Paper Number: P1068R1

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: SG1 (Parallelism & Concurrency)
Date: 2019-06-13

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.

II. Revision history

Key changes compared with R0:

- 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 of Engine algorithms and the scalar definition of underlying algorithms prevent auto-vectorization by compiler.

However, most existing algorithms for generation of pseudo- or quasi-random numbers 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 an order of magnitude 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(R) 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::vector<float> v(10);

std::mt19937 gen(777);
std::uniform_real_distribution<> dis(1.0f, 2.0f);

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();
}
```

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 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() is the key showstopper for the auto-vectorization.

VI. Possible approaches to address the problem

There are several approaches to address vectorization gap:

a) Internal bufferization

`operator()` of an Engine implementation can generate values in chunks of predefined size and store chunk in internal buffer. If buffer is not empty, implementation can pop value from the chunk and return it, otherwise generate next chunk and return first value from it.

```
std::array<float, arrayLength> stdArray;
std::minstd_rand0                      genStd(555);
std::uniform_real_distribution<float>  disFloat(0.0f, 1.0f);
for (int j = 0; j < arrayLength; ++j)
    stdArray[j] = disFloat(genStd);
```

Pros

- Existing standard API is not modified
- Optimization details are hidden from user space

Cons

- Compiler may not be always able to optimize out intermediate memory storage
- Low-level user code tuning is hard, due to no control on user level
- Instable performance of operator() (requirement of amortized constant complexity is fulfilled though)

b) Explicit iterators-based API

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

```
std::array<float, arrayLength> stdArray;
std::experimental::minstd_rand0                      genStd(555);
std::experimental::uniform_real_distribution<float> disFloat(0.0f, 1.0f);
disFloat(stdArray.begin(), stdArray.end(), genSvrng);
```

Note: **Additional design considerations** section address additional questions of naming and member-functions vs. standalone function aspects of iterators-based API.

Note: **Additional design considerations** section address additional questions of scalar-API values consistency.

Pros

- Optimization details are hidden from user space
- This API does not enforce any specific underlying implementation and can result in several possible optimization strategies to achieve vectorization
- Interface is similar to existing iterators-based API for algorithms with straightforward user-level optimization strategy
- API matches important use case of generation random numbers in block

Cons

- Low-level user code tuning is hard, due to no control on user level

- Changes required on `generate_canonical()` level may become a guarantee of Distribution API (as opposed to require Distribution to call `generate_canonical()` internally), which makes API less consistent.

c) Explicit simd-based API

API of Engines and Distributions is extended to allow simd-like type from Parallelism TS part 2 as a base type.

```
std::array<float, arrayLength > stdArray;
using simd32f = std::experimental::fixed_size_simd<float, 32>;
std::experimental::minstd_rand0                        genSvrng( 555 );
std::experimental::uniform_real_distribution<simd32f> disSimd(0.0f,1.0f);
for (int j= 0; j < arrayLength; j += simd32f::size())
{
    simd32f s = disSimd(genSvrng);
    for (int k = 0; k < simd32f::size(); k++)
        stdArray[j+k] = s[k];
}
int tail = arrayLength % simd32f::size();
if( tail > 0 )
{
    simd32f s = disSimd(genSvrng);
    for (int k = 0; k < tail; k++)
        stdArray[arrayLength - tail + k] = s[k];
}
```

Pros

- Optimizations details are very explicit, which allows low-level user code tuning

Cons

- User is responsible for dealing with blocking and tail calculation
- Changes required on `generate_canonical()` level may become a guarantee of Distribution API, which makes API less consistent.

d) Teach compiler to recognize specific engines and distributions

Intel® C/C++ compiler implements intrinsic functions in the Short Vector Random Number Generator Library [2]. These intrinsic functions can be used underneath existing C++ Standard Library scalar APIs and result in vectorization of the code without changing API.

