Document Number: N4185
Date: 2014-10-10
Project: Programming Language C++, Library Working Group
Reply-to: Matthias Kretz <kretz@compeng.uni-frankfurt.de>

SIMD Types: The Mask Type & Write-Masking

ABSTRACT

This paper describes a template class for portable SIMD Mask types. Most importantly it shows how conditional code can be expressed with SIMD types. Different variants of a syntax for write-masking will be discussed.

CONTENTS

1 About This Document

This document is derived from a larger document about the Vc library. For the sake of simplicity I refrained from changing the naming conventions of types/functions in this paper:

- I want to focus on functionality first. We can "correct" the names later.
- It is easier to find the reference to an existing implementation.

Disclaimer: I did not get everything "right" in the Vc implementation yet. Some details of the interface definitions I present here do not fully reflect the current state of the Vc SIMD types.

1.1 Shorthands in the document

- $\mathcal{W}_{\texttt{T}}$: number of scalar values (width) in a SIMD vector of type `T` (sometimes also called the number of SIMD lanes)

1.2 Relation to N4184

This document builds upon the `Vector<T>` type described in N4184. Many design decisions are not discussed in this document because they have been covered in N4184 before and can be applied analogously to `Mask<T>`.

2 General Introduction to Conditionals

Conditional statements are some of the most important language elements in C++. `if` statements enable programs to follow different code paths depending on arbitrary boolean conditions. In most cases an `if` statement is translated as a branching instruction. These instructions can be expensive on modern processors, if the branch prediction unit chooses the wrong branch. In such a case the pipeline has to be flushed and execution must restart at the other branch. This can incur penalties on the order of 100 cycles.

In order to overcome costly pipeline flushes on incorrect branch prediction, conditional move instructions have been introduced. A conditional move instruction typically executes a load or register copy iff one or more specific flag(s) is/are set. Thus, an optimizing compiler might translate the code `if (condition) { x = y; }` into a compare instruction and a subsequent conditional move instruction.

Not every conditional jump results from `if` statements. Conditional jumps are used for loop exit conditions in `while` or `for` statements. Furthermore, `switch` statements describe jumps into one code section out of several ones, where each one can be identified via one or more integral value(s). Instead of a `switch` statement, the logic can alternatively be expressed as several `if` statements. This is functionally equivalent, but often compilers optimize `switch` statements via jump tables, while `if` cascades typically are translated as consecutive compares and jumps.

3 CONDITIONALS IN SIMD CONTEXT

The SIMD types, as defined in N4184 do not return booleans from the compare operators. Instead they return `Vector<T>::MaskType`, which is an alias for `Mask<T>`. This mask type is the equivalent of a `Vector<bool>` type, but with additional type information about the associated `Vector<T>::EntryType`. (The need for this additional type information will be discussed in Section 4.) Thus, operations that return a definitive `true` or `false` answer with scalar types, return multiple `true` and/or `false` values in one return value with SIMD types. Obviously, these mask types cannot work directly with the builtin conditional statements in C++.

For SIMD code we have two principal choices for the semantics of `if`, `for`, `while`, and `switch`.

1. By enhancing the language it is possible to overload the meaning of conditional statements with operands of mask type. This has been implemented in Cilk Plus for the array notation extension [1]. Conditional statements subsequently do not disable a branch unless all entries of the mask are `false` (though essentially this is an optional optimization). Instead, all code branches are executed, but with some vector lanes implicitly disabled. Consider the example code in Listing 1 on a system with $\mathcal{W}_{\text{int}} = 4$ and `a = {1, 2, 3, 4}, b = {7, 0, 7, 7}`: The expression `a < b` then returns a mask with 4 boolean values: `{true, false, true, true}`. The compiler therefore has to translate the `if`-branch (line 3) into instructions that modify `a` only at the indexes 0, 2, and 3. Subsequently, `a` will be `a = {2, 2, 4, 5}`. The `else`-branch (line 5) then may only modify the SIMD vector entry at index 1. Thus `a` must become `a = {2, 1, 4, 5}`, which is the return value of the function `f`.

```
int_v f(int_v a, int_v b) {
  if (a < b) {
    a += 1;
  } else {
    a -= 1;
  }
  return a;
}
```

Listing 1: Example code relying on overloaded semantics for `if` statements with mask arguments.

