Vol.2011-HPC-130 No.10
2011/7/27

# 並列プログラミングフレームワークを活用した通貨オプション時価計算の実装と評価

鳥谷部　和孝[†1]　　飯塚　拓郎[†2]

本発表では，計算の並列化を CPU と GPU の両方に対して容易に行うために構築した並列プログラミングフレームワークと，それを用いて通貨オプションの価格付けの高速化を行った結果について報告する．

# CUDA implemantation of a currency option valuation using a parallel programming framework

Kazutaka Toyabe[†1] and Takuro Iizuka[†2]

In this study, we introduce to a parallel programming framework to implement pricing algorithm on both multi-core CPU and GPU efficiently, and compare their performance.

## 1. はじめに

金融機関では，計算負荷の高い複雑な金融商品の時価や取引相手のクレジット・リスク量の計算に，モンテカルロ法を用いることが多い．バミューダン型の早期償還条項のついた多資産を参照したスワップはその典型例で，その時価計算には最小二乗モンテカルロ法[1)]（least-squares Monte Carlo approach，以下 LSM）とよばれる方法を用いるのが一般的である．また，最近話題になっているカウンターパーティ（取引相手）リスクを考慮した時価計算に現れる CVA（credit valuation adjustment）の計算にも LSM は使用されており[2)]，LSM の高速化は金融機関にとって重要なテーマの一つである．

モンテカルロ法自体は並列化が非常に行いやすく，それへの GPU の活用はこれまでも様々な事例研究が行われているが[3)]，LSM に対する高速化研究はまだ十分には行われていないようである．また，GPU を用いた多くの高速化研究では比較に用いられている CPU コードはあまり最適化されていないものである，という批判も見受けられる[4)]．そこで本発表では，計算の並列化を CPU と GPU の両方に対して容易に行うために構築した並列プログラミングフレームワークと，それを用いた LSM の高速化を行った．

最適化した GPU コードは最適化した CPU シングルスレッド実装に比べて約 60 倍の高速化を達成した．並列プログラミングフレームワークを利用することで，CUDA やマルチスレッドプログラミングの知識が無くとも並列化の恩恵を受けることが可能である．CPU と GPU のコードの共有が可能であるため，CPU と GPU の二重開発を行う必要がなく，メンテナンスコストを減らすことができる．

## 2. 商品・計算手法

### 2.1 商　　品

マルチコーラブル・パワー・リバース・デュアル・スワップ（CPRD）と呼ばれる，バミューダン型の早期償還条件のついたスワップを計算対象商品とする．CPRD は計算負荷の高い為替ハイブリッド商品の典型例であり，この商品は為替に依存したキャッシュフローをもつ．そのため，時価評価を行うには為替および 2 カ国の金利の確率変動を記述する必要があり，3 ファクター以上の高次元のモデルが必要となるという意味で，計算負荷の高い商品の典型例となっている．

### 2.2 時価評価モデル

時価計算を行うには，為替と 2 カ国の金利をシミュレーションする必要がある．本発表では，為替と 2 カ国の金利の確率変動を記述するモデルとして，為替のスポットレートを Black-Scholes モデル，国内金利・海外金利を Hull-White モデルとした 3 ファクターモデルを用いる．

### 2.3 計算アルゴリズム

バミューダンオプションとは複数の日に権利行使可能なオプションである．バミューダンオプションの時価計算には各権利行使時点における継続価値が必要となるため，通常のモン

---

本稿の内容や意見は，筆者の個人的見解に基づくものであり，三菱ＵＦＪモルガン・スタンレー証券株式会社あるいは株式会社フィックスターズの公式見解を示すものではありません．

†1 三菱ＵＦＪモルガン・スタンレー証券株式会社
Mitsubishi UFJ Morgan Stanley Securities Co., Ltd.
†2 株式会社フィックスターズ
Fixstars Corporation

