# インテル® oneAPI マス・カーネル・ライブラリー (インテル® oneAPI MKL) データ並列 C++ 使用モデル (モンテカルロ・シミュレーションの例)

この記事は、インテル® デベロッパー・ゾーンに公開されている「Intel® oneAPI Math Kernel Library Data Parallel C++ Usage Models (on the Example of Monte Carlo Simulation)」の日本語参考訳です。

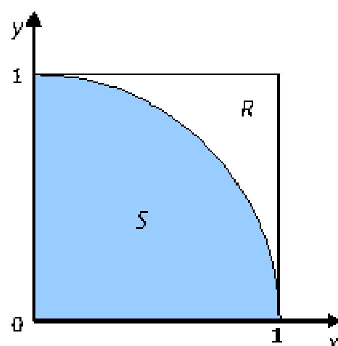
---

## はじめに

この記事では、ベータ版インテル® oneAPI マス・カーネル・ライブラリー (インテル® oneAPI MKL) (英語) で追加されたインテル® MKL 乱数ジェネレーターを利用して、モンテカルロ・シミュレーションの例にデータ並列 C++ 使用モデルを適用する方法を紹介します。リファレンス C++ ベース、インテル® oneMKL データ並列 C++ ベース、インテル® oneMKL データ並列 C++ 拡張の 3 つの例を示します。

## モンテカルロ・シミュレーションによる π の値の評価 - 論理的側面

モンテカルロ・シミュレーション (モンテカルロ法とも呼ばれる) は、数値結果を得るため繰り返しランダム・サンプリングを使用する計算アルゴリズムです [Knuth81]。次の例は、モンテカルロ法を用いて π 値を評価します。次の例に示す単位正方形に内接する象限について考えてみます。




セクター *S* の面積は Area(S) = 1.0/4*r^2 = Pi/4、正方形 *R* の面積は Area(R) = 1 に等しくなります。

ポイント C = (x,y) が単位正方形 *R* からランダムに選択される場合、*C* がセクター *S* 内にある確率は、次のように求められます。

Pr(C が S 内) = Area(S)/Area(R) = Pi/4　(1)

*n* 個のこのようなポイントについて考えます。ここで *n* は十分な大きさであり、*S* – *k* に該当するポイントの数をカウントします。確率 Pr (C が S 内) は、比率 *k*/n または次のように近似できます。

Pi/4 ~ *k*/n　(2)

π の近似は次のように求められます。

$$Pi \sim 4k/n \quad (3)$$

大数の法則に従って、n の数が大きくほど π の近似の精度は高くなります。任意の ξ > 0 について、次のようにベルヌーイの定理からより正確な値を求めることができます。

$$Pr(|k/n-4*Pi|>=\xi) <= 1/(4*n*\xi^2) \quad (4)$$

テストしたポイント *C* の x 座標と y 座標 (横座標と縦座標) が [0,1] の場合、ポイント C は次の場合にセクター *S* 内になります。

$$x^2 + y^2 <= 1 \quad (5)$$

**モンテカルロ･シミュレーションにより π 値を評価するワークフロー**

1. n 個の 2D ポイントを生成します (各ポイントは [0,1) 区間に均等に分布した 2 つの乱数で表されます)。
2. 条件 (5) を使用して、セクター S 内のポイントの数をカウントします。
3. 式 (3) を使用して π の近似値を計算します。

## モンテカルロ･シミュレーションを使用した π 値のリファレンス C++ の例

n_points 個の 2D ポイントを受け取り上記のワークフローを処理する関数 **estimate_pi** について考えてみましょう。

```
float estimate_pi( size_t n_points ) {
    float estimated_pi;          // Pi の推定値
    size_t n_under_curve = 0;    // 曲線の下にあるポイントの数

    // 乱数のストレージを割り当てる
    std::vector<float> x(n_points);
    std::vector<float> y(n_points);
    // ステップ 1. n_points 個の乱数を生成する
    // 1.1. ジェネレーターを初期化する
    std::default_random_engine engine(SEED);
    std::uniform_real_distribution<float> distr(0.0f, 1.0f);
    // 1.2. 乱数を生成する
    for(int i = 0; i < n_points; i++) {
        x[i] = distr(engine);
        y[i] = distr(engine);
    }
    // ステップ 2. 曲線の下にあるポイントの数をカウントする
    for ( int i = 0; i < n_points; i++ ) {
        if (x[i] * x[i] + y[i] * y[i] <= 1.0f)
            n_under_curve++;
    }
    // ステップ 3. Pi の近似値を計算する
    estimated_pi = n_under_curve / ((float)n_points) * 4.0;
    return estimated_pi;
}
```