2. The alternative keeps the semantics of the existing conditional statements unchanged. Then, mask types can only be used for conditional statements if a reduction function from a mask to a single boolean value is used (see Section 4.7). Still, the functionality described above (modifying a subset of a SIMD vector, selected via a mask) can be implemented via write-masking expressions (see Section 5).

3.1 Consequences of Implicit Masking

Consider the implications of `if` statements that accept SIMD masks. The code example in Listing 2 is a small modification of the example in Listing 1 that would be equivalent for scalar types. But with SIMD vector types both of the two `return` statements in the code must be taken. It is certainly possible to define that this code blends the SIMD vectors from the two `return` statements according to the implicit masks in the `if` and `else` branches. But already a seemingly small change, such as returning an `int` instead of `int_v` (Listing 3) leads to unresolvable ambiguity: Should the function return `+1` or `-1`? Similar ambiguity issues occur with non-complementary masked `return` statements and function calls inside the branches. Throwing exceptions and locking/unlocking mutexes would even have to be disallowed altogether.

```
int_v f(int_v a, int_v b) {
  if (a < b) {
    return a + 1;
  } else {
    return a - 1;
  }
}
```

Listing 2: Code example that shows unclear return semantics: both branches must execute but from where does the function return and what is the return value?

```
int f(int_v a, int_v b) {
  if (a < b) {
    return +1;
  } else {
    return -1;
  }
}
```

Listing 3: Code example that shows unresolvable ambiguity: both branches must execute but there can be only one return value because the return type is a scalar `int`.

There is a more fundamental uncertainty resulting from implicit masking via `if` statements on SIMD vector masks: How should different SIMD vector types interact? An `if` statement from `int_v` comparison returns $\mathcal{W}_{\texttt{int}}$ boolean answers. If the branch contains code with `short_v` or `double_v`, should it be implicitly write-masked or not? If yes, how? There is no natural and obvious behavior for applying write masks of different $\mathcal{W}_{\mathrm{T}}$.

This shows that `if` statements with non-boolean arguments limit the language features that are allowed in the `if`/`else` branches. This makes the feature much less intuitive. The implicit mask context changes the semantics significantly in different regions of the source code. And the problem is aggravated if a developer requires `else if` or `switch` statements.

3.2 Design Decision for Vc

For the Vc library I therefore decided that the semantics of `if`, `for`, `while`, and `switch` must not change for explicit SIMD programming.[1] Everything else would be too surprising and unintuitive to users, especially developers that read existing code without prior knowledge about SIMD programming. This may sound obvious, but consider that many developers will start from a scalar implementation of their algorithm. In the scalar code the conditional statements correctly express the logic of the algorithm. When a developer subsequently vectorizes the code (s)he starts with replacing scalar types with the Vc vector types. At this point it may appear like a logical simplification of the vectorization process to keep the conditional statements unchanged in order to minimize the effort for the user. But, as discussed above, this comes at a considerable cost in consistency of semantics.[2] Thus,

1 This is nice, because otherwise a pure library solution would not be possible.

2 There is not really a precedent in C++ for such differences in semantics / allowed operations for certain code regions. The transactional memory extensions for C++ [N3999] may introduce local semantics where actions inside a transaction are re-

part of the issue is the question whether it is more important to ease initial vectorization of an algorithm or whether maintenance effort is more important. Even then, whether implicit write-masking via conditional statements eases initial vectorization at all certainly depends on the algorithm: The restricted semantics might lead to an even larger effort required for converting a given scalar code to SIMD code.

4 THE Vc::Mask<T> CLASS

Analogous to the `Vector<T>` class discussed in N4184, there needs to be a type that acts as a SIMD vector of booleans. This is necessary for attaching the SIMD context only to types and never to some implicit context. There are three main approaches:

- Reuse/Specialize the `Vector<T>` class (`Vector<bool>`).
- Define a new class (`Mask<T>`) with a type as template parameter.
- Define a new class (`Mask<Size>`) with a size as template parameter.

