Document Number: N4184
Revision of: N3759
Date: 2014-10-10
Project: Programming Language C++, Library Working Group
Reply-to: Matthias Kretz <kretz@compeng.uni-frankfurt.de>

SIMD Types: The Vector Type & Operations

ABSTRACT

This paper describes a template class for portable SIMD Vector types. The class is portable because its size depends on the target system and only operations that are independent of the SIMD register size are part of the interface.

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

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

1.2 RELATION TO N3759

N3759 presented the main idea but skipped many details and features in order to focus on the concept. This document extends the types presented in N3759 and discusses interface decisions and expected behavior in detail.

2 INTRODUCTION

2.1 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 $\mathcal{W}_{\mathtt{T}}$ successive scalar values starting from a given address into a SIMD register.
- Store instructions: store from a SIMD register to $\mathcal{W}_{\mathtt{T}}$ 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.

The set of available operations can 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.

2.2 MOTIVATION FOR SIMD TYPES

There is no need to motivate explicit SIMD programming. It is very much needed. The open question is only: “How?”.

There have been several approaches to vectorization. I’d like to only discuss the merits of SIMD types here.

SIMD registers and operations are the low-level ingredients to SIMD programming. Higher-level abstractions can be built on top of these. If the lowest-level access to SIMD is not provided, users of C++ will be constrained to work within the limits of the provided abstraction.

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 not an 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.

2.3 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.)

3 SIMD VECTOR TYPE REQUIREMENTS

A thin abstraction on top of intrinsic or builtin types can provide the following desired properties:

- The value of an object of `Vector<T>` consists of $\mathcal{W}_{\mathtt{T}}$ scalar values of type `T`.
- The `sizeof` and `alignof` of `Vector<T>` objects is target-dependent.
- Scalar entries of a SIMD vector can be accessed via lvalue reference.
- The number of scalar entries ($\mathcal{W}_{\mathtt{T}}$) is accessible as a constant expression.
- Operators that can be applied to `T` can be applied to `Vector<T>` with the same semantics per entry of the vector.
- The result of each scalar value of an operation on `Vector<T>` does not depend on $\mathcal{W}_{\mathtt{T}}$.[1]
- The syntax and semantics of the fundamental arithmetic types translate directly to the `Vector<T>` types. There is an additional constraint for implicit type conversions, though: `Vector<T>` does not implicitly convert to `Vector<U>` if $\mathcal{W}_{\mathtt{T}} \neq \mathcal{W}_{\mathtt{U}}$ for any conceivable target system.

1 Obviously the number of scalar operations executed depends on $\mathcal{W}_{\mathtt{T}}$. But the resulting value of each scalar operation that is part of the operation on `Vector<T>` is independent.

```
 1 namespace Vc {
 2 namespace target_dependent {
 3 template <typename T> class Vector
 4 {
 5   implementation_defined data;
 6 
 7 public:
 8   typedef implementation_defined VectorType;
 9   typedef T EntryType;
10   typedef implementation_defined EntryReference;
11   typedef Mask<T> MaskType;
12 
13   static constexpr size_t MemoryAlignment = implementation_defined;
14   static constexpr size_t Size = implementation_defined;
15   static constexpr size_t size() { return Size; }
16 
17   // ... (see the following Listings)
18 };
19 template <typename T> constexpr size_t Vector<T>::MemoryAlignment;
20 template <typename T> constexpr size_t Vector<T>::Size;
21 
22 typedef Vector<             float>     float_v;
23 typedef Vector<            double>    double_v;
24 typedef Vector<  signed long long>  longlong_v;
25 typedef Vector<unsigned long long> ulonglong_v;
26 typedef Vector<  signed      long>      long_v;
27 typedef Vector<unsigned      long>     ulong_v;
28 typedef Vector<  signed       int>       int_v;
29 typedef Vector<unsigned       int>      uint_v;
30 typedef Vector<  signed     short>     short_v;
31 typedef Vector<unsigned     short>    ushort_v;
32 typedef Vector<  signed      char>     schar_v;
33 typedef Vector<unsigned      char>     uchar_v;
34 }  // namespace target_dependent
35 }  // namespace Vc
```

Listing 1: SIMD vector class definition

- The compiler can recognize optimization opportunities and apply constant propagation, dead code elimination, common subexpression elimintation, and all other optimization passes that apply to scalar operations.[2]

4 THE VC::VECTOR<T> CLASS

The boilerplate of the SIMD vector class definition is shown in Listing 1. There are several places in this listing where the declaration says "*target-dependent*" or "*implementation-defined*". All of the following listings, which declare functions of the `Vector<T>` class (to insert at line 17), do not require any further implementation-specific differences. All these differences in `Vector<T>` are fully captured by the code shown in Listing 1.

2 So much for the theory. In practice, there still are many opportunities for compilers to improve optimization of SIMD operations.

The only data member of the vector class is of an implementation-defined type (line 5). This member therefore determines the size and alignment of `Vector<T>`. Obviously, the SIMD classes may not contain virtual functions. Otherwise a virtual table were required and thus objects of this type would be considerably larger (larger by the minimum of the pointer size and the alignment of `VectorType`).

4.1 MEMBER TYPES

The member types of `Vector<T>` abstract possible differences between implementations and ease generic code for the SIMD vector types.

`VectorType`
: (line 8) is the internal type for implementing the vector class. This type could be an intrinsic or builtin type. The exact type that will be used here depends on the compiler and compiler flags, which determine the target instruction set. Additionally, if an intrinsic type is used it might not be used directly (in line 5) but indirectly via a wrapper class that implements compiler-specific methods to access scalar entries of the vector.

  The `VectorType` type allows users to build target- and implementation-specific extensions on top of the predefined functionality. This requires a function that returns an lvalue reference to the internal data (line 5). See Section 13 for such functions.

`EntryType`
: (line 9) is always an alias for the template parameter `T`. It is the logical type of the scalar entries in the SIMD vector. The actual bit-representation in the SIMD vector register may be different to `EntryType`, as long as the observable behavior of the scalar entries in the object follows the same semantics.

`EntryReference`
: (line 10) is the type returned from the non-const subscript operator. This type should be an lvalue reference to one scalar entry of the SIMD vector. It is not required for `EntryReference` to be the same as `EntryType &`. Consider an implementation that uses 32-bit integer SIMD registers for `Vector<short>`, even though a `short` uses only 16 bits on the same target. Then `EntryReference` has to be an lvalue reference to `int`.

If `EntryReference` were declared as `short &` then sign extension to the upper 16 bits would not work correctly on assignment.

`MaskType`
: (line 11) is the mask type that is analogous to `bool` for scalar types. The type is used in functions that have masked overloads and as the return type of compare operators. A detailed discussion of the class for this type is presented in [N4185].

4.2 CONSTANTS

The vector class provides a static data member (`Size`) and static member function (`size()`) which both identify the number of scalar entries in the SIMD vector (lines 14 and 15). This value is determined by the target architecture and therefore known at compile time. By declaring the `Size` variable `constexpr`, the value is usable in contexts where constant expressions are required. This enables template specialization on the number of SIMD vector entries in user code. Also it enables the compiler to optimize generic code that depends on the SIMD vector size more effectively. The additional `size()` function makes `Vector<T>` implement the standard container interface, and thus increases the reusability in generic code.[3]

The `MemoryAlignment` static data member defines the alignment requirement for a pointer passed to an aligned load or store function call of `Vector<T>`. The need for a `MemoryAlignment` static data member might be surprising at first. In most cases the alignment of `Vector<T>` will be equal to `MemoryAlignment`. But as discussed in Section **??**, implementations are free to use a SIMD register with different representation of the scalar entries than `EntryType`. In such a case, the alignment requirements for `Vector<T>` will be higher than $\mathcal{W}_{\mathrm{T}} \times$ `sizeof(T)` for an aligned load or store. Note that the load and store functions allow converting loads (Section 6.1). These functions need a pointer to memory of a type different than `EntryType`. Subsequently the alignment requirements for these pointers can be different. Starting with C++14 it may therefore be a good idea to declare `MemoryAlignment` as:

```
template <typename U>
static constexpr size_t MemoryAlignment = implementation_defined;
```

3 It would suffice to define only the `size()` function and drop `Size`. Personally, I prefer to not use function calls in constant expressions. Additionally, a 50% difference in the number of characters makes `Size` preferable because it is such a basic part of using SIMD types.

4.3 namespace

Line 2 defines a namespace that is not part of the standard interface. A user should be able to access the natural SIMD registers and operations via the `Vector<T>` type and its typedefs `float_v`, `double_v`, …. But consider the case of compiling two translation units with different target compiler flags. In that case `Vector<T>` could map to different SIMD widths even though they had the same type. Thus, the code would compile, link, and possibly even run—but not correctly—. The namespace turns the `Vector<T>` types into different symbols and thus ensures correct linkage of different translation units. An implementation may choose to document the target-dependent namespaces, as discussed below (Section 12.2).

4.4 simd type aliases

The code on lines 22–33 declares handy aliases for the `Vector<T>` class template. The intent of these aliases is to make SIMD vector code more concise and recognizable.

There is a design decision here: whether to use the types `char`, `short`, `int`, `long`, and `long long` or the `int8_t`, `int16_t`, `int32_t`, and `int64_t` typedefs. The problem with the latter list is that these types are optional. Thus, the definition of `int32_v` (≡ `Vector<int32_t>` is optional, too. Since the `int`*N*`_t` typedefs must map to one of the fundamental types in the first list of types, definition of the SIMD Types with the fundamental types of the first list is more general.

It is a sensible choice to additionally declare `typedef`s for `int`*N*`_v` in the presence of `int`*N*`_t` typedefs.

5 SIMD VECTOR INITIALIZATION

The interface for initialization—excluding loads (Section 6) and gathers (Section 11)—is shown in Listing 2. The Vc vector types are not POD types because we want to have full control over the implicit and explicit initialization methods (also because there is currently no guarantee about the POD-ness of `VectorType`). The decision for what constructors to implement follows from syntactical and semantical compatibility with the builtin arithmetic type `EntryType`. Thus the expressions in Listing 3 must compile and “do the right thing”.

```cpp
// init to zero
Vector();

// broadcast with implicit conversions
Vector(EntryType);

// disambiguate broadcast of 0 and load constructor
Vector(int);  // must match exactly

// implicit conversion from compatible Vector<T>
template <typename U>
  requires ImplicitConversionAllowed<U, EntryType>()
Vector(Vector<U>);

// static_cast from vectors of possibly (depending on target)
// different size (dropping values or filling with 0 if the size is
// not equal)
template <typename U>
  requires ExplicitConversionAllowed<U, EntryType>()
explicit Vector(Vector<U>);
```

Listing 2: Initialization and conversion constructors for `Vector<T>`

```cpp
double_v a0{}, a1();  // zero-initialized

float_v  b = 0, c = 2.f;
short_v  d = -1;          // -1
ushort_v e = -1;          // numeric_limits<unsigned short>::max()
int_v    f = short(-1);   // -1
uint_v   g = short(-1);   // numeric_limits<unsigned int>::max()

ushort_v h = d;  // numeric_limits<unsigned short>::max()
int_v    i = g;  // implementation-defined value for i

float_v  j = static_cast<float_v>(a);
double_v k = static_cast<double_v>(d);
```

Listing 3: A few statements that are valid initialization expressions, if the builtin scalar types were used. They should work equally well with the `Vector<T>` types (as shown).

