I.   Introduction

C++11 introduced a comprehensive mechanism to manage generation of random numbers in the <random> header file.

We propose to introduce an additional API based on iterators in alignment with algorithms definition. simd-type based interface presented in previous paper revisions will be submitted as a separate paper.

II.   Revision history

Only changes for R11 compared with R10 after Tokyo LWG review:

- Applied LWG feedback to wording

Key changes for R10 compared with R9 after Kona and follow up via LEWG Telecon:

- Changed the behavior of CPO to find non-static member function instead of ADL
- Added support of temporary engines and distributions for std::ranges::generate_random (changed to forwarding references)

Key changes for R9 compared with R8 after discussions via email:

- Added non-normative note to the wording about using vectorization-unsafe element access operations
- Added feature macro
- Fixed several typos

Key changes for R8 compared with R7 after discussions via email:

- Reworked CPO-based wording into algorithm-like wording
- Allowed span-based customization
- Added iterator overload for generate_random algorithm
- Extended design-considerations section

Key changes for R7 compared with R6 after LEWG telecon review (2023-03-14):

- Order of the arguments was changed to align with other algorithms
- Additional changes in wording

Key changes for R6 compared with R5 after LEWG telecon review (2022-08-23):

- API was changed from member functions to CPO-based approach

- Iterators-based API was dropped based on weak consensus poll results
- New concepts introduced in previous revisions were removed (not needed for CPO)
- The section of design considerations was considerably extended with past and new aspects documented

Key changes for R5 compared with R4 after LEWG mail list review (Feb-March 2021):

- Renamed member function from operator() to generate()
- Replaced legacy iterators with C++20 iterator and ranges concepts.
- Added ranges-based API
- Discussed iterator constraints from performance perspective
- Added reference to __generate() API added in GNU* libstdc++
- Renamed uniform_vector_random_bit_generator to uniform_bulk_random_bit_generator and added uniform_range_random_bit_generator

Key changes for R4 compared with R3 after LEWGI review (Prague):

- Reverted changes in existing concept uniform_random_bit_generator and introduced uniform_vector_random_bit_generator. Updated corresponding wording.
- Ensured std::random_device benefits from vector API.

Key changes for R3 compared with R2 after SG1 and SG6 review (Belfast):

- Removed execution policies from API, based on Cologne meeting decision.
- Removed simd-based API, for separate consideration as a follow up paper, based on corresponding TS results.
- Added formal wording section for iterators-based API.

Key changes for R2 compared with R1 after SG1 review (Cologne):

- Proposed API for switching between Sequentially consistent and Sequentially inconsistent vectorized results.
- Added performance data measured on the prototype to show price for sequentially consistent results.
- Extended description of the role of generate_canonical in distributions implementations.
- Reworked *Possible approaches to address the problem* chapter to focus on two main approaches under consideration.

Key changes for R1 compared with R0 after SG1 review (Rapperswil):

- Extended the list of possible approaches with simd type direct usage.
- Added performance data measured on the prototype.
- Changed the recommendation to a combined approach.

III.   Motivation and Scope

The C++11 random-number API is essentially a scalar one. Stateful nature and the scalar definition of underlying algorithms prevent auto-vectorization by the compiler.

However, most existing algorithms for generation of pseudo- [and quasi-]random sequences allow algorithmic rework to generate numbers in batches, which allows the implementation to utilize simd-based HW instruction sets.

Internal measurements show significant scaling over simd-size for key baseline Engines yielding a substantial performance difference on the table on modern HW architectures.

Extension and/or modification of the list of supported Engines and/or Distributions is out of the scope of this proposal.

IV. Libraries and other languages

Vector APIs are common for the area of random numbers generation. Examples:

* Intel® oneAPI Math Kernel Library (oneMKL)

  - Statistical Functions component includes Random Number Generators C vector based API

* Java* java.util.Random

  - Has doubles(), ints(), longs() methods to provide a stream of random numbers

* Python* NumPy* library

  - NumPy array has a method to be filled with random numbers

* NVIDIA* cuRAND

  - host API is vector based

* GNU* libstc++ extension

GNU* libstc++ library implementation has an extension, which introduces __generate() member function to all distributions (but not engines). It was added back in 2012 jointly with simd_fast_mersenne_twister_engine implementation in order to be able to benefit from simd instructions.