テカルロ法による前進計算では価格は求まらない．そこで，この継続価値を最小二乗法を用いてあらかじめ与えた基底関数の線形結合で近似する，LSM を用いて時価計算を行う．

時刻 $t$ における為替と金利の値を $X(t) \in R^3$ とする．権利行使日を $t_i (i = 1, 2, \ldots, m)$，評価基準日を $t_0$ とし，$X_i = X(t_i)$ と書く．$V_i(x)$ を時刻 $t_i$ におけるバミューダンオプションの価格とすると，時刻 $t_i$ における継続価値は期待値を用いて，$C_i(x) = E[V_{i+1}(X_{i+1})|X_i = x]$ で与えられる．継続価値の推定に使用する基底関数列を $\{\psi_j(x)\}_{j=1,\ldots,M}$ とし，$N$ を継続価値の推定に使用するパス数とする．このとき，次の 2 ステップで価格は求められる．

ステップ 1 では，最小二乗法による継続価値の推定を行う．あらかじめいくつかのパスを発生させ，そのパス上でのキャッシュフローをシミュレーションする．この結果をもとに，最小二乗法により各権利行使日時点での継続価値の推定量

$$\hat{C}_i(x) = \beta_i \psi(x) \tag{1}$$

を求める．ただし，$\beta_i = (\beta_{i1}, \ldots, \beta_{iM})^T$，$\psi(x) = (\psi_1(x), \ldots, \psi_M(x))$ であり，

$$\beta_i = E[\psi(X_i)\psi(X_i)^T]^{-1} E[\psi(X_i) V_{i+1}(X_{i+1})] \tag{2}$$

を満たす．数値計算上は $E[\psi(X_j)\psi(X_j)^T]$ は $M \times N$ 行列と $N \times M$ 行列の行列積，$E[\psi(X_j)V_{j+1}(X_{j+1})]$ は $M \times N$ 行列と $N$ 次元ベクトルの行列–ベクトル積となり，$\beta_i$ を算出する処理が計算上のボトルネックになりがちである．この過程で得られるバミューダンオプションの価値を interleaving estimator と呼ぶ．

ステップ 2 では，停止時刻型モンテカルロ法による価格付けを行う．新たにシミュレーション用のパスを発生させ，ステップ 1 で求めた継続価値の推定量 $\hat{C}_i(x)$ をもとに最適行使戦略を構成し，現在価値を求める．すなわち，停止時刻 $\tau$ を

$$\tau = \min\{i \in \{1, \cdots, m\} | h_i(X_i) \geq \hat{C}_i(X_i)\} \tag{3}$$

で定義すると，

$$V_0^{\tau}(X_0) = E[h_{\tau}(X_{\tau})] \tag{4}$$

で現在価値が求まる．

## 3. 並列プログラミングフレームワーク

計算の並列化を CPU と GPU の両方に対して容易に行うために並列プログラミングフレームワーク (parallel programming framework，以下 PPF) を作成した．

### 3.1 目　　的

PPF 作成の目的は，計算の並列化を CPU と GPU の両方に対して容易に行うことである．計算の並列化を行うフレームワークは，CPU 上では OpenMP や Intel Thread Building Blocks(以下，TBB)，GPU 上では Thrust C++ Template Performance Primitives ライブラリー (以下，Thrust) がよく知られている．しかし，CPU マルチコアと GPU の両方で使用することができるフレームワークはないように思われる．今回構築した PPF はコードの変更なく，GPU と CPU の両プラットフォームで動作させることができる．PPF を使用することで敷居の高い CUDA プログラミングやマルチスレッドプログラミングを行わなくとも並列化の恩恵を受けることができる．