```
std::array<float, arrayLength> stdArray;
std::minstd_rand0                       genStd(555);
std::uniform_real_distribution<float>   disFloat(0.0f, 1.0f);
for (int j = 0; j < arrayLength; ++j)
    stdArray[j] = disFloat(genStd);
```

Pros

- Existing standard API is not modified
- Optimization details are hidden from user space

Cons

- Implementation is based on compiler-specific extensions, which are not expressible in current state of OpenMP* `#pragma simd` APIs, which makes it vendor-specific
- Low-level user code tuning is hard, due to no control on user level

VII. Additional design considerations

a) Numerical results considerations

There is an open question, whether the results generated by vectorized implementation shall be equivalent to the sequence of scalar APIs.

It is natural to expect from user perspective, but things become more complicated, when `generate_canonical()` results in several calls to the underlying Engine.

Assuming we have a simd size equal `simd_size`, `generate_canonical()` enforces using 2 Engine values per one distribution value, engine values `e[i=0..7]` and distribution values `d[j=0..3]`.

Scalar implementation will use values `e[k*2]` and `e[k*2+1]` for `d[k]` value.

Optimal vector implementation will use `e[(k/simd_size)*simd_size*2 + k%simd_size]` and `e[(k/simd_size)*simd_size*2 + k%simd_size + simd_size]` (we use k-th value of first generated simd and k-th value of second generated simd), which is not only different from previous one, but also simd_size dependent.

Several options to address that:

- Explicitly allow different sequence for Distribution results (but enforce the same sequence for Engine results).
    - `generate_canonical()` can be extended with simd-based interface and/or iterators-based interface, but internal logic stays mostly similar to existing one
- Explicitly the same sequence with either:
    - Extend `generate_canonical()` logic to add values transposition, which may limit freedom of optimization strategies with predefined computational flow
    - Vector-centric APIs to take responsibility of ensuring generate_canonical-like implementation underneath without explicit calls of `generate_canonical()`
    - Drop the requirement to use more than one result of underlying Engine for single Distribution value
- Add user level switch to enable/disable same sequence requirement:
    - Iterators-based API can be extended with Execution policy with `seq` and `unseq` policies supported.

b) Implementation options for iterator-based interface

There are several API considerations for iterators based API

- Member function `operator()`

```
dist(stdArray.begin(), stdArray.end(), engine);
```

    - API described in previous chapter
    - Aligned with existing way to use scalar API via operator()
- Member function `generate()`

```
dist.generate(stdArray.begin(), stdArray.end(),engine);
```

    - Brings some connection with `std::generate` function, which has similar intention of filling a container with values
- Reuse `std::generate()`
    - Current API of `std::generate()` is insufficient to use with Distribution API directly, because the latter does not accept arguments for passing to `operator()`

```
std::generate(data.begin(), data.end(), [&]() {return dist(engine);});
```

   - This limits opportunities, for customization of behavior for the given Engine and/or Distribution with specific optimizations
- Introduce a new standalone function `std::generate_rng()`, which will pass the required argument to operator() of the distribution

