ケーススタディー: インテル® アドバンスト・ベクトル・エクステンションを使用したブラック-ショールズの計算

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: Computing Black-Scholes with Intel® Advanced Vector Extensions」の日本語参考訳です。

---

1. はじめに

革新をリードし、より高い計算能力とより低い消費電力を実現するという過酷な取り組みの一環として、業界で増大する要求を満たし、進化する使用モデルに対応するため、インテルは、インテル® アドバンスト・ベクトル・エクステンション (インテル® AVX) と呼ばれる新しい命令セットを広範なインテルのプラットフォームに実装しました。

インテル® AVX は、インテル® ストリーム SIMD 拡張命令 (インテル® SSE) の豊富な資産に基づいた、データ処理における柔軟なプログラミング環境、ベクトルとスカラー・データセットを用いた簡潔なコーディング、より電力効率に優れたパフォーマンスを実現する命令セット・アーキテクチャーの拡張を提供します。インテル® AVX は、金融解析やハイパフォーマンス・コンピューティング (HPC) 分野のようなアプリケーションで増大するニーズに対応するインフラストラクチャーとビルディング・ブロックを提供します。

この記事では、インテルの命令セット・アーキテクチャー (ISA) について簡単に説明した後、金融サービス業界で最も一般的なモデルの 1 つであるブラック-ショールズ・モデルを使用した計算をどのように向上できるかという点から、新しい拡張命令と機能の概要を説明します。

1.1 ISA と ISA 拡張命令の概要および使用する理由

ISA は、マイクロプロセッサー・アーキテクチャーのプログラミングに関連する部分で、ネイティブのデータ型、命令、レジスター、アドレッシング・モード、メモリー・アーキテクチャー、割り込み、I/O などを含みます。ISA は、マイクロプロセッサー・ファミリーにより実装されるマイクロプロセッサーの仕様を定義します。ISA は通常、プログラミング・モデルの互換性を維持したまま、時間とともに、より優れたパフォーマンスを提供する、新しく拡張された機能が追加されます。

ISA 拡張は、より高いアプリケーション・パフォーマンスとより低い消費電力をもたらし、ソフトウェアの下位互換性があるため、開発者にも大きなメリットがあります。例えば、8087 として知られる浮動小数点 (FP) 算術関数群は、8086 マイクロプロセッサーの拡張として定義され追加されたものです。浮動小数点ライブラリー・ルーチンの呼び出しに使用していた同じアプリケーション・ソフトウェアで 8087 命令を実行できます。x87 のような初期の拡張では単一データを処理する命令を追加していましたが、最近の拡張では、複数のデータを単一の命令で実行する命令、つまり SIMD (Single Instruction Multiple Data) 命令を追加しています。

インテル® MMX® テクノロジー対応のインテル® Pentium® II プロセッサーおよびインテル® Pentium® プロセッサー・ファミリーからこれまで、SIMD 演算を行う 6 つの拡張命令がインテル® 64 アーキテクチャーおよび IA-32 アーキテクチャーに追加されました。これらの拡張命令には、インテル® MMX® テクノロジー、インテル® SSE、インテル® SSE2、インテル® SSE3、インテル® SSSE3、インテル® SSE4 が含まれます。これらの拡張命令はそれぞれ、パックド整数やパックド浮動小数点データ要素で SIMD 演算を実行する一連の命令を提供します。

1.2 インテル® アドバンスト・ベクトル・エクステンション

インテル® AVX は、インテル® アーキテクチャーの新しい 256 ビット SIMD FP ベクトル拡張で、金融デリバティブの価格設定のような汎用科学計算の浮動小数点計算に SIMD テクノロジーを使用する機会を高めます。インテル® AVX は、インテル® 64 アーキテクチャーの包括的な ISA 拡張です。インテル® AVX の主な要素は次のとおりです。

- より幅の広いベクトルデータ (最大 256 ビット) のサポート。
- 3 オペランドおよび 4 オペランドの命令構文による、効率的な命令エンコーディング・スキーム。
- 分岐処理からメモリー・アライメント要件の緩和まで、広い範囲にわたるプログラミング環境の柔軟性。
- ブロードキャスト、置換、積和演算、その他を含む、新しいデータ操作および算術計算プリミティブ。