Intel oneMKL can be an example of the existing vectorized implementation for a variety of engines and distributions. Existing API is C [1] (and FORTRAN), but the key property which allows enabling vectorization is a vector-based interface.

Another example of implementation can be intrinsics for the Short Vector Random Number Generator Library [2], which provides an API on simd level and can be considered an example of internal implementation for proposed modifications.

V. Problem description

Main flow of random number generation is defined as a 3-level flow.

User creates Engine and Distribution and calls operator() of Distribution object, providing Engine as a parameter:

```cpp
std::array<float, arrayLength> stdArray;

std::mt19937 gen(777);
std::uniform_real_distribution dis(1.f, 2.f);

for(auto& el : stdArray) {
    el = dis(gen);
}
```

operator() of a Distribution implements scalar algorithm and typically (but not necessarily so) calls generate_canonical(), passing Engine object further down:

```
uniform_real_distribution::operator()(_URNG& __gen) {
    return (b() - a()) * generate_canonical<_RealType>(__gen) + a();

}
```

It is necessary to note that C++ standard does not require calling generate_canonical() function inside any distribution implementation and it does not specify the number of Engine numbers per distribution number. Having said that, 3 main standard library implementations share the same schema, described here.

generate_canonical() has a main intention to generate enough entropy for the type used by Distribution, and it calls operator() of an Engine one or more times (number of times is a compile-time constant):

```
_RealType generate_canonical(_URNG& __gen) {
    …

    _RealType _Sp = __gen() - _URNG::min();

    for (size_t __i = 1; __i < __k; ++__i, __base *= _Rp)
        _Sp += (__gen() - _URNG::min()) * __base;

    return _Sp / _Rp;

}
```

operator() of an Engine is (almost) always stateful with non-trivial dependencies between iterations, which prevents any auto-vectorization:

```
mersenne_twister_engine<…>::operator()() {

    const size_t __j = (__i_ + 1) % __n;

    …

    const result_type _Yp = (__x_[__i_] & ~__mask) | (__x_[__j] & __mask);
    const size_t __k = (__i_ + __m) % __n;
    __x_[__i_] = __x_[__k] ^ __rshift<1>(_Yp) ^ (__a * (_Yp & 1));
    result_type __z = __x_[__i_] ^ (__rshift<__u>(__x_[__i_]) & __d);
    __i_ = __j;

    …

    return __z ^ __rshift<__l>(__z);

}
```

operator() of the most distributions can be implemented in a way, which the compiler can inline and auto-vectorize. generate_canonical() adds an additional challenge for the compiler due to the loop, but it is resolvable. operator() on the engine level is the key showstopper for the auto-vectorization.

VI.   New API

The following API extension is targeting to cover generation of bigger chunks of random numbers, which allows internal optimizations hidden inside implementation.

A new algorithm is added in the std::ranges namespace to cover generation in chunks.

Ranges-based overload to cover engines, engine adaptors and random_device:

```
std::array<std::uint_fast32_t, arrayLength> intArray;
std::mt19937 g(777);
```

```
    std::ranges::generate_random(intArray, g);
```

It is equivalent to:

```
    for(auto& el : intArray)
        el = g();
```

Ranges-based overload to cover distributions:

```
    std::array<float, arrayLength> fltArray;
    std::mt19937 g(777);
    std::uniform_real_distribution d(1.f, 2.f);

    std::ranges::generate_random(fltArray, g, d);
```

It is equivalent to:

```
    for(auto& el : fltArray)
        el = d(e);
```

New CPO allows writing customizations primarily for generation of random numbers in chunks, which can rely on vectorization internally in implementation details.

