ケーススタディー: 段階的な最適化フレームワークを使用したモンテカルロ・ヨーロピアン・オプションの計算パフォーマンスの向上

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「Case Study: Achieving High Performance on Monte Carlo European Option Using Stepwise Optimization Framework」の日本語参考訳です。

---

1. はじめに

モンテカルロ法は、統計計算手法を使用して複雑な科学計算問題を解きます。問題の入力の不確実性をシミュレートするために乱数を使用し、パラメーターのサンプリングを繰り返して決定論的な結果を得ることにより、ほかの方法では解決不能な問題を解くことができます。この方法は、マンハッタン計画に参加していた数学者のグループにより 1940 年後半に考案され、カジノで有名なモナコ公国のモンテカルロ地区にちなんで名付けられました。

カナダの金融数学者である Phelim Boyle は、モンテカルロ法を用いてオプションの評価を行った最初の人物として知られています。彼は、「*Journal of Financial Economics*」の 1977 年 5 月号に掲載された歴史的な論文「*Options: a Monte Carlo Approach*」で、ヨーロピアン・オプション値問題を解くシミュレーション法を開発し、独自にブラック-ショールズ式を検証しました。それ以来、モンテカルロ法はウォールストリートのデリバティブ価格評価プロフェッショナルとリスク管理プロフェッショナルの間で非常に有名な計算法となりました。最近のレポートによれば、金融工学問題の約 40 パーセントにモンテカルロ法が含まれています。モンテカルロ・ファームと呼ばれる大規模なクラスター・コンピューター・サーバーのインストールは、ハイパフォーマンスの金融計算を実行するほとんどの企業で必須の構成となっています。1996 年には、Mark Broadie と Paul Glasserman により、モンテカルロ法がアジアンオプションに適用されました。2001 年には、Francis Longstaff と Eduardo Schwartz により、アメリカンオプションの価格を評価する最初のモンテカルロ・アルゴリズムが開発されました。2006 年には、オックスフォード大学の Mike Giles により、ユーザーが指定した精度で高次元問題を解くコスト効率に優れた方法として、マルチレベル・モンテカルロ法が開発されました。

モンテカルロ法は、単純な数式で多くの高次元問題が扱うことができ、直接 PDE 法やラティス法よりも必要なメモリーが少なくて済む一方、特に決定論的で正確な結果が必須の分野において膨大な計算能力を要求します。通常は、満足な結果を得るために数十万の (場合によってはさらに多くの) サンプリングを行います。精度を 1 桁上げるには、サンプリングの数を 100 倍にします。計算を多用する非常に多くの問題を扱う金融業界は、計算能力に対する際限のない要求を満たす革新的なアーキテクチャーを求めています。

1.1 インテル® メニー・インテグレーテッド・コア (インテル® MIC) アーキテクチャー

インテル® メニー・インテグレーテッド・コア (インテル® MIC) アーキテクチャーは、モンテカルロ関連アプリケーションにとって理想的な実行環境を提供します。IA-32/インテル® 64 と同じ命令セット・アーキテクチャーでビルドを行い、インテル® Xeon® プロセッサーとインテル® Xeon Phi™ コプロセッサー製品でほぼ同じプログラミング・モデルを使用できます。並列実行インフラストラクチャーをさらに拡張して、ソースコードをほとんど修正することなく、GPGPU をはるかに超えるレベルのパフォーマンスを備えた高度な並列アプリケーションを作成できます。ユーザーは、ソフトウェア開発と配備の容易さ、およびソフトウェア・パフォーマンスの向上による恩恵が得られます。

インテル® Xeon Phi™ コプロセッサーは、インテル® MIC アーキテクチャー・ベースの最初の製品で、インテル® Initial Many Core Instructions (インテル® IMCI) と呼ばれる全く新しい命令セットを実装しています。インテル® IMCI は、x87 命令を含む、IA-32/インテル® 64 のスカラー命令を継承し、新しく設計されたベクトル・プロセシング・ユニット (VPU) を操作する 512 ビット・ベクトル命令が追加されています。しかし、インテル® IMCI は、インテル® SSE、インテル® SSE2、インテル® AVX をサポートしていません。高度な並列アプリケーションでは、インテル® コンパイラーは VPU で実行するベクトル化されたコードを生成します。

1.1.1 プロセッサー・アーキテクチャー

マイクロプロセッサー・アーキテクチャー・レベルでは、このコプロセッサーは 2 つの双方向リングバスで接続された 61 以上のコアからなる 1 つの SMP プロセッサーです。8 つのオンダイ・メモリー・コントローラーは 16 の GDDR5 チャネルをサポートし、アクセス速度は最大 5.5 GT/秒です。コアレベルでは、各コプロセッサー・コアは個別に機能するインオーダー・コアであり、ほかのコアから独立して IA 命令を実行することができます。各コアはハードウェア・マルチスレッディングをサポートしており、4 つのハードウェア・コンテキストまたはスレッドを実行できます。クロックサイクルあたり、単一コンテキストから最大 2 つの命令を発行できます。

1.1.2 コアおよびベクトル・プロセシング・ユニット

コプロセッサー・コアは、16 の汎用 64 ビット・レジスターと 64 ビット拡張に関連する新しい命令のほとんどを実装しています。しかし、サポートしているベクトル命令はインテル® Initial Many Core Instructions (インテル® IMCI) のみです。スカラー数学ユニット (x87) は完全に機能しますが、コプロセッサー・コアでインテル® MMX® テクノロジー、インテル® SSE、インテル® AVX はサポートしていません。

ベクトル・プロセシング・ユニット (VPU) はゼロから設計された全く新しい 512 ビットの SIMD エンジンで、16 の単精度または 8 の倍精度浮動小数点 SIMD 演算を実行できます。IEEE 754R に表記されている 4 つの丸めモードを完全にサポートしています。VPU は、等価なビットの浮動小数点データ処理と似た方法で 32 ビット整数データと 64 ビット long 型整数データを処理することができます。

VPU 内部で、新しい拡張数学ユニット (EMU) は単精度超越関数 (ミニマックス 2 乗多項式近似を使用した逆数、平方根の逆数、基数 2 の対数、基数 2 の指数の関数) の高速実装を提供します。これらの 4 つのハードウェア実装された初等関数には 4 から 8 サイクルのレイテンシーが含まれ、1 から 2 サイクルの高スループットを達成します。ほかの超越関数はこれらの初等関数から導き出すことができます。

関数	算術	レイテンシー・サイクル	スループット・サイクル
RECIP	1/x		1
RSQRT	$1/\sqrt{x}$		1
EXP2	2^x		2
LOG2	$\log_2 x$		1

1.1.3 キャッシュ・アーキテクチャー

インテル® Xeon Phi™ コプロセッサーのレベル 1 (L1) キャッシュは、コアあたり 4 つのハードウェア・コンテキストに、より高いワーキングセット要件を提供するように設計されました。L1 命令キャッシュと L1 データキャッシュはそれぞれ 32KB です。連想性は 8 ウェイ、キャッシュラインは 64 バイト、バンク幅は 8 バイトです。データリターンはアウトオブオーダーにできます。アクセス時間には 3 サイクルのレイテンシーが含まれます。

512KB 統合レベル 2 (L2) キャッシュは、8 ウェイの連想性 (1 ウェイあたり 64 バイト)、1024 セット、2 つのバンク、32GB (35 ビット) のキャッシュ可能なアドレス範囲からなります。想定アイドルアクセス時間は約 80 サイクルです。

L2 キャッシュには、コード、読み取り、RFO (read-for-ownership) キャッシュラインを L2 キャッシュに選択的にプリフェッチできる、ストリーミング・ハードウェア・プリフェッチャーが用意されています。最大 4KB ページのデータを扱うことができる 16 のストリームをサポートしています。ストリームの方向が検出されると、プリフェッチャーは最大 4 つのプリフェッチ・リクエストを発行します。L2 キャッシュは ECC もサポートしています。L1 および L2 キャッシュの置換アルゴリズムは擬似 LRU 実装ベースです。

次の表は、コプロセッサーのキャッシュ・アーキテクチャーの主要パラメーターを要約したものです。

パラメーター	L1	L2
コヒーレンス	MESI	MESI
サイズ	32K (I) + 32K (D)	512K (統合)
連想度/ラインサイズ/バンク	8 ウェイ/64 バイト/8	8 ウェイ/64 バイト/8
アクセス時間 (クロック)	1	11
ポリシー	擬似 LRU	擬似 LRU
デューティサイクル	クロックあたり 1	クロックあたり 1
ポート	読み取りまたは書き込み	読み取りまたは書き込み

1.2 インテル® Xeon® プロセッサーおよびインテル® Xeon Phi™ コプロセッサー向けソフトウェア開発環境

1.2.1 インテル® Parallel Studio XE 2013

インテルは、2012 年 9 月にインテル® Parallel Studio XE 2013 を発表しました。このインテル製マルチコア・プロセッサーおよびインテル® MIC アーキテクチャー製品ライン向け統合ソフトウェア開発ツールキットには、インテル® C++ Composer XE、インテル® Fortran Composer XE、インテル® VTune™ Amplifier XE アプリケーション・パフォーマンス・プロファイラーをはじめ、開発者がアプリケーションとライブラリーのビルドに使用できる、インテル® スレッディング・ビルディング・ブロック (インテル® TBB) やインテル® マス・カーネル・ライブラリー (インテル® MKL) のようなインテル® パフォーマンス・ライブラリーが含まれています。インテル製マルチコア・プロセッサーおよびインテル® MIC アーキテクチャーのサポートは、最新リリースの主な機能の 1 つです。2013 年 1 月以降、5 つの異なるアップデートがリリースされています。この記事で紹介するプログラムは Update 5 で作成されたもので、パフォーマンスはインテル® C++ Composer XE 13.1 Update 3 で作成した実行ファイルを実行して記録したものです。

