Document Number: P0851R0
Date: 2017-11-06
Reply-to: Matthias Kretz <m.kretz@gsi.de>
Audience: LEWG / SG1

SIMD<T> IS NEITHER A PRODUCT TYPE NOR A CONTAINER TYPE

ABSTRACT

This paper provides context to the question on relational operators raised in P0820R0.

CONTENTS

1 Introduction 1
2 Design choices 2
3 Type category of simd<T> 3
4 relational operators 3
5 Some compromise? 6
6 Revisit the name? 6
A Bibliography 6

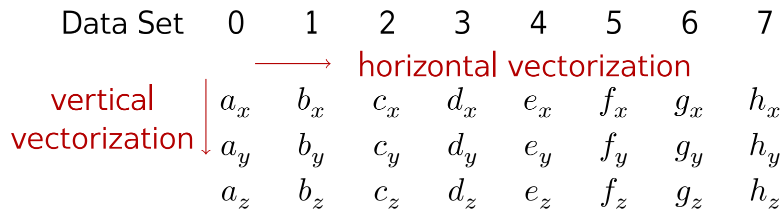

Figure 1: horizontal vs. vertical vectorization

1 INTRODUCTION

Shen [P0820R0] presents the view of the `simd<T>` specification of Kretz [P0214R5] in light of an existing library implementation for a similar problem. It is motivated by a different way of using `simd` types, which is why I'd like to start with an short detour into vectorization directions.

1.1 VECTORIZATION DIRECTION

HORIZONTAL same data member from several objects

VERTICAL different data members from one object [1]

As a simple example, consider a scalar product in a 3-dimensional space (cf. Figure 1). Vertical vectorization takes as input two 3-dimensional vectors and produces a single value. E.g. it places the 3 components of one input into one SIMD register and uses the `DPPS` instruction (on x86) to calculate the dot product. Horizontal vectorizing takes as input $\mathcal{W}_\mathrm{T}$ 3-dimensional vectors and produces $\mathcal{W}_\mathrm{T}$ values. It uses the same sequence of multiplications and additions, a classical scalar implementation would use, only applying it to a full SIMD register in parallel.

1.2 CHOOSING DIRECTION

In my experience, vertical vectorization is the traditional approach to using short vector extensions, such as SSE on x86. However, horizontal vectorization is where the true strength of data-parallel types lies. Horizontal vectorization reduces the need for reductions (e.g. shuffles) and thus can often yield the full speed-up of the SIMD vector width. Consequently, the code is easily scalable between targets with differing $\mathcal{W}_\mathrm{T}$.

The major reason for using vertical vectorization is that it can be used in small, contained areas of the code, typically hidden inside a single function. It is thus not

1 compare with “horizontal marekts” vs. “vertical markets”

apparent from the function signature, that the function uses a SIMD implementation. This allows localized changes and is much easier to take up in an existing code base.

Horizontal vectorization typically requires a much larger effort on an existing code base. Because a function, such as the horizontally vectorized scalar product above, requires the number of inputs and outputs to scale by $\mathcal{W}_{\mathtt{T}}$. Thus, every function signature changes, and vector types become a major part of the API. Data structure vectorization and AoVS (array of vectorized struct) become important tools.

2 DESIGN CHOICES

1. The main approach to vectorization is horizontal vectorization.
   Rationale: Because horizontal vectorization scales better, and thus produces better portability and even performance portability.
   Note: This does not preclude vertical vectorization. This is only about tailoring the API on one approach, while keeping the other approach possible.

2. Code that compiles (correctly) on one system compiles (correctly) on all others.
   Rationale: Because that's what everyone may expect of standard C++.
   Note: This is not always trivial, considering that `simd::size()` differs at compile-time. Some issues are not preventable, though.

3. The main vector type (`simd<T>`) is, as much as reasonably possible, a drop-in replacement for `T`. This means vectorization of a function may be as simple as a replacement of `float` with `simd<float>`.
   Rationale: Make it easy to use and understand by building on top of the fundamental types of the language. Generic code that works with built-in arithmetic types shall be reusable.

4. If the "drop-in replacement" cannot work without more input from the developer, the code shall not compile. The error message should be as helpful as possible, though; pointing out the missing "input".
   Rationale: It is dangerous to make assumptions when the intent is not clearly stated in the code (even if the user knows about them): this leads to hard to find bugs.
   Note: The main issue here is relational operators and branching.

5. Design the API from problems the user has to solve; and to make typical patterns easier to express. Or the inverse: do not simply expose every single special purpose instruction a hardware vendor came up with. Find the generality first.

Rationale: The user shall state intent; the library and compiler shall find the best instructions. This keeps user code readable and maintainable.

3 TYPE CATEGORY OF SIMD<T>

In private discussion with Tim Shen and Titus Winters the question came up whether `simd<T>` is a container or a product type. I strongly believe it is neither of those. The only precedence in the IS for the type category of `simd<T>` is `std::valarray`.

3.1 CONTAINER TYPE

Compare `simd<T>` and `std::array<T>`: Both types store a fixed number of values of type `T`. However, `simd<T>` is not a good choice for subscript access. The return type of `simd::operator[]` alone should make clear that `simd<T>` is not a container. While it does contain *values* of `T`, it does not necessarily contain *objects* of type `T`. Which is why the non-const subscript operator returns a smart reference, rather than an lvalue reference. For the same reason `simd<T>` does not even support iterators (at this point). Iterators could never return the required lvalue reference and thus would have to stick to being `InputIterators`.