4.1 WHY Vc::Vector<bool> IS NOT ENOUGH

The type `bool` is part of the *integral types* in C++. Since values of type `bool` "participate in integral promotions" [2, §3.9.1] they can be used in any expression where an `int` can be used.[3] Therefore, it appears as if the interface provided by `Vector<T>` is a good fit for boolean values, too. The additional functionality a SIMD vector of booleans should provide (such as the population count or reductions) could still be defined as non-member functions.

But, considering that $\mathcal{W}_{\texttt{T}}$ may be different for different `T` it follows that $\mathcal{W}_{\texttt{bool}} = \max(\{\mathcal{W}_{\texttt{T}} \mid \forall \texttt{T}\})$. Otherwise `Vector<bool>` would only be usable for a (target-dependent) subset of `Vector<T>` types. This definition of `Vector<bool>` implies that $\mathcal{W}_{\texttt{bool}}$ may be greater than $\mathcal{W}_{\texttt{T}}$ for some types `T`. Consider an SSE target, where $\mathcal{W}_{\texttt{short}} = 8$, $\mathcal{W}_{\texttt{float}} = 4$, and $\mathcal{W}_{\texttt{double}} = 2$. Thus, $\mathcal{W}_{\texttt{bool}}$ would need to be 8 (16 if `Vc::Vector<signed char>` were supported by Vc) and store 50% or 75% unused data for masks interacting with `float_v`

stricted to reversible operations. Approaches like explicit SIMD loops or the Intel Array Building Blocks framework also rely on restricted local semantics.

3 "A prvalue of type bool can be converted to a prvalue of type int, with false becoming zero and true becoming one." [2, §4.5]

and `double_v`, respectively. Considering the implementation implications, this issue turns out to have serious efficiency implications, as well: With the SSE instruction set boolean vectors are stored in the 128-bit SSE registers with 64/32/16/8 bits all set to either 0 or 1 for every associated value in the value vector. Thus, the hardware generates and expects booleans in different bit representations, depending on the SIMD vector type (or more accurately: `sizeof(T)`).

In addition to the size issue, there is good reason to use a single `bool` return value for the equal and not-equal comparison operators (see Section 4.6). Thus, `Vector<bool>` would need to specialize these functions, which is certainly possible, but, to a certain degree, defeats the purpose of using a generic class.

4.2 Vc::Mask<T> definition

As discussed in Section 4.1, it is beneficial to define several mask types instead of a single boolean vector type. By looking at the SSE instruction set, we have seen that `Mask<Size>` would suffice to define the minimal set of mask types for this target. But, consider that the AVX instruction set uses $\mathcal{W}_{\texttt{float}} = 8$ and $\mathcal{W}_{\texttt{double}} = 4$ on top of the SSE vector sizes. Using the SIMD vector size as template parameter for the mask type thus would lead to subtle portability issues (this is the same issue I discussed in N4184 for `Vector<T, Size>`): Consider the return types of the expressions `int_v() == int_v()` and `float_v() == float_v()`. With the SSE target they would both return the same type `Mask<4>`, whereas with AVX the types would differ: `Mask<4>` and `Mask<8>` respectively. The general solution (`Mask<T>`) therefore uses a different mask type for every SIMD vector type `Vector<T>`.[4] That way the types are different for every target and the user will be forced to use explicit type conversions.

Listing 4 shows the definition of the SIMD mask type. Except for the `EntryType` member type all member types in Listing 4 are *implementation-defined*. This is analogous to the definition of the `Vector<T>` class in N4184. The different types are used for abstracting the following concepts:

`VectorType` (line 8) This is the type that the implementation uses to store a SIMD vector of booleans. For some implementations this type may be equal to `Vector<T>::VectorType` but there is no such requirement.

