GPU 向け高性能のブラウン橋（Brownian Bridge）：
帯域幅の制約を受けるアプリケーションへの教訓

JACQUES DU TOI

概要：このレポートは非常に順応性のあるブラウン橋（Brownian Bridge）生成器について、NVIDIA C2050上でピークに近いパフォーマンスを実現するGPU 実装とともに説明します。そのパフォーマンスについて、高性能のx86-64システム上で稼働するOpenMPの実装と比較しています。GPU 実装は少なくとも10倍のパフォーマンスの向上を示しています。比較結果の詳細はセクション８で説明しています：特に、ブラウン橋アルゴリズムはメモリ帯域幅の制約を受けるためマルチコアCPU上ではスケーラビリティが低いことが観察されています。またGPUアルゴリズムの進化について記述されています。ピークのパフォーマンスの実現にはGPUプログラミングに関する「定説」への疑念、特に占有率の重要性、共有メモリの速度、分岐の影響について疑念を抱くことが必要でした。

目次

1. 概要とソフトウェア要件 1
2. アルゴリズム設計 4
3. １番目の GPU 戦略 7
4. ２番目の GPU 戦略 7
5. ３番目の GPU 戦略 8
6. ４番目の GPU 戦略 9
7. ５番目の GPU 戦略 10
8. 結果の概要と結論 11
参考文献 14

1. 概要とソフトウェア要件

ブラウン橋アルゴリズム（[2]参照）はブラウン運動のサンプルパスを構築するための有名な手法です。その手順は以下のように要約されます。

$t_0 < T$ を２度設定し、区間$[t_0, T]$のブラウン運動を $\mathrm{X} = (X_t)_{t_0 \le t \le T}$ と表します。また$\mathrm{N} \ge 1$ となる$t_0 < t_1 < \cdots < t_N < T$ を満たす時間を$(t_i)_{1 \le i \le N}$ とします。

目的は　標準の正規乱数 $Z_0, Z_1, \ldots, Z_N$ を用いて $\{X_{t_i}\}_{1 \le i \le N}$ と X_t の値をシミュレーションすることです。$X_{t_0} = x$（しばしば x = 0)の値は常に既知であると仮定します。また常に $X_T = x + \sqrt{T - t_0} Z_0$ と設定します。

$\{X_{t_i}\}_{1 \le i \le N}$ の残りの値を埋めるためにブラウン橋アルゴリズムは補間法を使用します。 ２つの点 X_{t_i} と X_{t_k}が既知である場合、 $t_i < t_j < t_k$である３つめの点 X_{t_j} は以下の式で計算する

ことができます。

(1) $$X_{t_j} = \frac{Xt_i(t_k - t_j) + Xt_k(t_j - t_i)}{t_k - t_i} + Z_j\sqrt{\frac{(t_k - t_j)(t_j - t_i)}{t_k - t_i}}$$

従ってアルゴリズムは反復します。開始値が $X_{t_0} = x$ で終了値が $X_T = x + \sqrt{T - t_0}Z_0$ の場合、$1 \le \mathrm{i} \le N$ となる３つめの点 Xt_i を計算することができます。

３つの点 X_{t_0}, X_T, X_{t_i} がある場合、$\mathrm{j} \neq \mathrm{i}$ である４つめの点 X_{t_j} を最近傍の補間法によって計算することができます。処理は全ての点が生成されるまで続きます。

もしブラウン運動が多次元である場合、アルゴリズムは若干修正することにより使用することができます。

X_{t_i} と Z_i はベクトルになり、相関は以下のように設定することにより導入されます。

(2) $$X_{t_j} = \frac{Xt_i(t_k - t_j) + Xt_k(t_j - t_i)}{t_k - t_i} + CZ_j\sqrt{\frac{(t_k - t_j)(t_j - t_i)}{t_k - t_i}}$$

ここで C は CC' がブラウン運動の共分散構造を表す行列です。ブラウン橋が確率微分方程式を解くために使用される場合、$\left(X_{t_{i+1}} - X_{t_i}\right) / \left(t_{i+1} - t_i\right)$ の式のスケーリングされた増分を生成するほうが適切です。ブラウン運動のサンプルパスの点 X_{t_i} を生成するよりもいくらか容易であることがわかります。スケーリングされた増分についてこれ以上述べませんが、増分生成についてのタイミングについてはセクション８で説明しています。

1.1. ブラウン橋構築順序。 ブラウン橋アルゴリズムはどの点 X_{t_j} がどの点 X_{t_i} と X_{t_k} から補間されるかを示すまでは完全には特定されません。例えば、N = 12 で時間が $\{t_i\}_{1 \le i \le 12}$ の場合、ブラウン橋を以下の順で構築できます。

(3) $$\mathrm{T}\ \ t_6\ \ t_3\ \ t_9\ \ t_1\ \ t_4\ \ t_7\ \ t_{11}\ \ t_2\ \ t_5\ \ t_8\ \ t_{10}\ \ t_{12}$$