DEFAULT CONSTRUCTOR The default constructor in line 2 creates a zero-initialized object. The constructor may not keep the object uninitialized. Because, if the expression `T()` is used with a fundamental type, a "prvalue of the specified type, which is value-initialized" [6, §5.2.3] is created. The term "value-initialized" implies "zero-initialized" for fundamental types.

DESTRUCTOR OR COPY/MOVE CONSTRUCTOR There is no need for a destructor and explicitly declared copy and/or move constructors as long as the vector type does not use external storage.[4] There might be a need for these functions if the `Vector<T>` type is used as a handle to remote data, for instance on an accelerator card. Such an implementation needs to be able to add the destructor and copy-/move constructors and assignment operators, though.

5.1 BROADCASTS

The constructor on line 5 declares an implicit conversion from any value of a type that implicitly converts to `EntryType`. This means that in places where a variable of type `Vector<T>` is expected, a variable of type `T` works as well. The constructor then broadcasts the scalar value to all entries of the SIMD vector. This kind of implicit conversion makes it very easy and natural to use numeric constants in SIMD code.

The constructor on line 8 is a special case of the preceding broadcast constructor on line 5. This constructor is required because initialization with the literal `0` is ambiguous otherwise. The load constructor (see Section 6 line 8 of Listing 7) and the `Vector(EntryType)` constructor match equally well, with just a single implicit type conversion. If `EntryType` is `int`, then this extra constructor overload on line 8 must be removed from overload resolution, because otherwise the signatures of the constructors on lines 5 and 8 are equal. For all the other `Vector<T>` types the `Vector(int)` constructor must not participate in overload resolution except when the argument to the constructor is exactly an `int`. Otherwise the expression `short_v v(1u)` would be ambiguous. This can be implemented with a template parameter which must be deduced to exactly be an `int` using an additional `enable_if` parameter:

4 *Implementation experience:* With the Linux ABI, a non-trivial copy constructor changes parameter passing. Using a trivial copy constructor, a `Vector<T>` parameter passed by value is passed in a single SIMD register. Using a non-trivial copy constructor, the same code is translated to pass the parameter via the stack.

```
template <typename A, typename B>
concept bool ImplicitConversionAllowed() {
  return is_integral<A>::value && is_integral<B>::value &&
         is_same<conditional_t<is_signed<A>::value,
                                     make_unsigned_t<A>, make_signed_t<A>>,
                 B>::value;
}
```

Listing 4: Possible implementation of the `ImplicitConversionAllowed` concept.

```
template <typename U>
Vector(U a, typename enable_if<is_same<U, int>::value &&
                                !is_same<U, EntryType>::value,
                          void *>::type = nullptr);
```

5.2 SIMD vector conversions

The fundamental arithmetic types implicitly convert between one another. (Not every such conversion is value-preserving, which is why some compilers emit warnings for type demotions.) For SIMD types conversions should work in the same manner. However, there is no guarantee that the number of scalar entries in a SIMD vector type is equal to the number of entries in a different type. Therefore, the conversions between `Vector<T>` types are split into implicit and explicit conversions. The idea is expressed with `requires` expressions [N3819].

It is important that code written with the `Vector<T>` types is as portable as possible. Therefore, implicit casts may only work if $\mathcal{W}_\mathtt{T} = \mathcal{W}_\mathtt{U}$ holds on every possible target system. There is no real guarantee for this to work with any type combination. It is a reasonable assumption, though, that $\mathcal{W}_\mathtt{T} = \mathcal{W}_{\mathtt{make\_signed\_t<T>}}$ for any unsigned integral type `T` (since `make_signed_t<T>` "occupies the same amount of storage" as `T` [6, §3.9.1]). Therefore, the `ImplicitConversionAllowed` concept (line 12) must check for both types to be integral and to differ only in signedness (Listing 4).

If only these implicit casts were allowed, then the interface would be too restrictive. The user needs to be able to convert between SIMD vector types that possibly have a different number of entries. The constructor in line 20 therefore allows all remaining conversions not covered by the preceding constructor. Since the constructor is declared `explicit` it breaks with the behavior of the builtin arithmetic types and only allows explicit casts (such as `static_cast` or explicit constructor calls).

```
float_v f(float_v x) {
  float_v r;
  for (size_t i = 0; i < float_v::Size; i += double_v::Size) {
    r = r.shiftIn(
        double_v::Size,
        static_cast<float_v>(g(static_cast<double_v>(x.shifted(i)))));
  }
  return r;
}
```

Listing 5: Sample code that portably calls the function `g(double_v)` on a full `float_v`.

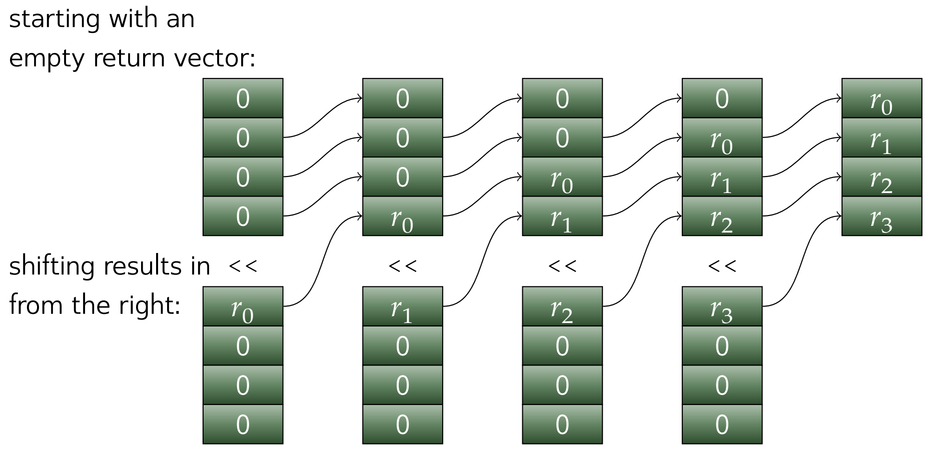

Figure 1: Vector filling algorithm used in Listing 5

It would certainly be possible to define additional guaranteed $\mathcal{W}_{\mathtt{T}}$ relations by requiring implementations to implement some vector types with multiple registers. As a matter of fact, this is how I initially implemented Vc: $\mathcal{W}_{\mathtt{int}} = \mathcal{W}_{\mathtt{float}}$ was guaranteed. This was easy to support with only SSE and Larrabee (now Xeon Phi) implementations, but required two SSE `int` vectors to implement the AVX target. This is due to the initial AVX revision not including instruction support for integer vectors of 256 bits. However, conversions between different integer types and floating-point types are very important to many algorithms and therefore must be supported.

The conversion constructor in line 20 converts $n = \min(\mathcal{W}_{\mathtt{T}}, \mathcal{W}_{\mathtt{U}})$ values for a conversion from `Vector<T>` to `Vector<U>`. The remaining $n_0 = \max(0, \mathcal{W}_{\mathtt{U}} - n)$ entries in the target variable are set to zero. This implies that user code that wants to portably implement an algorithm, which needs to convert between vectors of possibly different widths, must employ a pattern such as shown in Listing 5.

The idea of the code in Listing 5 is visualized in Figure 1. However, this code does not qualify as an intuitive solution. A developer

```
float_v f(float_v x) {
  float_v r;
  for (size_t i = 0; i < float_v::Size; i += double_v::Size) {
    r[{i, double_v::Size}] = static_cast<float_v>(
        g(static_cast<double_v>(x[{i, double_v::Size}])));
  }
  return r;
}
```

Listing 6: Possible syntax for implementing Listing 5 via the subscript operator.

will rather think of the subscript operator for accomplishing the task. Subscripting for a target-dependent number of entries in the vector requires an even more awkward and non-obvious interface. Listing 6 shows a possible syntax for a portable subscripting solution. It thus becomes clear that a better solution for the cast issue needs to be provided. One possible solution uses a new cast function (`simd_cast`) that can cast from multiple `Vector<T>` to one `Vector<U>` or from one `Vector<U>` to multiple `Vector<T>`. An even more convenient solution builds upon these casts and the `Vector<T>` type to create a `SimdArray<T, N>` abstraction for composing multiple `Vector<T>` into one type (see Document on SIMD Arrays—still to be written—).

6 LOADS AND STORES

The vector types need an interface for loading and storing SIMD vectors from/to memory. In contrast to the primary motivation of providing the same syntax and semantics as for fundamental types, these functions have no equivalent in their underlying fundamental types `T`. The load & store functions are required because they are a portable and efficient interface for converting between arrays of a fundamental scalar type and vector types. Without load & store functions, data could not reasonably be converted in a portable fashion: All input and output of data to a vectorized algorithm would be required to exclusively use the `Vector<T>` types. Obviously, this would be an unrealistic requirement for the majority of applications.

Nevertheless, loads and stores are an unfortunate requirement that should rather be eliminated from the set of required operations user code has to call. There are different ideas to hiding loads and stores behind abstractions on top of SIMD types and standard containers. These abstractions still build upon the load & store functions in the `Vector<T>` interface, though. Appendix A describes one of the ideas.

```cpp
// load member functions
void load(const EntryType *mem);
template <typename Flags> void load(const EntryType *mem, Flags);
template <typename U, typename Flags = UnalignedT>
void load(const U *mem, Flags = Flags());

// load constructors
explicit Vector(const EntryType *mem);
template <typename Flags>
explicit Vector(const EntryType *mem, Flags flags);
template <typename U, typename Flags = UnalignedT>
explicit Vector(const U *mem, Flags flags = Flags());

// store functions
void store(EntryType *mem) const;
void store(EntryType *mem, MaskType mask) const;
template <typename Flags>
void store(EntryType *mem, Flags flags) const;
template <typename Flags>
void store(EntryType *mem, MaskType mask, Flags flags) const;
template <typename U, typename Flags = UnalignedT>
void store(U *mem, Flags = Flags()) const;
template <typename U, typename Flags = UnalignedT>
void store(U *mem, MaskType mask, Flags = Flags()) const;
```

Listing 7: Declaration of the load and store functions.

Listing 7 shows the interface for loads and stores for the `Vector<T>` types. These functions convert $\mathcal{W}_{\mathrm{T}}$ consecutively stored scalar objects of type `T` to one object of type `Vector<T>` and back. Thus, the start address (pointer to the first scalar object) and the type of the scalar objects are sufficient to fully characterize the required load or store operation. (The case of randomly distributed scalars is handled by gather and scatter functions, which are described in Section 11.)