1.2.2 インテル® Xeon® プロセッサー・ホスト・システム

ホストシステムはデュアルソケットのインテル® Xeon® プロセッサー・ベースのプラットフォームで、PCI Express* (PCIe) インターフェイス経由でインテル® Xeon Phi™ コプロセッサーと接続されています。ホストの各ソケットには、インテル® Xeon® プロセッサー E5-2670 (8 コア、2.6GHz) が搭載されています。インテル® Xeon® プロセッサー E5-2670 は、SIMD 並列処理用のインテル® AVX をサポートしています。

統合ソフトウェア開発ツールスタックは、インテル® Xeon® プロセッサー・ベースのホストシステムで動作します。アプリケーション開発者は、インテル® Xeon Phi™ コプロセッサー用のオフロード・アプリケーションまたはネイティブ・アプリケーションを作成できます。

オフロード・アプリケーションはホストシステムで開始して個別に実行できます。プログラムの一部は、アプリケーション開発者の指示に基づいてコプロセッサーで実行できます。コンパイラーは、ホストとコプロセッサー用のバイナリーを含む 1 つの実行ファイルを作成します。統合ツールのランタイム・ライブラリーは、コプロセッサー用バイナリーファイルの抽出とコピー、入出力データの設定、コプロセッサー・プログラムの起動を行います。

ネイティブ・アプリケーションはコプロセッサー・カードで実行を開始できます。プログラマーは、コンパイラー・スイッチ (-mmic) を指定して、プログラム全体をコプロセッサー上で実行することをコンパイラーに伝えます。実行ファイルと共有ライブラリーのホストからコプロセッサーへのコピー、入力データと出力データの準備、コプロセッサーでのプログラムの起動はプログラマーの責任です。

この記事では、コプロセッサー向けのネイティブ・アプリケーションの作成についてのみ取り上げます。オフロード・アプリケーションについては取り上げません。

1.2.3 インテル® Xeon Phi™ コプロセッサー

このコプロセッサーは、独自のオペレーティング・システムである、インテル® メニーコア・プラットフォーム・ソフトウェア・スタック (インテル® MPSS) と呼ばれる修正版 Linux* オペレーティング・システムを実行する計算デバイスです。

コプロセッサー・デバイスは単体でブートできません。代わりに、ホストから Linux* デバイス操作コマンドを使用してコプロセッサーをブートまたはシャットダウンします。インテル® Parallel Studio XE 2013 Update 5 と組み合わせて動作するインテル® MPSS のバージョンは 2.1.6720-13 です。この記事では、61 コアのコプロセッサー・モデル 7120p を使用します。それぞれ 1.238GHz で動作し、16GB の GDDR メモリーを搭載しています。コアのメモリー速度は 5.50GHz です。

2. モンテカルロ・ヨーロピアン・オプション価格評価アルゴリズム

このセクションでは、金融派生商品、特にストックオプションの価格評価についていくつかの基本的な情報を紹介します。次に、どのようにモンテカルロ法を使用して入力の不確実性をモデル化するか、どのように各サンプルを扱うかを示します。ペイオフ関数はオプションに基づいて決定されます。最後に、標準 C++ を使用したモンテカルロの実装と、この実装のパフォーマンス・ボトルネックについて説明します。

2.1 オプションの価格評価とは

金融の世界においてデリバティブとは、値がほかのより基礎となる変数の値に依存する金融商品です。多くの場合、デリバティブの基礎となる変数は取引される資産の価格です。例えば、ストックオプションは値が株価に依存するデリバティブです。すべての変数やデリバティブが取引される資産に依存するとは限りません。例えば、特定のリゾートの降雪や、特定の時間間隔の平均気温などが変数になることもあります。

オプションは、一定価格での資産の取引 (権利行使) について買い手と売り手間の契約を指定するデリバティブです。オプションの買い手は、取引を行う権利を得ます (義務ではありません)。一方、オプションの売り手は、取引を行う義務を負います。2 種類のオプションがあります。コールオプションは、一定期日に一定価格で原資産を**買う**権利を保有者に与えます。プットオプションは、一定期日に一定価格で原資産を**売る**権利を保有者に与えます。契約の価格は権利行使価格と呼ばれます。契約の日付は満期日と呼ばれます。ヨーロピアン・オプションは、満期日にのみ権利行使が可能です。アメリカンオプションは、満期日までいつでも権利行使が可能です。

2.2 モンテカルロ法を使用したヨーロピアン・ストック・オプションの価格評価

モンテカルロ・シミュレーションは、リスク中立評価法を使用してオプションを評価します。パスをサンプリングしてリスク中立世界の期待ペイオフを取得した後、リスクフリー・レートを使用して現在の評価のペイオフを割り引きます。時価 S の株のストックオプションについて、時間 T におけるペイオフを計算してみましょう。利子は一定であると仮定し、次のようにデリバティブを評価します。

1. リスク中立世界で S のランダムパスをサンプリングします。
2. デリバティブのペイオフを計算します。
3. ステップ 1 と 2 を繰り返して、リスク中立世界のデリバティブから多くのサンプルペイオフ値を取得します。
4. サンプルペイオフの平均を計算してリスク中立世界の推定期待ペイオフを取得します。
5. リスクフリー・レートで期待ペイオフを割り引いてデリバティブの推定評価を取得します。

リスク中立世界の基礎的な市場変数が続くプロセスを次のように仮定します。

$$dS = \mu S dt + \sigma S dWt \qquad 3.1$$

ここで、dWt は Wiener 過程、μ はリスク中立ワークの期待収益、σ はボラティリティーです。lnS が続くパスをシミュレートするため、デリバティブの期間を長さ t の N 区間と近似方程式 3.1 に分割します。

$$lnS(t + \Delta t) - lnS(t) = (\mu - \frac{\sigma^2}{2})\Delta t + \sigma\epsilon\sqrt{\Delta t}$$

あるいは

$$S(t+\Delta t) = S(t)\,exp\left[\left(\mu - \frac{\sigma^2}{2}\right)\Delta t + \sigma\epsilon\sqrt{\Delta t}\right]$$

ここで、$\boldsymbol{\varepsilon}$ は $\boldsymbol{\varphi}$ (0,1) のランダムサンプルです。これで、時間 Δt の S の値は S の初期値から、2 $\boldsymbol{\Delta t}$ の値は時間 Δt の値から計算できます。

$$S(k\,\Delta t) = S(0)\,\exp\left[\sum_{i=1}^{k}\left(\mu - \frac{\sigma^2}{2}\right)\Delta t + \sigma\epsilon_i\sqrt{\Delta t}\right]$$

k はそれぞれ 1 と M の間です。ここで、各 $\boldsymbol{\varepsilon}_i$ は標準正規分布から得られます。

満期日のヨーロピアン・オプションの値は分かっているため、コールオプションとプットオプションの式は次のようになります。

$$f_{call}(S_T, T) = \max(S_T - X, 0.0)$$
$$f_{put}(S_T, T) = \max(X - S_T, 0.0)$$

モンテカルロを使用して、基礎となる Wiener 過程に対応する (0, 1) 分布の M の数値サンプルを生成した後、各サンプル値に対応する可能な終了時点における株の収益の平均を得ることができます。

$$f_{call}(S_T, T) = \frac{1}{M+1}\sum_{k=0}^{M}\max(S_{k\Delta t} - X, 0)$$

ヨーロピアン・プット・オプション用の同様の関数を得ることもできます。

結果はまだ時間 T のオプションの将来値です。この値を係数 e^{-rT} で割り引いて、ヨーロピアン・コール・オプションの現在値を取得します。

中心極限定理から、標準偏差を半分に減らすにはサンプリング・パスの数を 4 倍にする必要があります。つまり、モンテカルロの標準誤差は次の割合に収束します。

$$O\left(\frac{1}{\sqrt{N}}\right)$$

モンテカルロ・シミュレーションの利点は、基礎となる変数 S が続くパスにペイオフが依存する場合やオプションの期間中にペイオフが複数回発生する状況に使用できることです。ペイオフ関数に複数の独立した変数が含まれる場合は特に役立ちます。また、ほかのすべての解析手法が失敗する場合、モンテカルロが唯一の選択肢になります。

2.3 モンテカルロ・ヨーロピアン・オプション価格評価アルゴリズムの実装

モンテカルロ・ヨーロピアン・オプション価格評価アルゴリズムの実装は、前のセクションのペイオフ関数を利用すると比較的容易になります。まず最初に、φ (0, 1) から乱数を見つける必要があります。次に、ペイオフ関数を使用して基礎となるオプションの期待値と信頼区間を取得します。C/C++ 実装は次のようになります。

```
for (int pos = 0; pos < RAND_N; pos++)
    {
        float callValue =  max(0.0, Sval *exp(MuByT + VBySqrtT *  gen()) - Xval);
        val  += callValue;
        val2 += callValue * callValue;
    }
```

数百万の金融商品のヨーロピアン・オプションを計算しなければいけない状況を考えてみましょう。各商品には、1 セットのパラメーター (時価、権利行使価格およびオプション満期日) があります。モンテカルロ・シミュレーションを使用して、各データセットのヨーロピアン・コール・オプションを計算します。次のセクションでは、265K (266,144) のパス長を使用して 1,500 万オプションのデータセットについてヨーロピアン・コール・オプションの価格を評価するコードを紹介します。

```
#include "MonteCarlo.h"
#include <math.h>
#include <tr1/random>

#ifndef max
  #define max(a,b) (((a) > (b)) ? (a) ; (b))
#endif

void MonteCarlo(
    float *h_CallResult,
    float *h_CallConfidence,
    float *S,
    float *X,
    float *T
)
{
    typedef std::tr1::mt19937                     ENG;    // Mersenne Twister
    typedef std::tr1::normal_distribution<float> DIST;    // 正規分布
    typedef std::tr1::variate_generator<ENG,DIST> GEN;    // 変量ジェネレーター

    ENG  eng;
    DIST dist(0,1);
    GEN  gen(eng,dist);


    for(int opt = 0; opt < OPT_N; opt++)
    {
        float VBySqrtT = VOLATILITY * sqrt(T[opt]);
        float MuByT = (RISKFREE - 0.5 * VOLATILITY * VOLATILITY) * T[opt];
        float Sval = S[opt];
        float Xval = X[opt];
        float val = 0.0, val2 = 0.0;

        for(int pos = 0; pos < RAND_N; pos++)
        {
            float callValue =  max(0.0, Sval *exp(MuByT + VBySqrtT *  gen()) - Xval);
            val  += callValue;
            val2 += callValue * callValue;
        }

        float exprt = exp(-RISKFREE *T[opt]);
        h_CallResult[opt] = exprt * val / (float)RAND_N;
        float stdDev = sqrt(((float)RAND_N * val2 - val * val)/ ((float)RAND_N *
(float)(RAND_N - 1)));
        h_CallConfidence[opt] = (float)(exprt * 1.96 * stdDev / sqrtf((float)RAND_N));
    }
}
```

