Document Number: P1915R0
Date: 2019-10-07
Reply-to: Matthias Kretz <m.kretz@gsi.de>
Audience: SG1 / LEWG

# Expected Feedback from `simd` in the Parallelism TS 2

## ABSTRACT

This paper collects the questions (and some comments) we hope to answer via the TS process.

## CONTENTS

1 Introduction 1
2 Questions 1
A Bibliography 7

# 1 INTRODUCTION

ISO/IEC 19570:2018 [1] introduced data-parallel types to the standard library. Through the TS process we hope to get feedback on a number of issues that the committee was not comfortable to decide on before hands-on experience with the current choices. This paper collects the questions and design choices that SG1 / LEWG might want to revisit before moving `simd<T>` into the IS.

# 2 QUESTIONS

For every question, there is a link to a GitHub issue which can be used by anyone to give feedback.

## 2.1 name and ub of `popcount`, `find_first_set`, `find_last_set`

https://github.com/mattkretz/std-simd-feedback/issues/1

These need to be consistent and unambiguous with the corresponding functions introduced for builtin types (`countl_zero`, `countl_one`, `countr_zero`, `countr_one`, `popcount`). Note that there are two variants of these functions that are relevant to `simd<T>`:

1. Counting the number of entries in a `simd_mask<T>`. This returns a scalar integer or a `std::bitset`.
2. Element-wise counting the number of 1 bits in a `simd<T>`. This returns a `simd<int>`.

The former is present in ISO/IEC 19570:2018 [1]. The latter should be added, now that the [bit.count] functions are present in C++20.

The `count[lr]_(zero|one)` functions in C++20 are slightly different from `find_(first|last)_set`. The latter return in what place the first `true` value in the mask appears. However, if there is none, the behavior is undefined. The corresponding `count[lr]_zero` functions count the number of consecutive `false` values starting from the left or right and thus do not invoke undefined behavior.

## 2.2 names `copy_from`, `copy_to`

https://github.com/mattkretz/std-simd-feedback/issues/2

The names were discussed in SG1 and LEWG. If there is no new information via experience, WG21 should not spend any further time on renaming these functions. Consequently, this is a call for feedback from usage experience. We need arguments why a certain name works or dœs not work (clarity, ambiguity); no calls for preference. I.e. a code snippet where the `copy_to`/`copy_from` names are an issue would be new information. That I, who is used to the load & store terms, regularly start typing `.load` or `.store` is irrelevant.

## 2.3 name of `where` function

https://github.com/mattkretz/std-simd-feedback/issues/3

The name was discussed in LEWG. If there is no new information via experience, WG21 should not spend any further time on renaming this function. Consequently, this is a call for feedback from usage experience. We need arguments why a certain name works or dœs not work (clarity, ambiguity); no calls for preference.

Note that if Kretz [P0917R2] gets adopted, the `where` function could possibly removed in favor of conditional expressions for blending.

## 2.4 name of `simd<T>` misleading?

https://github.com/mattkretz/std-simd-feedback/issues/4

The name was discussed in SG1 and LEWG. SG1 chose `datapar` over `simd` and other options. LEWG overruled that choice and renamed it to `simd`.

We should use the TS to discover the following:

- Is there interest in implementations that use hardware parallelism that dœs not map (exclusively) to SIMD instructions & registers?

  I believe `fixed_size` itself is already a manifestation of such a feature. A single `fixed_size` object may map to multiple registers and thus operations on the object do not actually execute as a single instruction on multiple data.

- Do users choose to not use `simd`, because it appears too narrowly focused (from its name) on (certain) CPUs? Would such users choose differently if the name would focus on the parallelism it allows instead of a specific hardware implementation of such parallelism?