これは以下を意味します。X_{t_6} が X_{t_0} とX_T の間に補間されています; X_{t_3} はX_{t_0}と X_{t_6} の間に補間されています; X_{t_9} は X_{t_6} と X_T の間に補間されています; X_{t_1} は X_{t_0} と X_{t_3} の間に補間されています ; X_{t_4} は X_{t_3} と X_{t_6} の間に補間されています;しかし同様に以下の順で構築することもできます。

(4) $$\mathrm{T}\ \ t_2\ \ t_4\ \ t_3\ \ t_9\ \ t_1\ \ t_7\ \ t_{12}\ \ t_5\ \ t_{10}\ \ t_6\ \ t_{11}\ \ t_8$$

ここでは X_{t_2} は X_{t_0} と X_T の間に補間されています; X_{t_4} は X_{t_2} と X_T の間に補間されています; X_{t_3} は X_{t_2} と X_{t_4}の間に補間されています ; X_{t_9} は X_{t_4} と X_Tの間に補間されています。X_{t_1} は X_{t_0} と X_{t_2} の間に補間されています;どちらの構築順序もともに正しいです。実際に、時間 $\{t_i\}_{1 \le i \le N}$ の順列は正しいブラウン橋構築順序を指定しています。もし $\Theta \equiv \{\theta_i\}_{1 \le i \le N}$ が $\{t_i\}_{1 \le i \le N}$ の順列を表している場合，$\theta_i \in \Theta$ について以下の式で表すことができます。

(5) $$X_{\theta_i} = \frac{X_l(r - \theta_i) + X_r(\theta_i - l)}{r - l} + Z_i\sqrt{\frac{(r - \theta_i)(\theta_i - l)}{r - l}}$$

ここで$\mathrm{l} = \max\{t_0, \theta_j \mid 1 \le j < i, \theta_j < \theta_i\}$ は θ_i よりも小さい最大の「既知」の点であり、$\mathrm{r} = \min\{T, \theta_j \mid 1 \le j < i, \theta_j > \theta_i\}$ は θ_i よりも大きい最小の「既知」の点です。

ここで　もし対応する値 X_s が X_{θ_i} を計算するまでに計算済み（つまり既知である）の場合時間点 s は既知であることを意味します。 X_{θ_i} を補間する際に既知の最近傍に確実に補間します。例えば、上記の(4) のように、X_{t_0} と X_T の間ではなくX_{t_2} と X_T の間に X_{t_4} を補間します。

1.2. 準乱数。 Z_i が疑似乱数生成器から生成されている場合、一方のブラウン橋構築順序を他方よりも選ぶ理論的理由はありません。それぞれの Z_i が他の Z_i から独立していて同一だからです。しかしながらブラウン橋アルゴリズムは低ひずみ数列（Sobol列など）から生成される準乱数とともに頻繁に使用されており、この場合状況はかなり異なります。ブラウンアルゴリズムと共に準乱数点を何故使用するのかについての詳細は[2] を参照してください。基本的に、疑似乱数点でブラウン橋サンプルパスの空間を埋めるよりも準乱数点のほうがより均一に空間を埋めるという考えによるものです。その利点は $\mathrm{N}+1$ 次元関数のモンテカルロ積分で準乱数点を使用することと非常に似ています。

$\mathrm{N}+1$ 次元準乱数点$(Z_0, Z_1, \ldots., Z_N)$ はサンプルパス全体の構築のために使用されています。問題は、多くの準乱数生成器について低次元の準乱数点 $(Z_0, Z_1, \ldots.)$ は一般的に高次元の準乱数点 $(\ldots. Z_{N-1}, Z_N)$ よりもはるかに良い均一性の特性を示すということです。そのため低次元の準乱数点はより有益であり、ブラウン運動の最も重要な部分を構築するのに使用する必要があります。例えば時間 η でのブラウン運動の動きに特に敏感なモデルを考える場合、以下を確認します。

- 時間η は補間点の一つである
- X_η は低次元からの Z_i を用いて構成されている
- X_η は低次元からの Z_i を用いて構成されたデータ点の間に補間されている

この考えは上記の（3）と（4）で描かれているようにブラウン橋構築順序と非常に自然に結びついています。もしブラウン橋構築順序を時間 $\{t_i\}_{1\le i\le N}$ の数列 $\Theta \equiv \{\theta_i\}_{1\le i\le N}$ を通じて指定した場合、最も重要な時間点が θ_1, θ_2 によって与えられるようにします。それにより X_T を構築するのに Z_0 を使用することができ、X_{θ_1} を構築するのに Z_1 を使用することができ、X_{θ_2}を構築するのに X_{θ_1} を使用することができます。それぞれの点 X_{θ_i} を構築するのにどの次元が使用されるかがわかるように、構築順序 Θ は準乱数点の次元に直接結びつけられます。表記しやすいように、X_{θ_1}（$0 \le \mathrm{i} \le \mathrm{N}$）の構築に Z_i が使用されるよう $\theta_0 \equiv T$ を設定します。