上記のコードシーケンスは C++ TR1 (C++ Standards Committee Technical Report 1) 拡張の乱数生成機能を使用していることに注意してください。C++ 乱数生成クラスおよび関数は <random> ヘッダーで定義され、std::tr1 名前空間に含まれています。

擬似乱数生成ソフトウェアの中心となるのは、一様に分布されたランダム整数を生成するルーチンです。次に、このルーチンをブートストラップ・プロセスで使用して、一様に分布された浮動小数点数を生成します。一様に分布された浮動小数点シーケンスは、変換、採択棄却アルゴリズムによるほかの分布を生成するために使用されます。

C++ TR1 では、「エンジン」と呼ぶコア・ジェネレーターを選択できます。次の 4 つのエンジンクラスが GCC 4.3.x および Visual Studio* 2008 機能パックでサポートされています。

- linear_congruential
- mersenne_twister
- subtract_with_carry
- subract_with_carry_01

C++ TR1 ライブラリーは、分布クラスによる非一様乱数生成をサポートしています。これらのクラスは、operator() メソッドでランダムサンプルを返します。

variate_generator テンプレート・クラスは、エンジンと分布を保持するオブジェクトを記述し、ラップされたエンジン・オブジェクトを分布オブジェクトの operator() に渡して値を生成します。

GCC 4.4.6 でコンパイルした、モンテカルロ・ヨーロピアン・オプションの初期実装をデュアルソケットのインテル® Xeon® プロセッサー E5-2670 (8 コア、2.60GHz) ベースのホストシステムで実行したところ、計算レートは 34 オプション/秒をわずかに上回る程度でした。

```
[sli@localhost step1]$ ./MonteCarlo
Monte Carlo European Option Pricing in Single Precision
Pricing 32768 options with path length of 262144.
Completed in 955.2525 seconds.
Computation rate -   34.303 options per second.
```

明らかに、この実装には多くの改善の余地があります。次のセクションでは、段階的なフレームワークにより、このプログラムのパフォーマンスを向上します。

3. 段階的な最適化フレームワークとパフォーマンスの最適化

科学的なプロセスと同様に、パフォーマンスの最適化には体系的で構造的なアプローチが必要です。このセクションでは、アプリケーション・パフォーマンスの向上に全体的なアプローチを行うコード近代化フレームワークを取り上げます。このフレームワークの目標は、利用可能な最良のツールとライブラリーを使用して最高のパフォーマンスを達成することです。このフレームワークには 5 つの最適化ステップが含まれます。各ステップは、さまざまな手法を適用した 1 つの項目を使用して、アプリケーションのパフォーマンスを向上します。

この手法の目標は、可能な並列実行リソースをすべて利用して、アプリケーションのパフォーマンスに関連するすべての問題や障害に取り組み、インテル® アーキテクチャーで可能な最高のパフォーマンスを達成することです。

3.1 段階的なコード近代化フレームワーク

段階的なコード近代化フレームワークは、最適化を行うエンジニアができるだけ短い時間で最良のアプリケーション・パフォーマンスを実現すること支援する、5 ステップの並列化プロセスです。プログラムが実行環境のすべての並列ハードウェア・リソースを最大限に活用できるようにします。

- **ステップ 1** では、最適化開発環境を選択します。この環境で、最適化されたコードを作成して、既存の最適化ライブラリーを使用します。
- **ステップ 2** では、実行している演算を最適化します。意図した演算が適切に行われ、ほかに何も行われていないことを保証します。
- **ステップ 3** では、ベクトル化に関する作業を行います。SIMD 命令を活用する方法、アプリケーションで同期データ並列化を追加する方法、コンパイラーが理解できるように表現する方法を説明します。ここでの目標は、1 つのスレッドを使用して 1 つのコアのパフォーマンスを最大限に引き出すことです。
- **ステップ 4** では、スレッド化による並列化を行います。問題を並列化するために同期操作を追加します。最後の目標は、すべてのコアを最大限に活用することです。
- **ステップ 5** では、マルチコアからインテル® MIC にアプリケーションをスケーリングします。高度な並列アプリケーションでは、このステップが特に重要です。このステップの目的は、マイクロアーキテクチャー・レベルの独特な機能 (例えば、メモリー帯域幅と処理能力の違い、SIMD アライメント要件の変更、スレッドあたりのキャッシュなど) を対象にして、それらの機能に基づいて追加の最適化を行うことです。また、あるインテル® アーキテクチャー (インテル® Xeon® プロセッサー) から別のインテル® アーキテクチャー (インテル® Xeon Phi™ コプロセッサー) に実行ターゲットを変更する際に、必要な変更を最小限に抑え、パフォーマンスを最大限に引き出すことも目的です。

3.2 ステップ 1: 最適化ツールとライブラリーの活用

最適化プロジェクトの初めに、最適化開発環境を選択します。このステップで行った決定は、後のステップに大きな影響を及ぼします。得られる結果に影響するだけでなく、作業量を本質的に減らすことが可能です。正しい最適化開発環境を選択することにより、優れたコンパイラー・ツール、最適化されたライブラリー、ランタイムにコードが何を行っているか正確に特定するデバッグ/プロファイリング・ツールを利用できます。

すべての要件を満たす 1 つの環境を見つけることが困難な場合は、異なるツールを組み合わせたソリューションを利用します。例えば、C/C++ コードのパフォーマンス・クリティカルな部分を含む Java* プログラムを最適化する場合は、Java* SDK、C/C++ コンパイラー、JVM 環境、システムワイドのプロファイラーが必要になります。インテル® C++ Composer XE は、マルチコアおよびメニーコア開発環境を統合した、この記事の執筆時点でベクトル化が可能な唯一のアプリケーション・コンパイラーです。プロジェクトにインテル® Xeon® プロセッサーからインテル® Xeon Phi™ コプロセッサーへの高度な並列アプリケーションのスケーリングが含まれる場合、この記事の執筆時点で選択可能な開発ツールはインテル® C++ Composer XE 2013 のみです。統合パフォーマンス・プロファイリング・ツールは、パフォーマンス・モニター・イベントにより最適化を行う必要がある場所を正確に示す最良のツールです。暗闇で目標を照らしてくれる、暗視ゴーグルのようなものです。

インテル® C++ Composer XE 2013 for Linux* は、RHEL Server 6.3 でリリースされた GCC 4.4.6 と互換性があります。g++ でコンパイルしているものは icpc (インテル® C++ Composer に含まれる g++ と等価なコンパイラー) でコンパイルできます。例えば、g++ -o MonteCarlo -O2 MonteCarloStage1.cpp の代わりに、icpc -o MonteCarlo -O2 MonteCarloStage1.cpp を使用できます。インテル® C++ Composer で作成したバイナリーを実行すると、ソースコードを変更することなく、パフォーマンスは 1.37 倍になります。我々の目標は、正しい計算を行い、パフォーマンスを向上することです。

```
[sli@localhost step1]$ make -B CXX=icpc
icpc -c -O2  -o Driver.o Driver.cpp
icpc -c -O2  -o MonteCarloStep1.o MonteCarloStep1.cpp
icpc Driver.o MonteCarloStep1.o -o MonteCarlo
[sli@cthor-knc14 step1]$ ./MonteCarlo
Monte Carlo European Option Pricing in Single Precision
Pricing 32768 options with path length of 262144.
Completed in 694.6562 seconds.
Computation rate -   47.172 options per second.
```

インテル® MKL は、BLAS、LAPACK、乱数生成関数などの基本的な数学ルーチンからなるプリコンパイル済みライブラリーです。インテル® MKL は、これらの関数向けの C と Fortran の両方のインターフェイスを提供しています。C/C++ または Fortran からこれらの関数を呼び出すことができます。

乱数生成 (RNG) の効率は、モンテカルロ・シミュレーションのパフォーマンス考察の指標です。C++ TR1 と同様に、インテル® MKL にも乱数生成ルーチンが含まれています。C++ TR1 のように、インテル® MKL は、64 ビットおよび 32 ビットの 2 つの浮動小数点と 2 つの整数データを掛けた異なる基本的な RNG エンジンと異なる分布を個別に選択することもできます (適用可能な場合)。C++ TR1 とは異なり、インテル® MKL の RNG インターフェイスは C/C++ および Fortran の両方で使用できます。ほかのコンパイル済みライブラリーと同様に、最初にインターフェイス宣言 .h ファイルをインクルードして、C/C++ 実装ファイルでインテル® MKL API 呼び出しを行った後、プリコンパイル済みインテル® MKL ライブラリーとリンクします。例では、3 つのインテル® MKL ライブラリー (mkl_intel_lp64、mkl_sequential、mkl_core) を含める必要があります。これらのライブラリーは、-lmkl_intel_lp64 -lmkl_sequential -lmkl_core -lpthread のように、-lpthread とともに指定するか、リンカーに -mkl を渡してリンカーで処理します。