6.1 Converting Loads and Stores

Some SIMD hardware can convert between different data types without extra runtime overhead when executing a load or store instruction [4]. Therefore, and because it is very convenient for writing portable conversion code, the load & store functions provide a generic variant that can access arrays of different scalar types. Semantically, these functions behave as described in Listing 8. Thus, $\mathcal{W}_{\mathrm{T}}$ values of type `U` are converted with load/store functions in `Vector<T>`, independent of $\mathcal{W}_{\mathrm{U}}$, in contrast to the converting constructor in Section 5.2.

Not all conversions are equally efficient in terms of hardware support. But for reasons of portability, the full set of conversions between fundamental arithmetic types is made available through these functions.

```
void Vector<T>::load(const U *mem) {
  for (size_t i = 0; i < Size; ++i) {
    (*this)[i] = static_cast<T>(mem[i]);
  }
}
void Vector<T>::store(U *mem) const {
  for (size_t i = 0; i < Size; ++i) {
    mem[i] = static_cast<U>((*this)[i]);
  }
}
```

Listing 8: The semantics of a converting load.

6.2 load/store flags

SIMD hardware makes a difference between aligned and unaligned vector loads and stores. Additionally, most algorithms can be optimized if the developer can hint at the temporal usage of the data. The alignment can, in theory, be determined from the start address, and thus would not require additional specification in the function call. But, since the alignment can only be determined from the pointer value at runtime, such a check would incur a penalty. Using unaligned load/store instructions unconditionally would be more efficient than checking the alignment of the pointer. An unaligned load/store instruction in hardware can do the alignment test much more efficiently. Therefore, per default, the load/store functions translate to unaligned load/store instructions.

6.2.1 alignment

If the user can guarantee alignment, a tag type can be used as last argument to select the optimized load/store instructions at compile time, without any runtime overhead. It is important that the API is built via a `template` and tag type, rather than a boolean (or enum) function argument. A boolean function argument cannot guarantee compile-time optimization. Especially, such an API would allow passing a non-constant expression as flag variable which cannot be optimized at all. Via the tag type the user of the API is required to provide a constant expression and thus decide between aligned or unaligned memory access when he writes the code.

6.2.2 non-temporal access

Loads and stores can be further optimized for non-temporal accesses. Many data-parallel algorithms use a streaming pattern, where the input and/or output memory locations are used only once. Therefore, this data should not evict other data, which might be used repeatedly in the algorithm, from the CPU caches. The load/store functions in Vc can therefore be called with the `Vc::Streaming` tag type. This tag hints to the `Vector<T>` implementation that the data can be moved past the caches. (Most CPUs can use specialized load and store buffers for streaming loads and stores.) If the requested load/store operation cannot be executed as a streaming variant, the implementation will silently fall back to a non-streaming variant.

Streaming stores executed with the `Vc::Streaming` tag may use non-globally ordered stores if the target CPU supports this. Thus, two stores to the same memory location, where at least one is a streaming store, have undefined behavior unless a memory fencing operation is called between the stores. This allows to reach the highest store throughput, but requires a good understanding of the implications when used by a developer.

6.2.3 prefetching

A last flag that I implemented for the load/store functions makes prefetching in loops significantly simpler. By adding the `Vc::PrefetchDefault` tag type, the `Vector` implementation is asked to emit software prefetch instructions for a target-dependent predefined stride. Thus, a call to

```
float_v(memory, Vc::Aligned |
          Vc::PrefetchDefault)
```

may result in up to three instructions being called, one of which is the load instruction. In addition, prefetch instructions for the lowest level cache and second lowest level cache may be called. These prefetches are called with a predefined offset to the `memory` address that is passed to the load function.

The prefetch flag is therefore a shorthand to prefetching explicitly in many loops. But not all loops were created equal, which is why instead of the predefined strides the user may also set the strides explicitly. In almost all cases, a developer adds prefetches after the program or component is already working and is only modified for speed optimizations. The developer then determines the prefetch strides through intuition and/or trial and error.

Note that prefetches only need to be called once for any address inside one cache line. Thus, two subsequent loads/stores to neighboring SIMD vectors may result in more software prefetch instructions than necessary. This depends on the ratio of the cache line size to the vector register size. As this ratio is target dependent, the API appears to introduce a portability issue in this case. There is no easy solution

```
1  Vector &operator++();
2  Vector  operator++(int);
3  Vector &operator--();
4  Vector  operator--(int);
5
6  MaskType operator!() const;
7  Vector operator~() const;
8  Vector operator+() const;
9  Vector operator-() const;
```

Listing 9: Declaration of unary operators.

from the load/store interface side. But the compiler is, in theory, able to drop the superfluous prefetch instructions.[5]

7 UNARY OPERATORS

The unary operators (increment, decrement, logical negation, one's complement, unary +, and unary -) behave as $\mathcal{W}_{\mathrm{T}}$ applications of the operator to the scalar values in the vector. But, there is an API issue that results from *integral promotion*, which is applied to the operand of unary +, unary -, and one's complement. Integral promotion leads to operands of builtin integral types smaller than `int`/`unsigned int` getting promoted to `int` or `unsigned int` before the operator is applied. This implies that `ushort_v` and `short_v` would have to return `int_v` from unary +, unary -, and one's complement. But since $\mathcal{W}_{\texttt{short}} = \mathcal{W}_{\texttt{int}}$ does not hold for most targets, this is not possible (except if the return type were `SimdArray<int, short_v ::Size >` [Document on SIMD Arrays—still to be written—]). For the fundamental integral types, the integral promotion to `int` normally does not cause a performance hit because `int` is defined to be the natural integer type of the target CPU. But for vector types, integral promotion may require more overhead than is acceptable. Therefore the SIMD vector types do not perform integral promotion. This is also the case for binary operators (see Section 8). On the other hand, integral promotion makes much more sense for the `simdarray` types.

The declaration of the interface for the unary operators is shown in Listing 9. As discussed above, the return types of the functions in lines 7–9 do not follow the integral promotion rules. These operators can therefore lead to subtle differences to scalar code as the example

5 This is possible if the relative difference between prefetch instructions is considered by the compiler. It could apply an algorithm that keeps the first prefetch call and drops every subsequent prefetch call that would reference the same cache line as a previous call.

```
1 unsigned short a = 40000;
2 auto b = -a; // decltype(b) == int
3
4 ushort_v v = 40000;
5 auto w = -v; // decltype(w) == ushort_v
6
7 assert(b == w[0]); // this fails
```

Listing 10: Subtle differences between scalar code and vector code, because of differences in integer promotion.

in Listing 10 demonstrates. The assertion in line 7 fails because the builtin compare operator performs integral promotion, promoting the right hand side of the comparison to `int`. But while `b` holds the value -40000, `w` holds $\mathcal{W}_{\texttt{short}}$ values of $2^{16} - 40000 = 25536$. Conversion of one value in `w` to `int` does not change that value. Thus, line 7 compares `-40000 == 25536`.

8 BINARY OPERATORS

Binary operators provide arithmetic, comparison, bitwise, and shift operations. They implement the central part of the SIMD functionality by executing $\mathcal{W}_{\texttt{T}}$ operations in parallel on the SIMD vector entries. If the `EntryType` is not of integral type, the binary, shift, and modulo operators need to be disabled. They would certainly be implementable, but since the builtin non-integral types do not implement these operators, the SIMD types follow the same semantics.

The interface for these operators is shown in Listing 11. In this form of declaration, the compiler will allow the right-hand operand to be implicitly converted via a non-explicit conversion constructor. Thus, conversions from integer-vectors of differing signedness and broadcasts from scalars would be possible. But the resulting type would solely be determined by the type of the left-hand operand. Thus `int_v() + uint_v()` would result in an `int_v`, whereas `uint_v() + int_v()` would result in a `uint_v`. Also `int_v() + 1.f` would compile and result in an `int_v`, whereas any operation with a scalar value on the left-hand side (such as `1.f + float_v()`) would not compile at all. Thus, there is a need for further refinement of the binary operators, which can be done via non-member operators.

8.1 GENERIC NON-MEMBER BINARY OPERATORS

The definition of two of the non-member binary operators (one arithmetic and one comparison operator) is shown in Listing 12. There is

```
Vector operator* (Vector x) const;
Vector operator/ (Vector x) const;
Vector operator+ (Vector x) const;
Vector operator- (Vector x) const;
Vector operator% (Vector x) const {
  static_assert(std::is_integral<EntryType>::value,
                "the modulo operator can only be used with Vectors "
                "of integral type");
}

MaskType operator==(Vector x) const;
MaskType operator!=(Vector x) const;
MaskType operator>=(Vector x) const;
MaskType operator<=(Vector x) const;
MaskType operator> (Vector x) const;
MaskType operator< (Vector x) const;

Vector operator& (Vector x) const {
  static_assert(std::is_integral<EntryType>::value,
                "bitwise operators can only be used with Vectors of "
                "integral type");
}
Vector operator| (Vector x) const { ... }
Vector operator^ (Vector x) const { ... }
Vector operator<<(Vector x) const {
  static_assert(std::is_integral<EntryType>::value,
                "shift operators can only be used with Vectors of "
                "integral type");
}
Vector operator>>(Vector x) const { ... }
Vector operator<<(int x) const { ... }
Vector operator>>(int x) const { ... }
```

Listing 11: Declaration of binary operators.

```
template <typename L, typename R>
inline TypesForOperator<L, R> operator+(L &&x, R &&y) {
  using V = TypesForOperator<L, R>;
  return V(std::forward<L>(x)) + V(std::forward<R>(y));
}
// ...

template <typename L, typename R>
inline typename TypesForOperator<L, R>::MaskType operator==(L &&x,
                                                            R &&y) {
  using V = TypesForOperator<L, R>;
  return V(std::forward<L>(x)) == V(std::forward<R>(y));
}
// ...
```

Listing 12: Non-member operators for the Vc SIMD vector types.

only a slight difference in the return type between comparison operators and the remaining binary operators. Compares obviously must return a mask type and therefore require the `Vector<T> ::MaskType` return type. The operator's implementation simply forwards to the member operators using the same type for the left and right operands. The evaluation of this type does all the magic. Especially important is the fact that if `TypesForOperator<L, R>` leads to a substitution failure no error is emitted, but the operator is silently removed from the candidate list (SFINAE). Thus, `TypesForOperator<L, R>` determines every aspect of which binary operators are supposed to compile or throw an error and which implicit conversions are involved.

The `TypesForOperator<L, R>` type in Vc is defined as an alias template (see Listing 13, line 56) for the `TypesForOperatorInternal struct`. The alias template simplifies the implementation of `TypesForOperatorInternal` by swapping the types `L` and `R` if `L` is a non-vector type. Therefore, the first template argument is guaranteed to be a SIMD vector type, unless neither `L` nor `R` is a SIMD vector type. The third template argument (a boolean) provides a simple mechanism to specialize the `struct` for type combinations where a substitution failure should result (and thus inhibit the binary operator to participate in overload resolution). The conditions for this are simply that the non-member operators may only participate in overload resolution for type combinations that involve at least one vector type and where `L` and `R` are not equal, in which case the operator would lead to an ambiguity with the member operator in the vector class.

