P3024R0 Interface Directions for `std::simd`

Jeff Garland, David Sankel, Matthias Kretz, Ruslan Arutyunyan

Created: 2023-11-08 Wed 11:25

Goals & Context

- Improve the experience of all users of `std::simd`
- Increase the `std::simd` coherence with standard c++
- Continue to support best performance possible
- Outline path forward for c++26 `simd` interface
- Increase committee consensus on final direction forward
- Context
    - agree on almost everything
    - where we disagree we, agree on the characterization of the tradeoffs

Outline

- Construction
  - range constructors
  - safety
- Vectorizable type support: enum and `std::byte`
- operator[] and proxy references
- `simd` as a range
- input-output support
- algorithm naming
- regular redux

Construction of `std::simd`

Current api

```
constexpr simd() noexcept;  //does not initialize

template<class U>
constexpr simd(U&& value) noexcept;  //broadcast

template<class U, class UAbi>
constexpr explicit simd(const simd<U, UAbi>&) noexcept;

template<class G> constexpr explicit simd(G&& gen) noexcept;

template<contiguous_iterator It, class Flags = element_aligned_tag>
constexpr simd(const It& first, Flags = {})

template<class U, class Flags = element_aligned_tag>
void copy_from(const U* mem, Flags = {}) &&

template<class U, class Flags = element_aligned_tag>
void copy_to(U* mem, Flags = {}) const &&;
```

Current usage

```
#include <experimental/simd>
namespace stdx = std::experimental;
using intv8  = stdx::fixed_size_simd<int,8>;

intv8 add_v(const intv8& a, const intv8& b)
{
    return a + b;
}
int main()
{
    int a_data[] = {1, 2, 3, 4, 5, 6, 7, 8};
    intv8 a;
    a.copy_from(&a_data[0], stdx::element_aligned);
    int b_data[] = {7, 6, 5, 4, 3, 2, 1, 0};
    intv8 b;
    b.copy_from(&b_data[0], stdx::element_aligned);
    intv8 c = add_v( a, b );

    for (int i=0; i< c.size(); i++)
    {
       int val = c[i];
       print("{} ", val);
    }
}
```

https://godbolt.org/z/sPdzGWEnx

Concerns and omissions

- `copy_from` and `continguous_iterator` api
  - precondition: `[it, it + size)` must be a valid range
  - caller must ensure does not run off the end
  - occurs when: size mismatch between simd and data type
  - default simd size determined by implementation (compiler flags)
  - rationale: efficiency
- default construction does not initialize
  - `T x;` doesn't, `T x();` zero-initializes
  - UB to read from object in that state

Usage Desires - contiguous static extent types

- `array`, c array, `span`, `initializer_list`
  - P2876R0 Proposal to extend std::simd with more constructors and accessors
  - `initializer_list` interacts poorly with broadcast constructor
- size mismatch
  - too small –> default initialize remaining elements
  - too large –> not compile

```
namespace stdx = std::experimental;
using simd_int_8  = stdx::fixed_size_simd<int,8>;

std::array<int, 8> data = {0, 1, 2, 3, 4, 5, 6, 7};
simd_int_8 simd1{data};

std::span<int, 8> sdata{data};
simd_int_8 simd2{sdata};

int cdata[] = {0, 1, 2, 3, 4, 5, 6, 7};
simd_int_8 simd3{cdata};

simd_int_8 simd4 = {0, 1, 2, 3, 4, 5, 6, 7}; // initializer list*
```

Usage Desires - contiguous dynamic extent types

- `vector<data-parallel-type>`
- `string` and `string_view`
- `span<data-parallel-type>`

```
namespace stdx = std::experimental;
using simd_int_8  = stdx::fixed_size_simd<int,8>;

std::vector<int> vdata = {0, 1, 2, 3, 4, 5, 6, 7, 8, 9};
simd_int_8 simd1{data}; //drops 8, 9

std::span<int> data{vdata};
simd_int_8 simd2{sdata}; //drops 8, 9

std::string s{"hello");
simd_int_8 simd3{s}; //default initialize remaining
```