Also added iterator-based versions:

```
std::ranges::generate_random(intArray.begin(), intArray.end(), g);
std::ranges::generate_random(fltArray.begin(), fltArray.end(), g, d);
```

Implementers of engines and distributions are getting the instrument to develop *generate_random* customization, which would provide an optimized version of an algorithm for generating a batch of random numbers.

It should be sufficient to implement one customization for a std::span with result_type type, and the standard library will be able to use it with different range types, if result_type is convertible to range value_type. But customizations may go further with specific implementations for required ranges.

That said, it is assumed that the best performance will be achieved for a contiguous range with the value_type, which is the same as result_type of the generator/distribution.

VII.   Wording proposal

? Header <random> synopsis [rand.synopsis]

```
  namespace std {
  ...

  // [rand.util.canonical], function template generate_canonical
  template<class RealType, size_t bits, class URBG>
  RealType generate_canonical(URBG& g);

  namespace ranges {
    // [rand.alg.generate], Generate random
    template<class R, class G>
      requires output_range<R, invoke_result_t<G&>> &&
```

```
                  uniform_random_bit_generator<remove_cvref_t<G>>
        constexpr borrowed_iterator_t<R>
          generate_random(R&& r, G&& g);


      template<class G, output_iterator<invoke_result_t<G&>> O, sentinel_for<O> S>
        requires uniform_random_bit_generator<remove_cvref_t<G>>
        constexpr O
          generate_random(O first, S last, G&& g);

      template<class R, class G, class D>
        requires output_range<R, invoke_result_t<D&, G&>> &&
                 invocable<D&, G&> &&
                 uniform_random_bit_generator<remove_cvref_t<G>>
        constexpr borrowed_iterator_t<R>
          generate_random(R&& r, G&& g, D&& d);

      template<class G, class D, output_iterator<invoke_result_t<D&, G&>> O,
               sentinel_for<O> S>
        requires invocable<D&, G&> &&
                 uniform_random_bit_generator<remove_cvref_t<G>>
        constexpr O
          generate_random(O first, S last, G&& g, D&& d);
  }

  // [rand.dist.uni.int], class template uniform_int_distribution
  template<class IntType = int>
  class uniform_int_distribution;
  ...
  }
```

Add a new section after Specialized <memory> algorithms [specialized.algorithms]

? Specialized <random> algorithms [rand.alg]

The contents specified in [specialized.algorithms] are declared in the header <random>.

? Generate random [rand.alg.generate]

```
template<class R, class G>
requires output_range<R, invoke_result_t<G&>> &&
uniform_random_bit_generator<remove_cvref_t<G>>
constexpr borrowed_iterator_t<R> ranges::generate_random(R&& r, G&& g);
```

Effects:

–   Calls `g.generate_random(std::forward<R>(r))` if this expression is well-formed.

–   Otherwise, if `R` models `sized_range`, fills `r` with `ranges::size(r)` values of
    type `invoke_result_t<G&>` by performing an unspecified number of invocations of the
    form `g()` or `g.generate_random(s)`, if such an expression is well-formed for a value `N`
    and an object `s` of type `span<invoke_result_t<G&>, N>`.

    [*Note 1:* Values of `N` can differ between invocations. - *end note*]

–   Otherwise, calls `ranges::generate(std::forward<R>(r), ref(g))`.

Remarks: The effects of `generate_random(r, g)` shall be equivalent to
`ranges::generate(std::forward<R>(r), ref(g))`.