浮動小数点や整数 SIMD を多用するアプリケーションでもインテル® AVX を使用できますが、最良の結果が得られるのは、浮動小数点計算集約型のベクトル化可能なアプリケーションです。このようなアプリケーションの例としては、オーディオ/ビデオ処理やコーデックが一般的です。この記事では、インテル® AVX 拡張を活用できる別の種類のアプリケーションとして、金融サービス解析とモデリングを取り上げます。

2. インテル® AVX 対応インテル® プロセッサーにおけるブラック-ショールズ式の実装

2.1 ブラック-ショールズとブラック-ショールズ式

ブラック-ショールズ・モデルは、現代の金融工学理論で最も重要な概念の 1 つです。1973 年にフィッシャー・ブラックとマイロン・ショールズにより開発され、現在でも広く利用されており、金融デリバティブの公正価格を決定する最良の方法の 1 つと見なされています。

$$\frac{\partial f}{\partial t}+\frac{1}{2}\sigma^2 S^2\frac{\partial^2 f}{\partial S^2}+rS\frac{\partial f}{\partial S}-rf=0$$

その後、ロバート・マートンにより、(ブラック-ショールズ-マートン式と呼ばれる解法を用いて) ヨーロピアン・コール・オプション *c* およびヨーロピアン・プット・オプション *p* についてブラック-ショールズ方程式の数学的証明が行われました。

$$c = Scnd(d_1) - Ke^{-rT}cnd(d_2)$$
$$p = Ke^{-rT}cnd(-d_2) - Scnd(-d_1)$$

説明:

$$d_1 = \frac{\ln(S/K)+\left(r+\frac{\sigma^2}{2}\right)T}{\sigma\sqrt{T}}$$
$$d_2 = \frac{\ln(S/K)+\left(r-\frac{\sigma^2}{2}\right)T}{\sigma\sqrt{T}} = d_1 - \sigma\sqrt{T}$$

cnd(x) 関数は、累積正規分布関数です。x 未満の ? (0, 1) の標準正規分布の変数で確率を計算します。cnd(x) は、次のように定義された多項式を使用した近似関数です。

$$cnd(x)=\begin{cases}1-\frac{1}{\sqrt{2\pi}}e^{-x^2/2}(a_1k+a_2k^2+a_3k^3+a_4k^4+a_5k^5), & x\geq 0\\ 1-cnd(-x) & , x<0\end{cases}$$

ここで、

$k=\frac{1}{1+\gamma x},\quad \gamma=0.2316419$

$a_1=0.319381530, a_2=-0.356563782, a_3=1.781477937, a_4=-1.821255978, a_5=1.330274429$

とします。

2.2 ブラック-ショールズ式の実装

ブラック-ショールズ式は、金融工学のほぼすべての局面で利用されています。ブラック-ショールズ計算は、トレーダーやクオンツアナリストにより広く利用され、すべての金融工学ライブラリーに含まれています。現在は、グローバル金融 (FSI) 業界でコンピューター・アーキテクチャーを証明する指標となっています。この記事では、インテル製マイクロプロセッサーでインテル® AVX テクノロジーを利用してブラック-ショールズ計算を行う方法を説明します。