Contiguous Range Constructor Proposal

- recommendation:
  - add safe range constructors
  - keep opt in unsafe for performance (unless safe has no loss)
  - investigate other range constructors (`input_range`)
- range constructor correctly handles mismatched size
  - cost will be measured
- max performance still allowed using unsafe opt in
- similar changes for `copy_from`

```
template<contiguous_range R, class Flags = element_aligned_tag>
constexpr simd(R&& r, Flags = {})

template<contiguous_range R, class Flags = element_aligned_tag>
constexpr copy_from(R&& r, Flags = {})

template<contiguous_iterator It, class Flags = element_aligned_tag>
constexpr simd( simdunchecked_t{}, const It& first, Flags = {})

template<contiguous_iterator It, class Flags = element_aligned_tag>
constexpr unsafe_copy_from ( const It& first, Flags = {})
```

Default constructor and UB

- unfortunately `vector<simd>` is something we need
- performance is impacted
- not sure of a great approach

```
std::simd<int> simd, simd2;
auto simd_result = simd + simd2;

//opt in to unitialized?
std::simd<int> simd { simdunchecked_t{} };
```

initializer list

- P2876R0 Proposal to extend std::simd with more constructors and accessors.
- Recommendation:
  - leave `initializer_list` out in core
  - add it as P2876 progresses
  - consider using a broadcast wrapper to handle ambiguity

```
  simd<int> a(1);        // [1, 1, 1, 1]
  simd<int> b{1};        // [1, 1, 1, 1]
  simd<int> c = {1};     // [1, 1, 1, 1]
  simd<int> d{1, 0};     // [1, 0, 0, 0]
  simd<int> e = {1, 0}; // [1, 0, 0, 0]
// alternate
  simd<int> a(1);          // [1, 0, 0, 0]
  simd<int> b{1};          // [1, 0, 0, 0]
  simd<int> c = {1};       // [1, 0, 0 ,0]
  simd<int> d{1, 2};       // [1, 2, 0, 0]
  simd<int> e = {1, 2};    // [1, 2, 0, 0]
  simd<int> f(bcast(1));  // [1, 1, 1, 1]
  simd<int> b = bcast(1); // [1, 1, 1, 1]
```

Vectorizable type Type support enum and `std::byte`

- `std::byte` is a safer `unsigned char` for bitops
- makes sense to make simd from `span<byte>`
- generalized enum support is more complex
- recommendation: defer general enum support to later

`Operator[]` and proxy reference

- simd is not a container
  - having `operator[]` confusing
  - proxy can create issues (see also, `vector<bool>`)
- recommendation:
  - rename to `get` and `set`
  - leave `operator[]` when we can make it work well everywhere
- https://godbolt.org/z/cfodY4G1E

```
constexpr reference operator[](simd-size-type) &;
constexpr value_type operator[](simd-size-type) const&
```

simd as range

- discussed in several papers
- need `begin` `end` `iterators`
- get format for free
- problems
  - is it writable?
  - proxies and iterators tricky
- recommendation:
  - table simd as a range for now
  - focus on shipping needed core

Input-output support

- at a mimimum we'd like output support in format
- pretty much expect output like vector
- iostreams?
    - no lets not
- recommendation:
    - add formatter for `simd` and `simd_mask`

Algo naming

- naming differences between std algo and simd
- should try to have as much symmetry as possible
- examples
  - `reduce_count` -> `count_if_true`
  - `reduce_min_index` -> `find_if_true`

Regular redux

- after further discussion there are 2 possible paths
- first: current paper approach
- regular with `operator==` and `operator!=`
    - remove all the `operator<`, `operator>` etc
    - replace them with named functions
    - `xsimd` does this and calls them `eq`, `neq`, `gt`, etc

Comparison

- what do you value more: simd onboarding or standard library coherence?

current	regular
minimal change with existing scalar alg to work with simd	fundamental regular operations have an exclusive meaning in c++ (aside from valarray)
minimize cognitive overhead when learning simd	`vector<simd<T>>` is a use case and `operator==` works works
discoverability - if you say `if (simd == simd )` compile fail	default of operator== works with simd data members - secondary use case of simd can make use of existing generic algorithms

References

1. P1928 std::simd Matthias Kretz
2. P2876 Proposal to extend std::simd with more constructors and accessors Daniel Towner Matthias Kretz
3. P2664 Proposal to extend std::simd with permutation API Daniel Towner Ruslan Arutyunyan