The `struct` in line 48 defines an empty type for operands where the operators are not supposed to match. Note that the `struct` does not contain the member type `type`, thus leading to the intended

```cpp
template <typename T, bool> struct MakeUnsignedInternal;
template <typename T> struct MakeUnsignedInternal<Vector<T>, true> {
  using type = Vector<typename std::make_unsigned<T>::type>;
};
template <typename T> struct MakeUnsignedInternal<Vector<T>, false> {
  using type = Vector<T>;
};
template <typename Test, typename T>
using CopyUnsigned = typename MakeUnsignedInternal<
    T, isIntegral<T>() && isUnsigned<Test>()>::type;

template <typename From, typename To>
constexpr bool isNarrowingFloatConversion() {
  return is_floating_point<From>::value &&
         (is_integral<To>::value || (is_floating_point<To>::value &&
                                     sizeof(From) > sizeof(To)));
}

template <typename T> static constexpr bool convertsToSomeVector() {
  return is_convertible<T, double_v>::value ||
         is_convertible<T, float_v>::value ||
         is_convertible<T, int_v>::value ||
         is_convertible<T, uint_v>::value ||
         is_convertible<T, short_v>::value ||
         is_convertible<T, ushort_v>::value;
}

template <typename V, typename W>
constexpr bool participateInOverloadResolution() {
  return isVector<V>() && !is_same<V, W>::value &&
         convertsToSomeVector<W>();
}

template <typename V, typename W>
constexpr enable_if<isVector<V>(), bool> isValidOperandTypes() {
  using type = CopyUnsigned<W, V>;
  return isVector<W>() ? (is_convertible<V, W>::value ||
                          is_convertible<W, V>::value)
                       : (is_convertible<W, type>::value &&
                          !isNarrowingFloatConversion<
                              W, typename type::EntryType>());
}

template <
    typename V, typename W,
    bool VectorOperation = participateInOverloadResolution<V, W>() &&
                           isValidOperandTypes<V, W>()>
struct TypesForOperatorInternal {};

template <typename V, typename W>
struct TypesForOperatorInternal<V, W, true> {
  using type = CopyUnsigned<W, V>;
};

template <typename L, typename R>
using TypesForOperator = typename TypesForOperatorInternal<
    decay_t<conditional_t< isVector<L>(), L, R>>,
    decay_t<conditional_t<!isVector<L>(), L, R>>>::type;
```

Listing 13: The traits the non-member binary operators need for SFINAE and return type evaluation.

```
template <
    typename V, typename W,
    bool IsIncorrect = participateInOverloadResolution<V, W>() &&
                       !isValidOperandTypes<V, W>()>
struct IsIncorrectVectorOperands {};
template <typename V, typename W>
struct IsIncorrectVectorOperands<V, W, true> {
  using type = void;
};

template <typename L, typename R>
using Vc_does_not_allow_operands_to_a_binary_operator_which_can_have_\
different_SIMD_register_sizes_on_some_targets_and_thus_enforces_\
portability =
    typename IsIncorrectVectorOperands<
        Traits::decay<Conditional<isVector<L>(), L, R>>,
        Traits::decay<Conditional<!isVector<L>(), L, R>>>::type;

template <typename L, typename R>
Vc_does_not_allow_operands_to_a_binary_operator_which_can_have_\
different_SIMD_register_sizes_on_some_targets_and_thus_enforces_\
portability<L, R> operator+(L &&, R &&) = delete;
```

Listing 14: Declaration of explicitly deleted operators for improved diagnostics on incorrect usage.

substitution failure. The `struct` in line 51 specializes for the case where `V` is a SIMD vector type, the operand types are different, `W` is convertible to a SIMD vector type, and the combination of types yields valid implicit conversions. Since not all conversions to SIMD vector types or between vector types are implicit, the binary operator may not invoke such a conversion and turn an explicit conversion to an implicit one. This is determined via the `isValidOperandTypes` function defined in line 35. For all allowed type combinations `V` and `W` the member type `type` in line 52 determines the SIMD vector type to use as return type and for conversion of the operands before calling the member operator.

By additionally declaring operator overloads that are `!isValid-OperandTypes` as deleted, the interface catches incorrect use and gives some hint to the user why the code does not compile (Listing 14). Currently C++ only allows to encode an explanation in the type name. N4186 describes the issue of custom diagnostics for ill-formed function overloads in more detail and suggests a simple extension to the standard to improve the diagnostic output.[6]

6 Using a `static_assert` for improved error messages is also possible here and it can be used to explain the error directly, thus making correcting errors in the usage of the interface easier. On the other hand, with a `static_assert`, a trait that checks whether a binary operator for two given operands is defined will return a positive answer even though an actual call would fail to compile because of the static assertion. (see [N4186] for the details)

The isValidOperandTypes function ensures that the following type combinations for the operands are invalid:

- If both operands are SIMD vectors, at least one of them must be implicitly convertible to the other type.
- If one operand is a scalar type, then an implicit conversion from the scalar type to the return type must be possible.
- Furthermore, a conversion from scalar type to vector type may not lead to a narrowing conversion from a floating point type. This essentially forbids float_v × double because the double operand would have to be converted to the narrower single-precision float type. On the other hand double_v × float does not require a narrowing conversion and therefore works.

The return type is determined via the CopyUnsigned<Test, T> alias template in line 9. The rules are as follows:

- The return type is the unsigned variant of T if T is integral and Test is an unsigned type.
- Otherwise the return type is T.

Thus, if one operand is an unsigned integer vector or scalar and the other operand is a signed integer vector or scalar, then the operands are converted to the corresponding unsigned integer vector. But, in contrast to the semantics of builtin integer types, no full integer promotion is applied, leaving sizeof(T) of the vector entries, and thus $\mathcal{W}_{\mathtt{T}}$ unchanged. It follows that int_v × unsigned int yields uint_v and short_v × unsigned int yields ushort_v. The latter implicit conversion from unsigned int to ushort_v is unfortunate, but since short_v + 1 should be valid code and return a short_v, it is more consistent to also convert unsigned int implicitly to short_v or ushort_v.

The non-member operators were explicitly designed to support operator calls with objects that have an implicit conversion operator to a SIMD vector type. This is possible by leaving W less constrained than V.

8.2 Optimizing vector × scalar operations

Note the shift operator overloads for an argument of type int on lines 31–32 in Listing 11. This touches a general issue that is not fully

```
Vector &operator*= (Vector<T> x);
Vector &operator/= (Vector<T> x);
Vector &operator%= (Vector<T> x);
Vector &operator+= (Vector<T> x);
Vector &operator-= (Vector<T> x);

Vector &operator&= (Vector<T> x);
Vector &operator|= (Vector<T> x);
Vector &operator^= (Vector<T> x);
Vector &operator<<=(Vector<T> x);
Vector &operator>>=(Vector<T> x);
Vector &operator<<=(int x);
Vector &operator>>=(int x);
```

Listing 15: Declaration of assignment operators.

solved with the Vc binary operators interface, yet: Some operations can be implemented more efficiently if the operator implementation knows that one operand is a scalar or even a literal. A scalar operand would be converted to a SIMD vector with equal values in all entries via the non-member binary operators (which Vc therefore does not define for the shift operators).

The issue is certainly solvable. One solution could be to not call `V::operator···(V)` from the non-member operators but a template function such as

```
template <typename V, typename L, typename R>
V execute_operator_add(L &&, R &&)
```

. This function can then be overloaded such that one overload implements Vector + Vector and the other overload implements Vector + Scalar.

9 COMPOUND ASSIGNMENT OPERATORS

Apart from simple assignment, C++ supports compound assignment operators that combine a binary operator and assignment to the variable of the left operand. The standard defines the behavior “of an expression of the form E1 op = E2 [...] equivalent to E1 = E1 op E2 except that E1 is evaluated only once.” [6, §5.17] Thus, the simplest implementation calls the binary operator and the assignment operator.

But compound assignment operators can, in principle, do more than the combination of binary operator and assignment operator. The additional constraint of the compound assignment operators, that the result of the binary operator needs to be converted back to the type of the left operand makes it possible to allow any scalar arithmetic type as operand. This may be best explained with an example. Consider

```
EntryReference operator[](size_t index);
EntryType operator[](size_t index) const;
```

Listing 16: Declaration of the subscript operators for scalar access.

```
short_v f(short_v x) {
  return x *= 0.3f;
}
```

with SSE where $\mathcal{W}_{\texttt{short}} = 2\mathcal{W}_{\texttt{float}}$. In this case `Vector<short>::operator*=(float)` can be implemented as two `float_v` multiplications with the low and high parts of `x`. For binary operators this is not implemented because the return type would have to be something like `std::array<float_v, 2>` or `std::tuple<float_v, float_v>`. But for compound assignment operators this return type problem does not exist because the two `float_v` objects will be converted back to a single `short_v`.

At this point only the more restrictive compound assignment operators are implemented in Vc.

10 SUBSCRIPT OPERATORS

Subscripting SIMD vector objects is a very important feature for adapting scalar code in vectorized codes. Subscripting makes mixing of scalar and vector code intuitively easy (though sometimes at a higher cost than intended).

But while subscripting is desirable from a users point of view, the C++ language standard makes it close to impossible. The issue is that the non-const subscript operator needs to return an lvalue reference to the scalar type (`EntryReference`) and assignment to this lvalue reference needs to modify the SIMD vector, which is of type `VectorType`. This requires aliasing two different types onto the same memory location, which is not possible with standard C++. Even a `union` does not solve the issue because aliasing via `union`s is only well-defined for layout-compatible types.

Therefore, the return type of the non-const subscript operator uses the `EntryReference` member type (see Listing 16), which is an *implementation-defined* type. Most compilers provide extensions to standard C++ to work around the aliasing restriction. One popular extension is explicit aliasing via unions. But there are also other ways of allowing explicit aliasing, such as GCC's `may_alias` attribute. If Vc is compiled with GCC then the return type of the non-const sub-

```
EntryType operator[](size_t index) const {
  EntryType mem[Size];
  store(mem);
  return mem[index];
}
```

Listing 17: The offset in a SIMD register via `operator[]` is defined by the memory order.

script operator will be `EntryType` with the `may_alias` attribute attached.

The const subscript operator intentionally returns a prvalue and not a const lvalue reference. With a reference, read-only access would require the compiler to also violate the strict aliasing rules. And it is a very rare use-case to store a reference to the return value from the const subscript operator and then modify the object through a non-const reference. If the user wants to have a (mutable or immutable) reference to an entry in the SIMD vector he can (and must) use the non-const subscript operator. Finally, as noted in Section 4.1, the EntryReference type is not necessarily the same as EntryType &. Therefore, in order to actually return EntryType, the generic definition of the const operator needs to return a prvalue.

