value is not a multiple of memory_alignment\(<T>\), the behavior is undefined.

2.1.4.19 **mask subscript operators**

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

**Returns:** A temporary object acting as a smart reference wrapper to the \( i \)-th element.

**Postconditions:** Assignment of objects of type `bool` modify the \( i \)-th element without aliasing violations. Modification of `*this` does not invalidate references held to the return value. Subsequent reads from such references yield the new value of the \( i \)-th element.

```cpp
const_reference operator[](size_type) const;
```

**Returns:** A `const` value reference to the \( i \)-th element.

**Postconditions:** Modification of `*this` does not invalidate references held to the return value. Subsequent reads from such references yield the new value of the \( i \)-th element.
2.1.4.20 mask unary operators

\begin{verbatim}
mask operator!() const;
\end{verbatim}

1 \textit{Returns:} A mask object with the \( i \)-th element set to the logical negation for all elements.

2.1.4.21 mask native handles

\begin{verbatim}
native_handle_type &native_handle();
\end{verbatim}

1 \textit{Returns:} An lvalue reference to the implementation-specific object implementing the data-parallel semantics.

\begin{verbatim}
const native_handle_type &native_handle() const;
\end{verbatim}

2 \textit{Returns:} A \texttt{const} lvalue reference to the implementation-specific object implementing the data-parallel semantics.

2.1.5 mask non-member operations

2.1.5.22 mask binary operators

\begin{verbatim}
template<
class T, class Abi, class U>
bool operator==(mask<T, Abi>, const U &);

template<
class T, class Abi, class U>
bool operator!=(mask<T, Abi>, const U &);

template<
class T, class Abi, class U>
bool operator==(const U &, mask<T, Abi>);

template<
class T, class Abi, class U>
bool operator!=(const U &, mask<T, Abi>);
\end{verbatim}

1 \textit{Remarks:} Each of these operators only participates in overload resolution if \( U \) is implicitly convertible to mask\langle T, Abi\rangle.

2 \textit{Remarks:} The operators with \texttt{const} \( U \) as first parameter shall not participate in overload resolution if \texttt{is\_mask\_v\langle U\rangle == true}.

3 \textit{Returns:} The equality operator returns \texttt{true} if all boolean elements of the first argument equal the corresponding element of the second argument. It returns \texttt{false} otherwise.

4 \textit{Returns:} The inequality operator returns the negation of the equality operator.

2.1.5.24 mask reductions

\begin{verbatim}
template<
class T, class Abi>
bool all_of(mask<T, Abi>);

constexpr bool all_of(bool);
\end{verbatim}

1 \textit{Returns:} \texttt{true} if all boolean elements in the function argument equal \texttt{true}, \texttt{false} otherwise.
template <class T, class Abi> bool any_of(mask<T, Abi>);  
constexpr bool any_of(bool);  

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);  

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);  

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.

Note: some_of(bool) unconditionally returns false.

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

Returns: The number of boolean elements that are true.

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

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

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

constexpr int find_first_set(bool);  

Returns: 0 if the argument is true.

2.1.5.25 Masked assignment  

template <class T, class U, class Abi> implementation_defined where(mask<U, Abi> m, datapar<T, Abi> &v);  

Remarks: The function only participates in overload resolution if mask<U, Abi> is implicitly convertible to mask<T, Abi>.

Returns: A temporary object with the following properties:
1. The object is not CopyConstructible.
2. Assignment and compound assignment operators only participate in overload resolution if the corresponding operator for datapar<T, Abi> is usable.
3. Effects: Assignment and compound assignment implement the same semantics as the corresponding operator for datapar<T, Abi> with the exception that elements of v stay unmodified if the corresponding boolean element in m is false.
4. The assignment and compound assignment operators return void.

template <class T> implementation_defined where(bool, T &);
Remarks: The function only participates in overload resolution if \( T \) is a fundamental arithmetic type.

Returns: A temporary object with the following properties:

1. The object is not \textit{CopyConstructible}.
2. Assignment and compound assignment operators only participate in overload resolution if the corresponding operator for \( T \) is usable.
3. Effects: If the first argument is \textit{false}, the assignment operators do nothing. If the first argument is \textit{true}, the assignment operators forward to the corresponding builtin assignment operator.
4. The assignment and compound assignment operators return \textit{void}.