1.3. メモリ帯域幅の制約を受けるアルゴリズム。 上記(1)の乗算 $(t_k - t_j) / (t_k - t_i)$、$(t_j - t_i) / (t_k - t_i)$、 $\sqrt{(t_k - t_j)(t_j - t_i)/(t_k - t_i)}$ は事前に計算することができます：

それらのうち N 個だけあり、一度構築順序が決まるとそれらは変わりません。(1)を考慮する場合これらの値は既知の値として取り扱われます。それらは単純にメモリからフェッチされる必要があります。そのためブラウン橋はメモリ帯域幅の制約を受けるアプリケーションです：転送さ

れるデータ量に関して非常に小さな計算が行われます。従って、メインメモリからの Z_i の移動とメインメモリへの X_{t_i} の移動だけを考慮する場合、私たちの目標は最大のメモリ帯域幅（あるいはできるだけそれに近い帯域幅）を達成することでした。データの計量では必要以上のデータ通信量は消えていくような現実の世界を反映するよう（ユーザの視点から）それ以上のデータ移動は考慮されません。ユーザはブラウン橋構築順序を完全に自由に指定できる必要があります。ユーザが完全な構築順序を自分で作らなくてもいいようにヘルパールーチンが提供されます。アルゴリズムは従来のx86-64のアーキテクチャ上とNVIDIA GPU上でCUDAを用いて実装されます。私たちの考察はGPUの実装に重点がおかれています。x86-64 マルチコアの実装と比較した結果はセクション8 に記載されています。

2. アルゴリズム設計

ユーザがブラウン橋構築順序を指定するためにはこれを抽象する必要があり、従って全ての構築順序を共通の書式に変換することが必要です。これを達成するために２つのステップが採用されました。最初のステップ（初期化）で、ユーザはブラウン橋構築順序を与えます。この順番は処理され、実行戦略に変換されGPUにコピーされます。
ブラウン橋構築順序と時間点が変わらない限り、実行戦略は有効なままです。２つめのステップ（生成）でブラウン運動のサンプルパスが、入力された正規乱数（疑似乱数か準乱数どちらか）から生成されます。一連のブラウン運動のサンプルパスの生成のために生成ステップは何度か実行することができます。

2.1. ローカルスタック。 ブラウン橋アルゴリズムを効果的に実装するにはローカルスタックと呼ばれるローカルな作業領域の配列が必要です。前もって計算された X_{θ_i} が新しい点を補間するのに使用されるため、計算点はなるべくハードウェアキャッシュの処理ユニットにできるだけ近く保たれる必要があります。ハードウェアキャッシュは全てのブラウン点を保持するには小さすぎるため、それぞれのタイムステップでアルゴリズムはどの点を保持するか決定する必要があり、これらはローカルスタックに格納されます。またアルゴリズムはローカルスタックの点がいつ不要になるかを決定する必要があります。それにより新しい点を置き換えることができます。これはローカルスタックが大きくなりすぎなければ L1 あるいは L2 キャッシュにフィットするということが十分あり得るということです。ブラウン橋アルゴリズムの鍵となる要件はローカルスタックが小さいままであるということです。

2.2. 初期化と実行戦略。 計算された実際のブラウン運動の点を変更せずに任意のブラウン橋構築順序を変えることが可能であることがわかっています。
これを説明するため、上記の (3) を考えます。この構築順序は以下の構築順序と等しいです。

(6) $$\mathrm{T}\ \ t_6\ \ t_9\ \ t_3\ \ t_{11}\ \ t_7\ \ t_4\ \ t_1\ \ t_{12}\ \ t_{10}\ \ t_8\ \ t_5\ \ t_2$$

ここで X_{t_6} は X_{t_0} と X_T の間に補間されています。; X_{t_9} は X_{t_6} と X_T の間に補間されてい

ます。; X_{t_3} は X_{t_0} と X_6 の間に補間されています。; $X_{t_{11}}$ は X_{t_9} と X_T の間に補間されています。; この構築順序から得られる結果は、同じ Z_i が各ブラウン運動の点の作成に使用される限り(3) からの結果と同一です。従って(6)で X_T を生成するために Z_0 を使用します：X_{t_6} を生成するために Z_1 を使用します：X_{t_9} を生成するために Z_3 を使用します：X_{t_3} を生成するために Z_2 を使用します： $X_{t_{11}}$ を生成するために Z_7 を使用します：

任意のブラウン橋構築順序は多くのローカルスタックを容易に使用することができます。従って初期化ステップのタスクは、ユーザの構築順序を取り込み、必要最低限のスタックを使用する同等の構築順序を（ Z_i の順列と共に）見つけることです。この処理はかなり技術的であり、これ以上は説明をしませんが、出力結果は一般のステップに渡される実行戦略です。実行戦略は新しいブラウン橋構築順序と Z_i の順列から成り立ちます。

2.3. 定説： 占有率、共有メモリ及び分岐。 GPU プログラミングに関する定説とは、占有率はメモリ帯域幅の制約を受けるアプリケーションにとって重要であること、共有メモリは速いということ、分岐は避けるべきであるということです。 ここで分岐は、ワープ・ダイバージェンス（ワープ内の分岐）を意味していません。従来のシリアル分岐を意味しています。GPU上では、これはワープ内の全てのスレッドが同じ分岐をもつ場合のif-then-else　文と同じです。特に内側のループで分岐を避けることはCPUコードに対する標準の最適化戦略です。