The effect of the `const` subscript operator is shown in Listing 17. Most importantly, this defines the indexing order of the values in the SIMD vector: Access to the value at offset `i` via the subscript operator yields the same value a store and subsequent array access at offset `i` produces. Of course, the same indexing order must be used for non-const subscript operator.

11 GATHER & SCATTER

A gather operation in SIMD programming describes a vector load from a given base address and arbitrary (positive) offsets to this address. The scatter operation is the reverse as a vector store.

11.1 PRIOR ART

Portable and intuitive gather & scatter API in terms of function calls does not have an obvious solution. There is little prior art that is of much use.

IEEE and The Open Group [2] specify `readv` and `writev` in POSIX 1003.1 - 2004 (Listing 18). `readv` can be used to scatter from a file descriptor into a given number of arbitrarily sized buffers. `writev`

```
struct iovec {
  void *iov_base; // Pointer to data
  size_t iov_len; // Length of data
};
ssize_t readv (int fildes, const struct iovec *iov, int iovcnt);
ssize_t writev(int fildes, const struct iovec *iov, int iovcnt);
```

Listing 18: Scatter/gather functions in POSIX [2].

does the reverse of `readv`. Note that the functions are not type-safe (the `void*` erases the type information), which is fine for file descriptor I/O but not for a C++ API for SIMD vector types.

Much closer in functionality to the requirements of Vc's API are the SIMD gather/scatter intrinsic functions that Intel introduced with their compiler for the MIC architecture (Listing 19). In its simplest form the gather takes an `int` index vector, multiplies the values with the `scale` parameter (which may be 1, 2, 4, or 8) and uses these as Byte-offsets in the memory pointed to by `addr` to load 16 values. Thus `v = _mm512_i32gather_ps(index, addr, scale)` is equivalent to:

```
for (int i = 0; i < 16; ++i) {
  v[i] = *reinterpret_cast<const float *>(
           reinterpret_cast<const char *>(addr) + index[i] * scale);
}
```

In contrast to the POSIX functions, the memory regions that are read (buffers) are of fixed size (`sizeof(float)`). Instead of one pointer per memory location, here a single pointer with a fixed number of offsets is used. Thus, the `_mm512_i32gather_ps` intrinsic resembles a subscript operator applied to a pointer of `float`s (`v = addr[index]`). But note that the MIC intrinsics are not type-safe: they pass the pointer as `void *` and require the caller to determine the scaling factor.

The remaining gather functions provide additional features:

- Masked gathers allow to load fewer values from memory as determined by the corresponding bits in the mask. This allows `addr + index[i] * scale` to point to an invalid address for all `i` where `mask[i]` is `false`.
- The extended variants accept a parameter to do type conversions in addition to the load. For example, a `float` vector can be gathered from random `short`s in an array.
- The hint parameter is used to do cache optimizations, possibly marking the affected cache lines as LRU.

The scatter functions do the reverse of the gather functions. They store a (masked) set of scalar values from a vector register to memory locations determined by the `index` and `scale` parameters.

```
__m512 _mm512_i32gather_ps(__m512i index, void const *addr,
                           int scale);
__m512 _mm512_mask_i32gather_ps(__m512 v1_old, __mmask16 k1,
                                __m512i index, void const *addr,
                                int scale);
__m512 _mm512_i32extgather_ps(__m512i index, void const *mv,
                              _MM_UPCONV_PS_ENUM conv, int scale,
                              int hint);
__m512 _mm512_mask_i32extgather_ps(_m512 v1_old, __mmask16 k1,
                                   __m512i index, void const *mv,
                                   _MM_UPCONV_PS_ENUM conv, int scale,
                                   int hint);

void _mm512_i32scatter_ps(void *mv, __m512i index, __m512 v1,
                          int scale);
void _mm512_mask_i32scatter_ps(void *mv, __mmask16 k1, __m512i index,
                               __m512 v1, int scale);
void _mm512_i32extscatter_ps(void *mv, __m512i index, __m512 v1,
                             _MM_DOWNCONV_PS_ENUM conv, int scale,
                             int hint);
void _mm512_mask_i32extscatter_ps(void *mv, __mmask16 k1,
                                  __m512i index, __m512 v1,
                                  _MM_DOWNCONV_PS_ENUM conv,
                                  int scale, int hint);
```

Listing 19: The gather/scatter intrinsics for the Intel MIC architecture [4].

A third important interface is array notation, which is available as an extension to C/C++ with Cilk Plus [3]. With this extension the user can expression operations on whole arrays with very little code. For instance `A[:] = B[:] + 5` is equivalent to `for (i = 0; i < 10; i++) A[i] = B[i] + 5`. With this extension a gather is written as `C[:] = A[B[:]]` and a scatter accordingly as `A[B[:]] = C[:]`. The interface requires a single new concept (which is the central concept of array notation) to be type-safe, concise, and intuitively clear. This syntax also naturally expresses converting gather/scatters. Masking is possible as well by the generic masking support in the array notation syntax [5, §5.3.6], which extends the semantics of `if/else` statements.[7]

Finally, the standard library provides related functionality in the `std::valarray` and `std::mask_array` classes.

11.2 INITIAL VC INTERFACE

The initial approach to gather and scatter interfaces in the Vc SIMD vector classes was done via member functions, and for gathers additionally via constructors. A simple gather and scatter example, using this interface, is shown in Listing 20. A tricky issue were gather and

7 In Cilk Plus `if/else` statements are extended to accept arrays of booleans. Thus, both the `if` and `else` branches can be executed. I discuss this topic in depth in N4185.

```cpp
void maskedArrayGatherScatter(float *data, int_v indexes) {
  const auto mask = indexes > 0;
  float_v v(&data[0], indexes, mask);
  v.scatter(&data[1], indexes, mask);
}

struct S { float x, y, z; };

void structGatherScatter(S *data, int_v indexes) {
  float_v v(data, &S::x, indexes, indexes > 0);
  v.scatter(data, &S::y, indexes, indexes > 0);
  ...
}
```

Listing 20: Example usage of the first generation gather/scatter interface in Vc.

scatter operations on an array of a non-fundamental type, such as a struct, union, or array. In the case of an array of struct, the user needs to specify the member variable of the struct that he wants to access. A possible interface for this is shown at the bottom of the example in Listing 20.

This interface can certainly support all features of the underlying hardware, or emulate such features on SIMD hardware that does not have the respective instruction support. But the interface is hardly intuitive. The order of parameters does follow a logical pattern (outer array, possibly struct members, indexes, and optionally a mask as the last parameter), but even then I often looked up the interface in the API documentation. A more intuitive interface needs to relate closer to known C++ syntax, which is something the array notation in Cilk Plus nicely demonstrates.

Furthermore, the necessary overloads to support arbitrary nesting of structs and arrays quickly get out of hand. For every possible composition of structs, unions, and arrays two (unmasked and masked) gather and scatter function overloads are required. In some situations it may be necessary to supply more than one index vector: The first indexes subscript an outer array and the offsets in the inner array are not equal for all entries but require another index vector. The gather/scatter function approach simply does not scale in that respect.

11.3 Overloading the subscript operator

The problems discussed above do not exist for scalar types because the subscript operator and the member access operations support arbitrary access of members in such nested data structures. Thus, the question is whether the syntax that works for scalars can be made

```cpp
template <typename I>
inline auto operator[](I &&i)
    -> decltype(subscript_operator(*this, std::forward<I>(i))) {
  return subscript_operator(*this, std::forward<I>(i));
}

template <typename I>
inline auto operator[](I &&i)
    const -> decltype(subscript_operator(*this, std::forward<I>(i))) {
  return subscript_operator(*this, std::forward<I>(i));
}
```

Listing 21: Generic subscript member operator that forwards to a non-member function.

to work for SIMD vector types as well. It is possible to overload the subscript operator with one parameter of arbitrary type, and thus also with a parameter of SIMD vector type. But, it is not possible to overload the subscript operator of existing types, though, because C++ does not support non-member subscript operators. Thus, in order to implement gathers and scatters via the subscript operator, the array/container class needs to specifically have support for the SIMD types. On the other hand, adding a gather subscript operator directly to all container classes would make all of them depend on the declarations of the SIMD types. Luckily, there is a clean way around it that effectively creates opt-in non-member subscript operators.[8] A class simply needs the two subscript operators defined in Listing 21. Then, if the `subscript_operator` function is declared later (or before), the subscript operator can be used with the types the `subscript_operator` functions implement.

As long as the C++ standard library does not implement such a subscript operator, the container classes in the `std` namespace cannot support SIMD vector subscripting. Therefore, only a new type can implement these subscript operators. It is possible to adapt existing container classes with the `AdaptSubscriptOperator` class in Listing 35 and thus create a `Vc::vector` type that implements `std::vector` with the additional subscript operator.[9]

11.4 A subscript operator for SIMD gather and scatter

Once we have container classes that support subscripting with arbitrary, user-defined types, subscripting with Vc's vector types can be

8 For implementing the gather and scatter subscript operator overloads it would, of course, be better if non-member subscript operators were possible.

9 The exception is `std::array` and other container classes that need to be POD or aggregates. `Vc::array` therefore needs to be a verbatim copy of `std::array` plus the new subscript operator.

implemented (Listing 22). The requirements for the `subscript_operator` function are the following:

- It must accept only a specific subset of container classes, specifically those that use contiguous storage for its entries.
- The container may use storage on the free store. But nested containers are constrained: Only the outermost container may use the free store.
- The container and/or its entries may be arbitrarily cv-qualified.
- It must accept a specific set of types for the index parameter.
  - Any type that implements the subscript operator and contains as many entries as the gathered vector will contain / the vector to scatter contains.
  - SIMD vectors of integral type and `Size` equal for the value vector and index vector.

The `enable_if` statement allows to implement the function such that it only participates in overload resolution iff …

… the `has_subscript_operator` type trait finds a usable subscript operator in the `IndexVector` type.

… the `has_contiguous_storage` type trait determines that the `Container` type is implemented using contiguous storage for the entries in the container.[10]

… the `is_lvalue_reference` type trait determines that dereferencing the first iterator of the `Container` type returns a type that can be used to determine a pointer to the first element of the contiguous storage of the container.

Whether the `subscript_operator` function participates in overload resolution directly determines whether the generic forwarding member subscript operators in `Vc::array` and `Vc::vector` participate in overload resolution. This follows from the return type of these subscript operators, which lead to substitution failure (which is not an error) iff `subscript_operator` is not usable.

10 Such a trait cannot really be implemented for all I know. But it is possible to define a list of classes and class templates that will work as expected.

```
template <typename Container, typename IndexVector,
          typename = enable_if<
              Traits::has_subscript_operator<IndexVector>::value &&
              Traits::has_contiguous_storage<Container>::value &&
              std::is_lvalue_reference<decltype(
                  *begin(std::declval<Container>()))>::value>>
inline SubscriptOperation<
    typename std::remove_reference<
        decltype(*begin(std::declval<Container>()))>::type,
    typename std::remove_const<
        typename std::remove_reference<IndexVector>::type>::type>
subscript_operator(Container &&c, IndexVector &&indexes) {
  return {std::addressof(*begin(c)),
          std::forward<IndexVector>(indexes)};
}
```

Listing 22: Generic `subscript_operator` function that passes the context for a gather/scatter operation to the SubscriptOperation class.

