乱数生成と並列コンピューティング

この記事は、インテルのウェブサイトで公開されている「Fast Sub-Stream Parallelization for oneMKL MRG32k3a Random Number Generator」の日本語参考訳です。原文は更新される可能性があります。原文と翻訳文の内容が異なる場合は原文を優先してください。

GPU はさらに強力になり、乱数生成を含むさまざまな計算タスクで広く利用されています。乱数は、暗号化、シミュレーション、科学計算など、多くのアプリケーションで不可欠です。予知保全、定量的金融リスク評価、地震および津波緊急対応計画など、さまざまな予測シナリオに乱数シードを提供する上で重要な役割を果たします。GPU は並列処理機能を備えており、乱数生成には大きな利点があります。多数の乱数を同時に生成できることで、効率と速度が向上します。データのプライバシーと正確性が最優先であるデジタル時代において、GPU で乱数を生成することは、高精度の計算、シミュレーション、予測エンジンを拡張および高速化する上で不可欠であり、銀行、保健科学、企業セキュリティーにおける安全な暗号化を、エンドユーザー・エクスペリエンスに影響することなく実現できます。

インテル® oneAPI マス・カーネル・ライブラリー (インテル® oneMKL) は、数学計算向けに最適化された関数を提供するハイパフォーマンスな数学ライブラリーであり、線形代数、高速フーリエ変換、乱数生成などの複雑な数学演算を高速化するように設計されています。インテル® oneMKL は、高速計算プラットフォームをターゲットとする最適化された高性能アプリケーションを構築するための包括的な開発ツールとライブラリーのセットであるインテル® oneAPI ベース・ツールキットの一部です。

MRG32k3a[1] 乱数ジェネレーターは、高度なユースケースで疑似乱数を生成するために広く使用されているアルゴリズムです。長い周期の複数の再帰乱数ジェネレーターを組み合わせたもので、繰り返し前に大きな一意の乱数セットを生成できます。優れた統計特性で知られており、さまざまなアプリケーションに適しています。

アルゴリズムを同時に処理可能な複数の小さなタスクに分割することで、パフォーマンスをさらに向上させることができます。ここでは、アルゴリズムの並列処理レベルを上げてパフォーマンスを向上することを目指します。

この記事では、MRG32k3a 乱数ジェネレーターのサブシーケンス並列化メソッドをインテル® oneMKL に追加します。ライブラリーにはすでに乱数生成メソッドが存在しますが、この記事ではサブシーケンス並列化テクニックを組み込む利点に注目します。

並列化アプローチ

乱数を並列に生成するにはさまざまな方法があり、結果のシーケンスとジェネレーター周期の統計的正確性を考慮する必要があります。統計的ランダム性と、数字が繰り返されるまでの周期の長さは妥協できません。次のセクションで、このような並列化を実装するための主な戦略について説明します。

MRG32k3a 乱数並列化の標準的な実装では、スキップアヘッド・メソッド[2] (英語) を使用して、生成プロセスをスレッド間で均等に分割します (図 1)。例えば、10 個のスレッドで 100 個の乱数を生成する場合、1 つ目のスレッドが最初の 10 個の要素を生成、2 つ目のスレッドが次の 10 個の要素を生成、と続きます。このアプローチでは、一貫した出力が保証されますが、各スレッドで非常に計算が複雑なスキップアヘッド操作が必要になります。

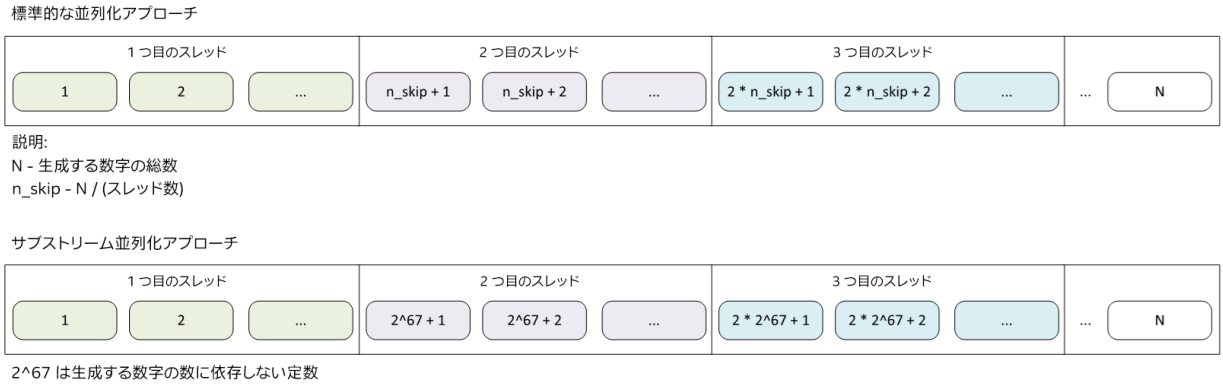

図 1. 並列化アプローチの種類

Pierre L'Ecuyer らによる論文「多くの長いストリームとサブストリームを使用したオブジェクト指向の乱数パッケージ」[3] で説明されている研究では、MRG32k3a ジェネレーターによって生成されたシーケンスは、2^67 要素の変位を持つサブシーケンスに分割でき (図 1)、結果のシーケンスに相関関係を生じさせることなく再結合できることが実証されています。