## 2.5 fixed_size vs. deduce_t

https://github.com/mattkretz/std-simd-feedback/issues/5

Whenever the interface requires a `simd` type that has a specific number of elements (typically deduced from one or more native types), we have a choice:

- Use `simd_abi::fixed_size<N>`. This is easy to memorize and understand. However, `simd_abi::fixed_size<N>` may be less efficient if such objects have to pass a function call boundary. Also, absent more implicit conversions, users that prefer native types will have to use an explicit cast.
- Use `simd_abi::deduce_t<T, N, Abis...>`. It is less obvious what the exact type will be (implementation-defined or `simd_abi::fixed_size<N>`) and the choice will be target-dependent.[1] This makes it slightly harder to write portable code (typically the logic is portable, it just needs an additional cast for some targets). The portability issue could be reduced by allowing more implicit casts.

The current situation is inconsistent: `simd_cast<`*`vectorizable_type`*`>` produces `fixed_size` whereas `split` and `concat` use `deduce_t`. Consequently WG21 might want to do either one of:

1. Modify the return types of split and concat to unconditionally use `simd_abi::fixed_size<N>`.
2. Modify `simd_cast` to use `rebind_simd` and thus `deduce_t`.

## 2.6 allowed template arguments for (static_)simd_cast

https://github.com/mattkretz/std-simd-feedback/issues/6

`static_simd_cast` and `simd_cast` in the TS support specifying either a `value_type` or a `simd` type. Any feedback on casts is interesting. Certainly, the omission of casting `simd_mask<T>` is a must-fix, in my opinion.

## 2.7 consider more implicit conversions

https://github.com/mattkretz/std-simd-feedback/issues/7

Links to public discussion relevant to the issue before finalizing the TS design:

1 Note that an implementation might deduce an ABI tag that is not `simd_abi::fixed_size<N>` for all combinations of `T` and `N`.

```
// set x[i] = 0 where y[i] is even
simd<float, Abi> zero_even(simd<float, Abi> x) {
  auto y = std::static_simd_cast<int>(x);
  //where((y & 1) == 0, x) = 0; // error unless Abi is fixed_size
  //where(std::static_simd_cast<float>(y & 1) == 0, x) = 0; // error unless Abi is fixed_size
  where(std::static_simd_cast<simd<float, Abi>>(y & 1) == 0, x) = 0; // works
  return x;
}

// scalar equivalent:
float zero_even(float x) {
  return ((static_cast<int>(x) & 1) == 0) ? 0 : x;
}

// How it should be for simd (also needs P0917R2):
simd<float, Abi> zero_even(simd<float, Abi> x) {
  return ((std::static_simd_cast<int>(x) & 1) == 0) ? 0 : x;
}
```

Listing 1: Conversion verbosity

- `https://github.com/mattkretz/wg21-papers/issues/26` and
- `https://github.com/mattkretz/wg21-papers/issues/3`.

The status quo of the TS:

- `simd_mask<T0, A0>` is implictly convertible to `simd_mask<T1, A1>` if both `A0` and `A1` are `fixed_size<N>` (same `N`).
- `simd<T0, A0>` is implictly convertible to `simd<T1, A1>` if
  - both `A0` and `A1` are `fixed_size<N>` (same `N`), and
  - converting `T0` to `T1` preserves the value of all possible values of `T0`.

As a consequence, the example in Listing 1 is too hard to write.

For the TS, the committee preferred to rather be too strict and discover the need for more implicit conversions through the TS process. Therefore it is important to submit enumerate the cases where implicit conversions would be helpful.

## 2.8 no default for the load/store flags

`https://github.com/mattkretz/std-simd-feedback/issues/8`

Consider:

```
1 std::simd<float> v(addr, std::vector_aligned);
2 v.copy_from(addr + 1, std::element_aligned);
3 v.copy_to(dest, std::element_aligned);
```

Line 1 supplies an optimization hint to the load operation. Line 2 says what really? "Please don't crash. I know this is not a vector aligned access[2]." Line 3 says: "I don't know whether it's vector aligned or not. Compiler, if you know more, please optimize, otherwise just don't make it crash." (To clarify, the difference between lines 2 and 3 is what line 1 says about the alignment of `addr`.) In both cases of `element_aligned` access, the developer requested a behavior we take as given in all other situations. Why do we force to spell it out in this case?