11.5 A proxy type for gather/scatter

The `subscript_operator` function then returns an object of type `SubscriptOperation` that contains a pointer to the beginning of the container storage and a `const` reference to the index vector. A naïve approach would return `Vector<T>` directly, where `T` is determined by the type of the entries in the container. But then converting gathers, nested subscripting, as well as any scatter operation would not be possible. Returning a proxy object allows to implement further subscript operators and delayed gather and scatter invocations to determine the SIMD vector entry type from the assignment operator.

The `SubscriptOperation` (Listing 23) class needs three template parameters: Obviously the type of the memory pointer and the type of the index vector/array/list need to be parameters. The third template parameter is needed for efficient implementation of the subscript operators in `SubscriptOperation`. This will be explained below.

11.6 Simple gather/scatter operators

The proxy type implements the conversion operator for gathers (line 14) and the assignment operator for scatters (line 15). Both of these functions may only participate in overload resolution if the memory pointer type is an arithmetic type and the template parameter type `V` is a SIMD vector type. The gather and scatter operations are then, together with the template parameter type `V`, fully defined and thus support type conversion on load/store. The conversion is defined by the entry type of the SIMD vector type and the type of the memory pointer. At this point the number of entries in the SIMD vector is known and therefore the size of the index vector can be checked. If the size of the index vector is encoded in the type, then this will be used to additionally determine participation of the functions in overload resolution.[11]

Since scatter is implemented via an assignment operator in `SubscriptOperation` we need to consider whether compound assignment should be supported as well. In Vc I decided against doing so, because compound assignment implies that an implicit gather operation needs to be executed. Since gather/scatter are rather expensive operations I believe the user should see more clearly that the memory access patterns of the code are sub-optimal. Not allowing compound assignment forces the user to explicitly execute a gather, thus making the memory accesses more obvious.

The two operators implement unmasked gather and scatter. Masked gather scatter will be discussed in N4185. The functions in lines 17 and 18 provide an interface to extract the necessary information in a minimal interface.

11.7 THE NESTED SCALAR SUBSCRIPT OPERATOR

The `SubscriptOperation` class implements three subscript operators. The scalar subscript operator on line 24 allows to use gather/scatter for nested containers. The second operator on line 31 implements gather/scatter operations for different offsets in nested containers.[12] The third operator on line 37 enables gather/scatter to use arrays of structs.

11.7.1 RETURN TYPE

Semantically, the scalar subscript operator (line 24) implements the same behavior as scalar subscripting. Thus, the expression

```
data[int_v::IndexesFromZero()][3]
```

references the elements

```
data[0][3], data[1][3], data[2][3], ...
```

11 An alternative to removing a function from overload resolution are diagnostics via `static_assert` statements. (Compare footnote 6 on page 21.) If the suggestion from [N4186] is incorporated in the standard the best solution will be to use deleted functions with custom diagnostics.

12 Basically, only nested arrays will work because of the requirement that only the outer container may allocate the data on the heap.

```
template <typename T,
          typename IndexVector,
          typename Scale = std::ratio<1, 1>>
class SubscriptOperation {
  const IndexVector m_indexes;
  T *const m_address;

  using ScaledIndexes = implementation_defined;

public:
  constexpr SubscriptOperation(T *address,
                               const IndexVector &indexes);

  template <typename V> operator V() const;
  template <typename V> SubscriptOperation &operator=(const V &rhs);

  GatherArguments<T, ScaledIndexes> gatherArguments() const;
  ScatterArguments<T, ScaledIndexes> scatterArguments() const;

  SubscriptOperation<
      std::remove_reference_t<decltype(m_address[0][0])>, IndexVector,
      std::ratio_multiply<
          Scale, std::ratio<sizeof(T), sizeof(m_address[0][0])>>>
      operator[](std::size_t index);

  template <typename IT>
  SubscriptOperation<
      std::remove_reference_t<
          decltype(m_address[0][std::declval<const IT &>()[0]])>,
      ScaledIndexes>
      operator[](const IT &index);

  template <typename U>
  SubscriptOperation<
      std::remove_reference_t<U>, IndexVector,
      std::ratio_multiply<Scale, std::ratio<sizeof(T), sizeof(U)>>>
      operator[](U T::*member);
};
```

Listing 23: The SubscriptOperation proxy type that implements conversion to/from SIMD vector types via gather/scatter calls and additional subscript operators.

. Obviously, the operator must use different template arguments for the `SubscriptOperation` return type:

- The memory pointer previously pointed to an array of arrays. The new pointer must point to the beginning of the first array in the outer array. Thus, the type changes from array of `U` to `U`.
- The `IndexVector` type does not change at this point, also because its value is not modified.
- Because of the above, the index vector would now contain incorrect offsets. Consider the expression (as above)

  ```
  data[int_v::IndexesFromZero()][3]
  ```

  and assume `data` is of type

  ```
  Vc::array<Vc::array<float, 100>, 100>
  ```

  . Then `data[int_v::IndexesFromZero()]` returns an object of type

  ```
  SubscriptOperation<Vc::array<float, 100>, int_v, ratio<1, 1>>
  ```

  . The subsequent call to the nested scalar subscript operator (`operator[](std::size_t)`) determines the memory pointer type to be `float` and retains the index vector as `{0, 1, 2, …}`. Since `&data[1]` $-$ `&data[0]` $= \frac{\texttt{sizeof(Vc::array<float,100>)}}{\texttt{sizeof(float)}} = 100$, the correct offsets to the new `float`-pointer are `{0, 100, 200, …}`. The pointer difference expression (`&data[1] - &data[0]`) is not a constant expression, but the `sizeof` fraction obviously is. Therefore, the `std::ratio` template argument is scaled with these two `sizeof` values (line 22).

  By using a template parameter, this fraction is built up in subsequent subscript operator calls and the division is evaluated at compile time inside the cast and assignment operators or the `gatherArguments` and `scatterArguments` functions. Thus, the multiplication of the index vector is delayed as far as possible. This is not only an optimization. It is necessary to delay the division to implement the member subscript operator (Section 11.10) correctly.

11.7.2 Participation in overload resolution

The scalar subscript operator (line 24) may not be instantiated with the `SubscriptOperation<T, I, S>` class if `T` does not implement the subscript operator for arguments of `std::size_t`. This

```
template <typename U = T>
auto operator[](enable_if_t<(has_no_allocated_data<T>::value &&
                             has_subscript_operator<T>::value &&
                             is_same<T, U>::value),
                            size_t> index)
    -> SubscriptOperation<
        remove_reference_t<decltype(m_address[0][index])>,
        IndexVector,
        ratio_multiply<
            Scale, ratio<sizeof(T), sizeof(m_address[0][index])>>>;
```

Listing 24: The complete declaration of the nested scalar subscript operator as used in Vc.

can be implemented via a dummy template parameter (`U`) to the subscript operator and modifying the subscript expression in the `decltype` expression such that it becomes a dependent type on `U`, while at the same time requiring `U` to be equal to `T`. In addition to delayed instantiation, the operator shall not participate in overload resolution if `T` does not implement the subscript operator or if `T` is a container that does not store its data inside the object. This last requirement is important to make the offset calculation work as described above. See Listing 24 for a possible declaration.

11.7.3 NESTED CONTAINER REQUIREMENTS

A container class that can be dynamically resized typically stores its data in an array outside of the container object. The object itself typically has only two or three member variables: pointers to begin and end of the data, and possibly a pointer to the end of the allocated memory. A well known example is `std::vector`. Consider what `Vc::array<std::vector<int>, N>` implies for SIMD gather and scatter operations, which require a single pointer to memory and a list of offsets to that pointer. In general, there is no guarantee about the dynamically allocated memory, thus the pointers could possibly cover the whole range of addressable memory.[13] On a system with a pointer-size of 64 bits, the index vector would be required to use 64-bit integers or risk pointing to incorrect addresses. Especially for SIMD vector gathers and scatters this is a very limiting requirement. The gather and scatter instructions in the MIC instruction set and the AVX2 gather instructions only support 32-bit integer offsets. In addition, these instructions assume unsigned integers and thus only positive offsets. The base pointer would therefore have to be the numerically smallest one. Overall, these constraints and complications show that an interface to gather/scatter for these kinds of

13 or worse: different segments of memory

nested container types is not worth the effort. The user would be much better served with writing a loop that assigns the scalar values sequentially.[14]

11.8 Type of the index vector

As mentioned above (and more below), at some point the index vector values must be scaled with the ratio stored in the `std::ratio` template parameter. But when the multiplication is executed, the index vector type must be able to store these larger values. It would be easiest if the scaled index vector type were equal to the type used for the initial subscript. But the scale operation of the indexes is implicit and not at all obvious to the user, who did not study the implementation of nested subscripting. The user only sees the need for the type of subscript argument to be able to store the offsets for the outer subscript. Thus, a vector of 16-bit integers may appear to be sufficient.

Obviously, a 16-bit integer can quickly overflow with the scale operations involved for nested arrays.[15] Therefore, the index vector type needs to be promoted transparently before applying the scale operation. The safest type for promotion would be a `std::size_t`. But as discussed before, we would rather not use 64-bit offsets, since they cannot be used for gather/scatter instructions on at least one major platform. Thus, integral promotion to `int` or `unsigned int` is the most sensible solution.

The promoted index vector type is captured in the `ScaledIndexes` member type (line 8). The discussion showed that there is no definite answer on the type promotion. Since the type is only used internally, the implementation may choose the exact rules.

For Vc I chose the following logic for the `ScaledIndexes` member type:

- If `IndexVector` is `Vector<T>` or `SimdArray<T, N>` and `sizeof(T)` $\geq$ `sizeof(int)`, then `ScaledIndexes` is set to `IndexVector`.

14 Or even better: The API limitation uncovers the flaw in the data structure and leads to a redesign and better data structures.

15 Consider `Vc::array<Vc::array<float, 1000>, 1000> data`: The first subscript operator only works with values from 0–999, which easily fit into a 16-bit integer. But with the second subscript those indexes must be scaled by 1000, thus exceeding the representable range.

- If `IndexVector` is a `Vector<T>` or `SimdArray<T, N>` and `sizeof(T)` < `sizeof(int)`, then `ScaledIndexes` is set to `SimdArray<int, IndexVector::Size >`.

- If `IndexVector` is an array with known size (`std::array`, `Vc::array`, or fixed-size C-array), then `ScaledIndexes` is set to `SimdArray<promoted_type<T>, N>` (where `T` is the value type of the array).

- If `IndexVector` is an `initializer_list<T>`, then `ScaledIndexes` is set to `vector<promoted_type<T>>`.

- If `IndexVector` is a `vector<T>`, then `ScaledIndexes` is set to `vector<promoted_type<T>>`.