MRG32k3a アルゴリズムには 2^{191} 要素の大きな周期があるため、この処理を実現できます。各サブストリームは乱数を生成する役割を担い、生成された乱数はサブストリームの総数に等しいオフセットでメモリーに格納されます (図 2)。例えば、サブストリームが 10 個あり、100 個の乱数を生成する場合、1 つ目のサブストリームは 1 番目、11 番目、...、91 番目の要素を生成、2 つ目のサブストリームは 2 番目、12 番目、...、92 番目の要素を生成、と続きます。

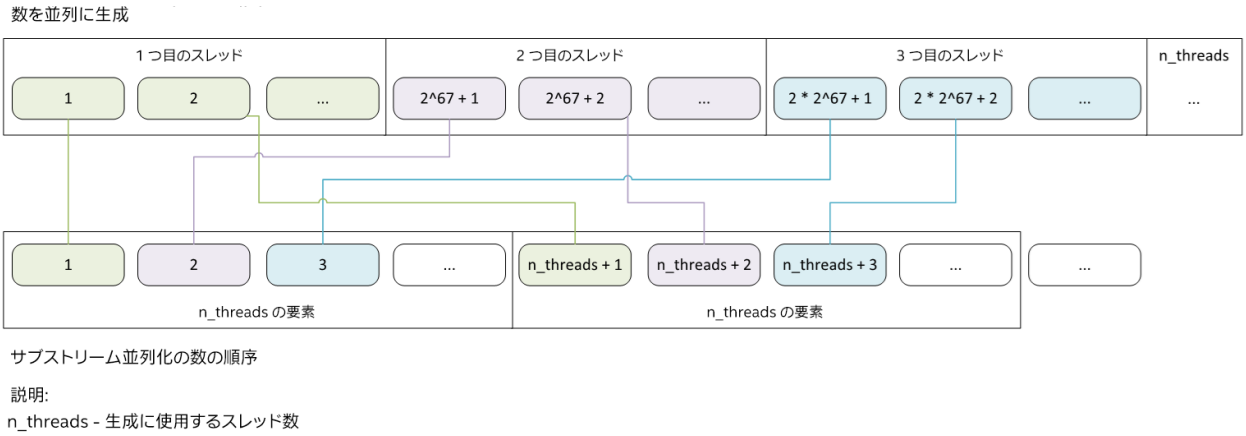

図 2. 生成した数をメモリーに分散するアプローチ

乱数を並列に生成するもう 1 つの一般的なメソッドは、各スレッドに一意のシードを割り当てることです。このアプローチはパフォーマンスのスケーラビリティーを実現できますが、異なるシードを持つシーケンスの統計的独立性は保証されません。異なるシードを使用して並列に乱数を生成すると、類似したまたは相関性がある数のシーケンスが生成される可能性が高くなります。

サブシーケンス並列化メソッドは、生成に単一のシードを使用するため、適切なパフォーマンスが得られ、高品質の出力シーケンスが保証されます。

インテル® oneMKL でのサブシーケンスの使用

乱数ジェネレーターの初期化ステージでこのメソッドを提供することにより、インテル® oneMKL ホスト API でサブシーケンス並列化を使用できます (表 1 を参照)。

表 1. MRG32k3a で均一に分散された数の生成メソッドを更新

`// ...`
 	
`namespace host_rng = oneapi::mkl::rng;`
 	
`host_rng::mrg32k3a`
`  engine(queue, seed);`
 	
`host_rng::uniform distribution;`
 	
`host_rng::generate(distribution, engine, n, data);`
 	
`// ...`
	`// ...`
	
	`namespace host_rng = oneapi::mkl::rng;`
	
	`host_rng::mrg32k3a`
	`  engine(queue, seed,`
	
	`host_rng::mrg32k3a_mode::optimal{});`
	
	`host_rng::uniform distribution;`
	
	`host_rng::generate(distribution, engine, n, data);`
	
	`// ...`

標準的な並列化アプローチで *n* 個の乱数を生成します。	サブシーケンス並列化アプローチで *n* 個の乱数を生成します。

ハードウェアに基づいてサブストリームのデフォルトの数を設定するには、*optimal{}* を指定します。*custom{num_streams}* オプションを使用して、特定のタスクに最適なサブストリームの数を手動で設定します。

サブストリーム並列化によるシーケンス内での数の生成の順序は、上記のようにサブストリームの数によって決まります。「custom」オプションを利用すると、異なるハードウェアとジェネレーター実装を使用した場合でも同じ結果が得られます。