現在の例は、C++ 11 標準の乱数ジェネレーターを使用しています。ステップ 1 では、エンジン (engine) と分布 (distr) の 2 つのインスタンスを作成して乱数ジェネレーターを初期化します。

```
// 1.1. ジェネレーターを初期化する
    std::default_random_engine engine(SEED);
    std::uniform_real_distribution<float> distr(0.0f, 1.0f);
```

engine はジェネレーターの状態を保持し、独立した一様分布の確率変数を提供します。distr は適切な統計とパラメーターを使用してジェネレーター出力の変換を表します。この例では、uniform_real_distribution は区間 [a, b) に均一に分布したランダムな浮動小数点値を生成します。

engine を distr の operator() に渡して、単一の浮動小数点変数 (乱数) を取得します。次のループは、ベクトル x と y に乱数を格納します。

```
// ステップ 2. 曲線の下にあるポイントの数をカウントする
    for ( int i = 0; i < n_points; i++ ) {
        if (x[i] * x[i] + y[i] * y[i] <= 1.0f)
            n_under_curve++;
    };
```

ステップ 3 では、π 値を計算してメインプログラムに返します。

```
estimated_pi = n_under_curve / ((float)n_points) * 4.0;
    return estimated_pi;
```

## モンテカルロ・シミュレーションを使用した π 値のインテル® oneMKL データ並列 C++ の例

**estimate_pi** 関数に **cl::sycl::queue** の生成を追加します。データ並列 C++ では、セレクター・インターフェイスを利用して実行するデバイスを選択できます [DPC++ Spec reference]。

```
float estimate_pi(size_t n_points) {
    float estimated_pi;          // Pi の推定値
    size_t n_under_curve = 0;    // 曲線の下にあるポイントの数

    // 乱数のストレージを割り当てる
    cl::sycl::buffer<float, 1> x_buf(cl::sycl::range<1>{n_points});
    cl::sycl::buffer<float, 1> y_buf(cl::sycl::range<1>{n_points});

    // 実行するデバイスを選択してキューを作成する
    cl::sycl::gpu_selector selector;
    cl::sycl::queue queue(selector);

    std::cout << "Running on: " <<
        queue.get_device().get_info<cl::sycl::info::device::name>()
<<std::endl;
    // ステップ 1. n_points 個の乱数を生成する
    // 1.1. ジェネレーターを初期化する
    mkl::rng::philox4x32x10 engine(queue, SEED);
    mkl::rng::uniform<float, mkl::rng::standard> distr(0.0f, 1.0f);

    // 1.2. 乱数を生成する
    mkl::rng::generate(distr, engine, n_points, x_buf);
    mkl::rng::generate(distr, engine, n_points, y_buf);
```

```
    // ステップ 2. 曲線の下にあるポイントの数をカウントする
    auto x_acc = x_buf.template get_access<cl::sycl::access::mode::read>();
    auto y_acc = y_buf.template get_access<cl::sycl::access::mode::read>();
    for ( int i = 0; i < n_points; i++ ) {
        if (x_acc[i] * x_acc[i] + y_acc[i] * y_acc[i] <= 1.0f)
            n_under_curve++;
    }

    // ステップ 3. Pi の近似値を計算する
    estimated_pi = n_under_curve / ((float)n_points) * 4.0;
    return estimated_pi;
}
```

cl::sycl::queue はインテル® oneMKL 機能の入力引数であり、インテル® oneMKL ライブラリーのカーネルはこのキューで送信されます。デバイスを切り替えるためコードを変更する必要はありません (ベータ版インテル® oneMKL ではホスト、CPU、および GPU を選択できます)。

乱数の格納には、std::vectors の代わりに cl::sycl::buffers を使用します。

```
// リファレンス:
std::vector<float> x(n_points);
std::vector<float> y(n_points);

// データ並列 C++:
cl::sycl::buffer<float, 1> x_buf(cl::sycl::range<1>{n_points});
cl::sycl::buffer<float, 1> y_buf(cl::sycl::range<1>{n_points});
```

