Paper Number:	P1068R2
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), SG6 (Numerics)
Date:	2019-10-07

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 and based on std::experimental::simd for low level optimizations.

II. Revision history

Key changes for R2 compared with R1:

- 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:

- 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 an 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(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();
}
```

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 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. Iterators-based API

The following API extension is targeting to cover generation of bigger chunks of memory with internal optimizations hidden inside implementation.

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

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

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

```
for(double& d : arrayLength) {
    d = distFloat(genStd);
}
```

Sequentially consistent result can be a valuable property for the application, thus we introduce ExecutionPolicy based API.

```
template< class RealType = double,
          class ExecutionPolicy = std::sequenced_policy >
class uniform_real_distribution;
```

Two policies support:

- `class sequenced_policy`
  - Provides sequentially consistent result
- `class unsequenced_policy`
  - The results is not necessarily sequentially consistent

See Performance results chapter for performance implications.

VII. std::experimental::simd-based API

The following API extension is targeting to cover fine-grain optimization on user side, providing low-level optimization blocks.

```
std::array<double, arrayLength> stdArray;
using simd32d = std::experimental::fixed_size_simd<double, 32>;
std::experimental::minstd_rand0 genStd( 555 );
std::experimental::uniform_real_distribution<simd32d,
                                             std::execution::unsequenced_policy>
                                                 disSimd(0.0, 1.0);
for (int j = 0; j < arrayLength; j += simd32d::size()) {
    simd32d s = disSimd(genStd);
    for (int k = 0; k < simd32f::size(); k++)
        stdArray[j+k] = s[k];
}
int tail = arrayLength % simd32d::size();
if( tail > 0 ) {
    simd32d s = disSimd(genStd);
    for (int k = 0; k < tail; k++)
        stdArray[arrayLength - tail + k] = s[k];
}
```

The output of single operator() will be equivalent to the following sequence of the scalar API calls if **`std::execution::sequenced_policy`** is used in Distribution type:

```
simd32d s{};
for (int j = 0; j < simd32d::size(); j++) {
    s[j] = distFloat(genStd);
}
```

VIII. Design considerations

a) Numerical results and generate_canonical

C++ standard does not prescribe using generate_canonical for distribution implementation, but it is a common practice to implement it via the call of the function. Which results in using to consecutive numbers from 32-bits Engines for double-precision distributions.

Assuming we have a simd size equal `simd_size`, 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.

It may be more performant to 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.

API of generate_canonical() function does not fit the usage within proposed implementations, thus it is not used, but its behavior is reproduced. Performance of the implementation is not optimal in that case.

b) Numerical results and normal distribution

All key standard libraries use the acceptance-rejection method for implementation of normal distribution, which implies an internal loop with generation of a pair of uniform values and checking if they pass some criteria:

```
uniform_real_distribution<result_type> _Uni(-1, 1);
do {
    __u = _Uni(__g);
    __v = _Uni(__g);
    __s = __u * __u + __v * __v;
} while (__s > 1 || __s == 0);
…
```

Such implementation is not friendly for SIMD-based vectorization. There are other methods possible like Box-Muller method [3] or inverse transform method [4].

Performance of sequentially consistent result is affected, if implemented in alignment with acceptance-rejection method.

IX. Performance results

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).

Two benchmarks were chosen to collect performance data.

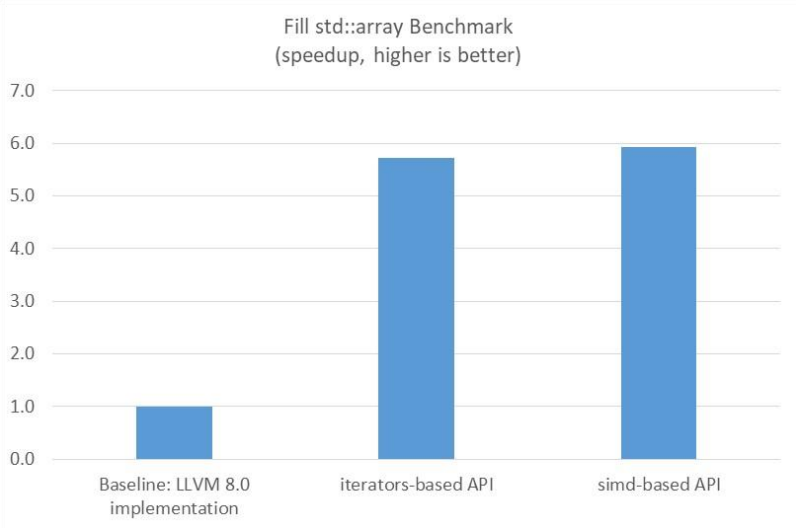
Benchmarks compiled with Intel® C++ Compiler 19.0, measured on Intel® Xeon® Silver 4116 CPU @ 2.10GHz.

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

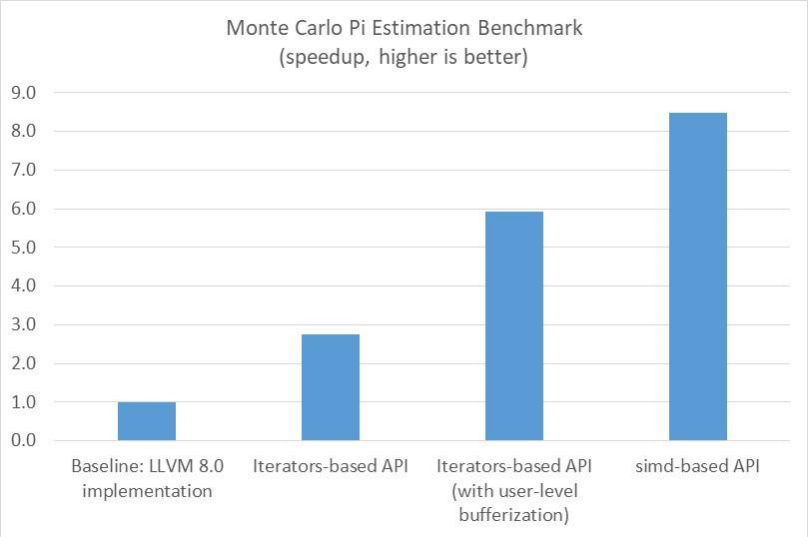
This is an implementation of reference benchmark:

```
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;
```

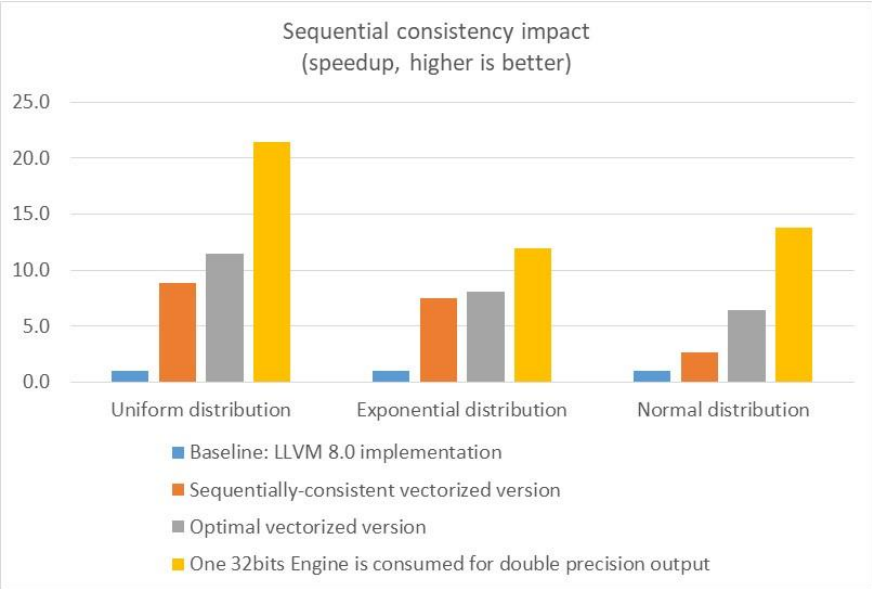


This benchmark showed different requirements needed for user level tuning of the implementation:

- 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

c) Sequential consistency impact (on fill std::array benchmark)

Simd-based API was used, to compare the impact of sequential consistency requirement on the performance of vectorized version. Double-precision distributions were used for this case (which implies generate_canonical function usage, as described in previous sections).



Sequential consistency result is 30% slower in case of uniform distribution.

Sequential consistency result is 8% slower in case of exponential distribution.

Sequential consistency result is 2.3x slower in case of normal distribution (inverse transform method was used for optimal vectorization). If baseline implementation be changed to inverse transform method, the impact will be similar to exponential distribution numbers

As it is seen from graphs, using only single 32-bit engine value provides additional performance benefits and remains valid from perspective of existing C++ standard.

X. Recommendation

Proceed further with both approaches, using SIMD-based approach as a low level building block for iterators-based API.

XI. Impact on the standard

This is a library-only extension. It adds new member functions to some classes. Due to changes in list of template parameters of distributions, this change brings in an ABI change, localized to random numbers generation functionality.

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
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

Legal Disclaimer & Optimization Notice

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