4 Implicit and explicit conversions between `Mask<T>` and `Mask<U>` can be a no-op whenever `sizeof(T) = sizeof(U)` $\wedge\ \mathcal{W}_{\texttt{T}} = \mathcal{W}_{\texttt{U}}$.

```
namespace Vc {
namespace target_dependent {
template <typename T> class Mask
{
  implementation_defined data;

public:
  typedef implementation_defined VectorType;
  typedef bool EntryType;
  typedef implementation_defined EntryReference;

  static constexpr size_t MemoryAlignment = implementation_defined;
  static constexpr size_t Size = implementation_defined;
  static constexpr size_t size() { return Size; }
  // ... (see below)
};
template <typename T> constexpr size_t Mask<T>::MemoryAlignment;
template <typename T> constexpr size_t Mask<T>::Size;

typedef Mask<             float>     float_m;
typedef Mask<            double>    double_m;
typedef Mask<  signed long long>  longlong_m;
typedef Mask<unsigned long long> ulonglong_m;
typedef Mask<  signed      long>      long_m;
typedef Mask<unsigned      long>     ulong_m;
typedef Mask<  signed       int>       int_m;
typedef Mask<unsigned       int>      uint_m;
typedef Mask<  signed     short>     short_m;
typedef Mask<unsigned     short>    ushort_m;
typedef Mask<  signed      char>     schar_m;
typedef Mask<unsigned      char>     uchar_m;
}  // namespace target_dependent
}  // namespace Vc
```

Listing 4: SIMD mask class definition

`EntryType` (line 9) This is an alias for `bool`. The member type is defined for generality/interface compatibility with `Vector<T>`. This type signifies the conceptual entry type. The actual entries in `VectorType` may use a different binary representation than `bool`.

`EntryReference` (line 10) This type is used as the return type of the non-const subscript operator. It is therefore used to reference a single boolean entry in the internal mask representation. Note that the most compact binary representation for a SIMD vector of booleans uses a single bit per boolean value. In this case there cannot be a type that represents the actual bits of the boolean value of a single mask entry.[5] Thus, `EntryReference` can also be a wrapper type that can access (read and write) individual bits of such a mask via the assignment operators and cast-to-`bool` operator.

The `Mask<T>` type needs a single data member of an *implementation-defined* type (line 5). This member defines the size and alignment of the `Mask<T>` type.

The number of entries in the SIMD vector, in general, is different from `sizeof(Mask<T>)`, which is why the `Size` constant (line 13) defines this value. For compatibility with STL containers, `Mask<T>` contains the `size()` member function, which also returns the number of scalar entries in the SIMD vector.

The `Mask<T>` type also defines a `MemoryAlignment` static data member, just as `Vector<T>` does. Analogously, its value is the alignment requirement of pointers to `EntryType` (i. e.`bool`) in aligned load and store calls (Section 4.4). Implementation experience tells that in most cases the alignment of `Mask<T>` will not be equal to `Mask<T>::MemoryAlignment`. This is due the SIMD mask register either using several Bytes or only a single bit per boolean entry.

Finally, analogous to the type aliases for `Vector<T>`, the mask types that the Vc library implements are aliased to the type names `float_m`, `double_m`, … (lines 20–31).

```
1  Mask();
2  explicit Mask(bool);
3  template <typename U> Mask(const Mask<U> &);
4  template <typename U> explicit Mask(const Mask<U> &);
```

Listing 5: Declaration of the `Mask<T>` copy and cast constructors.

4.3 CONSTRUCTORS

The constructors for the `Mask<T>` class need to replicate the semantics of the `bool` type as much as possible. The necessary declarations are shown in Listing 5.

The default constructor of `Mask<T>` initializes the value of all entries in the mask to `false`. This is required for compatibility with the expression `bool()`, which constructs a `bool` with the value `false`.

The copy and move constructors and operators are omitted for the same reason as for `Vector<T>` [N4184].