数百万の金融商品のヨーロピアン・オプションを計算する場合について考えてみましょう。各商品には、時価、権利行使価格およびオプション満期日があります。これらのデータの各セットについて、数千のブラック-ショールズ計算 (近隣の株価、権利行使価格および異なるオプション満期日の計算) を行います。

```
const float      RISKFREE = 0.02f;
const float    VOLATILITY = 0.30f;

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;
}

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

    float
        K = 1.0 / (1.0 + 0.2316419 * abs(d));

    float
        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 BlackScholesBody(float& callResult,
                    float& putResult,
                    float S,
                    float X,
                    float T,
                    float R,
                    float V )
{
    float sqrtT = sqrt(T);
    float d1 = (log(S / X) + (R + 0.5f * V * V) * T) / (V * sqrtT);
    float d2 = d1 - V * sqrtT;
    float CNDD1 = CND(d1);
    float CNDD2 = CND(d2);

    float expRT = exp(- R * T);
    callResult   = S * CNDD1 - X * expRT * CNDD2;
    putResult = X * expRT * (1.0f - CNDD2) - S * (1.0f - CNDD1);
}

///////////////////////////////////////////////////////////////////////
//// Process an array of optN options
///////////////////////////////////////////////////////////////////////
void BlackScholes(float *CallResult,
                    float *PutResult,
                    float *StockPrice,
                    float *OptionStrike,
                    float *OptionYears,
                    float Riskfree,
                    float Volatility,
                    int optN)
{
        for(int i = 0; i < NUM_ITERATIONS; i++)
                for(int opt = 0; opt < OPT_N; opt++)
                        BlackScholesBody(
                                CallResult[opt],
                                PutResult[opt],
                                StockPrice[opt],
                                OptionStrike[opt],


                                OptionYears[opt],
                                Riskfree,
                                Volatility);
}

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

    float
          *CallResult,
          *PutResult,
          *StockPrice,
          *OptionStrike,
          *OptionYears;
    double
          sTime, eTime;
    int
          mem_size = sizeof(float) * OPT_N;

          CallResult = (float *)malloc(mem_size);
          PutResult  = (float *)malloc(mem_size);
          StockPrice    = (float *)malloc(mem_size);
          OptionStrike  = (float *)malloc(mem_size);
          OptionYears   = (float *)malloc(mem_size);

          for(int i = 0; i < OPT_N; i++)
          {
             CallResult[i] = 0.0f;
             PutResult[i]  = -1.0f;
             StockPrice[i]    = RandFloat(5.0f, 30.0f);
             OptionStrike[i]  = RandFloat(1.0f, 100.0f);
             OptionYears[i]   = RandFloat(0.25f, 10.0f);
          }
          sTime = second();
          BlackScholes( CallResult,
                     PutResult,
                     StockPrice,
                     OptionStrike,
                     OptionYears,
                     RISKFREE,
                     VOLATILITY,
                     OPT_N);
          eTime = second();

        printf("Parallel version runs at %7.5f Billion option per
second :\n", (2.0f*(NUM_ITERATIONS)*OPT_N)/(1e9*(eTime-sTime)));
        printf("Completed pricing %d million options in %f
seconds:\n", 2*(NUM_ITERATIONS)*(OPT_N/1000000), eTime-sTime);

          free(CallResult);
          free(PutResult);
          free(StockPrice);
          free(OptionStrike);
          free(OptionYears);
}
```

並列バージョンは毎秒 589 万のオプションを実行し、11 億 5200 万のオプションの価格設定は 234.880353 秒で完了しました。

図 1: ブラック-ショールズのコードと gcc –O2 を指定してコンパイルしたときの出力

3. ブラック-ショールズの実装の最適化

ブラック-ショールズ式をそのまま実装してもハイパフォーマンスが保証されるわけではありません。インテル® Xeon® プロセッサー EP 3.10GHz システムで –O2 最適化スイッチを指定して GNU* コンパイラー・コレクション 4.4.6 を使用すると、11 億 5200 万のオプションの価格設定は 234.88 秒で完了しました。スループット・レートは毎秒 589 万です。このセクションでは、できるだけ低い経過時間でできるだけ高いスループットを達成できるように、ブラック-ショールズ実装のパフォーマンスを向上させます。

3.1 段階的な最適化フレームワーク

まず、体系的なアプローチでアプリケーションのパフォーマンスを向上させる最適化フレームワークについて説明します。このフレームワークは、アプリケーションを 4 つの最適化ステージで改良します。各ステージ

は、それぞれ 1 つの手法を適用した 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。この方法に従うことにより、アプリケーションはインテル® マイクロプロセッサー・アーキテクチャーで可能な最も高いパフォーマンスを達成することができます。

- ステージ 1: 最適化ツールとライブラリーの活用
- ステージ 2: スカラー/シリアル最適化
- ステージ 3: ベクトル化
- ステージ 4: 並列化

図 2: 最適化フレームワーク

3.2 ステージ 1: 最適化ツールとライブラリーの活用

最初のステージでは、最適化を開始する前に、すでに分かっていることを最初からやり直そうとしていないか、自分自身に確認します。問題がほかの人によってすでに解決されている場合、最良の手法は、その問題には既存の解法を利用し、まだ解決されていない問題に労力をかけることです。