11.9 The nested vector subscript operator

The second subscript operator on line 31 also implements gather and scatter on nested arrays. But, in contrast to the above, it allows to use different offsets on all nesting levels. Thus, `data[i][j]` references the values `{data[i[0]][j[0]], data[i[1]][j[1]], …}`. The second subscript's offsets therefore need to be added to the scaled original offsets. This is why the return type of the subscript operator is `SubscriptOperation<U, ScaledIndexes, ratio<1, 1>>`.

As for the scalar subscript operator, the vector subscript operator may not be instantiated together with the `SubscriptOperation<T, I, S>` class unless `T` implements the scalar subscript operator. Since the subscript operator is declared as a template function and the subscript expression in the deduction of type `U` of the return-type `SubscriptOperation<U, ScaledIndexes>` depends on the template type, this requirement is already fulfilled. In addition, the operator may only participate in overload resolution iff …

… `T` implements the subscript operator.

… `T` is a container that stores its data inside the object. (compare Section 11.7.3)

… the operators function parameter type implements the subscript operator. This requirement fully disambiguates the function with the scalar subscript operator.

... the number of values in `IndexVector` and the function parameter type `IT` are equal or at least one of them cannot be determined as constant expression.

11.10 THE NESTED STRUCT MEMBER SUBSCRIPT OPERATOR

The third subscript operator on line 37 in Listing 23 enables gather and scatter for arrays of structs. The API defined here does not have a direct counterpart in scalar code. The reason for this is that the dot-operator is not overloadable in C++ (yet).

Consider a `Vc::vector<S>` of a simple struct for the following discussion: `struct S { float x, y, z; }`. Then, to access `S::x` of a given array element with offset `i`, the required scalar code is `data[i].x`. The data member access via `.x` is not overloadable in a generic proxy-class returned from `data[indexes]` (where `indexes` is a vector/array of indexes). Therefore, with C++14, the only way to implement a vectorized struct gather requires `data[indexes]` to return a struct of proxy objects that are named `x`, `y`, and `z`. It follows that such an implementation must know the members of the struct before template instantiation. Such a return type therefore is not generically implementable with C++14.

11.10.1 EMULATING MEMBER ACCESS

Member access is too important to postpone its use with Vc until some future C++ standard provides the capabilities, though. Therefore, consider an alternative that, at least when seen in code, is suggestive enough to a developer that (s)he can understand intuitively what it does and how it can be used in new code.

The solution I chose in Vc therefore had to rely on a different operator overload or member function. The subscript operator is the semantically closest relative to accessing member variables via the dot operator. But instead of an integral argument, the subscript operator needs to know the member variable offset in the struct, which can be expressed with pointers to members. Thus, `data[indexes][&S::x]` in Vc expresses the equivalent of `data[index].x` in scalar code.

11.10.2 PARTICIPATION IN OVERLOAD RESOLUTION

The nested struct member subscript operator may only participate in overload resolution if the template parameter `T` of the `SubscriptOperation` class template is a `class` or `union` type. Otherwise,

```cpp
template <typename U>
SubscriptOperation<
    std::remove_reference_t<U>, IndexVector,
    std::ratio_multiply<Scale, std::ratio<sizeof(S), sizeof(U)>>>
    operator.(U T::*member);
```

Listing 25: A possible `operator.()` overload that can capture the requirements of `SubscriptOperation`.

the pointer to member type in the function parameter list would be an ill-formed expression.

11.10.3 RETURN TYPE

The return type of `operator[](U S::*)` follows the same considerations as the return type for `operator[](std::size_t)` (compare Section 11.7.1). But, now the importance of using a fraction template parameter instead of immediate scaling or a single integer for the scaling parameter becomes clear. Consider a struct that contains an array: `struct S2 { float x[4]; float y; }`. The index scale factor for `&S2::x` thus would be $\frac{\texttt{sizeof(S2)}}{\texttt{sizeof(float[4])}} = \frac{20}{16}$, which is not an integral value. In order to access a scalar element the user must call another subscript operator for the `S2::x` array, which will result in the scaling fraction $\frac{\texttt{sizeof(float[4])}}{\texttt{sizeof(float)}} = \frac{16}{4}$. The final scaling that is applied in e.g. `SubscriptOperation::operator V()` thus becomes $\frac{20 \cdot 16}{16 \cdot 4} = 5$.

11.10.4 DIGRESSION: IMPLICATIONS FOR OPERATOR.()

The `SubscriptOperation` class is a use case for an overloadable `operator.()` that needs to do more than just forward to an object returned by the operator (i.e. the current `operator->()` cannot be used either). A possible `operator.()` declaration for `SubscriptOperation` could look like the one in Listing 25. For a container of type `Vc::vector<S>`, the code `data[indexes].x` would thus call `data[indexes].operator.(&S::x)` The type `S` could be deduced from the parameter type of the `operator.()` declaration.

12 SELECTING THE DEFAULT SIMD VECTOR TYPE

In Section 4, I showed that the `Vector<T>` class is defined inside a *target-dependent* namespace. Thus, the class (and its type aliases) need to be imported into a public namespace. The `Vc` namespace therefore must import all vector types from one of the target-dependent namespaces with `using` declarations (Listing 26). The default

```
namespace Vc {
using target_dependent::Vector;
using target_dependent:: float_v;
using target_dependent::double_v;
using target_dependent::   int_v;
// ...
}
```

Listing 26: Declaration of the default SIMD vector type(s).

choice is very important to the idea of a portable SIMD vector type because it allows using a different target-dependent implementation of `Vector<T>` with just a recompilation (using a different compiler or compiler flags).

12.1 THE SCALAR `VECTOR<T>` IMPLEMENTATION

In addition to the vector types in the `Vc` namespace, the `Vc::Scalar` namespace is always defined. The `Vc::Scalar::Vector<T>` class is implemented with `T` as VectorType member type, thus storing a single scalar value. But in contrast to the fundamental type `T`, `Vc::Scalar::Vector<T>` implements the complete SIMD types interface and thus is always available as a drop-in replacement for `Vc::Vector<T>`.

The scalar implementation is especially useful for

- making debugging a vectorized algorithm easier.
- testing that a given code works with a different vector width.
- targets without SIMD registers/instructions.
- implementing generic algorithms that need to be able to process chunks of data that are smaller than the native SIMD width (compare Appendix A).

12.2 SEVERAL SIMD GENERATIONS IN ONE TRANSLATION UNIT

For some target architectures it is possible to support more than one SIMD register width. This can be supported by using documented namespace names for the different SIMD targets. Then a user that knows that he targets this architecture can explicitly use SIMD registers and operations that are not declared as the default type.

As an example consider a x86 target with AVX instruction support. In addition to 256-bit SIMD registers the target supports also supports 128-bit SIMD registers (SSE). Thus the types `Vc::AVX::float_v` and `Vc::SSE::float_v` as well as `Vc::Scalar::float_v` are available to the user. (The type `Vc::float_v` would then be an alias for `Vc::AVX::float_v`.)

For a problem where only 4 entries in a single-precision float SIMD vector are needed `Vc::SSE::float_v` will perform better than `Vc::AVX::float_v` because it requires half of the memory bandwidth and because not all instructions on the CPU work equally fast for AVX vectors as for SSE vectors. As we will see later (Document on SIMD Arrays—still to be written—), the different vector types can be abstracted into a higher level type, alleviating the need for `#ifdefs` checking for a specific target architecture.

Experience has shown that it is useful to forward declare all user-visible types from target-specific namespaces even in case they are incompatible with the target system of the current compilation unit. This makes it easier for users to write target-specific code without using the preprocessor, only relying on template instantiation rules.

13 INTERFACE TO INTERNAL DATA

The `Vc::Vector` class does not implement all operations that a user might want to use. Most importantly, there exist specialized instructions for specific application domains that are not easy to capture in a generic interface. Thus, if the Vc types should be usable also for such special application domains it must be possible to access internal and implementation-dependent data.

The `Vc::Vector` class therefore defines the `data()` member function and a constructor that converts from a VectorType object:

```
  Vector(VectorType);
  VectorType &data();
  const VectorType &data() const;
```

A user might want to use the SSE intrinsic `_mm_adds_epi16`[16] with `Vc::SSE::short_v` and can thus write his own abstraction:

```
Vc::SSE::short_v add_saturating(Vc::SSE::short_v a,
                               Vc::SSE::short_v b) {
  return _mm_adds_epi16(a.data(), b.data());
}
```

The choice of the `data()` function is rather arbitrary. Alternatively, the VectorType could also be returned via a conversion oper-

16 This intrinsic adds 16-bit integers with signed saturation, thus returning `SHRT_MIN`/`SHRT_MAX` if the addition would underflow/overflow. This functionality may certainly be abstracted for the `Vc::Vector` types, but currently this is not the case.

```
namespace SSE {
  template <typename T> class Vector;
}
namespace AVX {
  template <typename T> class Vector;
}
namespace ...
#if DEFAULT_IS_SSE
using SSE::Vector;
#elif DEFAULT_IS_AVX
using AVX::Vector;
#elif
...
#endif
template <typename T, std::size_t N> class SimdArray;
```

Listing 27: The `Vector<T>` Design

ator (`operator VectorType &()`) or a non-member friend function such as (`VectorType &internal_data(Vector &)`). The conversion operator seems very convenient, and possibly too convenient as the type can implicitly convert to VectorType then. But if the conversion operator is declared as `explicit` then it is more convenient to use the `data()` member function.

I believe it is useful to have such a function in a standardized SIMD vector interface. But at this point I am looking for feedback from the community. In Vc the function exists, but has never been documented as a public function.

14 DESIGN ALTERNATIVES

There is no obvious natural choice for the class that provides the SIMD types. The choices I considered are:

1. Declare multiple `Vector<T>` class templates in separate namespaces for each possible SIMD target (Listing 27). This design has been discussed in Section 4 in detail and is the class design used in the Vc library.

2. Declare a single `Vector<T, SimdInterface>` class template (Listing 28).

3. Declare a single `Vector<T, Width>` class template (Listing 29).

At first glance the choices appear equivalent. It is possible to additionally provide the other two interfaces with any of the above choices. But there are important differences: not every interface adaption can be done transparently. `Vector<T, SimdInterface>`

```
namespace Common {
  enum class SimdInterface {
    ...
  };
  template <typename T, SimdInterface I> class Vector;
}
template <typename T>
using Vector = Common::Vector<T,
#if DEFAULT_IS_SSE
                              Common::SimdInterface::SSE
#elif DEFAULT_IS_AVX
                              Common::SimdInterface::AVX
#elif
                              ...
#endif
                              >;
template <typename T, std::size_t N> class SimdArray;
```

Listing 28: The `Vector<T, SimdInterface>` Design

```
template <typename T> constexpr std::size_t defaultWidth();
template <typename T, std::size_t N = defaultWidth<T>()> class Vector;
```

Listing 29: The `Vector<T, Width>` Design

can be declared as an alias template for `SimdNamespace::Vector<T>`. In this case the types are equal.