```
std::generate_rng(data.begin(), data.end(), dist, engine);
```

   - This implementation leaves opportunities for customization on a library level, having both types of Engine and Distribution in the function template arguments

VIII. Performance results

Possible implementation approaches were prototyped in part of Distribution API (and Engine API, where required for the usecase). Short Vector Random Number Generator Library [2] was used as an underlying vectorization engine. LLVM* libc++ 8.0 implementation was used as a baseline implementation.

`std::minstd_rand0` was chosen as an Engine (generated numbers were verified to be bit-to-bit identical with LLVM baseline implementation).

`std::uniform_real_distribution<float>` was chosen as a Distribution (`generate_canonical()` for this pair of Engine and Distribution shall result in single Engine `operator()` call, and thus avoids complexity described in Additional design considerations section).

Two benchmarks were chosen to collect performance data.

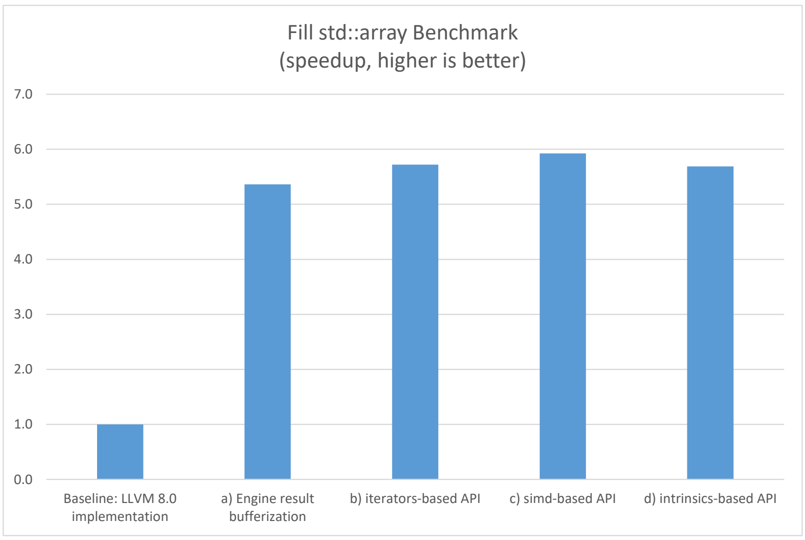
Benchmarks compiled with Intel® C++ Compiler 19.0, measured on Intel® Xeon® Silver 4116 CPU @ 2.10GHz.

a) Fill std::array benchmark

This is an implementation of reference benchmark:

```
std::array<float, 128> stdArray;
std::minstd_rand0                      genStd(555);
std::uniform_real_distribution<float>  disFloat(0.0f, 1.0f);
for (int j=0; j < 128; ++j)
    stdArray[j] = disFloat(genStd);
```

The difference in implementation of the benchmark is discussed in possible approaches chapter.



The results show up to 6x speedup, with options a-d) show comparable performance.

b) Monte Carlo Pi estimation benchmark

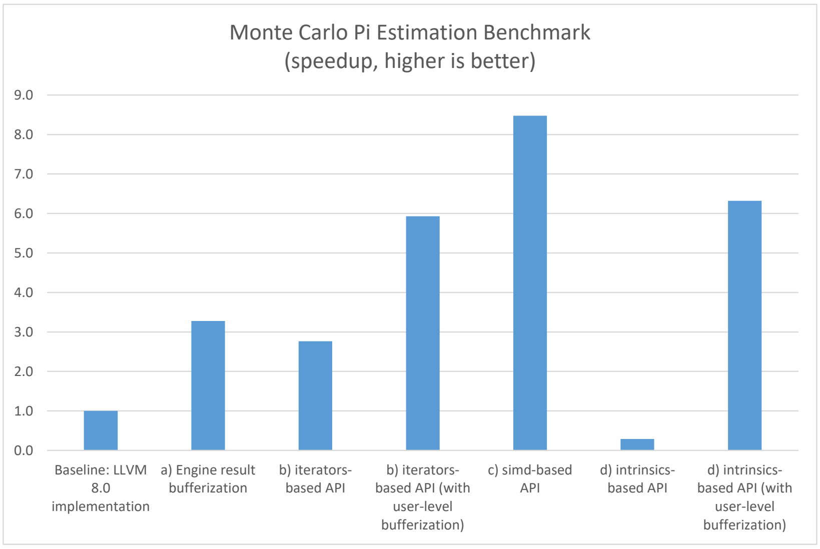
This is an implementation of reference benchmark:

```
int nsamples = 128000000;
std::minstd_rand0                        genStd( 555 );
std::uniform_real_distribution<float>    disFloat( 0.f, 1.f );

int dbUnderCurve = 0;

for (int i = 0; i < nsamples; ++i)
{
    float dbX = disFloat(genStd);
    float dbY = disFloat(genStd);
    if ( dbX*dbX + dbY*dbY <= 1.0 )
        dbUnderCurve++;
}

float dbPiEst = 1.f * dbUnderCurve / nsamples * 4.f;
```



This benchmark showed different requirements needed for user level tuning of the implementation:

- Internal bufferization inside engine, leaves vectorization on low-level loop, which limits vectorization opportunities of user–level loops
- Straightforward usage of iterators-based API results in generation of all required random numbers in the intermediate buffer, which improves the performance from the baseline, but has additional potential for results bufferization on user side to reuse CPU L1 cache
- Simd-based API requires low-level programming by API definition
- Straightforward usage of intrinsics-based version, meets implementation limitation, were compiler cannot auto-vectorize generation of 2 random numbers in a loop, because of requirement to maintain scalar-like RNG sequence. Additional bufferization is needed to overcome this limitation

The results show up to 8.5x speedup with options c), a 6.3x speedup for options b) and d), up to 3.3x with option a).

IX. Recommendation

Options b) Iterators-based API and c) Simd-based API are not mutually exclusive and can be recommended for implementation in the standard:

- c) simd-based API addresses request of low level optimization, when user is willing to get maximum performance for the price of additional coding;
- b) iterators-based API addresses the common use case of generating numbers in blocks with straightforward performance achieving strategy.

Options a) and d) do not require standard modifications to be applied, but have significant downsides, which does not allow them replacing options with explicit API extensions:

- a) internal bufferization option is sub-par performance-wise by design;
- d) compiler intrinsics-options required non-standard compiler modifications and prevents portable library-only implementations.

X. Impact On the Standard

This is a library-only extension. It adds new member functions to some classes but does not change any existing functions, nor enforce adding additional data members or virtual functions. It can, therefore, be ABI compatible with existing implementations.

XI. Summary of changes

The following wording is relative to the C++17 standard. Future revisions of this proposal will include exact sections and deltas.

The engine classes are modified with an additional member function in generating functions section

```
class linear_congruential_engine;
class mersenne_twister_engine;
class subtract_with_carry_engine;
class discard_block_engine;
class independent_bits_engine;
class shuffle_order_engine;
```

Added function (with example of trivial implementation):

```
template<class OutputIt>
void operator()(OutputIt first, OutputIt last) {
    for (; first != last; ++first) {
        *first = operator()();
    }
}
```

Additional iterator version of generate canonical function (with example of trivial implementation):

```
template<class RealType, size_t bits, class URBG, class OutputIt>
void generate_canonical(OutputIt first, OutputIt last, URBG& g) {
    for (; first != last; ++first) {
        *first = generate_canonical<RealType, bits, URBG>(g);
    }
}
```

The distribution classes are modified with two additional member functions in generating functions section

```
class uniform_int_distribution;
class uniform_real_distribution;
```

```
    class bernoulli_distribution;
    class binomial_distribution;
    class geometric_distribution;
    class negative_binomial_distribution;
    class poisson_distribution;
    class exponential_distribution;
    class gamma_distribution;
    class weibull_distribution;
    class extreme_value_distribution;
    class normal_distribution;
    class lognormal_distribution;
    class chi_squared_distribution;
    class cauchy_distribution;
    class fisher_f_distribution;
    class student_t_distribution;
    class discrete_distribution;
    class piecewise_constant_distribution;
    class piecewise_linear_distribution;
```

Added functions (with example of trivial implementation):

```
template<class OutputIt, class URBG>
result_type operator()(OutputIt first, OutputIt last, URBG& g) {
    for (; first != last; ++first) {
        *first = operator()(g);
    }
}

template<class OutputIt, class URBG>
result_type operator()(OutputIt first, OutputIt last, URBG& g, const
param_type& parm) {
    for (; first != last; ++first) {
        *first = operator()(g, parm);
    }
}
```

An optimized implementation should ensure exactly the same result as trivial implementation based on scalar function calls.

Existing library components do not depend on the proposed change, only new APIs added.

XII. 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