ブラック-ショールズ実装では、優れたコンパイラーや C++ ランタイム・ライブラリーを利用しているかどうか確認すべきです。その結果、我々はインテル® Composer XE 2013 を使用することにしました。GCC は汎用アプリケーションの開発をターゲットにしていますが、インテル® Composer XE はハイパフォーマンス・アプリケーションの開発をターゲットにしています。ターゲット実行環境が最近リリースされたインテル® マイクロプロセッサー製品ベースで、コンパイラーが結合や分布などの法則に基づいた数学的変換を行ってもかまわないのであれば、インテル® コンパイラーのツールを使用すべきです。同じコンパイラー・スイッチを使用して g++ を icpc に変更するだけで、プログラム実行は 46 秒と、コードの変更前よりも約 5 倍の速度になりました。

並列バージョンは毎秒 2986 万のオプションを実行し、11 億 5200 万のオプションの価格設定は 46.295983 秒で完了しました。

インテル® Composer XE 2013 には、インテルが拡張した C/C++ ランタイム・ライブラリーも含まれています。一般的な超越関数の 1 つである、誤差関数 `erf()` はインテル® Composer XE のランタイム・ライブラリー libm に含まれていますが、GCC のランタイム・ライブラリーには含まれていません。インテル® コンパイラーのツールを使用することに決めたため、libm に含まれる誤差関数とブラック-ショールズ式の累積正規分布関数を同時に使用することができます。

$$cnd(x) = \frac{1}{2} + \frac{1}{2} * erf(\frac{1}{\sqrt{2}} * x)$$

```
const float     INV_SQRT2 = 1.0/sqrt(2.0);
const float          HALF = 0.5;
float CND(float d)
{
    return HALF + HALF*erf( INV_SQRT2*d);
}
```

この結果、パフォーマンスはさらに 13% 向上しました。
並列バージョンは毎秒 3382 万のオプションを実行し、11 億 5200 万のオプションの価格設定は 40.875621 秒で完了しました。

各自のアプリケーションに注目する時間を増やすには、まず、既存のハイパフォーマンス・ソリューションを最大限に利用することです。コンパイラーを変更して超越関数を使用するだけで、大幅なスピードアップを達成することが可能です。この例では、ブラック-ショールズ式のパフォーマンスが 5.75 倍に向上しました。

3.3 ステージ 2: スカラー/シリアル最適化

利用可能な最適化ソリューションをすべて適用した後、アプリケーションのパフォーマンスをさらに向上するには、アプリケーションのソースコードを入手して、最適化プロセスを開始します。実際に並列プログラミングを開始してベクトル化や並列化を行う前に、アプリケーションが正しい結果を生成していることを確認する必要があります。最小限の演算数でその正しい結果を得ていることを確認することも同じく重要です。通常は、次のようなデータとアルゴリズム関連の問題に注意します。

- 正しい浮動小数点精度を選択する。
- 正しい近似法を選択する (多項式または有理関数)。
- ジャンプ・アルゴリズムを使用しない。
- 反復計算を使用して、ループ演算の重みを減らす。
- アルゴリズムで条件分岐を使用しない、または最小限にする。
- 以前の計算結果を使用して、同じ計算を繰り返さない。

言語関連のパフォーマンス問題にも注意する必要があります。ここでは C/C++ を選択したため、次のような C/C++ 関連の問題に注意します。

- 自動入力を使用しないで、すべての定数を明示的に入力する。
- C ランタイム関数の正しい型を選択する (`exp()` と `expf()`、`abs()` と `fabs()` など)。
- コンパイラーにエイリアスを明示的に伝える。
- オーバーヘッドを避けるためインライン関数呼び出しを明示的に行う。

『Numerical Recipes in C』に掲載されている多くのルーチンと同様に、ブラック-ショールズ式は float で記述されています。ほとんどの金融工学アプリケーションは、32 ビット浮動小数点のダイナミック・レンジで十分です。入力データ、出力データ、計算はすべて 32 ビットです。32 ビットから 64 ビットへの変換を行うと、パフォーマンスは低下しますが精度は向上しません。定数およびライブラリー関数呼び出しはすべて、明示的に float にしてください。