The constructor on line 2 initializes a mask object with all values set to the boolean value passed in the argument. Therefore, this constructor implements a broadcast from one scalar value to all entries in a SIMD vector. Note that, in contrast to the broadcast constructor from `Vector<T>`, the broadcast constructor of `Mask<T>` is declared as `explicit`. This is a deviation from the behavior of the scalar `bool` type. But, for boolean vectors the usefulness of a broadcast is mostly limited to initialization of mask objects. If a developer really needs to use a mask with all entries set to either `true` or `false`, then it is very likely that a scalar control-flow statement (such as `if`) is much better suited for the task. On the other hand, if implicit conversions from scalar `bool` to `Mask<T>` were possible, a user might fail to notice that an expression produces a `bool` instead of the intended mask object.

Finally, the two constructor functions in lines 3 and 4 implement implicit and explicit (`static_cast`) conversions between mask objects. The two functions, as declared in Listing 5, are ambiguous. They need to be adjusted, such that the implicit constructor only participates in overload resolution iff $\mathcal{W}_U = \mathcal{W}_T$ for all possible targets. According to the discussion of implicit conversions in N4184 this can be decided via the following `enable_if` expression:

```
enable_if<differs_only_in_signedness<U, T>::value>
```

The `explicit` constructor then simply requires the inverse condition to `enable_if`.

5 The object representation of any type in C++ takes up N bytes, where N is integral. This is also evident from the `sizeof` operator which returns a `size_t` denoting the number of bytes in the object representation of the type.

4.4 Loads and Stores

Mask types can implement load and store functions, reading from / writing to arrays of `EntryType` (which is `bool`). These functions can be useful to write code that is independent of the SIMD register width and to interface with non-SIMD code (or I/O in general). Listing 6

```
1  explicit Mask(const bool *mem);
2  template<typename Flags> explicit Mask(const bool *mem, Flags f);
3
4  void load(const bool *mem);
5  template<typename Flags> void load(const bool *mem, Flags);
6
7  void store(bool *) const;
8  template<typename Flags> void store(bool *mem, Flags) const;
```

Listing 6: Declaration of the `Mask<T>` load and store functions.

shows the declaration of the necessary functions. The `Flags` argument is analogous to the one for the `Vector<T>` load/store functions. The default uses unaligned loads and stores and can be set to aligned loads and store via the second argument.

4.5 Logical and Bitwise Operators

```
1  Mask operator!() const;
2
3  Mask &operator&=(Mask);
4  Mask &operator|=(Mask);
5  Mask &operator^=(Mask);
6
7  Mask operator&(Mask) const;
8  Mask operator|(Mask) const;
9  Mask operator^(Mask) const;
10
11 Mask operator&&(Mask) const;
12 Mask operator||(Mask) const;
```

Listing 7: Declaration of logical and bitwise operators for `Mask<T>`.

Listing 7 shows the declaration of the operators for logical and bitwise operations. Each operator simply applies the operation component-wise. There is no need for non-member overloads as was required for `Vector<T>`, because the conversion rules are much simpler for different vectors of booleans. The implicit and explicit conversion constructors fully suffice.

4.6 comparison operators

Listing 8 shows the declaration of comparison operators that I implemented for `Vc::Mask`. Note, that the return type is a scalar `bool` and not a SIMD type. Returning another mask type would make the compare operator basically an alias for the xor operator. Typically, it is more interesting to determine whether two given mask are equal (or not) and this requires a single boolean.

```
1  bool operator==(Mask rhs) const;
2  bool operator!=(Mask rhs) const;
```

Listing 8: Declaration of the `Mask<T>` comparison operators.

It is certainly possible to define a meaning for relational compare operators (less/greater). The most obvious definition would be an interpretation of the boolean entries as bits of an integer and then compare the integers. Up to now I did not come across a use case for such operators, though. I am looking for input from the community on this question.

4.7 reduction functions

In order to use a mask object in an `if` statement or loop condition there needs to be a reduction function from the multiple boolean values in the mask to a single `bool`. There are four useful reduction functions:

`all_of`: Returns `true` iff all entries in the mask are `true`.

`any_of`: Returns `true` iff at least one entry in the mask is `true`.

`none_of`: Returns `true` iff all entries in the mask are `false`.

`some_of`: Returns `true` iff there is at least one entry that is `true` and at least one entry that is `false` (note that this is always `false` for `Vc::Scalar::Mask<T>`).

