Paper Number: P1068R4
Title: Vector API for random number generation
Authors: Ilya Burylov <ilya.burylov@intel.com>
Pavel Dyakov <pavel.dyakov@intel.com>
Ruslan Arutyunyan <ruslan.arutyunyan@intel.com>
Andrey Nikolaev <andrey.nikolaev@intel.com>

Audience: LEWG
Date: 2020-07-14

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

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 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 generation random numbers. Examples:

* Intel® Math Kernel Library (Intel® MKL)

  - 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

Intel MKL can be an example of the existing vectorized implementation for verity of engines and distributions. Existing API is C [1] (and FORTRAN), but the key property which allows enabling vectorization is 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:

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

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

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

operator() of a Distribution typically (but not necessarily so) implements scalar algorithm and 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 compiler can inline and auto-vectorize. generate_canonical() adds additional challenge for the compiler due to loop, but it is resolvable. operator() on the engine level is the key showstopper for the auto-vectorization.

VI. Iterators-based API

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

API of Engines and Distributions is extended with iterators based API.

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

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

dis(stdArray.begin(), stdArray.end(), gen);
```

The output of this function may or may not be equivalent to the scalar calls of the scalar API:

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

VII. Wording proposal

26.6.1 Header <random> synopsis [rand.synopsis]

```
namespace std {
// 26.6.2.3, uniform random bit generator requirements
template<class G>
concept uniform_random_bit_generator = see below ;
template<class G>
concept uniform_vector_random_bit_generator = see below ;
…
```

26.6.2.3 Uniform random bit generator requirements [rand.req.urng]

A *uniform random bit generator* g of type G is a function object returning unsigned integer values such that each value in the range of possible results has (ideally) equal probability of being returned. [*Note*: The degree to which g's results approximate the ideal is often determined statistically. — *end note*]

```
…
template<class G>
  concept uniform_random_bit_generator =
    invocable<G&> && unsigned_integral<invoke_result_t<G&>> &&
    requires {
      { G::min() } -> same_as<invoke_result_t<G&>>;
      { G::max() } -> same_as<invoke_result_t<G&>>;
    };
```

Let g be an object of type G. G models uniform_random_bit_generator only if

- both G::min() and G::max() are constant expressions (7.7),
- G::min() < G::max(),
- G::min() <= g(),
- g() <= G::max(), and
- g() has amortized constant complexity.

A class G meets the *uniform random bit generator* requirements if G models uniform_random_bit_generator, invoke_result_t<G&> is an unsigned integer type (6.8.1), and G provides a nested *typedef-name* result_type that denotes the same type as invoke_result_t<G&>.