3.2 PRODUCT TYPE

Is `simd<T>` a product type then? Or, in other words, has an object of type `simd<T>` *one value* as a whole. (After all that's where the term "product" in product type stems from.) The answer to this question differs for users that do vertical vs. horizontal vectorization. Consider the scalar product example again: In the vertical vectorization one `simd<T>` object contains a 3-dim euclidean vector and thus the whole `simd<T>` object has "one value". However, in the case of horizontal vectorization, one `simd<T>` object contains $\mathcal{W}_T$ values of the 1st, 2nd, or 3rd component of $\mathcal{W}_T$ euclidean vectors. A single `simd<T>` object has no sensible value as a whole. It really just stores $\mathcal{W}_T$ values.

So is it a product type? According to the design choices, if horizontal and vertical vectorization disagree, horizontal vectorization is preferred. Thus, `simd<T>` is not a product type.

4 RELATIONAL OPERATORS

The type category has important consequences for the definition of relational operators. If one considers `simd<T>` a product type, it is reasonable to expect `operator==`

to return `bool`. However, when vectorizing horizontally, the only reasonable result for an equality test is to return one boolean answer per element of the SIMD vector. Therefore, `simd<T>` in [P0214R6] returns `simd_mask<T>`.

4.1 EXAMPLE STD::COMPLEX

Consider `std::complex<simd<float>>`. While officially unspecified, at least libstdc++ has a useful implementation that mostly works. Example: `https://godbolt.org/g/fHJemB`. However, operations such as `abs`, require `operator==` in its implementation. As of the current spec, this does not compile (`https://godbolt.org/g/QiJcZn`). If we were to change `operator==` to return `bool`, the code would compile, but not do the right thing. According to design choice 4, having `operator==` return `bool` is a no-go.

4.2 WILL EXTENDING IF TO ACCEPT MASKS HELP?

Lets ignore that we'd ask for a rather narrow language extension, which has low chances of passing through EWG. Is an extension allowing `if (simd_obj0 == simd_obj1) foo();` a solution/compromise?

Cilk Plus has implemented write-masking via extending `if` statements for the array notation extension (cf. *Tutorial: Array Notation | Cilk Plus* [1]). Conditional statements thus do not disable a branch unless all entries of the mask are `false` (though essentially this is an optional optimization). Instead, all code branches are executed with an implicit state that determines the disabled vector lanes. Consider the example code in Listing 1 on a system with $\mathcal{W}_{\texttt{int}} = 4$ and `a = {1, 2, 3, 4}, b = {7, 0, 7, 7}`: The expression `a < b` returns a mask with 4 boolean values: `{true, false, true, true}`. The compiler therefore has to translate the `if`-branch (line 3) into instructions that modify `a` only at the indexes 0, 2, and 3. Subsequently, `a` will be `a = {2, 2, 4, 5}`. The `else`-branch (line 5) then may only modify the SIMD vector entry at index 1. Thus, `a` must become `a = {2, 1, 4, 5}`, which is the return value of the function `f`.

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

```cpp
simd<int> f(simd<int> a, simd<int> b) {
  if (a < b) {
    a += 1;
  } else {
    a -= 1;
  }
  return a;
}
```

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

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

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

the branches. Throwing exceptions and locking/unlocking mutexes would even have to be disallowed altogether.

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

This shows that `if` statements with non-boolean arguments limit the language features allowed in the `if`/`else` branches. This makes the feature much less intuitive.

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

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

The implicit mask context changes the semantics significantly in different regions of the source code. And the problem is aggravated if a developer requires `else if` or `switch` statements.

I therefore strongly recommend not to extend `if`-statements for masking.

5 SOME COMPROMISE?

If the committee feels uneasy to overload relops for `simd` (returning `simd_mask`), the TS could experiment with an opt-in mechanism: Define the relational operators in a namespace (e.g. `std::experimental::simd_relops`, or `std::experimental::simd_mask_relops` and `std::experimental::simd_bool_relops`). The user thus has to select the preferred behavior. My recommendation then would be to drop this opt-in behavior after TS feedback on this issue was collected.

6 REVISIT THE NAME?

I still believe the name "simd" is potentially misleading, since short vector SIMD instructions have traditionally (only) been used for special purpose optimziations and vertical vectorization. I don't recommend to reopen the discussion, though.

A BIBLIOGRAPHY

[P0214R5] Matthias Kretz. *P0214R5: Data-Parallel Vector Types & Operations*. ISO/IEC C++ Standards Committee Paper. 2017. URL: http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2017/p0214r5.pdf.

[P0214R6] Matthias Kretz. *P0214R6: Data-Parallel Vector Types & Operations*. ISO/IEC C++ Standards Committee Paper. 2017. URL: http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2017/p0214r6.pdf.

[P0820R0] Tim Shen. *P0820R0: Feedback on P0214R5*. ISO/IEC C++ Standards Committee Paper. 2017. URL: https://wg21.link/p0214r0.

[1] *Tutorial: Array Notation | Cilk Plus*. Intel Corporation. URL: https://www.cilkplus.org/tutorial-array-notation (visited on 01/11/2014).