数学者は完璧な対称性を好みます。その結果、式にその好みが反映されることがあります。しかし、式を実装するときに、その好みが冗長となりパフォーマンス・ペナルティーの原因になることがあります。これは、コールオプションとプットオプションの計算にも当てはまります。

$$c = Scnd(d_1) - Ke^{-rT}cnd(d_2)$$
$$p = Ke^{-rT}cnd(-d_2) - Scnd(-d_1)$$

マートンの式はかなり対称的です。しかし、コールオプションのサイクルに時間を費やしたら、プットオプションのサイクルに同じ時間を費やす必要はありません。その理由は、コールオプションとプットオプションは *put-call* パリティーを満たしているからです。

$$p - c = Ke^{-rT} - S$$

put-call パリティーも数学的に派生することがあります。

$$cnd(-d_2) = 1 - cnd(d_2)$$

コールオプションが完了したら、プットオプションは 2 つ加算を追加するだけです。以下に変更後のコードを示します。

```
#include <sys/time.h>
#include <time.h>

#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <math.h>
#include <omp.h>

//const int OPT_N = 3*40000000;
const int OPT_N = 3*400000;
const int  NUM_ITERATIONS = 3*32*6; //64*128;

const float      RISKFREE = 0.02f;
const float    VOLATILITY = 0.30f;
const float     INV_SQRT2 = 1.0f/sqrtf(2.0f);
const float          HALF = 0.5f;

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;
}
///////////////////////////////////////////////////////////////////////
///////////
//// Process an array of optN options
///////////////////////////////////////////////////////////////////////
/////////////
inline void BlackScholes(
        float *CallResult,
        float *PutResult,
        float *StockPrice,
        float *OptionStrike,
        float *OptionYears
        )
{
        for(int i = 0; i < NUM_ITERATIONS; i++)
                for(int opt = 0; opt < OPT_N; opt++)
                {
                        float CallVal =0.0f, PutVal  = 0.0f;
                        float T = OptionYears[opt];
                        float X = OptionStrike[opt];
                        float S = StockPrice[opt];
                        float sqrtT = sqrtf(T);
                        float d1 = (logf(S / X) + (RISKFREE + 0.5f *
VOLATILITY * VOLATILITY) * T) / (VOLATILITY * sqrtT);
                        float d2 = d1 - VOLATILITY * sqrtT;
                        float CNDD1 = HALF + HALF*erff(INV_SQRT2*d1);
                        float CNDD2 = HALF + HALF*erff(INV_SQRT2*d2);
                        float expRT = expf(-RISKFREE * T);
                        CallVal  += S * CNDD1 - X * expRT * CNDD2;
                        PutVal  += CallVal  +  expRT - S;
                        CallResult[opt] = CallVal ;
                        PutResult[opt] = PutVal ;
                }
}
int main(int argc, char* argv[])
{

   float
        *CallResult,
        *PutResult,
        *StockPrice,
        *OptionStrike,
        *OptionYears;
   double
        sTime, eTime;
//        delta, ref, sum_delta, sum_ref, max_delta, L1norm, gpuTime;
   int mem_size = sizeof(float) * OPT_N;

        CallResult = (float *)malloc(mem_size);
        PutResult  = (float *)malloc(mem_size);
        StockPrice    = (float *)malloc(mem_size);
        OptionStrike  = (float *)malloc(mem_size);
        OptionYears   = (float *)malloc(mem_size);

        for(int i = 0; i < OPT_N; i++)
        {
            CallResult[i] = 0.0f;
            PutResult[i]  = 0.0f;
            StockPrice[i]    = RandFloat(5.0f, 30.0f);
            OptionStrike[i]  = RandFloat(1.0f, 100.0f);
            OptionYears[i]   = RandFloat(0.25f, 10.0f);
         }

        sTime = second();
        BlackScholes( CallResult,
                PutResult,
                StockPrice,
                OptionStrike,
                OptionYears);
        eTime = second();

        printf("Parallel version runs at %7.5f Billion option per
second :\n", (2.0f*(NUM_ITERATIONS)*OPT_N)/(1e9*(eTime-sTime)));
        printf("Completed pricing %d million options in %f
seconds:\n", 2*(NUM_ITERATIONS)*(OPT_N/1000000), eTime-sTime);

        free(CallResult);
        free(PutResult);
        free(StockPrice);
        free(OptionStrike);
        free(OptionYears);
}
```