The usefulness of the first three functions should be obvious. The `some_of` reduction, on the other hand, is not used that often. It is a useful check for knowing whether some conditions in the SIMD lanes diverged, though. For example, it could signify that a program still needs to continue iterating, but at least one vector lane is idle and a reorganization of the data vectors might increase the throughput.

The template functions that reduce a mask object need to be declared in such a way that they do not participate in overload resolution

```
namespace Vc {
template <typename Mask> bool all_of(const Mask &m);
template <typename Mask> bool any_of(const Mask &m);
template <typename Mask> bool none_of(const Mask &m);
template <typename Mask> bool some_of(const Mask &m);

inline bool all_of(bool b) { return b; }
inline bool any_of(bool b) { return b; }
inline bool none_of(bool b) { return !b; }
inline bool some_of(bool) { return false; }

namespace target_dependent {
using Vc::all_of;
using Vc::any_of;
using Vc::none_of;
using Vc::some_of;
}  // namespace target_dependent
}  // namespace Vc
```

Listing 9: Declaration of the `Mask<T>` reduction functions.

unless the template argument actually is a `Mask<T>` type (from any internal namespace).

In addition to the declarations for the `Vc::Mask` types, the reduction functions are also declared for `bool` arguments. That way the functions can be used in generic code where scalar types and `Vc::Vector` types can be used at the same time.

5 WRITE-MASKING

The term write-masking is used to denote the expression that disables an arbitrary set of vector lanes for writes to the destination register (or memory location). This is equivalent to the conditional move operation for scalars, applied to several values in parallel. Hardware support for write-masking requires a rather simple operation: instead of writing all bits from some temporary buffer to the destination register, some lanes are disabled, thus keeping the old value in the destination register unchanged. But, from the language side, this operation has only been implemented via implicit masking (such as the masked `if` statements in Cilk Plus [1]) or blend functions, which essentially implement the SIMD equivalent of the C++ ternary operator (conditional operator).

5.1 CONDITIONAL OPERATOR

For SIMD blend operations, the conditional operator (`a < b ? 1 : -1`) would be a very natural solution. It is straightforward to translate this conditional expression from scalar context into SIMD context. The

operator expresses, that for a given condition, its result should be the value of either the first or the second expression after the question mark. In the SIMD case, where a boolean is replaced by a vector of booleans, the conditional operator states that the results of the first expression must be blended with the results of the second expression according to the mask in the conditional expression before the question mark.

But with the current C++ standard, overloading the conditional operator is not allowed [2, §13.5]. According to Stroustrup [5] "there is no fundamental reason to disallow overloading of `?:`". Therefore, until C++ gains this ability, conditional operators have to be replaced by a function call for supporting SIMD types.

For the Vc library, I defined the function

```
Vector<T> iif(Mask<T>, Vector<T>, Vector<T>).
```

The name `iif` is an abbreviation for *inline-if*. To allow generic use of this function, Vc provides the overload

```
T iif(bool, T, T).
```

Thus `iif` can be used in template functions where both `bool`s and Vc mask types may be used as the first argument to `iif`. Listing 10

```
void filter(/*...*/) {
  // ...
  float_t sigma2 = measurementModel.sigma2;
  float_t sigma216 = measurementModel.sigma216;
  float_t hch = measurementModel * F.slice<0, 2>();
  float_t denominator = Vc::iif (hch < sigma216, hch + sigma2, hch);
  // ...
}
```

Listing 10: Part of the Kalman-Filter code that uses `iif`. The `float_t` type can be defined as either `float` or `float_V`

shows how `iif` is used inside the Kalman-Filter. The `float_t` type can be defined to anything that returns either a boolean or a Vc mask on `operator<`. Thus the implementation of the algorithm is generically usable for SIMD and scalar types.