---

3.1 member types

The member types may not seem obvious. Rationales:

\textbf{value\_type}

In the spirit of the \textit{value\_type} member of STL containers, this type denotes the \textit{logical} type of the values in the vector.

\textbf{register\_value\_type}

On some targets it may be beneficial to implement \textit{datapar} instantiations of some \( T \) with a different type \textit{register\_value\_type}, which has higher precision than \( T \). This is mostly an implementation detail, but can be important to know in some situations, especially whenever \textit{native\_handle\_type} is involved.

\textit{Requesting Guidance: A better name might be} \textit{native\_value\_type}.

\textbf{native\_handle\_type}

The type used for enabling access to an implementation-defined member object (via the \texttt{native\_handle()} function).

\textbf{reference}

Used as the return type of the non-const scalar subscript operator. This may use implementation-defined means to solve possible type aliasing issues.

\textbf{const\_reference}

Used as the return type of the const scalar subscript operator. From my experience with Vc, it is safest to actually not use a const lvalue reference here, but a temporary.
3 Discussion

3.1 mask_type

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

3.2 size_type

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

3.3 abi_type

The Abi template parameter to `datapar`.

3.2 Conversions

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

3.3 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.

3.4 Aliasing of Subscript Operators

Note that the way the subscript operators are declared, some kind of type punning needs to happen. An lvalue reference to `register_value_type` needs to reference the same object as is contained as one element of the `datapar` object. An alternative to an lvalue reference would be a smart reference object. This would require progress on language improvements for smart references first.

The subscript operator of the `mask` type, on the other hand, may not use lvalue references and must use a smart reference wrapper instead. This is necessary because there are systems where a single boolean element is stored as a single bit. To ensure source compatibility the return type must therefore be a smart reference on all implementations.

---

1 Note: The vector builtins of clang do not suffice to implement the subscript operators, even though they support subscripting the vector object. An implementation might have to use a mechanism such as the `gnu::may_alias` attribute.
3.5 parameters of binary and compare operators

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

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

The motivation for using a variant where at least one function parameter is constrained to the `datapar` class template is compilation speed. The compiler can drop the operator from the overload resolution set quicker. With concepts it might be worthwhile to revisit this decision.

3.6 compound assignment

The semantics of compound assignment would allow less strict implicit conversion rules. Consider `datapar<int>() *= double()`: the corresponding multiplication operator would not compile because the implicit conversion to `datapar<float>` 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 suddenly would not compile anymore.

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.

3.7 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`).
3 Discussion

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

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

3.8 FUNDAMENTAL SIMD TYPE OR NOT?

3.8.1 THE ISSUE

There has been renewed discussion on the reflectors over the question whether C++ should define a fundamental, native SIMD type (let us call it `fundamental<T>`) and a generic data-parallel type on top 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.

3.8.2 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]

Listing 1: Possible declaration of the class template parameters of a `datapar` class with arbitrary width.
The author has implemented a library where a clear distinction is made between fundamental\(<T, \text{Abi} >\) 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).

3.8.3 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:

\[
\text{arbitrary\(\<\text{float} >\)(\() * \text{arbitrary\(\<\text{int} >\)(\()); // compiles for some targets, fails for others}
\]
\[
\text{fundamental\(\<\text{float} >\)(\() * \text{fundamental\(\<\text{int} >\)(\()); // never compiles, requires explicit cast}
\]

The datapar\(<T>\) operators 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:

\[
\text{using floatvec = datapar\(<\text{float} >\);}
\text{using intvec = arbitrary\(<\text{int} >, \text{floatvec}\::\text{size()}\> ;}
\text{using doublevec = arbitrary\(<\text{int} >, \text{floatvec}\::\text{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:

\[
\text{using floatvec = arbitrary\(<\text{float} >\);}
\text{using intvec = arbitrary\(<\text{int} >, \text{floatvec}\::\text{size()}\> ;}
\text{using doublevec = arbitrary\(<\text{int} >, \text{floatvec}\::\text{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\(<\text{float} >() * \text{arbitrary\(\<\text{int} >()\)) 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.
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).

BIBLIOGRAPHY