並列バージョンは毎秒 3909 万のオプションを実行し、11 億 5200 万のオプションの価格設定は 35.364703 秒で完了しました。

図 3: put-call パリティーを使用するように変更されたコード

上記の変更を行った後、-O2 -xAVX オプションを指定し、ベクトル化 (必要な場合は -no-vec を使用) と並列化を行わないように指定してコンパイルしたところ、同じハードウェアでパフォーマンスは GCC の 6.64 倍になりました。

3.4 ステージ 3: ベクトル化

スカラーコードが最適化され、ベクトル化の準備が整いました。このセクションでは、ブラック-ショールズのソースコードにベクトル化を追加します。ベクトル化の意味は、人によって異なります。ここでは、プロセッサー・レベルで SIMD レジスターと SIMD 命令を活用することを意味します。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追加することができます。これらのコンパイラー・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量、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。

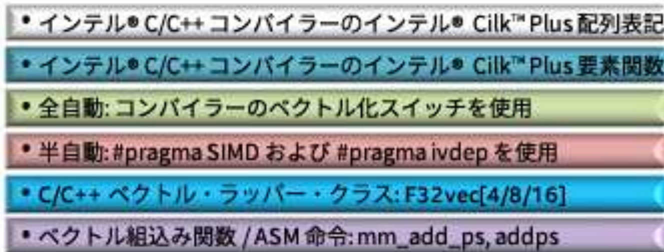

図 4: ベクトル化の手法

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、プログラマーはメモリー・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、パフォーマンスは低下します。適切なメモリー・アライメントを保証する 1 つの方法は、厳密にアライメントされたメモリーを常に要求して操作することです。インテル® Composer XE 2013 では、__attribute__(align(32)) 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。32 バイト境界は、YMMX 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。

```
        CallResult = (float *)_mm_malloc(mem_size, 32);
        PutResult  = (float *)_mm_malloc(mem_size, 32);
        StockPrice    = (float *)_mm_malloc(mem_size, 32);
        OptionStrike  = (float *)_mm_malloc(mem_size, 32);
        OptionYears   = (float *)_mm_malloc(mem_size, 32);
...
        _mm_free(CallResult);
        _mm_free(PutResult);
        _mm_free(StockPrice);
        _mm_free(OptionStrike);
        _mm_free(OptionYears);
```

図 5: メモリーの割り当て/解放の呼び出しの例

メモリー・アライメントを指定したら、ブラック-ショールズ実装に使用するベクトル化手法を選択します。作業量を最小化し、移植性を維持するため、ユーザーが開始する半自動ベクトル化アプローチを使用してベクトル化します。ユーザーは、#pragma SIMD を使用して、ループをベクトル化するようにコンパイラーに指示します。コンパイラーは、#pragma SIMD でマークされたループがベクトル化できる場合はベクトル化を行い、ベクトル化できない場合はエラーメッセージを出力します。この動作は、ユーザーが #pragma IVDEP で推

奨事項を提示し、ベクトル化されたコードがシリアルコードよりも速く実行できるかどうかはコンパイラーが評価する点で、以前のモデルとは異なります。ベクトル化されたコードは、シリアルバージョンよりも速いとコンパイラーが判断した場合にのみ生成されます。この新しいモデルでは、ベクトル化のオーバーヘッドがスピードアップよりも大きくならないことを保証するのはプログラマーの責任です。

```
for(int i = 0; i < NUM_ITERATIONS; i++)
#pragma simd
    for(int opt = 0; opt < OPT_N; opt++)
    {
        float CallVal =0.0f, PutVal  = 0.0f;
        float T = OptionYears[opt];
        float X = OptionStrike[opt];
        float S = StockPrice[opt];
        float sqrtT = sqrtf(T);
        float d1 = (logf(S / X) + (RISKFREE + 0.5f * VOLATILITY * VOLATILITY) * T)
        / (VOLATILITY * sqrtT);
        float d2 = d1 - VOLATILITY * sqrtT;
        float CNDD1 = HALF + HALF*erff(INV_SQRT2*d1);
        float CNDD2 = HALF + HALF*erff(INV_SQRT2*d2);
        float expRT = expf(-RISKFREE * T);

        CallVal  += S * CNDD1 - X * expRT * CNDD2;
        PutVal  += CallVal  +  expRT - S;
        CallResult[opt] = CallVal ;
        PutResult[opt] = PutVal ;
    }
```