主な違いは、C++ TR1 では、乱数が variate_generator クラスからの gen() メソッド呼び出しで一度に得られることです。一方、インテル® MKL では、RNG エンジン・オブジェクトの作成プロセスが RNG ストリームの作成に置き換えられ、分布オブジェクトの作成プロセスが正しい RNG インターフェイス API 呼び出しの選択に置き換えられます。1 回のインテル® MKL RNG インターフェイス呼び出しで任意の数の乱数を扱うことができます。この問題では、1 回の RNG インターフェイス呼び出しで 256K の乱数を扱います。


```
#include "MonteCarlo.h"
#include "math.h"
#include "mkl_vsl.h"
#define RANDSEED 123

#ifndef max
  #define max(a,b) (((a) > (b)) ? (a) : (b))
#endif

void MonteCarlo(
    float *h_CallResult,
    float *h_CallConfidence,
    float *S,
    float *X,
    float *T
)
{
    float random [RAND_N];
    VSLStreamStatePtr Randomstream;
    vslNewStream(&Randomstream, VSL_BRNG_MT19937, RANDSEED);

    vsRngGaussian (VSL_METHOD_SGAUSSIAN_ICDF, Randomstream, RAND_N, random, 0.0, 1.0);

    for(int opt = 0; opt < OPT_N; opt++)
    {
        float VBySqrtT = VOLATILITY * sqrt(T[opt]);
        float MuByT = (RISKFREE - 0.5*VOLATILITY*VOLATILITY)*T[opt];
        float Sval = S[opt];
        float Xval = X[opt];
        float val = 0.0, val2 = 0.0;

        for(int pos = 0; pos < RAND_N; pos++)
```

```
        {
            float callValue =  max(0.0, Sval*exp(MuByT+VBySqrtT*random[pos])-Xval);
            val  += callValue;
            val2 += callValue * callValue;
        }

        float exprt = exp(-RISKFREE *T[opt]);
        h_CallResult[opt] = exprt * val / (float)RAND_N;
        float stdDev = sqrt(((float)RAND_N*val2-val*val)/((float)RAND_N*(float)(RAND_N-
1)));
        h_CallConfidence[opt] = (float)(exprt * 1.96 * stdDev / sqrtf((float)RAND_N));
    }
    vslDeleteStream(&Randomstream);
}
```

インテル® MKL の RNG を使用することにより、コードの行数は少なくなりました。パフォーマンスは大きく向上し、オリジナルコードの 5.53 倍以上になりました。

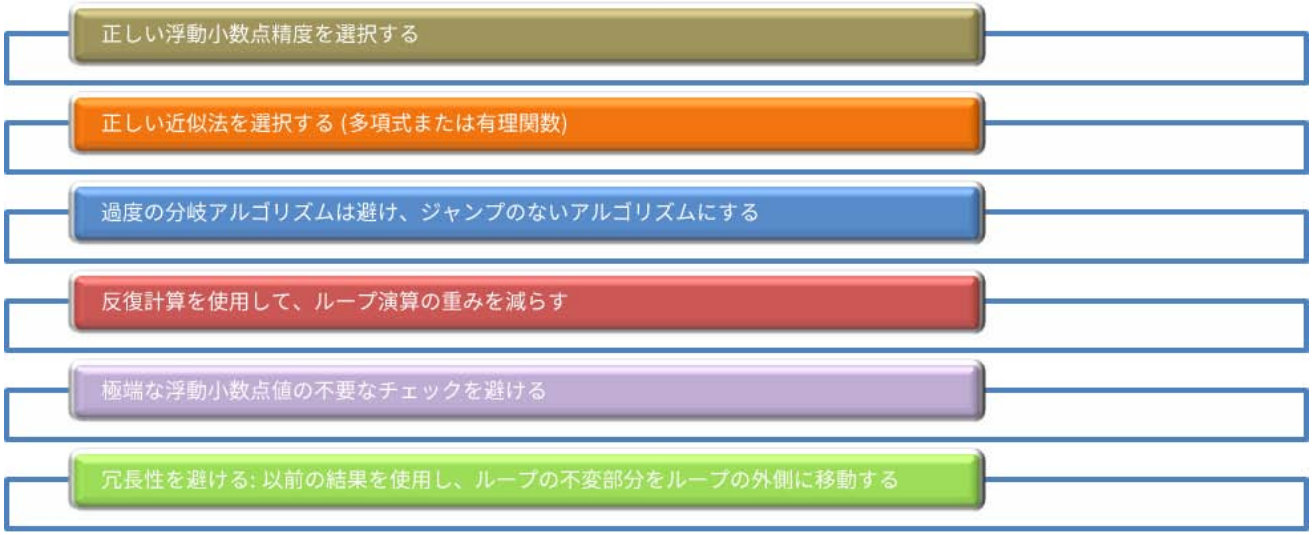
```
[sli@cthor-knc14 .solution]$ ./MonteCarlo
Monte Carlo European Option Pricing in Single Precision
Pricing 32768 options with path length of 262144.
Completed in 125.5210 seconds.
Computation rate -  261.056 options per second.
```

C++ TR1 からインテル® MKL に RNG を変更することにより、パフォーマンスが 5.53 倍になっただけでなく、コンパイラーでインライン関数呼び出しを行うことが可能になり、内側のループのコードもベクトル化されました (ステップ 3 で説明)。

最適化ソフトウェア開発環境として利用可能な最適化ソフトウェアを最大限に活用した結果、多くの時間が節約され、独自の IP の開発や新しい問題の作業に取り組むことができるようになりました。さらにパフォーマンスを向上するには、ソースコードに変更を加える必要があります。並列メカニズムを適用する前に、まずシリアルコードが適切であることを確認しましょう。

3.3 ステップ 2: スカラー/シリアル最適化

スカラー/シリアル最適化を使用して、冗長性がないことを調べ、最高の効率でコードを実行していることを確認します。このステップでは、問題に適した (過剰でない) データ構造および数値メソッドの精度を選択していることを確認します。必要な場合は、単精度のような低コストの演算を高コストの倍精度演算に置換します。



C/C++ は、C の初期からある弱い型付けをまだ継承しています。自動プロモーションは特定のケースでは高精度になると考えられています。しかし、たとえ精度が向上した場合でも、パフォーマンスを向上する可能性が失われることは明白です。スカラー/シリアル最適化段階で注目すべきポイントを次にリストします。

モンテカルロ・アプリケーションには、1 つの超越関数呼び出し (内側のループ内にある自然ベースの指数関数) が含まれています。この関数呼び出しを回避できれば、大きなパフォーマンスの向上につながることが予想されます。

```
#include "MonteCarlo.h"
#include "math.h"
#include "mkl_vsl.h"
#define RANDSEED 123

#ifndef max
  #define max(a,b) (((a) > (b)) ? (a) : (b))
#endif

static const float RVV = RISKFREE-0.5f*VOLATILITY*VOLATILITY;
static const float INV_RAND_N = 1.0f/RAND_N;
static const float F_RAND_N = static_cast<float>(RAND_N);
static const float STDDEV_DENOM = 1 / (F_RAND_N * (F_RAND_N - 1.0f));
static const float CONFIDENCE_DENOM = 1 / sqrtf(F_RAND_N);

void MonteCarlo(
    float *h_CallResult,
    float *h_CallConfidence,
    float *S,
    float *X,
    float *T
)
{
    float random [RAND_N];
    VSLStreamStatePtr Randomstream;
    vslNewStream(&Randomstream, VSL_BRNG_MT19937, RANDSEED);

    vsRngGaussian (VSL_METHOD_SGAUSSIAN_ICDF, Randomstream, RAND_N, random, 0.0f, 1.0f);

    for(int opt = 0; opt < OPT_N; opt++)
    {
        float VBySqrtT = VOLATILITY * sqrtf(T[opt]);
        float MuByT = RVV * T[opt];
        float Sval = S[opt];
        float Xval = X[opt];
        float val = 0.0, val2 = 0.0;

        for(int pos = 0; pos < RAND_N; pos++)
        {
            float callValue =  max(0.0, Sval *expf(MuByT + VBySqrtT *  random[pos]) -
Xval);
            val  += callValue;
            val2 += callValue * callValue;
        }
```

```
        float exprt = expf(-RISKFREE *T[opt]);
        h_CallResult[opt] = exprt * val * INV_RAND_N;
        float stdDev = sqrtf((F_RAND_N * val2 - val * val)* STDDEV_DENOM);
        h_CallConfidence[opt] = (exprt * 1.96f * stdDev * CONFIDENCE_DENOM);
    }
    vslDeleteStream(&Randomstream);
}
```

定数で分割する演算をループの外に移動したことに注意してください。これらの演算は内側のループの外にあり、コンパイラーが最適化できなかったものです。パフォーマンスは 41.11% 向上しました。

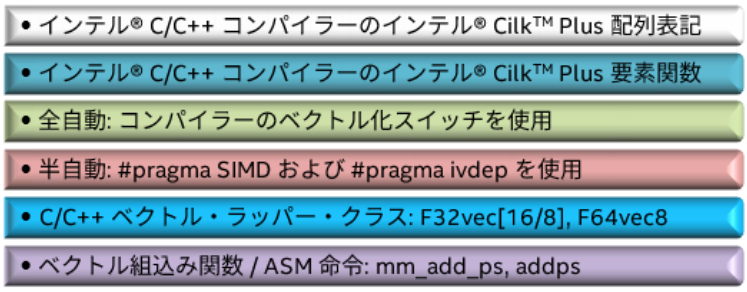
```
[sli@localhost step3]$ make -B
icpc -c -O3 -ipo -fimf-precision=low -fimf-domain-exclusion=31 -fimf-accuracy-bits=11 -no-
prec-div -no-prec-sqrt -o Driver.o Driver.cpp
icpc -c -O3 -ipo -fimf-precision=low -fimf-domain-exclusion=31 -fimf-accuracy-bits=11 -no-
prec-div -no-prec-sqrt -o MonteCarloStep2.o MonteCarloStep2.cpp
icpc Driver.o MonteCarloStep2.o -o MonteCarlo -mkl
[sli@cthor-knc14 .solution]$ ./MonteCarlo
Monte Carlo European Option Pricing in Single Precision
Pricing 32768 options with path length of 262144.
Completed in  88.9513 seconds.
Computation rate -  368.381 options per second.
```

向上率が低いのは、コードをスカラー/シリアルモードで実行している間は、これ以上パフォーマンスを向上する余地がほとんどないためです。そのため、並列処理に目を向ける必要があります。

3.4 ステップ 3: ベクトル化

ベクトル化は、状況によって意味が異なることがあります。この記事では、ベクトル化とは、CPU 内部のベクトルレジスターを使用する、コンパイラーが生成する SIMD 命令を意味します。ここでは、多くのデータ要素を備えた一部の命令の同期実行を活用します。