Since C++20, we also have another option:

```
1 std::simd<float> v(std::assume_aligned<std::memory_alignment_v<std::simd<float>>>(addr));
2 v.copy_from(addr + 1);
3 v.copy_to(dest);
```

This seems to compose well, except that line 1 is rather long for a common pattern in this interface. Also, this removes implementation freedom because the library cannot statically determine the alignment properties of the pointer.

Consequently, I suggest to keep the load/store flags as in the TS but default them to `element_aligned`. I.e.:

```
1 std::simd<float> v(addr, std::vector_aligned);
2 v.copy_from(addr + 1);
3 v.copy_to(dest);
```

## 2.9 should converting loads/stores be safer wrt. conversions

https://github.com/mattkretz/std-simd-feedback/issues/9

Since implicit conversions on broadcasts are restricted to value-preserving conversions, the load/store interface is inconsistent with this strictness. I.e.:

```
using V = simd<float>;
V(1.);  // error, double -> float conversion is not value-preserving
V(short(1));  // good, short -> float is value-preserving
double mem[V::size()] = {1.};
V(mem, flags::element_aligned);  // converting load, compiles (but shouldn't?)
short mem[V::size()] = {1};
V(mem, flags::element_aligned);  // value-preserving conversion, compiles
```

2 Of course, vector aligned is equivalent to element aligned if `simd<float>::size() == 1`

The conversion is not obvious when you look at the line of code that requests the load.

Options:

1. Drop converting loads and stores. (That would be unfortunate, since doing converting loads and stores efficiently and portably is potentially a hard problem.)
2. Use different functions, e.g. `v.copy_from(mem, flags)` requires `mem` to be a `value_type` array. `v.memload_cvt_safe(mem, flags)` allows value-preserving conversions. `v.memload_cvt_unsafe(mem, flags)` allows all conversions.
3. Use a flag to enable conversions (without cvt flag, no conversions are allowed), e.g.

   ```
       V(mem, flags::element_aligned | flags::safe_cvt);
       V(mem, flags::element_aligned | flags::any_cvt);
       V(mem, flags::element_aligned | flags::saturating_cvt);  // new feature
   ```

   As a variation, safe conversions could be enabled per default and only unsafe conversions would require extra typing.

## 2.10 relation operators

https://github.com/mattkretz/std-simd-feedback/issues/10

Shen [P0820R1] argues for a substantial change to the definition of relation operators. Kretz [P0851R0] presents the reason for the status quo. The choices were:

1. All relops are defined and return `simd_mask` (status quo).
2. Compares returning `simd_mask` are provided via new functions (member vs. non-member?)
   a) No relops are defined.
   b) `operator==` and `operator!=` are defined and return `bool`.

If there is new information showing that the current behavior is problematic and a different behavior is more useful the discussion should be reopened.

# A BIBLIOGRAPHY

[1] ISO/IEC 19570:2018. Programming Languages — Technical Specification for C++ Extensions for Parallelism. Standard. ISO/IEC JTC 1/SC 22, 2018. URL: `https://www.iso.org/standard/70588.html`.

[P0851R0] Matthias Kretz. P0851R0: simd<T> is neither a product type nor a container type. ISO/IEC C++ Standards Committee Paper. 2017. URL: `https://wg21.link/p0851r0`.

[P0917R2] Matthias Kretz. P0917R2: Making operator?: overloadable. ISO/IEC C++ Standards Committee Paper. 2019. URL: `https://wg21.link/p0917r2`.

[P0820R1] Tim Shen. P0820R1: Feedback on P0214R5. ISO/IEC C++ Standards Committee Paper. 2017. URL: `https://wg21.link/p0820r1`.