[*Note 2*: This implies that `g.generate_random(a)` fills `a` with the same values as produced by invocation of `g()`. - *end note*]

Returns: `ranges::end(r)`.

```
template<class G, output_iterator<invoke_result_t<G&>> O,
        sentinel_for<O> S>
  requires uniform_random_bit_generator<remove_cvref_t<G>>
  constexpr O ranges::generate_random(O first, S last, G&& g);
```

Effects: Equivalent to: `return generate_random(subrange<O, S>(std::move(first), last), g);`

```
template<class R, class G, class D>
requires output_range<R, invoke_result_t<D&, G&>> &&
invocable<D&, G&> &&
uniform_random_bit_generator<remove_cvref_t<G>>
constexpr borrowed_iterator_t<R>  ranges::generate_random(R&& r, G&& g, D&&
d);
```

Effects:

- Calls `d.generate_random(std::forward<R>(r), g)` if this expression is well-formed.
- Otherwise, if `R` models `sized_range`, fills `r` with `ranges::size(r)` values of type `invoke_result_t<D&, G&>` by performing an unspecified number of invocations of the form `invoke(d, g)` or `d.generate_random(s, g)`, if such an expression is well-formed for a value `N` and an object `s` of type `span<invoke_result_t<D&, G&>, N>`.

  [*Note 3:* Values of `N` can differ between invocations. - *end note*]

- Otherwise, calls `ranges::generate(std::forward<R>(r), [&d, &g] { return invoke(d, g); })`

Remarks: The effects of `generate_random(r, g, d)` shall be equivalent to `ranges::generate(std::forward<R>(r), [&d, &g] { return invoke(d, g); })`.
[*Note 4*: This implies that `d.generate_random(a, g)` fills `a` with the values with the same random distribution as produced by invocation of `invoke(d, g)`. - *end note*]
Returns: `ranges::end(r)`.

```
template<class G, class D, output_iterator<invoke_result_t<D&, G&>> O,
        sentinel_for<O> S>
  requires invocable<D&, G&> &&
          uniform_random_bit_generator<remove_cvref_t<G>>
  constexpr O ranges::generate_random(O first, S last, G&& g, D&& d);
```

Effects: Equivalent to: `return generate_random(subrange<O, S>(std::move(first), last), g, d);`

Add a predefined macro to [version.syn]:

```
#define __cpp_lib_generate_random 20????L // also in <random>
```

VIII.   Past poll results

LEWG telecon review (2023-11-28):

POLL: Change `generate_random` customization point to find a `generate_random` member function, rather than use ADL (i.e. change the paper to do what LWG wants).

S F	F	N	A	S A
-	-	-	-	-
0	9	3	2	0

Outcome: Consensus in favor.

LEWG telecon review (2023-08-08):

POLL: Modify P1068R8 (Vector API for random number generation) by adding a feature test macro, and then send the revised paper to LWG for C++26, to be confirmed with a Library Evolution electronic poll.

S F	F	N	A	S A
-	-	-	-	-
2	7	0	0	0

Outcome: Unanimous consensus in favor of forwarding.

LEWG telecon review (2023-03-17):

POLL: Make the argument order of `generate_random` output range, engine, and optionally distribution, in order to be consistent with `fill` and similar algorithms.

S F	F	N	A	S A
-	-	-	-	-
5	4	1	0	0

Outcome: Strong Consensus

POLL: We should promise more committee time to pursuing a fused generate random numbers and copy algorithm (what the paper proposes), knowing that our time is scarce and this will leave less time for other work.

SF	F	N	A	SA
-	-	-	-	-
3	4	1	1	1

Outcome: Weak Consensus

POLL: .generate should be an algorithm e.g. generate_random/random_fill instead of a member function

SF	F	N	A	SA
-	-	-	-	-
8	4	7	0	0

Outcome: Strongly in favor

POLL: This proposal should drop the iterator overloads and provide only the ranges one, assuming it's an algorithm.

SF	F	N	A	SA
-	-	-	-	-
2	6	8	2	0

Outcome: Weak consensus in favor.

IX. Design considerations

a) Performance without API modification

Why is a new API needed and why can't we achieve the same performance gains within the existing API?

There are two major ways to try achieving similar performance gains within existing scalar API:

Library only solution (buffer under the hood)

Engines and/or distributions can implement buffers under the hood, generating a pack of numbers at a time, then returning them back these numbers in a scalar way.