### 3.2 サンプルコード

以下のコードは PPF を用いたベクトルの足し算 $c = a + b$ の実装例である．

```
template<typename T> class Add{
public:
    Add(const Array<T> a, const Array<T> b) : _a(a), _b(b){}
    __device__ void operator()(int i, const Ary<T> result) const{
        result[i] = _a[i] + _b[i];
    }
private:
    const Array<T> _a, _b;
};

template<typename T> void addVector()
{
    // CPU メモリの確保
    Array<T, Host> h_A(n), h_B(n);
    ...
    // CPU メモリから GPU メモリにコピー
    Array<double, Device> d_A = h_A.copy<Device>();
    Array<double, Device> d_B = h_B.copy<Device>();
    // kernel 実行時のパラメタを設定
    Invoker invoker(maxBlock, maxThread);
    Array<double, Device> d_C
        = invoker.map<double>(Add<double>(d_A, d_B), n);
    // GPU メモリから CPU メモリにコピー
    Array<double, Host> h_C = d_C.copy<Host>();
}
```

### 3.3 特　　徴

PPF は TBB や Thrust と同じ template ライブラリであり，使用方法も似ている．ユーザは各スレッドが行うべき計算を関数オブジェクトとして作成し，Invoker インスタンスに関数オブジェクトを渡すことで，並列に実行される．

コア部分はメモリアロケータ，map や reduce といった並列アルゴリズム，Array で構成される．

メモリアロケータは，GPU と CPU のメモリ構造の違いを吸収する．GPU 上では，host memory，device memory，constant memory，mapped memory を利用することが可能であり，CPU 上では全て host memory となる．

並列アルゴリズムは，map，each，reduce があり，GPU と CPU の並列プログラムの実装方法の違いを吸収する．実装には，GPU は CUDA を使い，CPU マルチコアには TBB を使用した．

Array は，メモリアロケータと並列アルゴリズムを繋ぐユーティリティである．メモリの取り扱いを簡単にするため，参照カウント方式により自動的にメモリは解放される．また，高速化のために多次元配列の場合は自動的に align してメモリを確保する．内部のポインタを取得することも可能であり，既存の CUDA や C++ コードと接続することも可能である．

PPF の欠点は，スレッド間に依存があるアルゴリズムを記述できないこと，shared memory を使用したプログラムを書けないことである．このようなコードを CUDA や C++ で記述し，それ以外を PPF を用いて書くことで開発コストを抑えつつ，並列処理の恩恵を受けることが可能である．

## 4. 検　　証

金融機関は同じ商品を大量に保有しており，更に 1 取引につき大量のシナリオシミュレーション[⋆1]を日々行っている．そのため，大量の時価計算のスループットを上げることで，大幅なコスト削減が可能となる．また，時価計算のレイテンシが数十倍高速化されると，これまで不可能だったインディケーションの多様化やヘッジ取引の自動発注などを通じて，ビジネス・プロモーションやトレーディングの世界の景色が変わってくる可能性が強い．

本節では，3 ファクターモデルの LSM での実装を CPU と GPU に対して行い，それぞれの計算性能をスループット，レイテンシに関して比較した結果について説明する．なお，1PV 計算と書いた場合，1 取引明細の時価計算を意味する．

### 4.1 ハードウェア構成

GPU は Tesla C2050，CPU は Xeon X5650 プロセッサ 2.67GHz × 2 スロット[⋆2]，メモリは 3.3GBytes，OS は Windows7 Professional 32 bit，開発環境は Visual Studio 2008 SP1，C++ Composer XE 2011, CUDA3.2 を利用した．

### 4.2 計 算 設 定

満期が 10 年，クーポンの支払い間隔が半年，権利行使日が半年毎の CPRD に対する時価を計算する．数値計算手法は LSM を使用し，シミュレーショングリッドが半年毎，継続価値推定 (ステップ 1) に使用するパス数は 20000，最適停止モンテカルロに使用するパス数は 80000，継続価値推定に使用する基底関数の次元は 30 とする．

### 4.3 実 装 方 法

本検証では，以下の 3 つの実装を行った．

(1)　PPF で全て実装 (CPU と GPU のコード共通)
(2)　PPF は使用せずに実装 (CPU のみ)
(3)　PPF + CUDA による実装 (GPU のみ)