プロセッサーの組込み関数を使用する方法から、インテル® Cilk™ Plus の配列表記 (アレイ・ノーテーション) を使用する方法まで、さまざまな方法でプログラムにベクトル化を実装することができます。コンパイラー・ベースのベクトル化手法は、生成するコードに対するプログラマーの制御の割合、構文の表現力、シリアルプログラムに要求する変更の量の点で異なります。



コンパイラーでシリアルコードをベクトル化して SIMD 命令を生成する前に、メモリー・アライメントが適切に行われていることを保証する必要があります。メモリー・アライメントが適切に行われていないと、最悪の場合はプロセッサー・フォルトが発生します。動作に影響がない場合でも、キャッシュラインが分割され、オブジェクト・コードが冗長になることで、アプリケーションのパフォーマンスは大幅に低下します。適切なメモリー・アライメントを保証する 1 つの方法は、明示的なアライメント要件を常に要求して操作することです。インテル® C++ Composer XE 2011 では、__attribute__(align(64)) をメモリー定義の前に指定することにより、静的に割り当てられたメモリーを要求することができます。64 バイト境界は、インテル® Xeon Phi™ コプロセッサーのベクトルレジスターで指定されるメモリーの最小アライメント要件です。_mm_malloc

および _mm_free を使用して、動的に割り当てられたメモリーを要求および解放することもできます。Linux* 用インテル® コンパイラーと GCC では、メモリーの割り当て組込み関数がサポートされています。

関数呼び出しを行うとオーバーヘッドが発生するため、インテル® コンパイラー・ベースのベクトル化は、ホットなループの関数呼び出しが最小限に抑えられている場合に最も適切に動作します。すべての関数呼び出しで CPU のベクトルエンジンが SIMD モードでの実行を継続できるとは限りません。関数呼び出しをインライン展開することで関数呼び出しのオーバーヘッドを回避でき、コンパイラー・ベースのベクトライザーで呼び出し元のコードと呼び出し先のコードをともに解析してベクトル化できるため、できるだけ関数呼び出しをインライン展開します。

```
#include "MonteCarlo.h"
#include "math.h"
#include "mkl_vsl.h"
#define RANDSEED 123

static const float  RVV = RISKFREE-0.5f*VOLATILITY*VOLATILITY;
static const float  INV_RAND_N = 1.0f/RAND_N;
static const float  F_RAND_N = static_cast<float>(RAND_N);
static const float STDDEV_DENOM = 1 / (F_RAND_N * (F_RAND_N - 1.0f));
static const float CONFIDENCE_DENOM = 1 / sqrtf(F_RAND_N);

void MonteCarlo(
    float *h_CallResult,
    float *h_CallConfidence,
    float *S,
    float *X,
    float *T
)
{
    __attribute__((align(4096))) float random [RAND_N];
    VSLStreamStatePtr Randomstream;
    vslNewStream(&Randomstream, VSL_BRNG_MT19937, RANDSEED);

    vsRngGaussian (VSL_METHOD_SGAUSSIAN_ICDF, Randomstream, RAND_N, random, 0.0f, 1.0f);
    for(int opt = 0; opt < OPT_N; opt++)
    {
        float VBySqrtT = VOLATILITY * sqrtf(T[opt]);
        float MuByT = RVV * T[opt];
        float Sval = S[opt];
        float Xval = X[opt];
        float val = 0.0, val2 = 0.0;

#pragma vector aligned
#pragma simd reduction(+:val) reduction(+:val2)
#pragma unroll(4)
        for(int pos = 0; pos < RAND_N; pos++)
        {
            float callValue  = Sval * expf(MuByT + VBySqrtT * random[pos]) - Xval;
            callValue = (callValue > 0) ? callValue : 0;
            val  += callValue;
            val2 += callValue * callValue;
        }

        float exprt = expf(-RISKFREE *T[opt]);
        h_CallResult[opt] = exprt * val * INV_RAND_N;
        float stdDev = sqrtf((F_RAND_N * val2 - val * val)* STDDEV_DENOM);
        h_CallConfidence[opt] = (exprt * 1.96f * stdDev * CONFIDENCE_DENOM);
    }
    vslDeleteStream(&Randomstream);
}
```

```
#include <stdlib.h>
#include <stdio.h>
#include <sys/time.h>
#include <string.h>
#include <math.h>
```

```
#include "MonteCarlo.h"
#include <iostream>
using namespace std;

#define SIMDALIGN 64

double second()
{
    struct timeval tv;
    gettimeofday(&tv, NULL);
    return (double)tv.tv_sec + (double)tv.tv_usec / 1000000.0;
}

inline float RandFloat(float low, float high){
    float t = (float)rand() / (float)RAND_MAX;
    return (1.0f - t) * low + t * high;
}

////////////////////////////////////////////////////////////////////////////////
// 累積正規分布関数の多項式近似
////////////////////////////////////////////////////////////////////////////////

double CND(double d){
    const double       A1 = 0.31938153;
    const double       A2 = -0.356563782;
    const double       A3 = 1.781477937;
    const double       A4 = -1.821255978;
    const double       A5 = 1.330274429;
    const double RSQRT2PI = 0.39894228040143267793994605993438;

    double
        K = 1.0 / (1.0 + 0.2316419 * fabs(d));

    double
        cnd = RSQRT2PI * exp(- 0.5 * d * d) *
        (K * (A1 + K * (A2 + K * (A3 + K * (A4 + K * A5)))));

    if(d > 0)
        cnd = 1.0 - cnd;

    return cnd;
}

void BlackScholesFormula(
    double& callResult,
    double Sf, // 株価
    double Xf, // 権利行使価格
    double Tf, // 期間
    double Rf, // リスクフリー・レート
    double Vf  // ボラティリティー・レート
){
    double S = Sf, X = Xf, T = Tf, R = Rf, V = Vf;

    double sqrtT = sqrt(T);
    double    d1 = (log(S / X) + (R + 0.5 * V * V) * T) / (V * sqrtT);
    double    d2 = d1 - V * sqrtT;
    double CNDD1 = CND(d1);
    double CNDD2 = CND(d2);

    double expRT = exp(- R * T);
    callResult   = (S * CNDD1 - X * expRT * CNDD2);
}

int main(int argc, char* argv[])
{
    float
        *CallResultParallel,
```

```
        *CallConfidence,
        *StockPrice,
        *OptionStrike,
        *OptionYears;
    double
         sTime, eTime;
    int
        mem_size, rand_size, verbose = 0;

    const int RAND_N = 1 << 18;

    if (argc > 2)
    {
         printf("usage: MonteCarlo <verbose>  where verbose = 1 for validtating result,
the default is not to validate result. \n");
         exit(1);
    }
    if (argc == 1)
         verbose = 0;
    if (argc == 2)
         verbose = atoi(argv[1]);

    printf("Monte Carlo European Option Pricing in Single Precision\n");
    mem_size = sizeof(float)*OPT_N;
    rand_size = sizeof(float)*RAND_N;

    CallResultParallel = (float *)_mm_malloc(mem_size, SIMDALIGN);
    CallConfidence     = (float *)_mm_malloc(mem_size, SIMDALIGN);
    StockPrice         = (float *)_mm_malloc(mem_size, SIMDALIGN);
    OptionStrike       = (float *)_mm_malloc(mem_size, SIMDALIGN);
    OptionYears        = (float *)_mm_malloc(mem_size, SIMDALIGN);

    if (verbose)
    {
        printf("...generating the input data.\n");
    }
    for(int i = 0; i < OPT_N; i++)
    {
        CallResultParallel[i] = 0.0;
        CallConfidence[i]= -1.0;
        StockPrice[i]    = RandFloat(5.0f, 50.0f);
        OptionStrike[i]  = RandFloat(10.0f, 25.0f);
        OptionYears[i]   = RandFloat(1.0f, 5.0f);
    }

    printf("Pricing %d options with path length of %d.\n", OPT_N, RAND_N);
    sTime = second();
    MonteCarlo(
        CallResultParallel,
        CallConfidence,
        StockPrice,
        OptionStrike,
        OptionYears);
    eTime = second();
    printf("Completed in %8.4f seconds.\n",eTime-sTime );
    printf("Computation rate - %8.3f options per second.\n", OPT_N/(eTime-sTime));

    if (verbose)
    {
        double
           delta, sum_delta, sum_ref, L1norm, sumReserve;
        double CallMaster;

        sum_delta = 0;
        sum_ref   = 0;
        sumReserve = 0;

        for(int i = 0; i < OPT_N; i++)
```

```
        {
            BlackScholesFormula(CallMaster,
                (double) StockPrice[i],
                (double) OptionStrike[i],
                (double) OptionYears[i],
                (double) RISKFREE,
                (double) VOLATILITY);
            delta = fabs(CallMaster - CallResultParallel[i]);
            sum_delta += delta;
            sum_ref   += fabs(CallMaster);
            if(delta > 1e-6)
                sumReserve += CallConfidence[i] / delta;
        }
        sumReserve /= (double)OPT_N;
        L1norm = sum_delta / sum_ref;
        printf("L1 norm: %E\n", L1norm);
        printf("Average reserve: %f\n", sumReserve);

        printf("...freeing CPU memory.\n");
        printf((sumReserve > 1.0f) ? "PASSED\n" : "FAILED\n");
    }
    _mm_free(CallResultParallel);
    _mm_free(CallConfidence);
    _mm_free(StockPrice);
    _mm_free(OptionStrike);
    _mm_free(OptionYears);
    return 0;
}
```