Problem with engines: buffers will increase the state of an engine. This size can be extremely important. Heavy monte carlo simulation may require huge numbers of engines existing at the same time and increasing their size will increase pressure on cache/memory, which will overcome all benefits from vectorization.

This is why the state of engines is predefined in the standard. E.g. see [rand.eng.lcong]: "The state x_i of a linear_congruential_engine object x is of size 1 and consists of a single integer."

If we can't do it on the engines side, we can try to do it on the distributions side.

Problem with distributions: buffer is impossible on the distribution side, because:

1. API allows using different engines for each scalar call and thus implementation can not speculatively assume that the same engine is used further on;
2. API allows using the same engine for the different purpose in between the scalar distribution calls and thus implementation can not speculatively assume that the state of the engine will not change between the calls.

Library only solution (overloads for std::ranges::generate)

Library may provide a number of dedicated overloads for std::ranges::generate, which would apply optimization when available.

Problem with engines: there are not so many. There can be a shadow implementation for existing engines, which would be executed when used via std::ranges::generate with a suitable chunk size. This API is not guiding users that it is a path for performance and it is easy to overlook, but it is partially solvable via more documentation.

Problem with distributions: the key problem is that distribution is not invokable without arguments, it always needs an engine at least, which brings in limitations. Distribution should be wrapped with std::bind and optimization is applied to expressions of the following form only:

```
std::ranges::generate(range, std::bind(distribution, engine));
```

1. Engine is copied inside std::bind result without access to this copy for the user, thus engine cannot be used to continue generation after this call.
2. Implementation should heavily rely on std::bind unspecified implementation details, like function type for overloading, a way to access std::bind engine to get its state for vectorization, etc.
3. An average user will unlikely to rely on std::bind in this case, and compiler would likely to see the following code, which is completely unusable for optimizations via overloads:

```
std::ranges::generate(range, [&](){ distribution(engine); });
```

Compiler+library solution

As it was discussed earlier, a compiler can not vectorize the scalar engine implementation on its own. But if a library implementation can provide internal tips for a compiler that there is a dedicated unrolled implementation for the engine and it can be used for this vectorization, then it can be implemented. E.g. with OpenMP* **declare simd** directive (see section 7.7 of OpenMP* 5.2 specification [6]).

Similar approach is implemented in Intel® C++ Compiler in the Short Vector Random Number Generator Library (see section Usage Model [2]).

While this model is attractive in its simplicity, its performance outcome should be carefully controlled on the user side, because the loop, which will be vectorized, is in the user code and any small modification or change in compiler heuristics may affect the outcome of the vectorization.

b) Why not represent a proposed API as a C++20 range?

Algorithms with ranges are scalar by nature. For example the following pseudo-code shows that std::ranges::copy expects a single output value per iteration.

std::ranges::copy(*engine_view(e)*, output_it);

This leads to the very similar problems described in the previous section.

Problem with engines: We would need to keep a buffer within *engine_view* that stores a generated pack of numbers returning them one by one per algorithm iteration. That increases the size of *engine_view,* which is basically the same as increasing the size of the engine itself. See the "monte carlo simulation" use-case considerations in the previous section.

Problem with distributions: For the imagining pipable API like *engine_view(e) | distribution_view(d);* the buffer is impossible to keep in *distribution_view* because of the same problems for distributions described in the previous section. For alternative API like *engine_view(e,d)* we have the same problems as described for engines and, furthermore, it's unclear how to reuse the same engine object in other RNG APIs since *engine_view* could keep the tail of an already generated pack of random numbers.

c) Standalone function vs. member function

Original revisions of the paper proposed operator(begin, end) as the interface for vector generation.

We changed it to member function generate(begin, end) starting revision R5 based on discussion in LEWG mailing list it was renamed to member function .generate() in R5.

It was refactored to become an algorithm-like interface based on POLL on LEWG telecon (2022-08-28).