```
template<class G, class ForwardIterator >
  concept uniform_vector_random_bit_generator =
    uniform_random_bit_generator<G> &&
    requires(G&& g, ForwardIterator begin, ForwardIterator end) {
      { g(begin, end) } -> same_as<void>;
    };
```

Let g be an object of type G, rb and re meet the Cpp17ForwardIterator requirements (23.3.5.4). G models uniform_vector_random_bit_generator only if

- G models uniform_random_bit_generator, and
- Invoke result of g(rb, re) is void.

A class G meets the *uniform vector random bit generator* requirements if G models uniform_random_bit_generator.

26.6.2.4 Random number engine requirements [rand.req.eng]

A *random number engine* (commonly shortened to *engine*) e of type E is a uniform vector random bit generator that additionally meets the requirements (e.g., for seeding and for input/output) specified in this subclause.

…

The template argument for parameters named ForwardIterator shall meet the Cpp17ForwardIterator requirements (23.3.5.4).

Table 92: Random number engine requirements [tab:rand.req.eng]

Expression	Return type	Pre/post-condition	Complexity
		…	
e()	T	Advances e's state e_i to e_{i+1} = TA(e_i) and returns GA(e_i)	per 26.6.2.3
e(ForwardIterator first, ForwardIterator last)	void	With N = last – first, assigns the result of evaluations of e() through each iterator in the range [first, first + N).	$O(N)$
		…	

26.6.2.5 Random number engine adaptor requirements [rand.req.adapt]

No changes

26.6.2.6 Random number distribution requirements [rand.req.dist]

The template argument for parameters named ForwardIterator shall meet the Cpp17ForwardIterator requirements (23.3.5.4).

Table 93: Random number distribution requirements [tab:rand.req.dist]

Expression	Return type	Pre/post-condition	Complexity
		…	
d(g)	T	With p = d.param(), the sequence of numbers returned by successive invocations with the same object g is randomly distributed according to the associated p(z | {p}) or P(zi | {p}) function.	amortized constant number of invocations of g
d(g,p)	T	The sequence of numbers returned by successive invocations with the same objects g and p is randomly distributed according to the associated p(z | {p}) or P(zi | {p}) function.	amortized constant number of invocations of g
d(ForwardIterator first, ForwardIterator last, g)	void	With N = last – first and p = d.param(), the sequence of numbers assigned through each iterator in [first, first + N) is randomly distributed according to the associated p(z | {p}) or P(zi | {p}) function.	$O(N)$

d(ForwardIterator first, ForwardIterator last, g, p)	void	With N = last – first, the sequence of numbers assigned through each iterator in [first, first + N) is randomly distributed according to the associated p(z | {p}) or P(zi | {p}) function.	$O(N)$
...			

26.6.6 Class random_device [rand.device]

A random_device uniform vector random bit generator produces nondeterministic random numbers.

...

VIII. Design considerations

a) Naming

Current paper proposes operator(begin, end) as the interface for vector generation, which makes the object of an Engine similar to algorithm.

At the same time, Sseq uses member function generate(begin, end) for the purpose of using underneath the engine for seeding its state.

P0205R0 [5] proposed adding generate() functions to std::random_device with the similar signature, for it to be usable as a Sseq.

Using generate member function, might be considered as an alternative solution.

IX. Performance results

Implementation approaches were prototyped in part of Distribution API (and Engine API, where required for the use case). See P1068R2 for performance results.

X. Impact on the standard

This is a library-only extension. It adds new member functions to some classes. This change is ABI compatible with existing random numbers generation functionality.

XI. References

1. Intel MKL documentation:
   https://software.intel.com/en-us/mkl-developer-reference-c-2019-beta-basic-generators
2. Intrinsics for the Short Vector Random Number Generator Library
   https://software.intel.com/en-us/node/694866
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

Legal Disclaimer & Optimization Notice

INFORMATION IN THIS DOCUMENT IS PROVIDED "AS IS". NO LICENSE, EXPRESS OR IMPLIED, BY ESTOPPEL OR OTHERWISE, TO ANY INTELLECTUAL PROPERTY RIGHTS IS GRANTED BY THIS DOCUMENT. INTEL ASSUMES NO LIABILITY WHATSOEVER AND INTEL DISCLAIMS ANY EXPRESS OR IMPLIED WARRANTY, RELATING TO THIS INFORMATION INCLUDING LIABILITY OR WARRANTIES RELATING TO FITNESS FOR A PARTICULAR PURPOSE, MERCHANTABILITY, OR INFRINGEMENT OF ANY PATENT, COPYRIGHT OR OTHER INTELLECTUAL PROPERTY RIGHT.

Software and workloads used in performance tests may have been optimized for performance only on Intel microprocessors. Performance tests, such as SYSmark and MobileMark, are measured using specific computer systems, components, software, operations and functions. Any change to any of those factors may cause the results to vary. You should consult other information and performance tests to assist you in fully evaluating your contemplated purchases, including the performance of that product when combined with other products.

Optimization Notice

Intel's compilers may or may not optimize to the same degree for non-Intel microprocessors for optimizations that are not unique to Intel microprocessors. These optimizations include SSE2, SSE3, and SSSE3 instruction sets and other optimizations. Intel does not guarantee the availability, functionality, or effectiveness of any optimization on microprocessors not manufactured by Intel. Microprocessor-dependent optimizations in this product are intended for use with Intel microprocessors. Certain optimizations not specific to Intel microarchitecture are reserved for Intel microprocessors. Please refer to the applicable product User and Reference Guides for more information regarding the specific instruction sets covered by this notice.

Notice revision #20110804