バッファーは、ホスト・アプリケーションとデバイスカーネルのデータを管理します。buffer クラスと accessor クラスはメモリー転送を追跡し、異なるカーネル間でデータの一貫性を保証します。

ステップ 1 では、インテル® oneMKL RNG API も 2 つのエンティティーを初期化します。基本乱数ジェネレーター (エンジン) と分布です。ステップ 1.1 は次のように表すことができます。

```
mkl::rng::philox4x32x10 engine(queue, SEED);
mkl::rng::uniform<float, mkl::rng::standard> distr(0.0f, 1.0f);
```

エンジンは、コンストラクターの入力として cl::sycl::queue と初期値 (SEED) を受け取ります。分布 mkl::rng::uniform には、出力値の型とエンジンの出力の変換に使用されるメソッドのテンプレート引数 (詳細は『インテル® oneAPI マス・カーネル・ライブラリー (インテル® oneMKL) - データ並列 C++ デベロッパー・リファレンス』を参照)、および分布のパラメーターがあります。

ステップ 1.2 では、乱数を取得するため mkl::rng::generate 関数を呼び出します。

```
mkl::rng::generate(distr, engine, n_points, x_buf);
mkl::rng::generate(distr, engine, n_points, y_buf);
```

この関数は、以前のステップで作成された分布とエンジン、生成する要素の数、バッファー内の結果のストレージを受け取ります。RNG API の mkl::rng::generate() は、C++ 標準 API とは異なりベクトル化されます。多くの場合、ベクトルバージョンのライブラリー・サブルーチンのほうがスカラーバージョンよりもはるかに効率良く実行されます。スカラーバージョンのオーバーヘッドは、特に高度に最適化された RNG では、ベクトルバージョンの計算に必要な合計時間と同程度になることがよくあります [VS Notes]。

ステップ 2 では、CPU で生成したすべての乱数を後処理し、データアクセスのためバッファーのホストアクセサーが生成されます。

```
auto x_acc = x_buf.template get_access<cl::sycl::access::mode::read>();
auto y_acc = y_buf.template get_access<cl::sycl::access::mode::read>();
```

その他のステップは、リファレンス C++ の例と同じです。

## モンテカルロ・シミュレーションのインテル® oneMKL データ並列 C++ の例 - 拡張版

インテル® oneMKL データ並列 C++ ベース例のステップ 2 は、データ並列 C++ の Parallel STL 関数を使用して最適化できます。これにより、デバイスからホストへのデータ転送を軽減して、シミュレーションのパフォーマンスを向上できます。最適化後のステップ 2 を次に示します。

```
auto policy = dpstd::execution::make_sycl_policy<class count>(queue);
    auto x_buf_begin = dpstd::begin(x_buf);
    auto y_buf_begin = dpstd::begin(y_buf);
    auto zip_begin = dpstd::make_zip_iterator(x_buf_begin, y_buf_begin);
    n_under_curve = std::count_if(policy, zip_begin, zip_begin + n_points,
        [](auto p) {
            using std::get;
            float x, y;
            x = get<0>(p);
            y = get<1>(p);
            return x*x + y*y <= 1.0f;
        });
```

zip イテレーターは、**count_if** 関数の入力として乱数のペアを提供します。

その他のステップは、インテル® oneMKL データ並列 C++ ベース例と同じです。

## モンテカルロ・シミュレーションのインテル® oneMKL データ並列 C++ の例 - 統合共有メモリー (USM) ベース

データ並列 C++ では、cl::sycl::buffers の代わりに cl::sycl::malloc_shared によって割り当てられた生のポインターを操作することが可能です。このため、ポインター演算を使用できます。

ベータ版インテル® oneMKL の BLAS およびベクトルマス・コンポーネントはすでに USM をサポートしており、インテル® oneMKL の RNG やその他のコンポーネントでは将来のリリースでサポートする予定です。

## インテル® oneMKL データ並列 C++ 例外処理

非同期例外によるエラー処理は、次の場合にサポートされます。

```
auto exception_handler = [] (cl::sycl::exception_list exceptions) {
    for (std::exception_ptr const& e : exceptions) {
        try {
            std::rethrow_exception(e);
        }
        catch(cl::sycl::exception const& e) {
            std::cout << "Caught asynchronous SYCL exception:\n"
            << e.what() << std::endl;
        }
    }
```