(2) は PPF を用いず最適化した CPU 向けの実装であり，マルチスレッディングには OpenMP を用いている．(3) は (1) の GPU の最適化実装であり，LSM の継続価値の推定部分を CUDA 実装を用いて最適化を行い，残りの部分は (1) をそのまま利用した．

### 4.4 PPF と PPF を使用しなかったコードの性能比較

1 取引明細の時価計算の性能比較を PPF と PPF を使用しなかった場合について行う．

表 1 は，1 取引明細の時価計算を CPU 実装 (1) と CPU 実装 (2) の比較，GPU 実装 (1) と GPU 実装 (3) の比較を行ったものである．

PPF を使用しない場合に性能が向上するのは，2 つの要因がある．1 点目は，PPF 自体のオーバーヘッドである．これは表 1 の#1 の結果から分かる．2 点目は，PPF に適合しにくいアルゴリズムに PPF を使用すると性能が出にくいことである．GPU の実装 (1) は，LSM の継続価値の推定が実行時間の大部分を占めている．この部分は PPF のみを利用すると，PPF の shared memory を使えないという制限から非常に効率の悪い実装を行う必要がある．そのため，この部分を CUDA で実装すると性能が劇的に向上する．

⋆1 市場データなどの計算パラメタを変化させて同一取引の計算を行うこと

⋆2 合計 12 コア

**表 1** PPF と最適化後の性能比較
Table 1 Comparison of PPF and optimized code

| # | CPU/GPU | 使用コア/SM 数 | 実装 (1) | 実装 (2) | 実装 (3) | 実装 (1) を基準とした性能向上 (倍) |
|---|---|---|---|---|---|---|
| 1 | CPU | 1 コア | 1.135 | 0.870 | - | 1.305 |
| 2 | CPU | 12 コア | 0.462 | 0.327 | - | 1.415 |
| 3 | GPU | 1SM | 0.381 | - | 0.210 | 1.814 |
| 4 | GPU | 14SM | 0.242 | - | 0.098 | 2.469 |

### 4.5 LSM の性能比較

表 2 は，CPU，GPU について，PPF を用いず最適化した性能を比較したものである．CPU は 12 コアを利用し，GPU は 14SM を利用して，168 個の PV 計算を行った．CPU は実装 (2)，GPU は実装 (3) を用い，#1 の PV 計算に比べて，スループットとレイテンシがどの程度の性能が出ているのかを調べる．ここでは，スループットを 1 秒間当たりの PV 計算数，レイテンシを 1PV 計算にかかる時間 (秒) とする．

レイテンシについて考えると，当然のことであるが，GPU の 14SM 全部を使用して 1PV 計算を行うのが 1PV 計算単独としては最も高速である．ただし，14SM を使用した場合は 1SM しか使用しない場合の 2 倍程度しか速くならず，使用 SM 数に関してスケールしにくい．同様に，CPU に関しても，使用コア数に関してスケールしにくい．この原因は，本計算のボトルネックが LSM の継続価値推定であり，この部分の計算が使用コア数に関してスケールしにくいからである．

次に，スループットについて考える．高スループットを達成するのであれば，1PV は 1SM で計算し，14SM を用いて複数 PV の同時計算を行うのが最もよく，1CPU コア比 58.8 倍 ($4.90 \times 12$) 性能が高い．これは，CPU でも同様である．しかし，この手法では GPU のメモリ量がボトルネックとなり，タイムグリッド数の増加，サンプルパス数の増加などのメモリ量を増やす操作を行うと，同時に計算できる PV 計算数を減少させなければならず，スループットが落ちてしまうという問題がある．