並列バージョンは毎秒 2 億 7955 万のオプションを実行し、11 億 5200 万のオプションの価格設定は 4.945128 秒で完了しました。

図 6: ユーザーが開始したベクトル化の例

ベクトル化されたコードは、最近のマイクロプロセッサーの SIMD 命令を利用して、周波数を上げたり、コア数を増やすことなくアプリケーションのパフォーマンスを向上します。今回の例では、ベクトル化によりパフォーマンスは 7.15 倍から 8 倍になりました。

3.5 ステージ 4: 並列化

マイクロプロセッサー・パッケージの CPU コアの数は、この 10 年で大きく増加しています。ベクトル化されたブラック-ショールズ式は、1 つのプロセッサー・コアを最大限に活用するようになりましたが、追加のコアを利用するには、コードをマルチスレッド化する必要があります。ここでは、コアをすべて同時に動作させて、それぞれベクトル化されたブラック-ショールズ式を実行して作業を完了するようにします。

ベクトル化と同じように、マルチスレッド化でもプログラマーにはさまざまなスレッド化の選択肢があります。選択肢は、明示的なプログラマーの制御、容易性、コードの保守性の点で異なります。

- インテル® C/C++ コンパイラー cilk_spawn
- Rogue Wave スレッドクラス
- スレッディング・ビルディング・ブロック
- OpenMP* スレッド
- pthread/win32 スレッド・ライブラリー

図 7: マルチスレッド化の選択肢

このブラック-ショールズ式には 2 つのループが含まれています。内側のループはすでにベクトル化されています。マルチスレッド化を行う最も簡単な方法は、各スレッドで異なるデータの外側のループ全体を実行することです。この方法に適した選択肢は OpenMP* です。

```
    float RLOG2E =-RISKFREE*M_LOG2E;
    kmp_set_defaults("KMP_AFFINITY=scatter,granularity=thread");
#pragma omp parallel
    for(int i = 0; i < NUM_ITERATIONS; i++)
#pragma omp for
#pragma simd
#pragma vector aligned
        for(int opt = 0; opt < OPT_N; opt++)
        {
            float CallVal =0.0f, PutVal  = 0.0f;
            float T = OptionYears[opt];
            float X = OptionStrike[opt];
            float S = StockPrice[opt];
            float sqrtT = sqrtf(T);
            float d1 = (logf(S / X) + (RISKFREE + 0.5f * VOLATILITY *
                            VOLATILITY) * T) / (VOLATILITY * sqrtT);
            float d2 = d1 - VOLATILITY * sqrtT;
            float CNDD1 = HALF + HALF*erff(INV_SQRT2*d1);
            float CNDD2 = HALF + HALF*erff(INV_SQRT2*d2);
            float expRT = exp2f(RLOG2E * T);

            CallVal  += S * CNDD1 - X * expRT * CNDD2;
            PutVal  += CallVal  +  expRT - S;
            CallResult[opt] = CallVal ;
            PutResult[opt] = PutVal ;
        }
```

並列バージョンは毎秒 40 億 7992 万のオプションを実行し、11 億 5200 万のオプションの価格設定は 0.338830 秒で完了しました。

図 8: スレッド化されたコード

この例で、`#pragma omp parallel` はスレッドを作成または fork します。各スレッドは、データのサブセットで外側のループを実行します。#pragma omp for は、スレッドを内側のループにバインドします。

このプログラムを実行したところ、パフォーマンスはベクトル化されたスカラー実装の 14.21 倍に向上しました。16 コアのシステムで OpenMP* スレッド作成による多少のオーバーヘッドがあることを考えると、このパフォーマンス向上率は非常に高いものと言えます。