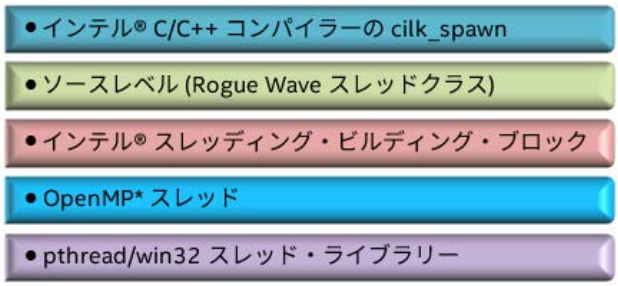
```
icpc -c -O3 -ipo -xAVX -fimf-precision=low -fimf-domain-exclusion=31 -fimf-accuracy-
bits=11 -no-prec-div -no-prec-sqrt -fno-alias -vec-report2 -o Driver.o Driver.cpp
icpc: リマーク #10346: 最適化レポートはプロシージャー間の最適化を実行したリンク時に使用できます。
icpc -c -O3 -ipo -xAVX -fimf-precision=low -fimf-domain-exclusion=31 -fimf-accuracy-
bits=11 -no-prec-div -no-prec-sqrt -fno-alias -vec-report2 -o MonteCarloStep4.o
MonteCarloStep4.cpp
icpc: リマーク #10346: 最適化レポートはプロシージャー間の最適化を実行したリンク時に使用できます。
icpc Driver.o MonteCarloStep4.o -o MonteCarlo -mkl
Driver.cpp(130): (列 5) リマーク: ループはベクトル化されませんでした: ベクトル依存関係が存在しています。
Driver.cpp(141): (列 5) リマーク: SIMD ループがベクトル化されました。
Driver.cpp(141): (列 5) リマーク: ループはベクトル化されませんでした: 内部ループではありません。
Driver.cpp(161): (列 9) リマーク: ループがベクトル化されました。
[sli@localhost step4]$ ./MonteCarlo
Monte Carlo European Option Pricing in Single Precision
Pricing 32768 options with path length of 262144.
Completed in  10.9624 seconds.
Computation rate - 2989.117 options per second.
```

ベクトル化によるパフォーマンスの向上は 8.11 倍でした。これは予想値の 8 倍をわずかに上回るものです。ほかのコンパイラー機能の変更、インテル® MKL の追加、シリアル最適化、ベクトル化を組み合わせることにより、パフォーマンスの向上は約 87.14 倍になりました。1 つのコアで 1 つのスレッドのみを実行する場合に必要な処理は、これですべて行いました。

3.5 ステップ 4: 並列化

最近のマイクロプロセッサーはすべてマルチコアです。ハイパースレッドから、デュアルコア、クアッドコア、8 コアへと発展し、10 コアや 12 コアも一般的になっています。インテル® Xeon Phi™ コプロセッサー・ファミリーの最初の製品は、最大 61 コアでした。現在では、コア数によりスケーリングするプログラムを設計することがハイパフォーマンス・ソフトウェア開発の要件となっています。一般的には、ジョブ全体を多くの小さなタスクに分割し、これらの小さなタスクを実行する小さな軽いプロセスやスレッドを暗黙的または明示的に作成して、各スレッドをハードウェア・スレッドや OS がスケジュール可能なエンティティーにバインドします。この処理はスレッド化やマルチスレッド化として知られ、この処理を行うプログラムはマルチスレッド・プログラムと呼ばれます。

インテル® C++ Composer XE 2013 では、制御の度合いや問題の要件に応じて、マルチスレッド・プログラムを作成するさまざまな方法を選択できます。



このモンテカルロ実装のスレッド化では、ステップ 3 でベクトル化したシリアルコードをできる限り変更しない方法を利用します。このモンテカルロ・ヨーロピアン・オプション・プログラムは、サブタスクへの分割も比較的容易です。約 32K のデータセットがあるため、デュアルソケットのインテル® Xeon® プロセッサーに合わせて、コードを 16 のサブタスクに分割し、各タスクで 2K のデータセットを処理します。この条件には、OpenMP* が非常に適しています。

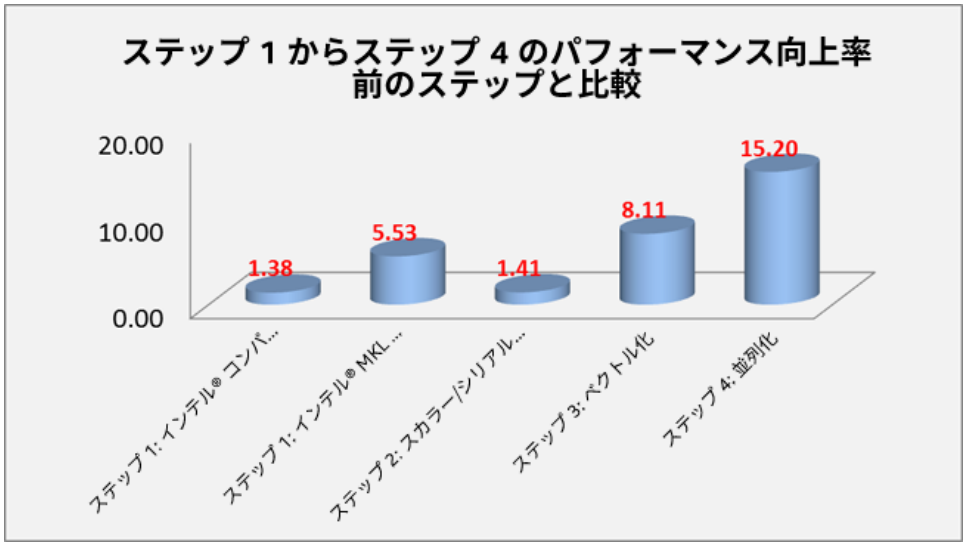
```
#include "MonteCarlo.h"

#include "math.h"
#include "mkl_vsl.h"
#define RANDSEED 123

static const float  RVV = RISKFREE-0.5f*VOLATILITY*VOLATILITY;
static const float  INV_RAND_N = 1.0f/RAND_N;
static const float  F_RAND_N = static_cast<float>(RAND_N);
static const float STDDEV_DENOM = 1 / (F_RAND_N * (F_RAND_N - 1.0f));
static const float CONFIDENCE_DENOM = 1 / sqrtf(F_RAND_N);

void MonteCarlo(
    float *h_CallResult,
    float *h_CallConfidence,
    float *S,
    float *X,
    float *T
)
{
    __attribute__((align(4096))) float random [RAND_N];
    VSLStreamStatePtr Randomstream;
    vslNewStream(&Randomstream, VSL_BRNG_MT19937, RANDSEED);

    vsRngGaussian (VSL_METHOD_SGAUSSIAN_ICDF, Randomstream, RAND_N, random, 0.0f, 1.0f);
#pragma omp parallel for
    for(int opt = 0; opt < OPT_N; opt++)
    {
        float VBySqrtT = VOLATILITY * sqrtf(T[opt]);
        float MuByT = RVV * T[opt];
        float Sval = S[opt];
        float Xval = X[opt];
        float val = 0.0, val2 = 0.0;

#pragma vector aligned
#pragma simd reduction(+:val) reduction(+:val2)
#pragma unroll(4)
        for(int pos = 0; pos < RAND_N; pos++)
        {
            float callValue  = Sval * expf(MuByT + VBySqrtT * random[pos]) - Xval;
            callValue = (callValue > 0) ? callValue : 0;
            val  += callValue;
            val2 += callValue * callValue;
        }
```

```
        float exprt = expf(-RISKFREE *T[opt]);
        h_CallResult[opt] = exprt * val * INV_RAND_N;
        float stdDev = sqrtf((F_RAND_N * val2 - val * val)* STDDEV_DENOM);
        h_CallConfidence[opt] = (exprt * 1.96f * stdDev * CONFIDENCE_DENOM);
    }
    vslDeleteStream(&Randomstream);
}
```

```
[sli@localhost step5]$ make -B
icpc -c -g -O3 -ipo -openmp -xAVX -fimf-precision=low -fimf-domain-exclusion=31 -fimf-
accuracy-bits=11 -no-prec-div -no-prec-sqrt -fno-alias -vec-report2 -o Driver.o Driver.cpp
icpc: リマーク #10346: 最適化レポートはプロシージャー間の最適化を実行したリンク時に使用できます。
icpc -c -g -O3 -ipo -openmp -xAVX -fimf-precision=low -fimf-domain-exclusion=31 -fimf-
accuracy-bits=11 -no-prec-div -no-prec-sqrt -fno-alias -vec-report2 -o MonteCarloStep5.o
MonteCarloStep5.cpp
icpc: リマーク #10346: 最適化レポートはプロシージャー間の最適化を実行したリンク時に使用できます。
icpc Driver.o MonteCarloStep5.o -o MonteCarlo -L /opt/intel/composerxe/mkl/lib/intel64 -
lmkl_intel_lp64 -lmkl_sequential -lmkl_core -lpthread -openmp
Driver.cpp(130): (列 5) リマーク: ループはベクトル化されませんでした: ベクトル依存関係が存在しています。
Driver.cpp(161): (列 9) ループはベクトル化されませんでした: ベクトル依存関係が存在しています。
MonteCarloStep5.cpp(69): (列 9) リマーク: SIMD ループがベクトル化されました。
MonteCarloStep5.cpp(58): (列 5) リマーク: ループはベクトル化されませんでした: 内部ループではありません。
[sli@localhost step5]$ setenv KMP_AFFINITY "compact,granularity=fine"
[sli@localhost step5]$ ./MonteCarlo
Monte Carlo European Option Pricing in Single Precision
Pricing 1998848 options with path length of 262144.
Completed in  43.9896 seconds.
Computation rate - 45439.090 options per second.
```

2 ソケットの Sandy Bridge システム (2.60GHz) で、パフォーマンスは 15.20 倍に向上しました。16 コア、32 スレッドの実行環境でパフォーマンスが 15.20 倍に向上したことは、各スレッドが実行するコードの多くが並列部分に含まれることを表しています。スレッド・アフィニティーを共有キャッシュの効率が最大限になるコンパクトモードに設定し、粒度セットを細粒度に設定する必要があることに注意してください。最終的には、最初に GCC 4.4.6 でコンパイルしたベースラインから、パフォーマンスは 1324.6 倍に向上しました。

次の表は、各テクノロジーを適用したときのパフォーマンスの向上を要約したものです。値は、これらのテクノロジーの適用前と適用後の比率を示しています。例えば、インテル® MKL を追加することにより、インテル® コンパイラーを使用したベースラインからパフォーマンスは 5.53 倍に向上しました。