占有率を増加させるために、カーネルはできるだけ少ないレジスタを使用する必要があります。カーネルはストリーミングマルチプロセッサ（SM）ごとにできるだけ多くのスレッドで起動されなくてはなりません。これらのスレッド全てからのメモリ要求はメモリバスを満たすということです。いくつかのスレッドに対してデータがフェッチされるとき（そのためデータの到着を待っている間何もできません）、データが到着した他のスレッドは引き続き実行が可能です。

ブラウン運動の点 X_{θ_i} を補間するために、以下のステップを実行する必要があります。

(a) X_{θ_i} の左隣と右隣を決定し、ローカルスタックのそれらの位置を見つけます。これらは X_{θ_i} が補間される上記(5)の点 X_l と X_r です。
(b) ローカルスタックから左隣と右隣を読み取ります。
(c) どの乱数点Z_i を使用するか決定します。
(d) グローバルメモリからZ_i 点を読みます。
(e) (5)を使用して X_{θ_i}を計算します。
(f) メインメモリのどこに X_{θ_i} を格納するかを決定します。正しい順、つまり $X_{t_1}, X_{t_2}, \ldots, X_T$ で確実に点は格納される必要があります。
(g) メインメモリに X_{θ_i} を格納します。
(h) ローカルスタックのどこに X_{θ_i} を格納するか決定します。
(i) ローカルスタックに X_{θ_i} を格納します。

ステップ (a)、(c)、(f) と (h) は全て初期化のステップで行われ、共に実行戦略を構成します。物理的に実行戦略はGPUにコピーされる整数の配列で構成されます。またサンプルパスを生

成するので各 CUDAスレッドにより読み取られます。
各タイムステップで、新しいブラウン運動の点を計算するためにスレッドは５つの正数を読む必要があります：ローカルスタックの左隣と右隣のインデックス;点 Z_i のインデックス;
グローバルメモリの X_{θ_i} の記憶域インデックス；ローカルスタックの X_{θ_i} の記憶域インデックス。これらの情報により、生成ステップは簡単に以下で表されます。
(a) ローカルスタックから左隣と右隣を読み取ります。
(b) グローバルメモリから Z_i 点を読みます。
(c) (5)を使用してX_{θ_i} を計算します。
(d) メインメモリに X_{θ_i} を格納します。
(e) ローカルスタックに X_{θ_i} を格納します。
これは分岐なしの処理です。従って非常に有効です。さらにローカルスタックは速度の速い共有メモリに格納することができます。ワープ・ダイバージェンスはありません。(各スレッドは別々のブラウン運動のサンプルパスを生成しますーワープ内のスレッドは同時に同じ処理を行います。) グローバルメモリへの全てのアクセスは調整され、完全にまとめられます。共有メモリをアクセスする際にバンクコンフリクトはありません。全体として、アルゴリズムは理想的にGPUに適していると思われ、非常に良く機能しています。

2.4. 検証システムと検証問題。検証システムは8GB RAM を搭載した2.8GHz で動くIntel Core i7 860 とError Checking and Correction (ECC) を搭載した Tesla C2050で構成されています。このシステムはCUDA Toolkit v4.0 を備えた64 bit Linux です。基本検証問題は64のタイムステップを持つ1,439,744の一次元のブラウン橋サンプルパスを生成するものです。構築順序は(3)で示されているような標準の二分法の順序です。全ての計算は単精度で実行されます。メモリバスを8バイトの転送で飽和させるよりも4バイトの転送で飽和させる方が難しいです。検証問題は計算コアへの351MBのデータ（Z_i）の移動とグローバルメモリへの351MBのデータ（X_{t_i}）の戻しで構成されています。全てのパフォーマンス測定はパフォーマンスカウンターの利用が可能なNVIDIA の Compute Visual Profilerを通じて得られました。もしアルゴリズムがローカルメモリバストラフィック(L2キャッシュあるいはグローバルメモリへあふれたL1キャッシュの転送量)を取り込んだとすれば、これは記録されました。生成ステップのみが計測され、初期ステップは無視されました。
C2050 のピークのメモリ帯域幅は144GB/s です。しかしながらこれは ECC が作動してない場合の数字です。ECC が作動している場合、カードのピークの帯域幅は120GB/s より若干低い値に減少します（[1]を参照）。従って目標はできるだけ全体の転送速度を 120GB/s 近くにすることです。
以下で引用されている全てのパフォーマンスについての数字は、カーネル実行時間の測定で見つかった最適な起動設定に関するものです。占有率についての数字は起動設定に関するものです。

3. 1番目の GPU 戦略

実行戦略がローカルスタックのインデックスを含む整数の配列、Z_i の配列とブラウン橋記憶域（出力）配列から成り立っていることを思い出してください。これらの整数は生成ステップの間固定であり、ワープ内の各スレッドは同時に同じ要素(ブロードアクセス)にアクセスするので、一定のメモリにそれらを格納するのは明らかであると思われます。従って最初のGPUの実装は以下のようになります。