The main arguments for the refactoring are:

- It is consistent with current approach to introduce customizable API via CPO
- It is aligned with other algorithm-like functionality
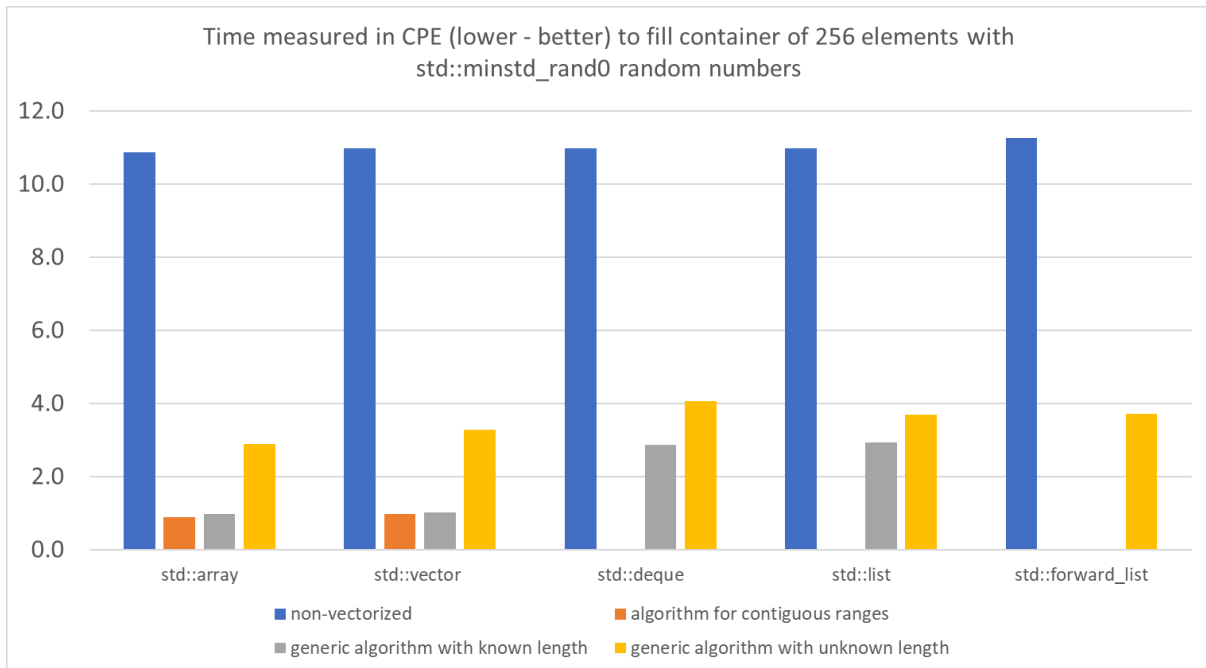- Additional concepts are not required, which simplify the feature

Drawbacks:

- It is different from existing experimental feature in GNU C++ standard library where it is member function called .__generate()
- Using .generate() member function could allow a use case of applying random_device or simpler engines to initialize a big initial state for more sophisticated engines. See, for example, *subtract_with_carry_engine* description, which is initialized by *linear_congruential_engine* by default. But it may be considered a controversial feature.

d) Constraining iterators and ranges

Additional performance results were collected to support discussion regarding additional constraints for iterators and ranges. One would likely need 3 different internal implementations to get optimal performance on existing sequence container:

- Algorithm optimal for contiguous ranges
- Generic algorithm for ranges on known length
- Generic algorithm for ranges on unknown length

Time measured in CPE (lower - better) to fill container of 256 elements with std::minstd_rand0 random numbers

Legend: non-vectorized; algorithm for contiguous ranges; generic algorithm with known length; generic algorithm with unknown length

CPE – cycles per element of the container filled with numbers. Measured on Intel® Xeon® Gold 6152 Processor.

- Contiguous-aware implementation is simpler than generic implementation, but performance benefit can be marginal.
- Knowledge of length allows one to make decisions on vectorization of small chunks of numbers and we get a measurable overhead if length is unavailable.