このベクトル化されたブラック-ショールズ式の並列実装は、任意のスレッド数のシステムで動作します。各スレッドは、NUM_ITERATION (反復数) / NumThreads (スレッド数) 回、反復を行います。このスレッドあたりの反復数は倍数でないことがあります。そのため、コンパイラーは、(ベクトル化されたループ用とSIMD 長未満のデータ用の) 2 つのバージョンのループ本体を生成します。その結果、ベクトル化された並列プログラムのオブジェクト・コードのサイズは、ベクトル化されたシリアルコードよりも大きくなります。

4. まとめ

この記事では、最新のインテル® ISA 拡張と大量の数値計算を行う金融/科学計算向けアプリケーションの概要を説明しました。サンプルとしてブラック-ショールズ式を使用し、段階的なアプリケーション最適化フレームワークを紹介しました。この最適化フレームワークに従うことで、ブラック-ショールズ計算は、インテル® Composer XE 2013、インテル® アドバンスト・ベクトル・エクステンション、OpenMP* の組み合わせを使用して、637 倍という、とてつもないパフォーマンス向上を達成しました。この最適化の中心となった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® Composer XE 2013 の次のバージョンでは、科学プログラマーがより簡単に、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、できるだけ並列化を達成できるようにする予定です。

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 13.0 ユーザー・リファレンス・ガイド

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。

著作権と商標について

本資料に掲載されている情報は、インテル製品の概要説明を目的としたものです。本資料は、明示されているか否かにかかわらず、また禁反言によるとよらずにかかわらず、いかなる知的財産権のライセンスを許諾するものではありません。製品に付属の売買契約書『Intel's Terms and Conditions of Sale』に規定されている場合を除き、インテルはいかなる責任を負うものではなく、またインテル製品の販売や使用に関する明示または黙示の保証 (特定目的への適合性、商品適格性、あらゆる特許権、著作権、その他知的財産権の非侵害性への保証を含む) に関してもいかなる責任も負いません。

インテルによる書面での合意がない限り、インテル製品は、その欠陥や故障によって人身事故が発生するようなアプリケーションでの使用を想定した設計は行われていません。

インテル製品は、予告なく仕様や説明が変更される場合があります。機能または命令の一覧で「留保」または「未定義」と記されているものがありますが、その「機能が存在しない」あるいは「性質が留保付である」という状態を設計の前提にしないでください。これらの項目は、インテルが将来のために留保しているものです。インテルが将来これらの項目を定義したことにより、衝突が生じたり互換性が失われたりしても、インテルは一切責任を負いません。この情報は予告なく変更されることがあります。この情報だけに基づいて設計を最終的なものとしないでください。

本資料で説明されている製品には、エラッタと呼ばれる設計上の不具合が含まれている可能性があり、公表されている仕様とは異なる動作をする場合があります。現在確認済みのエラッタについては、インテルまでお問い合わせください。

最新の仕様をご希望の場合や製品をご注文の場合は、お近くのインテルの営業所または販売代理店にお問い合わせください。

本資料で紹介されている資料番号付きのドキュメントや、インテルのその他の資料を入手するには、1-800-548-4725 (アメリカ合衆国) までご連絡いただくか、インテルの Web サイトを参照してください。

性能に関するテストに使用されるソフトウェアとワークロードは、性能がインテル® マイクロプロセッサー用に最適化されていることがあります。SYSmark* や MobileMark* などの性能テストは、特定のコンピューター・システム、コンポーネント、ソフトウェア、操作、機能に基づいて行ったものです。結果はこれらの要因によって異なります。製品の購入を検討される場合は、他の製品と組み合わせた場合の本製品の性能など、ほかの情報や性能テストも参考にして、パフォーマンスを総合的に評価することをお勧めします。

本資料に含まれるソフトウェア･ソース･コードはソフトウェア･ライセンス契約に基づいて提供されるものであり、その使用および複製はライセンス契約で定められた条件下でのみ許可されます。

Intel、インテル、Intel ロゴ、Cilk、Pentium、MMX、Xeon は、アメリカ合衆国および / またはその他の国における Intel Corporation の商標です。

© 2016 Intel Corporation. 無断での引用、転載を禁じます。
* その他の社名、製品名などは、一般に各社の表示、商標または登録商標です。

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