スペースを節約するため、実行戦略はreadインデックスとwriteインデックスをローカルスタックに格納するために 8 ビットの符号なしの整数を使用し、Z_i readインデックスと X_{θ_i} write インデックスを格納するために16 ビットの符号なし整数を使用しました。従って各ブラウン運動の点の実行戦略の総サイズは、7 バイトの総コストに対して2 X 16 ビット + 3 X 8 ビットでした。これらの値は 16 bit と 8 bit のブロードキャストを通じてコンスタントメモリから読み込まれました。

スタックは L1キャッシュに存在するよう、ローカルメモリに保持され、L1 キャッシングは停止されました。L1キャッシュと共有メモリ両者に対して同じ物理ハードウェアが使用されているので、これは共有メモリにスタックを置くのと同じはずです。

3.1. 並列化と多次元ブラウン運動。 ブラウン運動が一次元であるとき、各 CUDAスレッドは他のCUDAスレッドのブラウン橋サンプルパスを単独で作成することができるのは明らかです。この場合、各スレッドブロックが十分な量の作業をするように、各スレッドにいくつかのブラウンパスを作成させます。しかしもしブラウン運動が多次元である場合、行列ベクトルの積 CZ_i を計算する必要があります。この実装は、コンスタントメモリに C を入れ、スレッドに各 CZ_i ベクトルの値を共有メモリに読みこませる方法を用いました。スレッドは同時に発生し、各スレッドは行列ベクトルの積 CZ_i の行を計算します。C へのアクセスはブロードキャストのアクセスパターンに従っています。ブラウン運動の点 X_{θ_i} は前に述べたように他のすべての CUDA スレッドとは独立して計算されます。このアプローチは２つの _syncthreads() 呼び出しが各タイムステップで行われる必要があることを意味します。

3.2. パフォーマンス。 カーネルは20 個のレジスタを使用し、占有率が100%であり、パフォーマンスは 29GB/s です。

4. 2番目の GPU 戦略

帯域幅の制約を受けるアプリケーションについて頻繁に耳にするアドバイスは、各スレッドがより多くのデータを転送し処理できるよう、ベクトルデータタイプを使用するということです。各スレッドがfloat2、float3 あるいは float4のように Z_i を読みこみ X_{θ_i} を書けるようにセクション３のアルゴリズムを調整しました。

必要とされるローカルスタックの量が２倍、３倍あるいは４倍と増えるように、各スレッドは２、３あるいは４個のブラウン橋サンプルパスを一度に処理します。その後は、アルゴリズムは変わりませんでした：実行戦略はコンスタントメモリから読み込まれ、タイムステップごとに２つの同期があり、行列　C はコンスタントメモリに保持されました。

4.1. パフォーマンス。 カーネルは20個から34個のレジスタを使用し、占有率は100%から54%の間で、一般にパフォーマンスは低いものでした。グローバルメモリの転送量は増加しましたが、これはローカルスタックが L2 キャッシュへあふれ、かなり大きくなることによるものです。スタックはローカルメモリに保持され、そのため L1 キャッシュに存在します。占有率を高めるため、各スレッドブロックは多くのスレッドで起動されます。しかしレジスタや共有メモリに対して、ランタイムがSMごとに１つあるいは２つのブロックしか置かないよう強制する制限がないため、ランタイムは各SM上にいくつかのブロックを置いています。Ｌ１キャッシュを使いはたし、スタックがL2 とグローバルメモリにあふれる原因となります。少ないブロックと少ないスレッドでカーネルを動かすことはメモリへのあふれを軽減しますが、Z_i と X_{θ_i} のグローバルメモリへのトランザクションが少ないためパフォーマンスの悪化をもたします。
全体で、各カーネルはセクション３のカーネルよりゆっくり動きます。これは有効なパフォーマンスが 29GB/s に満たないことを意味します。

5. 3番目の GPU 戦略

アルゴリズムの通常の動きがメモリバスを飽和させるための十分なメモリの命令を生成できないため、明示的なデータプリフェッチを取り入れました。この発想はGPUの計算コアがデータロード命令を失速させないという事実を生かしています：ロードのために使用されたレジスタ（つまりグローバルメモリからのデータがロードされるレジスタ）が演算の引数になると失速します。従ってもし一つのスレッドがいくつかのロード操作を異なるレジスタ上で発生させる場合、プログラマはプログラムの後半までこれらのレジスタを使用しないよう気をつけます。ロードはスレッドが実行を続けている間非同期的に生じます。
パスの計算を始める前に各スレッドが P個の正規乱数をプリフェッチできるよう私たちはセクション３のアルゴリズムを変更しました。パスの各点 X_{θ_i} が計算されると、対応する正規乱数が使用され、$X_{\theta_{i+P}}$ に対応する新しい正規乱数が同じレジスタにロードされます。このように、P個の正規フェッチが常にin-flight（実行中）となります。この戦略では、レジスタが正しく扱われているかを確認するためメインの計算ループをP回展開することが必要です。