The `Vector<T>` types in the different namespaces in design 1 are different types. Therefore they cannot be unified in an equal type `Vector<T, SimdInterface>`. This means, if a `Vector<T, SimdInterface>` type is desired, it must be a new type and does not match objects of type `Vector<T>` (though providing implicit conversion is easy). The implication is that template argument deduction may be less convenient to use (see below).

The `Vector<T, Width>` type can easily be aliased to `Vector<T>` in different namespaces. Equally, it can be aliased to or from `Vector<T, SimdInterface>` (see Listing 30). Still, `Vector<T, Width>` and `Vector<T, SimdInterface>` are not equivalent: Consider the type `Vector<float, 8>`. It is the same type irrespective of the target system supporting vectors of $\mathcal{W}_{\texttt{float}} = 8$ or not. Thus, a symbol (such as the function `void f(Vector<float, 8>)` which is compiled for SSE and AVX would link, but crash or simply fail to work as expected at runtime. (This is due to `Vector<float, 8>`

```
template <typename T, std::size_t Width>
using Vector2 = Vector<T, simdInterfaceForWidth<T, Width>()>;

template <typename T, SimdInterface I>
using Vector2 = Vector<T, widthForSimdInterface<T, I>()>;
```

Listing 30: Possible aliasing of `Vector<T, Width>` to or from `Vector<T, SimdInterface>`

```
template <typename T> void f(Vector<T> v);
void g() {
  f(float_v());          // compiles
  f(double_v());         // compiles
  f(AVX::double_v());    // only compiles if AVX is the
                         // compile-time default anyway
}

// with Concepts-Lite:
template <typename T> requires IsSimdVector<T>() void f2(T v);
// alternative:
template <typename T> enable_if_t<is_simd_vector<T>::value> f2(T v);

void g2() {
  f2(SSE::float_v());
  f2(Scalar::double_v());
  f2(AVX::double_v());
}
```

Listing 31: Generic Function with Vector Parameter for design 1

using two `__m128` member objects for SSE and for AVX using a single `__m256` member.) The type `Vector<T, Width>` therefore does not ensure type safety and either needs a third template parameter to identify the SIMD architecture, or needs to be placed into a namespace as was done for `Vector<T>`. Therefore, the `Vector<T, Width>` type is a bad choice for the fundamental SIMD vector type. It is still a useful interface, though. I will discuss the details of a `Vector<T, Width>` class template in [Document on SIMD Arrays—still to be written—].

In design 2, the fundamental SIMD vector class is more generic than in design 1. This is relevant for generic functions that are supposed to work for any possible SIMD vector type, independent of the current compile-time default. On the other hand, most code should only use the compile-time default SIMD vector type, therefore alleviating the issue. With design 1, a generic function that wants to accept any vector type, including from different namespaces, needs to use an unconstrained template parameter as function parameter type. In order to ensure that the generic function only works with a SIMD vector type parameter, `std::enable_if` can be used (see Listing 31). It is expected that a future C++ revision will introduce *concepts*, a solution that allows expressing requirements for template parameters [N3819]. Listing 32 shows that such a generic function can be expressed more naturally with the current C++ standard with design 2. (If, on the other hand, a fully generic function wants to support both fundamental scalar types and SIMD vector types, `std::enable_if` or a concept are needed anyway.)

As a minor downside, design 2 can make compilation errors more verbose, because of the `SimdInterface` template parameter. The

```cpp
template <typename T, SimdInterface I> void f(Vector<T, I> v);
void g() {
  f(SSE::float_v());
  f(Scalar::double_v());
  f(AVX::double_v());
}
```

Listing 32: Generic Function with Vector Parameter for design 2 (design 3 is equivalent)

`SimdInterface` parameter could also be implemented as a tag type, in which case the additional template argument in the diagnostic output will be more readable than the numerical value of the enumeration.

Concluding, designs 1 and 2 appear like equally good design choices. I am looking for feedback from the community on this issue.

15 CONCLUSION

I have presented a SIMD vector class template and some of the operations for this type. The interface is declared in a way that it facilitates portable SIMD code. The set of presented operations is not complete but at this point a useful start.

I am looking for guidance how far a first revision of possible wording should go. I.e. what features are a must-have and which ones should be considered lower priority.

```
std::vector<int> data = ...;
for_each(par_vec, data.begin(), data.end(), [](auto &x) {
  x = x * x + 3;
});
```

Listing 33: Example use of the `for_each` algorithm with the `std::vec` policy.

A VECTORIZED STL ALGORITHMS

N4071 "describes requirements for implementations of an interface that computer programs written in the C++ programming language may use to invoke algorithms with parallel execution". As such, it deals with parallelization in terms of multi-threading and SIMD. The specification of the `std::par_vec` policy requires the compiler to treat code called from such algorithms special in such a way that "multiple function object invocations may be interleaved on a single thread". Therefore the library does not actually vectorize the code, it only annotates code so that the compiler might do it.

There is an alternative approach using vector types, which works nicely with generic lambdas and is a useful abstraction to hide load and store functions of the vector types. Consider the example use of `for_each` in Listing 33. The implementation of `for_each` can call the lambda with any type that implements the required operators. Therefore, we can use the `int_v` type and a scalar type. The scalar type could be `Vc::Scalar::int_v` to always have the full Vc interface available when the lambda is called. The need for using both types arises from the possibility that `data.size()` is not a multiple of $\mathcal{W}_{\texttt{int}}$. In that case there will be a number of entries in `data` that cannot be loaded or stored as a full SIMD vector without risking an out-of-bounds memory access. Additionally an implementation may choose to do a prologue that processes initial elements as scalars if they are not aligned on the natural alignment of SIMD objects.

With this solution for implementing the vectorization policy of STL algorithms the restrictions on lock usage and throwing exceptions become unnecessary.

Listing 34 shows a possible implementation of a vectorized `for_each`. The implementation can be generalized further to support containers that do not store their values in contiguous memory. In the same manner support for the restrictive `InputIterator` class of iterators can be implemented. Obviously, memory access would not use efficient vector loads and stores anymore.

Note that vectorization of composite types becomes possible with the `simdize<T>` work which I will describe in a future document.

```cpp
template <typename It, typename UnaryFunction>
inline enable_if<
    is_arithmetic<typename It::value_type>::value &&
        is_functor_argument_immutable<
            UnaryFunction, Vector<typename It::value_type>>::value,
    UnaryFunction>
simd_for_each(It first, It last, UnaryFunction f) {
  typedef Vector<typename It::value_type> V;
  typedef Scalar::Vector<typename It::value_type> V1;
  for (; reinterpret_cast<uintptr_t>(addressof(*first)) &
                 (V::MemoryAlignment - 1) &&
             first != last;
       ++first) {
    f(V1(addressof(*first), Vc::Aligned));
  }
  const auto lastV = last - (V::Size + 1);
  for (; first < lastV; first += V::Size) {
    f(V(addressof(*first), Vc::Aligned));
  }
  for (; first != last; ++first) {
    f(V1(addressof(*first), Vc::Aligned));
  }
  return move(f);
}

template <typename It, typename UnaryFunction>
inline enable_if<
    is_arithmetic<typename It::value_type>::value &&
        !is_functor_argument_immutable<
            UnaryFunction, Vector<typename It::value_type>>::value,
    UnaryFunction>
simd_for_each(It first, It last, UnaryFunction f) {
  typedef Vector<typename It::value_type> V;
  typedef Scalar::Vector<typename It::value_type> V1;
  for (; reinterpret_cast<uintptr_t>(addressof(*first)) &
                 (V::MemoryAlignment - 1) &&
             first != last;
       ++first) {
    V1 tmp(addressof(*first), Vc::Aligned);
    f(tmp);
    tmp.store(addressof(*first), Vc::Aligned);
  }
  const auto lastV = last - (V::Size + 1);
  for (; first < lastV; first += V::Size) {
    V tmp(addressof(*first), Vc::Aligned);
    f(tmp);
    tmp.store(addressof(*first), Vc::Aligned);
  }
  for (; first != last; ++first) {
    V1 tmp(addressof(*first), Vc::Aligned);
    f(tmp);
    tmp.store(addressof(*first), Vc::Aligned);
  }
  return move(f);
}

template <typename It, typename UnaryFunction>
inline enable_if<!is_arithmetic<typename It::value_type>::value,
                 UnaryFunction>
simd_for_each(It first, It last, UnaryFunction f) {
  return for_each(first, last, move(f));
}
```

Listing 34: A possible implementation of a vectorized `std::for_each`.

```cpp
template <typename Base> class AdaptSubscriptOperator : public Base {
public:
  using Base::Base;

  // explicitly enable Base::operator[] because the following would
  // hide it
  using Base::operator[];

  // forward to non-member subscript_operator function
  template <
      typename I,
      typename = typename std::enable_if<
          !std::is_arithmetic<typename std::decay<I>::type>::value>::
          type  // arithmetic types should always use Base::operator[]
                // and never match this one
      >
  auto operator[](I &&arg)
      -> decltype(subscript_operator(*this, std::forward<I>(arg))) {
    return subscript_operator(*this, std::forward<I>(arg));
  }

  // const overload of the above
  template <typename I,
            typename = typename std::enable_if<
                !std::is_arithmetic<
                     typename std::decay<I>::type>::value>::type>
  auto operator[](I &&arg) const
      -> decltype(subscript_operator(*this, std::forward<I>(arg))) {
    return subscript_operator(*this, std::forward<I>(arg));
  }
};
```

Listing 35: Generic adaptor class to add the forwarding subscript operator to existing container classes.

B THE ADAPTSUBSCRIPTOPERATOR CLASS

Listing 35 shows an adaptor class template to easily add non-member subscript functionality to an existing class.

C 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.

D REFERENCES

[1] Jared Hoberock. N4071: Working Draft, Technical Specification for C++ Extensions for Parallelism. ISO/IEC C++ Standards Committee Paper, 2014. URL `http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2014/n4071.htm`.

[2] IEEE and The Open Group. The Open Group Base Specifications, Issue 6, IEEE Std 1003.1, 2004. URL `http://pubs.opengroup.org/onlinepubs/009695399/`.

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

[4] Intel Corporation. *Intel® Xeon Phi™ Coprocessor Instruction Set Architecture Reference Manual*. Intel Corporation, 2012.

[5] Intel Corporation. Intel® Cilk™ Plus Language Extension Specification, September 2013. URL `https://www.cilkplus.org/sites/default/files/open_specifications/Intel_Cilk_plus_lang_spec_1.2.htm`.

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

[7] 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`.

[8] Matthias Kretz and Jens Maurer. N4186: Supporting Custom Diagnostics and SFINAE. ISO/IEC C++ Standards Committee Paper, 2014. URL `http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2014/n4186.pdf`.

[9] Andrew Sutton, Bjarne Stroustrup, and Gabriel Dos Reis. N3819: Concepts Lite Specification. ISO/IEC C++ Standards Committee Paper, 2013. URL `http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2013/n3819.pdf`.