3.6 ステップ 5: インテル製マルチコア・プロセッサーからインテル® MIC アーキテクチャーへのスケーリング

モンテカルロのような高度な並列アプリケーションは、インテル® MIC アーキテクチャー・ベースのインテル® Xeon Phi™ コプロセッサーの恩恵を直ちに得ることができます。インテル® MIC アーキテクチャー・ベースの最初の製品であるこのコプロセッサーには、最大 61 のコアと 244 のスレッドが搭載されており、スケーラブルなソフトウェアのパフォーマンスをこれまでにないレベルに引き上げます。61 のコアにはそれぞれ、512 ビットの SIMD エンジンが含まれており、8 つの 64 ビット・データまたは 16 の 32 ビット・データ (整数または浮動小数点) を処理できます。

3.6.1 インテル® MIC アーキテクチャー向けリビルド

インテル® C++ Composer XE 2013 の最も重要な機能の 1 つは、インテル製マルチコア・プロセッサーとインテル® MIC アーキテクチャー開発環境の統合です。インストールするツールスイートは 1 つです。ネイティブ・マルチコア・アプリケーションをコンパイルして実行するか、同じソースコードをコンパイルしてインテル® Xeon Phi™ コプロセッサー用の実行ファイルをビルドします。

段階的な最適化フレームワークに従って得られたソフトウェアを再コンパイルするだけで、コプロセッサー上でネイティブに実行するバイナリーファイルを作成することができます。

```
icpc -c -g -O3 -ipo -openmp -mmic -fimf-precision=low -fimf-domain-exclusion=31 -fimf-
accuracy-bits=11 -no-prec-div -no-prec-sqrt -fno-alias -vec-report2 -opt-threads-per-
core=4 -o Driver.o Driver.cpp
icpc: リマーク #10346: 最適化レポートはプロシージャー間の最適化を実行したリンク時に使用できます。
icpc -c -g -O3 -ipo -openmp -mmic -fimf-precision=low -fimf-domain-exclusion=31 -fimf-
accuracy-bits=11 -no-prec-div -no-prec-sqrt -fno-alias -vec-report2 -opt-threads-per-
core=4 -o MonteCarloStep5.o MonteCarloStep5.cpp
icpc: リマーク #10346: 最適化レポートはプロシージャー間の最適化を実行したリンク時に使用できます。
icpc -mmic Driver.o MonteCarloStep5.o -o MonteCarlo -L /opt/intel/composerxe/mkl/lib/mic -
lmkl_intel_lp64 -lmkl_sequential -lmkl_core -lpthread -openmp
Driver.cpp(130): (列 5) リマーク: ループはベクトル化されませんでした: ベクトル依存関係が存在しています。
Driver.cpp(161): (列 9) ループはベクトル化されませんでした: ベクトル依存関係が存在しています。
MonteCarloStep5.cpp(69): (列 9) リマーク: SIMD ループがベクトル化されました。
MonteCarloStep5.cpp(58): (列 5) リマーク: ループはベクトル化されませんでした: 内部ループではありません。
```

以前の makefile で作成したモンテカルロのバイナリーを実行するには、scp を使用してホストシステムからカードにコピーする必要があります。また、3 つのインテル® MKL 共有ライブラリーもコピーする必要があります。その後、ssh を使用してデバイスの OS コマンドシェルを開く必要があります。正しい OpenMP* ランタイム動作を保証するには、一部の環境変数を設定する必要があります。

```
[sli@cthor-knc14 step5]$ scp MonteCarlo cthor-knc14-mic1:
MonteCarlo                                                    100%  493KB 492.8KB/s   00:00
[sli@cthor-knc14 step5]$ ssh cthor-knc14-mic1
~ $ export LD_LIBRARY_PATH=.
~ $ export KMP_AFFINITY="compact,granularity=fine"
~ $ ./MonteCarlo
Monte Carlo European Option Pricing in Single Precision
Pricing 1998848 options with path length of 262144.
Completed in   4.9851 seconds.
Computation rate - 400964.798 options per second.
```

ホストシステムで完了までに 44 秒かかっていた同じプログラムが 5 秒未満で完了するようになりました。これだけで、パフォーマンスが 8.82 倍向上したことになります。このセクションでは、アプリケーションのパフォーマンスをさらに向上する方法を調べます。

3.6.2 ターゲット・アーキテクチャー向けの最適化

インテル® アーキテクチャー (IA) の異なる実装は、ISA 間でソフトウェアを移動できる、独自のパフォーマンス・プロファイルを含む ISA を作成します。インテルは、コプロセッサーに拡張数学ユニットと呼ばれるハードウェア関数近似ユニットを実装しました。log2()、exp2()、rsqrt()、rcp() の 4 つの初等関数はハードウェアに直接実装されます。スループットは 1-2 サイクル、レイテンシーは 4-8 サイクルです。高速初等関数は、初等関数と単純な変換を組み合わせて 4 つのほかの関数を直接高速化できます。

例えば、基数 2 の指数と基数 e の指数は基数の変換式と呼ばれる方程式で次のように関連付けられています。

$e^x = 2^{x*\log_2 E}$

$\ln(x) = \log_2(x)/\log 2E = \log 2(x)*\ln 2 = \log 2(x)*M\_LN2$

$\exp(x) = \exp_2(x * \log 2E) = \exp 2(x * M\_LOG2E)$

LOG_2E (e の基数 2 の対数) は、math.h で M_LOG2E として定義されている定数です。

初等関数	レイテンシー	スループット	導関数	レイテンシー	スループット
exp2()	8	2	pow()	16	4
log2()	4	1	sqrt()	8	2
rcp()	4	1	div()	8	2
rsqrt()	4	1	ln()	8	2

pow() は exp2()、乗算、log2() から、sqrt() は rsqrt() と rcp() から、div() は rcp() と乗算から、それぞれ構成されます。

これは、モンテカルロ・コードを変更する良い機会です。ここでは、これらの関数の基数 2 のバージョンが呼び出されるようにコードを変更します。

```
        float VBySqrtT = VLOG2E * sqrtf(T[opt]);
        float MuByT = RVVLOG2E * T[opt];
        float Sval = S[opt];
        float Xval = X[opt];
        float val = 0.0, val2 = 0.0;

#pragma vector aligned
#pragma simd reduction(+:val) reduction(+:val2)
#pragma unroll(4)
        for(int pos = 0; pos < BLOCKSIZE; pos++)
        {
            float callValue  = Sval * exp2f(MuByT + VBySqrtT * random[pos]) - Xval;
            callValue = (callValue > 0) ? callValue : 0;
            val  += callValue;
            val2 += callValue * callValue;
        }
```

このケースでは、exp2f() の内側に 2 つの式があり、両方の式を調整して、内側のループの外に調整を反映する必要があります。

必要なのは、exp(x) と log(x) を exp2(x*M_LOG2E) と LOG2(x)*M_LN2 (または LOG2(x)/M_LOG2E) に置換することのみと考えるかもしれません。しかし、追加の乗算があるため、EMU 関数の利点をすべて引き出すには、まだ十分ではありません。ほとんどのケースでは、内側のループの外側の乗算をあらかじめ調整できれば、乗算のコストが抑えられ、EMU 関数を使用することによるパフォーマンスの利点を活用することができます。

3.6.3 メモリー・アクセス・パターン最適化

モンテカルロの現在の実装は、すべてのスレッドが事前に生成された 256K の乱数 (単精度浮動小数点用の 1MB のフットプリントを含む) を共有しているため、これらの乱数はワーカースレッドの L2 キャッシュに収まりません。各スレッドは、データの一部のみキャッシュに入れて、現在の部分を完了すると、L2 ミスイベントを生成します。その結果、リング・インターコネクトはより多くのデータを L2 キャッシュに入れようとします。これらのデータのアクセスが各スレッドで調整されていない場合、各スレッドが 1MB のデータの一部にアクセスし、244 スレッドすべての L2 キャッシュミスに対応するためリング・インターコネクトがビジーになります。リング・インターコネクトの帯域幅は、わずか 1MB のデータが含まれる場合でもボトルネックになります。調整されていないメモリーアクセスでは、1MB のデータ取得が 244 回行なわれ、244MB 相当の帯域幅が要求されます。

調整されたアクセスでは、最初のスレッドがデータを要求したときにデータをメモリーから 1 回のみ取得することが保証されます。取得したデータは、リング・インターコネクトを経由して最初のスレッドの L2 キャッシュに到着します。ほかのスレッドでデータが必要な場合、最初のスレッドの L2 キャッシュからデータを取得します。各スレッドはデータへのアクセスをすべて作成する一方で、データは L2 キャッシュに残されたままです。メモリーアクセスが調整され、1MB のデータはリング・インターコネクトを 1 回のみ経由するため、メモリー帯域幅の負荷は減ります。

L2 キャッシュの処理能力を最大限にするには、各ワーカースレッドの L2 キャッシュに収めることができるデータサイズを計算する必要があります。各コアには 512K のキャッシュと 4 つのスレッドが含まれます。各スレッドで利用可能なキャッシュは 128K です。ランタイムの必要分を引くと、アプリケーションの管理下にあるのは 64K になります。64KB ということは、単精度浮動小数点数で 16K、倍精度浮動小数点数で 8K です。