Our experience shows that a contiguous memory use case is the most important for this API. Random access use case is possible in an unfortunate case of AOS data layout. Other use cases are exotic for random numbers generation and are not expected to be the target for vectorization.

APIs proposed starting from revision 6 of this paper are no longer limited by range constraints: default implementation allows any *output_range* and choosing a constraint for specific engines and/or distributions becomes a quality of implementation question.

e) Bit-to-bit Reproducibility

This paper allows customizations for engines, engine adaptors, distributions and random_device.

"generate_random customization effects shall be equivalent to std::ranges::generate(R, std::ref(E))" wording for *engines* and *engine adaptors* is intended to guarantee bit-to-bit equivalence of the following use case:

```
std::ranges::generate_random(range, g);
```

and

```
for(auto& el : range) {
    el = g();
}
```

This guarantee is needed to keep statistical properties of random numbers sequence, which was justified for each specific engine by its authors in corresponding papers.

random_device sequences are not guaranteed to be anyhow reproducible by fundamental design of this feature.

Bit-to-bit reproducibility of *distributions* sequences is not guaranteed by current wording of the standard even for scalar API. One of the main reasons for that is to allow different distribution algorithms to be implemented by different standard libraries. The main guarantee provided by the standard in this case is a statistical property of the sequence of the numbers.

"generate_random customization effects shall be equivalent to std::ranges::generate(R, [&E,&D]() {return D(E);})" wording for distributions is intended to provide the same guarantees – those two code snippets will result in values, which have the same statistical properties, but are not required to be bit-to-bit exact values:

```
std::ranges::generate_random(range, g, d);
```

and

```
for(auto& el : range) {
    el = d(g);
}
```

f) std::span as a special range

Revision R6 of the paper introduced a CPO-based approach, where default implementation on the standard library side was to reproduce:

```
std::ranges::generate(range, [&](){ distribution(engine); });
```

Each engine and distribution could implement a customization, which would deal with provided Range. If range::value_type is not the same as distribution::result_type and/or range is not contiguous range, each engine/distribution would deal with that on their own. At the same time the approach for vectorization of such cases is similar across the board.

With that we introduced a special case for std::span. If implementation provides customization for std::span<distribution::result_type, size>, the standard library algorithm may deal with bufferization and conversions in a generic way.

This is expected to have zero cost in case of proper value_type and container, when the call would be forwarded to span customization without any additional manipulations.

X.   Implementation experience

Proposed extensions were prototyped on godbolt: https://godbolt.org/z/4Kzh45cE7

XI.   Impact on the standard

This is a library-only extension. This change is ABI compatible with existing random numbers generation functionality.

XII.   References

1. Intel oneMKL documentation:
   https://software.intel.com/content/www/us/en/develop/documentation/onemkl-developer-reference-c/top/statistical-functions/random-number-generators/basic-generators.html
2. Intrinsics for the Short Vector Random Number Generator Library
   https://software.intel.com/content/www/us/en/develop/documentation/cpp-compiler-deve

loper-guide-and-reference/top/compiler-reference/intrinsics/intrinsics-for-the-short-vector-random-number-generator-library.html

3. Box-Muller method
https://en.wikipedia.org/wiki/Box%E2%80%93Muller_transform

4. Inverse transform sampling
https://en.wikipedia.org/wiki/Inverse_transform_sampling

5. Allow Seeding Random Number Engines with std::random_device
http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2016/p0205r0.html

6. OpenMP API 5.2 Specification
https://www.openmp.org/wp-content/uploads/OpenMP-API-Specification-5-2.pdf

Notices & Disclaimers

Performance varies by use, configuration and other factors. Learn more on the Performance Index site.

Performance results are based on testing as of dates shown in configurations and may not reflect all publicly available updates. See backup for configuration details. No product or component can be absolutely secure.

Your costs and results may vary.

Intel technologies may require enabled hardware, software or service activation.

© Intel Corporation. Intel, the Intel logo, and other Intel marks are trademarks of Intel Corporation or its subsidiaries. Other names and brands may be claimed as the property of others.