5.2 Write-masked assignment operators

The `iif` function would suffice to translate any scalar conditional code to a vectorized code. But it is neither a good general interface, nor does it properly express intention of the code, hiding behind unnecessarily complex expressions. Therefore, I created a new syntax for the `Vector<T>` types to express conditional assignment with any assignment operator:

```
x(x < 0) *= -1;
```

This line of code reads as: multiply `x` with `-1` where `x` is less than `0`. The general syntax is *vector-object* (*mask-object*) *assignment-operator initializer-clause*. The `Vector<T>` class template therefore declares the function call operator as shown in Listing 11. This oper-

```
1  WriteMaskedVector<T> operator()(MaskType);
```

Listing 11: Declaration of the function call operator for write-masking support in `Vector<T>`.

ator returns a temporary object which stores a (non-const) lvalue-reference to the `Vector<T>` object and a copy of the mask object. The `WriteMaskedVector` class template overloads all assignment operators which implement the write-masked assignment to the `Vector<T>` object.

In addition to assignment operators the `WriteMaskedVector` can also implement the increment and decrement operators.

5.2.1 Alternative: `Vc::where`

The function call operator syntax has a significant downside: It is impossible to write generic functions with conditional assignment that work with SIMD vector types and fundamental types. It would require an operator overload for fundamental types, or rather a change to the language specification. Therefore, I worked on alternative solutions:

```
Vc::where(x < 0, x) *= -1;   // variant (1)
Vc::where(x < 0) | x *= -1;  // variant (2)
Vc::where(x < 0) (x) *= -1;  // variant (3)
```

The goal was to have a function/expression that can return a `WriteMaskedVector` object for vector types and fundamental types.

- The first variant uses less “magic” but does not have such an obvious connection between the modified variable `x` and the assignment operator.
- The second variant states more clearly that an assignment to `x` is executed. But it requires an operator between the `where` function and the assignee that has lower precedence than assignment operators. In any case, this operator will be deprived of its normal semantics, which is a potentially confusing solution.
- The third variant is a compromise of the first two variants. It uses the function call operator of the return type of the `where`

function to make it clearer that assignment is applied to the `x` variable.

All three variants of the `where` function can be overloaded with fundamental types.

All four solutions for write-masking (`where` and `Vector<T>::operator()`) can be translated to optimal SIMD code and thus only differ in syntax and semantics. I am looking for feedback from the community on the preferred solution for an interface for write-masking.

5.2.2 Return type of masked assignment operators

The assignment operators that are declared in the `WriteMaskedVector` type can return either:

- A reference to the `Vector<T>` object that was modified.
- A temporary `Vector<T>` object that only contains the entries where the mask is `true`.
- The `WriteMaskedVector` object.
- Nothing (`void`).

The most sensible choice seems to be a reference to the modified `Vector<T>` object. But then the statement `(x(x < 0) *= -1) += 2` may be surprising: it adds `2` to all vector entries, independent of the mask. Likewise, `y += (x(x < 0) *= -1)` has no obvious interpretation anymore because of the mask in the middle of the expression.

If we consider that a write-masked assignment is used as a replacement for an `if`-statement, using `void` as return type is a more fitting choice. An `if`-statement has 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.

6 Masked Gather & Scatter

Finally, let us look at masked gather and scatter operations. (Gather/scatter was introduced in N4184.) A gather expression creates a temporary `Vector<T>` object that can be assigned to an lvalue. If the user wants to assign only a masked subset of the gathered values, the write-masked assignment as described in Section 5 suffices.

But write-masked gather is special in that there are memory reads which are unnecessary (and thus should be omitted for performance reasons) and potentially even invalid, out-of-bounds accesses. Therefore, we rather want write-masked assignment from a gather operation to propagate to the gather function itself. Then the gather function can use the mask to omit loads for the SIMD lanes that will not be used on assignment.

The scatter function, called from a scatter expression, must use the mask information for the same reasons: it should avoid unnecessary stores and must omit out-of-bounds stores. But for scatters the scatter expression is on the left hand side of the assignment operator and thus basically follows the same logic as normal write-masking.

To support masked gathers, the `WriteMaskedVector` class declares an assignment operator for an rvalue-reference to `SubscriptOperation`:

```
template <typename T, typename I, typename S>
void operator=(SubscriptOperation<T, I, S> &&);
```

The operator will call `gatherArguments` on the `SubscriptOperation` object and use that information to execute a masked gather and assign the result to the referenced `Vector<T>` object.