5.1. パフォーマンス。 いくつかの実験後、私たちは最適値が　P = 8 であることわかりました。カーネルが34個のレジスタを使用し、占有率が 58% であり、パフォーマンスが 58GB/s であることを表しています。

6. 4番目の GPU 戦略

この点から、アルゴリズムの大局的な再考が必要であることは明らかでした。基本的な問題はスレッドと全てのメモリ操作との間にある間接参照のレベルであると思われました。メモリがアクセスされる前に、各スレッドは最初にアクセスが生じるインデックスをフェッチする必要があります。またデータをアクセスするのにインデックスを使用する必要があります。従って各メモリ操作はメモリへの２つのトリップを必要とします。１つはインデックスへフェッチするためのトリップで、２つ目はインデックスを用いてデータをアクセスするためのトリップです。
これはプロセス全体の速度を落としています。グローバルメモリバスを飽和させる速度でインデックスを配布できるほどコンスタントメモリは速いわけではありません。実行中に各スレッドがインデックスをその場で「計算した」合成実験ではメモリバスの飽和に近づきました。従ってボトルネックがインデックスのフェッチにあるように思われました。
実際にコンスタントメモリから共有メモリへのインデックスの移動には改善が見受けられません。コンスタントメモリと同様に、共有メモリは遅すぎてグローバルメモリバスを飽和させる速度でインデックスを提供することができません。その場でインデックスを計算することが不可能なので、ボトルネックを回避する方法を見つける必要があります。
データのプリフェッチは何らかの希望を与えるようです。しかし正規乱数をプリフェッチしたい場合、それらのインデックスもプリフェッチしなければなりません。（インデックスはロードをもたらすために必要とされるからです。）またもしこれらのインデックスをプリフェッチしている場合、おそらく全てのインデックスをプリフェッチすることは意味があります。
従って、アルゴリズムを以下のように変更しました：

(a) スレッド同士は多次元のブラウン橋サンプルパスの生成をもはや協力し合っては行いません。各スレッドは各サンプルパスの全次元を計算します。行列ベクトル積CZ_i がレジスタ内のデータ上で実行できるよう行列 C はレジスタに移動されます。全ての同期バリアが除去されます。

(b) Z_i の値はもはや共有メモリに読み込まれないため、ローカルスタックはコンスタントメモリから共有メモリへ移動されます。

(c) 実行戦略はコンスタントメモリから共有メモリへ移行されます。しかし共有メモリが小さすぎて保持できないため、実行戦略はグローバルメモリに残る必要があります。その一部は必要に応じて共有メモリにコピーされます。

(d) 実行戦略を定義するインデックスはプリフェッチされます。各ステップで、５個のインデックスが必要とされ、２ステップ分のインデックスがプリフェッチされます。これは最初のブラウン運動の点 X_T が計算される前に10個のインデックスがプリフェッチされることを意味します。

(e) インデックスのプリフェッチに加えて、P個の正規乱数もグローバルメモリからプリフェッチされます。

(f) アルゴリズムの残りの部分は同じままです：左隣と右隣がスタックから読み込まれ、新しいブラウン運動の点が計算され、スタックとグローバルメモリに順に格納されます。

6.1. パフォーマンス。 上記の変更によりコードの複雑さは激増しました。考え方は比較的に単純ですが、コードを書く際にコードを理解するのにある程度時間がかかります。プリフェッチのレジスタは正しく扱われる必要があるだけでなく、実行戦略のセクションを共有メモリにコピーすることによって生じる稀なケースとクリーンアップコードに対して例外的に注意を払う必要があります。

これにはプリフェッチを伴いますが、これはどのセクションをコピーするか知るための注意深い記録処理が必要であるからです。つまり発想をロバスト実装に転換することは簡単ではありません。結果的に、カーネルはスレッドごとに42個のレジスタを使用し、占有率は 45% で、 85GB/s で動きます。

GPU戦略	レジスタ	占有率	パフォーマンス	ピークの %
１番目	20	100%	29GB/s	24.1%
２番目	20 to 34	100% to 54%	<29GB/s	<24%
３番目	34	58%	58GB/s	48.3%
４番目	42	45%	85GB/s	70.8%
５番目 (プリフェッチなし)	24	43%	79GB/s	65.8%
５番目 (プリフェッチあり)	33	58%	102GB/s	85%

表 1. 異なる GPU 戦略の比較

7. 5番目の GPU 戦略

前述の GPU カーネルのパフォーマンスは理論上のピークのわずか70%です。プリフェッチの余地はあまりありません。アルゴリズムの中で最も可能性の高いボトルネックはローカルスタックへの転送量やローカルスタックからの転送量、関連するインデックスのフェッチです。各タイムステップでスタックから２つの値を読んで一つの値をスタックに書き出します。これは３つのインデックスをフェッチすることを意味します。インデックスは共有メモリに存在します。これは、各タイムステップで共有メモリから３つの値をロードすることを意味します。さらに２つの値をロードし最終的に共有メモリに一つの値を格納するためにこれらの値を使用します。この転送量は共有メモリバスにとって多すぎるため処理できません:プログラムのボトルネックになります。初期化ステップに戻って、ローカルスタックの転送量を減らすことが可能かどうか検討しました。生成関数の最も内側のループでいくつかの分岐を入れることによってのみ、その回答はイエスになりました。X_l と X_r がブラウン運動の点 X_{θ_i} の左隣と右隣である場合の(5)の状況を検討し