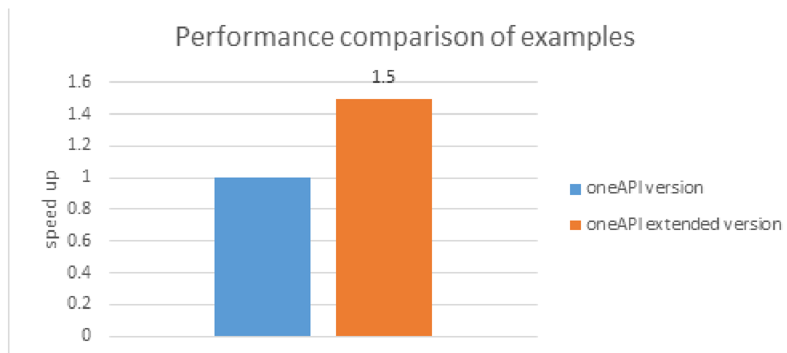
```
};
// 実行するデバイスを選択してキューを作成する
cl::sycl::gpu_selector selector;
cl::sycl::queue queue(selector, exception_handler);
```

例外ハンドラーは cl::sycl::queue コンストラクターで渡されます。DPC++ ランタイムとインテル® oneMKL ライブラリー例外を処理するため、メインの DPC++ 計算部分 (ステップ 1 - 2) は try-catch ブロックにラップされる可能性があります。

```
try {
    // DPC++ 計算部分
}
catch(cl::sycl::exception const& e) {
    std::cout << "\t\tSYCL exception \n" << e.what() << std::endl;
}
```

## パフォーマンスの比較

次のグラフは、インテル® oneMKL データ並列 C++ のベース例と拡張例の比較結果を示します。




システム構成: ハードウェア: インテル® Core™ i7-6770HQ CPU @ 2.60GHz、第 9 世代インテル(R) HD グラフィックス、NEO。OS: Ubuntu* 18.04.1 LTS。ソフトウェア: ベータ版インテル® oneMKL。

測定条件: 生成した 2D ポイントの数:10^8。基本乱数ジェネレーター: mkl::rng::philox4x32x10。乱数分布: 単精度一様分布。測定範囲: estimate_pi() 関数の計算部分 (メモリー割り当てオーバーヘッドを除く)。

## 使用したコード

ここで使用した 3 つの例のコードです。

- pi_bench_reference.cpp (2.48 KB) - リファレンス実装
- pi_bench_oneAPI.cpp (3.04 KB) - インテル® oneMKL ベース実装
- pi_bench_oneAPI_extended.cpp (3.36 KB) - インテル® oneMKL 拡張実装

## Linux* でのサンプルコードのビルドコマンド

```
clang++ -std=c++11 -O3 -DMKL_ILP64 pi_bench_reference.cpp -o
pi_bench_reference.out
clang++ -fsycl -std=c++11 -O3 -DMKL_ILP64 pi_bench_oneAPI.cpp -o
pi_bench_oneAPI.out -lmkl_intel_ilp64 -lmkl_sequential -lmkl_core -lmkl_sycl
-lsycl -lOpenCL
clang++ -fsycl -std=c++14 -O3 -DMKL_ILP64 pi_bench_oneAPI_extended.cpp -o
pi_bench_oneAPI_extended.out -lmkl_intel_ilp64 -lmkl_sequential -lmkl_core -
lmkl_sycl -lsycl -lOpenCL
```

## 文献目録


[Knuth81] Knuth, Donald E. The Art of Computer Programming, Volume 2, Seminumerical Algorithms, 2nd edition, Addison-Wesley Publishing Company, Reading, Massachusetts, 1981.

[VS Notes] Intel® MKL Vector Statistics Notes, https://software.intel.com/en-us/node/810895

[Intel MKL Documentation] https://software.intel.com/en-us/mkl/documentation/view-all

[DPC++ Spec reference] https://spec.oneapi.com/oneAPI/Elements/dpcpp/dpcpp_root.html#

[oneMKL Data Parallel C++ Developer reference] https://software.intel.com/en-us/onemkl-developer-reference-c

[PSTL documentation]
https://spec.oneapi.com/oneAPI/Elements/onedpl/onedpl_root.html#extensions-to-parallel-stl


## 著者


Elizarova, Alina alina.elizarova@intel.com

Dyakov, Pavel pavel.dyakov@intel.com

Fedorov, Gennady Gennady.Fedorov@intel.com



コンパイラーの最適化に関する詳細は、最適化に関する注意事項を参照してください。