MRG32k3a ジェネレーターを使用して数を生成し、インテル® oneMKL デバイス API を使用してサブシーケンス並列化を行うこともできます (図 3)。

```
// ...

namespace rng_device = oneapi::mkl::rng::device;

// kernel with initialization of generators

queue.parallel_for({num_streams}, [=](std::size_t id) {
        // initialize generators with an offset of 2^67 elements
        rng_device::mrg32k3a<1> engine(seed, {0, 4096 * (id % num_streams)});
        engine_ptr[id] = engine;

}).wait();

// generate random numbers

queue.parallel_for({num_streams}, [=](std::size_t id) {
        auto engine = engine_ptr[id];
```

```
        rng_device::uniform<Type> distr;

        std::uint32_t count = 0;

        while (id + (count * num_streams) < n) {
            auto res = rng_device::generate(distr, engine);
            result_ptr[id + (count * num_streams)] = res;
            count++;
        }

}).wait();

// ...
```

図 3. インテル® oneMKL デバイス API を使用したサブストリーム並列化実装の例

CuRAND とインテル® oneMKL RNG の互換性

アプリケーションを CUDA 乱数生成ライブラリーからインテル® oneMKL に移行する場合、機能が完全に同一になるように、出力シーケンスを維持することが重要です。これは、インテル® oneMKL の MRG32k3a 乱数ジェネレーターでサブシーケンス並列化モードを使用することにより解決できます。

cuRAND ホスト API では、MRG32k3a 乱数ジェネレーターはサブシーケンス並列化モードでのみ動作します。対照的に、インテル® oneMKL の標準実装はシーケンシャル・アプローチを使用します。その結果、同じ乱数ジェネレーターの 2 つの実装で異なるシーケンスになります。

サブシーケンス・モードに切り替えると、両方の乱数ジェネレーターの初期化シードが同じであれば、同じシーケンスになります。

自動コード変換、分析、最適化機能を備えたインテル® DPC++ 互換性ツールを使用すると、インテル® oneMKL への移行を簡素化できます。さらに、出力シーケンスを維持したまま、cuRAND からインテル® oneMKL の RNG ドメインへ自動的に移行できます。

ランダム性を維持したままスケーラビリティーを実現

インテル® oneMKL ライブラリーの MRG32k3a 乱数ジェネレーターにサブシーケンス並列化メソッドを追加すると、GPU で高品質の疑似乱数を効率的に生成できます。出力シーケンスを並列処理できる複数のサブストリームに分割すると、統計品質を損なうことなく並列に処理できます。

サブシーケンス手法は、MRG32k3a 乱数ジェネレーターのスケーラビリティーとランダム性の適切なバランスを見つけます。提供されている使用例は、このメソッドを既存のインテル® oneMKL ワークフローに簡単に統合して、開発者が乱数生成に GPU の並列計算を利用できる方法を示しています。このメソッドをインテル® oneMKL に組み込むと、cuRAND などのほかの GPU 数学ライブラリーからのアプリケーションの移行が簡単になり、使いやすさと互換性が向上します。アプリケーションがディスクリートのインテル® Arc™ グラフィックス GPU、インテル® Xᵉ2 グラフィックス・ベースの統合 GPU、インテル® データセンター GPU、サードパーティーのコンピューティング・アクセラレーターで実行するように最適にスケーリングされているかどうかに関係なく、シミュレーション、数値解析、大量の高品質ランダムデータに依存するその他のアプリケーションが最終的に高速化されます。

ソフトウェアを入手

インテル® oneAPI マス・カーネル・ライブラリー (インテル® oneMKL) はスタンドアロン (英語)、または多様なアーキテクチャー向けにハイパフォーマンスでデータセントリックなアプリケーションの開発を支援するランタイム・ライブラリーのコアセット、インテル® oneAPI ベース・ツールキットの一部としてダウンロードできます。

関連情報

- 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 (英語)
    - スキップアヘッド・メソッド (英語)
- インテル® マス・カーネル・ライブラリーのリンク行アドバイザー (英語)
- インテル® DPC++ 互換性ツール
- インテル® oneMKL 乱数ジェネレーター・デバイス・ルーチン (英語)
- cuRAND* とインテル® oneMKL での乱数生成 (英語)

参考文献

[1] - Pierre L'Ecuyer, (1999) Good Parameters and Implementations for Combined Multiple Recursive Random Number Generators. Operations Research 47(1):159-164. https://pubsonline.informs.org/doi/10.1287/opre.47.1.159 (英語)

[2] - oneMKL Data Parallel C++ Developer Reference Skip Ahead Method (英語)

[3] - Pierre L'Ecuyer, Richard Simard, E. Jack Chen, W. David Kelton, (2002) An Object-Oriented Random-Number Package with Many Long Streams and Substreams. Operations Research 50(6):1073-1075. https://doi.org/10.1287/opre.50.6.1073.358 (英語)