ます。
実行戦略を慎重に調整することによって、$\{X_l, X_r, X_{\theta_i}\}$ がブラウン運動の点 $X_{\theta_{i+1}}$ の左隣と右隣両方を含むようにすることが可能です。これは全ての点 $X_{\theta_{i+1}}$ に対して行うことはできませんが、多くの点に対して行うことができます。それがいつ保持するかを特定するため何らかのフラグを取り入れることが必要です。それに対応して生成ステップのメインの計算ループがいくつかの分岐を含む必要があります。点 $X_{\theta_{i+1}}$ を計算する際の分岐はおおよそ以下のようになります：
$X_{\theta_{i+1}}$の左隣と右隣は$\{X_l, X_r, X_{\theta_i}\}$ の中にありますか？
✓ Yes.

- 点 X_l, X_r, X_{θ_i} のどれが隣かを特定し、それらを使用して $X_{\theta_{i+1}}$ を計算します。

X No.

- ローカルスタックから左隣と右隣を読みそれらを使用して $X_{\theta_{i+1}}$ を計算します。

通常プログラマはこのような分岐を特に内部ループでは避けますが、この場合は有効です。

7.1. パフォーマンス。プリフェッチを追加しない場合、上記で説明されたカーネルは24個のレジスタを使用し、占有率は 43% で、79GB/s で動きます。セクション6のプリフェッチが追加されると、カーネルは 33個のレジスタを使用し、占有率は58%で、102GB/s で動きます。倍精度の同一コードでは38個のレジスタを使用し、占有率は 33% で、115.66GB/s で動きます。

8. 結果の概要と結論

セクション2.4で説明されているシステムで計測したとおりにブラウン橋サンプルパス生成器とスケーリングされたブラウン橋増分生成器両者のパフォーマンスの結果を要約して説明します:

- ブラウン橋サンプルパス生成器
  - 単精度： ランタイムはグローバルメモリスループットを達成する速度102GB/s で10.01ms。これは正規乱数を生成するのにかかる時間よりも1.9倍速い 。[1]
  - 倍精度：ランタイムはグローバルメモリスループットを達成する速度115.6GB/s で17.12ms。これは正規乱数を生成するのにかかる時間よりも2.2倍速い。[2]
- スケーリングされたブラウン橋増分生成器
  - 単精度:ランタイムはグローバルメモリスループットを達成する速度109GB/sで9.09ms 。これは正規乱数を生成するのにかかる時間よりも2.09倍速い。[1]
  - 倍精度：ランタイムはグローバルメモリスループットを達成する速度116.3GB/s で17.01ms 。これは正規乱数を生成するのにかかる時間よりも2.2倍速い。[2]

セクション7 からのアルゴリズム（プリフェッチなし）はCでコーディングされ、OpenMPと並列

化され、多くの異なる x86-64システム上で稼働しています。結果は表２に記載されています。３つのCPUシステムはIntel Core i7 デスクトッププロセッサからサーバ市場向けのトップエンドのXeon X5680までパフォーマンスの範囲を表します。AMD マシンは２つのMagny Cours チップを備えているデュアルソケットシステムで、英国のHECToR スーパーコンピュータのフェーズ２のノードに似ています。Xeon マシンもまた２つのWestmere を備えたデュアルソケットシステムです。CPU コードのスケーリングを観察して下さい。Core i7 では２スレッド以降は限られたスケーリングを示しています。さらに、３つ以上のスレッドではパフォーマンスは 極めて変化しやすくなります（ここで示される値は平均です）。Xeon　では実質的に８スレッド以降はスケーリングがないことを示しています。AMD はスケーリングがハードウェアの能力の最大まで現れる唯一のシステムです。とは言ってもAMDシステムのパフォーマンスが特別にすばらしいというわけではなく Xeon より70% ほど遅いです。

[1] NAG GPU MRG32k3a　生成器使用、単精度

[2] NAG GPU MRG32k3a　生成器使用、倍精度

生成器		Intel Core i7 860, 4 cores @ 2.8GHz (with hyperthreading)					GPU 増速
		1 Thread	2 Threads	3 Threads	4 Threads	8 Threads	
ブラウン橋	float	1212.5ms	1142.6ms	1189.3ms	787.2ms	720.3ms	72.0x
	double	1771.1ms	930.5ms	704.4ms	803.6ms	593.4ms	34.7x
ブラウン橋 増分	float	1170.5ms	613.2ms	857.1ms	639.1ms	605.0ms	67.1x
	double	1452.8ms	782.6ms	742.1ms	653.5ms	549.7ms	32.3x