そこで、一度に 1 つの 16K ブロックの乱数を処理し、すべてのスレッドが処理を完了したときに次のブロックを処理するようにモンテカルロを変更します。中間結果はメモリーに保存します。すべてのデータブロックを処理したら、保存した中間結果を処理して最終結果を得ます。

```
#include "MonteCarlo.h"
#include "math.h"
#include "mkl_vsl.h"
#include "omp.h"
#define RANDSEED 123

static const float  RVVLOG2E = (RISKFREE-0.5f*VOLATILITY*VOLATILITY)*M_LOG2E;
static const float  INV_RAND_N = 1.0f/RAND_N;
static const float  F_RAND_N = static_cast<float>(RAND_N);
static const float STDDEV_DENOM = 1 / (F_RAND_N * (F_RAND_N - 1.0f));
static const float CONFIDENCE_DENOM = 1 / sqrtf(F_RAND_N);
static const int BLOCKSIZE = 32*1024;
static const float  RLOG2E = RISKFREE*M_LOG2E;
static const float  VLOG2E = VOLATILITY*M_LOG2E;

void MonteCarlo(
    float *h_CallResult,
    float *h_CallConfidence,
    float *S,
    float *X,
    float *T
)
{
    __attribute__((align(4096))) float random [BLOCKSIZE];
    VSLStreamStatePtr Randomstream;
    vslNewStream(&Randomstream, VSL_BRNG_MT19937, RANDSEED);
#ifdef _OPENMP
    kmp_set_defaults("KMP_AFFINITY=compact,granularity=fine");
#endif
#pragma omp parallel for
    for(int opt = 0; opt < OPT_N; opt++)
    {
        h_CallResult[opt]     = 0.0f;
        h_CallConfidence[opt] = 0.0f;
```

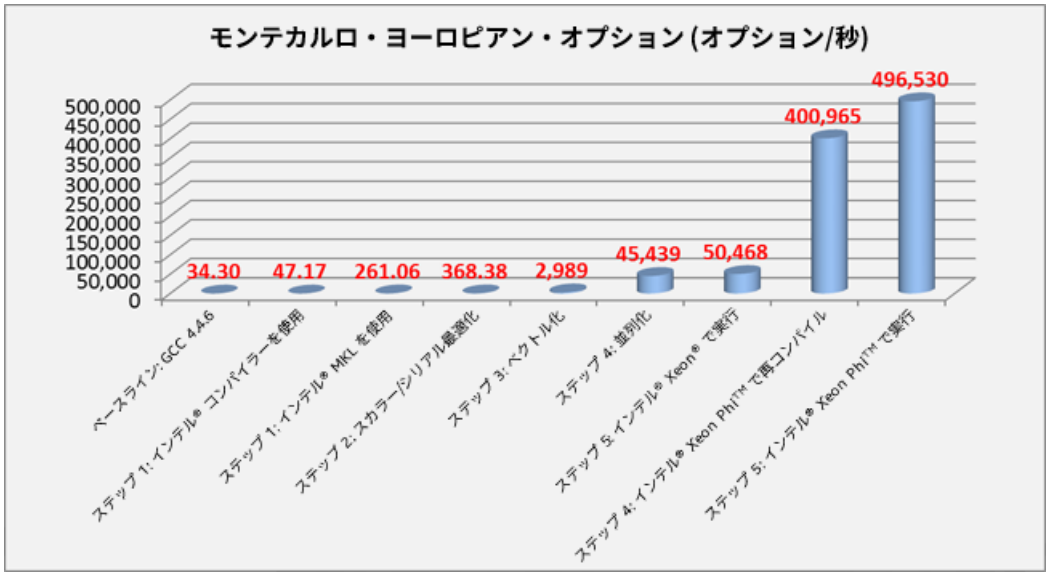
```
    }

    const int nblocks = RAND_N/BLOCKSIZE;
    for(int block = 0; block < nblocks; ++block)
    {
        vsRngGaussian (VSL_METHOD_SGAUSSIAN_ICDF, Randomstream, BLOCKSIZE, random, 0.0f,
1.0f);
#pragma omp parallel for
        for(int opt = 0; opt < OPT_N; opt++)
        {
           float VBySqrtT = VLOG2E * sqrtf(T[opt]);
           float MuByT = RVVLOG2E * T[opt];
           float Sval = S[opt];
           float Xval = X[opt];
           float val = 0.0, val2 = 0.0;

#pragma vector aligned
#pragma simd reduction(+:val) reduction(+:val2)
#pragma unroll(4)
           for(int pos = 0; pos < BLOCKSIZE; pos++)
           {
              float callValue  = Sval * exp2f(MuByT + VBySqrtT * random[pos]) - Xval;
              callValue = (callValue > 0) ? callValue : 0;
              val  += callValue;
              val2 += callValue * callValue;
           }

           h_CallResult[opt]     +=  val;
           h_CallConfidence[opt] +=  val2;
        }
    }

#pragma omp parallel for
    for(int opt = 0; opt < OPT_N; opt++)
    {
        const float val      = h_CallResult[opt];
        const float val2     = h_CallConfidence[opt];
        const float  exprt    = exp2f(-RLOG2E*T[opt]);
        h_CallResult[opt]     = exprt * val * INV_RAND_N;
        const float  stdDev   = sqrtf((F_RAND_N * val2 - val * val) * STDDEV_DENOM);
        h_CallConfidence[opt] = (float)(exprt * stdDev * CONFIDENCE_DENOM);
    }
    vslDeleteStream(&Randomstream);
}
```

帯域幅の負荷を減らすことにより、モンテカルロは真の計算集約ワークロードになりました。合計実行時間が 1 秒近く短縮され、パフォーマンスは 1.24 倍に向上しました。

```
~ $ ./MonteCarlo
Monte Carlo European Option Pricing in Single Precision
Pricing 1998848 options with path length of 262144.
Completed in   4.0256 seconds.
Computation rate - 496530.481 options per second.
```

メモリー・アクセス・パターン最適化は、インテル® Xeon® プロセッサーおよびインテル® Xeon Phi™ コプロセッサーで有効です。スレッドごとに利用できる L2 キャッシュの処理能力の計算は偶然にも両方の IA で同じになりました。そのため、インテル® MIC アーキテクチャー用に最適化されたプログラムをインテル製マルチコア・プロセッサー上でコンパイルして実行してみることにしました。その結果、以前よりも 1.11 倍以上高速に実行することができました。

```
[sli@localhost .solution]$ make -B
icpc -c -O3 -openmp -xAVX -fimf-precision=low -fimf-domain-exclusion=31 -fimf-accuracy-
bits=11 -no-prec-div -no-prec-sqrt -fno-alias -vec-report2  -o Driver.o Driver.cpp
Driver.cpp(130): (列 5) リマーク: ループはベクトル化されませんでした: ベクトル依存関係が存在しています。
Driver.cpp(161): (列 9) リマーク: ループがベクトル化されました。
icpc -c -O3 -openmp -xAVX -fimf-precision=low -fimf-domain-exclusion=31 -fimf-accuracy-
bits=11 -no-prec-div -no-prec-sqrt -fno-alias -vec-report2  -o MonteCarloStep5.o
MonteCarloStep5.cpp
MonteCarloStep5.cpp(60): (列 5) リマーク: ループはベクトル化されませんでした: ベクトル依存関係が存在しています。
MonteCarloStep5.cpp(53): (列 5) リマーク: ループがベクトル化されました。
MonteCarloStep5.cpp(75): (列 12) リマーク: SIMD ループがベクトル化されました。
MonteCarloStep5.cpp(64): (列 5) リマーク: ループはベクトル化されませんでした: 内部ループではありません。
MonteCarloStep5.cpp(88): (列 5) リマーク: ループがベクトル化されました。
icpc -xAVX -openmp Driver.o MonteCarloStep5.o -o MonteCarlo -L
/opt/intel/composerxe/mkl/lib/intel64 -lmkl_intel_lp64 -lmkl_sequential -lmkl_core
[sli@localhost .solution]$ ./MonteCarlo
Monte Carlo European Option Pricing in Single Precision
Pricing 1998848 options with path length of 262144.
Completed in  39.6060 seconds.
Computation rate - 50468.304 options per second.
```



4. まとめ

この記事では、デリバティブ価格評価アプリケーションにおけるモンテカルロ法の概要を説明しました。また、インテル® Xeon Phi™ コプロセッサーの新製品とインテル® C++ Composer XE 2013 の統合ソフトウェア開発環境について紹介しました。ケーススタディーとして、モンテカルロ・ヨーロピアン・コール・オプションの価格評価に独自の C/C++ 実装を行い、段階的な最適化フレームワークを示しました。

インテル® C++ Composer XE 2013 を使用した 5 ステップのパフォーマンスの最適化フレームワークに従うことにより、モンテカルロ・ヨーロピアン・オプション価格評価のパフォーマンスはインテル® Xeon Phi™ コプロセッサーで 1471 倍以上に向上しました。また、インテル® Xeon® プロセッサー用に最適化されたアプリケーションをコンパイル、リビルドしてインテル® Xeon Phi™ コプロセッサーでネイティブに実行できることを示しました。インテル® Xeon Phi™ コプロセッサー上でアプリケーションをさらに最適化して、スケーラビリティーの問題に取り組むことができます。最適化したコードを再コンパイルしてインテル® Xeon® プロセッサーで実行することにより、より高いパフォーマンスを得ることができます。

段階的な最適化フレームワークは、大量の数値計算を行う金融アプリケーションだけでなく、一般的な科学計算アプリケーションにも有効であることが判明しています。インテル® C++ Composer XE 2013 により、プログラマーはより簡単に、構造化されたアクティビティーとしてパフォーマンスの最適化に取り組み、プログラムの制限を迅速に理解して、最大限の並列化を達成できるでしょう。

関連情報

- インテル® Composer XE 2013 for Linux* (インテル® MIC アーキテクチャー対応)**
- インテル® C++ コンパイラー XE 12.0 ユーザー・リファレンス・ガイド**
- インテル® MIC アーキテクチャー最適化ガイド***
- C++ Technical Report 1

** このドキュメントは、インテル® Composer とともにインストールされます。

*** このドキュメントは、MIC 開発者ポータル (https://mic-dev.intel.com) から入手できます。

著者紹介

Shuo Li
インテル コーポレーションのソフトウェア & ソリューション・グループに所属しており、ソフトウェア開発において 24 年の経験があります。主に、並列プログラミング、金融工学、アプリケーション・パフォーマンスの最適化に取り組んでいます。現在は、ソフトウェア・パフォーマンス・エンジニアとして金融サービス業界を担当しており、ソフトウェア開発者やモデラーがインテルのプラットフォームで最良のパフォーマンスを引き出せるように支援しています。オレゴン大学でコンピューター・サイエンスの修士号を、デューク大学で MBA を取得しています。