Note that this only allows direct assignment from the gather expression. The user can not execute operations in addition (though this could be supported via expression templates).

7 Conclusion

I have presented the `Mask<T>` class and associated functions and operators that can be used to vectorize conditional statements with little effort and in an understandable and intuitive syntax. There are still a few open questions on how to create the best write-masking syntax. Also there are some useful functions that I have implemented in Vc, such as population count, index of first `true` value, subscript operator for reading and setting individual mask entries, and a few more that are not described here. This document is a work in progress on the mask type, as I am looking for guidance how to proceed.

A EXAMPLE: MANDELBROT

```
typedef SimdArray<int, float_v::Size> IV;
for (int y = 0; y < imageHeight; ++y) {
  const float_v c_imag = y0 + y * scale;
  for (IV x = IV::IndexesFromZero(); any_of(x < imageWidth);
       x += float_v::Size) {
    const std::complex<float_v> c(x0 + x * scale, c_imag);
    std::complex<float_v> z = c;
    IV n = 0;
    auto inside = norm(z) < 4.f;
    while (any_of(inside && n < 255)) {
      z = z * z + c;
      where(inside) | n += 1;
      inside = norm(z) < 4.f;
    }
    IV colorValue = 255 - n;
    colorizeNextPixels(colorValue);
  }
}
```

Listing 12: A Vc implementation of the Mandelbrot algorithm.

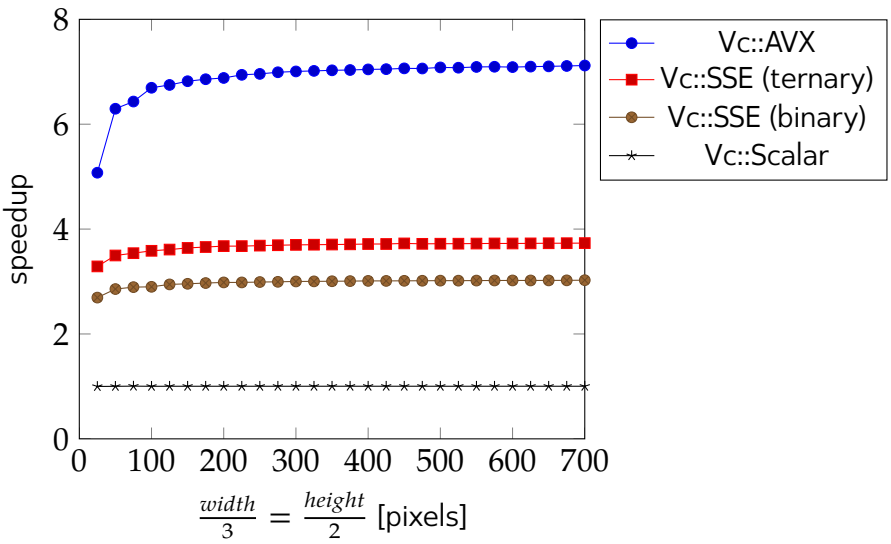

Figure 1: Runtime of the Vc implementation of the Mandelbrot algorithm normalized to an optimized implementation using `float` and `int`.

B 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).

- Thanks to all the useful and encouraging feedback from Vc users in the community.

C REFERENCES

[1] Intel Corporation. Tutorial: Array Notation | Cilk Plus. URL https://www.cilkplus.org/tutorial-array-notation.

[2] ISO/IEC JTC1/SC22/WG21. ISO International Standard ISO/IEC 14882:2011(E) – Programming Language C++, 2011. URL http://isocpp.org/.

[3] 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.

[4] Victor Luchangco, Jens Maurer, Michael Wong, et al. N3999: Standard Wording for Transactional Memory Support for C++. ISO/IEC C++ Standards Committee Paper, 2014. URL http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2014/n3999.pdf.

[5] Bjarne Stroustrup. Stroustrup: C++ Style and Technique FAQ, 2013. URL http://www.stroustrup.com/bs_faq2.html#overload-dot.