生成器		AMD Opteron 6174, 24 cores @ 2.2GHz (dual socket Magny Cours, 2 x 12 cores)					GPU 増速
		1 Thread	8 Threads	12 Threads	16 Threads	24 Threads	
ブラウン橋	float	2402.6ms	1019.6ms	676.1ms	452.4ms	355.4ms	35.5x
	double	2948.8ms	900.3ms	576.4ms	454.2ms	328.4ms	19.2x
ブラウン橋 増分	float	2173.6ms	637.2ms	434.4ms	321.1ms	257.0ms	28.6x
	double	2694.1ms	804.9ms	555.3ms	418.4ms	298.8ms	17.6x

生成器		Intel Xeon X5680, 12 cores @ 3.33GHz (dual socket Westmere, 2 x 6 cores with heyperthreading)					GPU 増速
		1 Thread	8 Threads	12 Threads	16 Threads	24 Threads	
ブラウン橋	float	1392.1ms	192.4ms	195.6ms	171.2ms	171.9ms	17.2x
	double	1463.8ms	205.7ms	222.7ms	193.6ms	227.6ms	11.4x
ブラウン橋 増分	float	1207.4ms	168.8ms	168.8ms	146.0ms	207.6ms	16.2x
	double	1308.1ms	183.2ms	184.9ms	166.2ms	211.8ms	9.8x

表 2. Tesla C2050 対　高パフォーマンス CPU についてのベンチマーク図

表２の動きはメモリ帯域幅の制約を受けるアプリケーションの典型的な動きです。問題に対し新たな計算コアを投入することはパフォーマンスの改善にはなりません:実際のメモリハードウェアのスピードを増す必要があります。ここで GPU は速いGDDR5 グラフィックメモリによって明らかな強みがあります。

8.1. 結論。 ブラウン橋生成器を作りだす過程で、私たちはGPUプログラミングの定説に関して多くのことを学びました：

- 高い占有率はメモリ帯域幅の制約を受けるアプリケーションに対しては必ずしも高いパフォーマンスを意味しません。
- データのレジスタへの明示的なプリフェッチは著しくパフォーマンスを高めることができます。
- 共有メモリは人が思うほど速くはありません。特に、インデックスが共有メモリから別の値をフェッチするのに使用される場合、共有メモリからインデックスがフェッチされる間接参照の挿入は、処理をかなり減速させます。
- レジスタはGPU上の最も速いメモリであるため、帯域幅バインドアプリケーションに対しても、レジスタを積極的に使用することはパフォーマンスを高める良い方法です。

- 分岐が最も内側の計算ループにある場合でも、慎重に分岐を使用することによりパフォーマンスを増加することができます。
- 従来のマルチコアプラットホーム上でスケーリングをうまく行わないことの多いメモリ帯域幅バインドアプリケーションを加速させるのにGPU は非常に効果的です。

8.2. 謝辞。ニューメリカルアルゴリズムズグループは実装に有益なGPUブラウン橋ルーチンに尽力いただいた Mike Giles 教授に感謝申し上げます。 またNAG はブラウン橋アルゴリズムがどのように実際に使用されているかについて洞察を示していただいた二人のシニア計量アナリストのご意見やフィードバックに感謝します。

8.3. ソフトウェアへのアクセス。ユーザの皆様がもしブラウン橋ルーチンを入手されたい場合はWebサイト (http://www.nag-j.co.jp) もしくはメール (sales@nag-j.co.jp) を通じて NAG へお問い合わせ下さい。
GPU と CPU （シングルスレッド）の両方の実装は GPU [3] 向けの NAG 数値計算ルーチンで利用いただけます。ルーチンのドキュメントは[3]で参照いただけます。GPU モンテカルロプライシングアプリケーションを作成するための GPU Sobol 生成器の使用法を説明しているサンプルプログラム（ドキュメントを含む）は[4]で参照いただけます。
非常に順応性のあるブラウン橋アルゴリズムのシリアル及びマルチスレッドの実装はGPU 向けのNAG 数値計算ルーチンで大きな特徴となっていますが、さらに NAG CPU ライブラリとNAG Toolbox for MATLABの今後のリリースにおいても大きな特徴となるでしょう。早期リリースはリクエストに応じて入手いただくことができます。

参考文献

[1] CUDA Optimization : Memory Bandwidth Limited Kernels + Live Q&A by Tim Schroeder. NVIDIA GPU Computing Webinar Series.
http://developer.download.nvidia.com/CUDA/training/Optimizing_Mem_limited_kernels.mp4

[2] Glasserman, P. (2004). Monte Carlo Methods in Financial Engineering. Springer.

[3] GPUルーチンドキュメント用 NAG数値計算ルーチン
http://www.nag.co.uk/numeric/GPUs/doc.asp

[4] GPU向け NAG 数値計算ルーチンを用いたデモ
http://www.nag.co.uk/numeric/GPUs/gpu_demo_applications

Numerical Algorithms Group Ltd
E-mail address: jacques@nag.co.uk

[3] http://www.nag.co.uk/numeric/gpus/index.asp