## 5. 考　　察

### 5.1 PPF

PPF を利用することで，比較的容易にマルチコアや GPU の恩恵を受けることができる．更に，CPU と GPU で同じコードを利用することができるため，開発コストを減らすことができるという利点がある．しかし，計算粒度の小さい計算は PPF 自体のオーバーヘッドが無視できず，性能が出にくい．また，スレッド間に依存があるアルゴリズムは PPF に向きにくいという問題がある．速度を追い求めると CUDA で実装する必要があるが，開発コストや保守コストが増加してしまい，ビジネスでは利用しにくいと思われる．そのため，ビジネスで GPU を活用するには，オーバーヘッドが小さく，柔軟な記述が可能であり，かつユーザが使いやすいフレームワークを作成する必要があると感じる．

**表 2** LSM の性能比較
Table 2 Perfomance comparision of LSM

| # | 実装方法 | 1PV 計算に利用するコア/SM 数 | スループット (PV 数/秒) | レイテンシ (秒) | スループット性能向上率 (#1 基準) | レイテンシ性能向上率 (#1 基準) |
|---|---|---|---|---|---|---|
| 1 | CPU 実装 (2) | 1 コア | 13.78 | 0.87 | 1.00 | 1.00 |
| 2 | CPU 実装 (2) | 6 コア | 4.96 | 0.40 | 0.36 | 2.16 |
| 3 | CPU 実装 (2) | 12 コア | 3.06 | 0.33 | 0.22 | 2.66 |
| 4 | GPU 実装 (3) | 1SM | 67.50 | 0.21 | 4.90 | 4.20 |
| 5 | GPU 実装 (3) | 3SM | 36.56 | 0.14 | 2.65 | 6.37 |
| 6 | GPU 実装 (3) | 14SM | 10.17 | 0.10 | 0.74 | 8.86 |

### 5.2 GPU による高速化

PPF + 一部 CUDA 実装による約 60 倍の高速化は，最適化された CPU コードを元にした場合は十分に高速化されていると考えることができるが，コスト削減効果の面から見ると十分に高速であるとは言えない．金融計算へ GPU を適用する際には，保守コストや開発コストなども勘案して，効果のある計算対象を慎重に選択する必要がある．

## 6. 結　　論

PPF を利用することで CPU と GPU で同一コードを共有することができ，容易に計算の並列化が可能であった．しかし，PPF 自体のオーバーヘッドが大きく，また，PPF に適さないアルゴリズムの性能が出ないという問題があった．

PPF と一部 CUDA 実装を行った PV 計算のスループットは十分に高速化された CPU コードと比較して 1CPU コア比 60 倍の性能向上を達成した．十分に最適化された CPU コードと比較すると十分高速化されていると考えられるが，コスト削減効果の観点からは十分に高速化されていると言えないという結論に至った．

Vol.2011-HPC-130 No.10
2011/7/27

## 7. 今後の課題

保守コストやプログラミングコストなども勘案して，GPU の利用効果のある計算対象を慎重に選択する必要がある．どのような計算が GPU の計算に向いているのかを検証することは今後の課題である．

また，現状，PPF 上でプリミティブな処理を組み合わせてアプリケーションを構築すると，PPF 自身のオーバヘッドが無視できない．高い汎用性と小さいオーバーヘッドを実現するために，実行時のカーネルコード生成などの PPF の拡張を行い，GPU 導入コストの削減を実現したい．

## 参 考 文 献

1) Longstaff, F.A. and Schwartz, E.S.: Valuing American Options by Simulation: A Simple Least-Squares Approach , *Review of Financial Studies.*, Vol. 14, No. 1, 113–117, 2001
2) Giovanni, C. et al.: *Modelling, Pricing, and Hedging Counterparty Credit Exposure: A Technical Guide* , Springer-Verlag (2010).
3) Joshi, M.S.: Graphical Asian Options, *Wilmott Magazine*, Vol. 2, No. 2, pp.97–107(2010).
4) Victor, W.L. et al.: Debunking the 100x GPU vs. CPU Myth: An Evaluation of Throughput Computing on CPU and GPU, *SIGARCH Computer Architecture News*, Vol. 38, No. 3, pp. 451–460, 2010.