GPGPUを用いた不確実時系列データ類似検索の高速化

黄　　峻† 小澤　佑介† 天笠　俊之†† 北川　博之††

† 筑波大学大学院システム情報工学研究科　〒 305–8573 茨城県つくば市天王台 1–1–1
†† 筑波大学システム情報系　〒 305–8573 茨城県つくば市天王台 1–1–1
E-mail: †{fand,kyusuke}@kde.cs.tsukuba.ac.jp, ††{amagasa,kitagawa}@cs.tsukuba.ac.jp

あらまし　計算機によって処理されるデータ量は日々急速に増大しており，計算手法の高速化及び効率化が求められている．これは科学研究の分野においても同様である．誤差を含む時系列データ間の類似度を評価する手法はいくつか存在するが，いずれも確率計算を行うためコストが大きく，大量のデータに対して類似度計算を行うことが難しいという問題がある．本研究では DUST [1] と呼ばれる類似検索手法を対象に GPGPU を用いた高速化手法を提案する．具体的には，DUST 内の距離計算において必要となる積分計算や確率計算を GPU 上で並列に行い，高速な処理を実現する．

キーワード　不確実時系列データ，時系列データマイニング，GPGPU

1. はじめに

近年，計算機性能の向上に伴い，日常生活から科学まで様々な分野に渡って，日々蓄積され処理されるデータは非常に膨大なものとなっている．その中から所望する情報を検索し，新しい知識を得ることは極めて重要であり，そのための技術が求められている．

時系列データとは，時間によって変化するデータを連続的あるいは周期的に記録して得られたデータの系列をいう．代表的な例では音声や映像，または気温や湿度など自然現象の観測データがある．複数の時系列データを分析するにあたって，時系列データ間の類似度を求める処理がよく用いられる．これを時系列データの類似検索と呼ぶ．時系列データの類似検索は，特に科学研究においては，観測データを比較することで特定の現象の分析や新たな事実の発見するために用いられている．

計算機で扱われるデータは正確なものだけでなく，誤差を含む不確実なデータも数多く存在する．時系列データの類似検索において，従来の手法ではデータの不確実性を考慮しておらず，類似検索精度の向上の妨げとなっていた．このため，昨今では不確実性を考慮した時系列データの類似検索手法が提案されている．その一つに DUST がある． これは確率計算によって不確実なデータ間の距離指標を与えるものであり，データ間の距離計算に用いることで従来の時系列データの類似検索手法に適用出来る．しかし，DUST は計算負荷が大きく，多くのデータを処理するには時間がかかるという問題がある．

この問題に対して，本研究では GPU を用いることによって大規模データに対する処理の実現を図る．GPU は元来グラフィック処理向けに開発されたデバイスであるが，数百個の演算ユニットを搭載し，並列計算において極めて優れた性能を示す．GPU は比較的安価に高性能な計算を行うことが出来，また単純な構造であることから，近年様々な分野で応用研究がなされている．

本研究では，GPU を用いることで DUST に基づく不確実時系列データに対する類似検索処理の高速化を目指す．具体的には，不確実データの比較での積分計算，及び時系列データの時刻ごとの処理を並列化することで高速化するまた，合成データを用いた実験により実際に類似検索を高速化できること，及びその効率を示す．

本稿の構成は以下の通りである．まず，第 2 節で研究対象となる不確実時系列データについて述べ，第 3 節でその類似検索手法の一つである DUST を説明する．第 4 節では本研究で用いる GPGPU と呼ばれる技術について説明し，続いて第 5 節では具体的な高速化手法を説明する．第 6 章で今回行った実験の評価及び考察を行い，最後に第 7 章をまとめと今後の課題について述べる．

2. 不確実時系列データ

2.1 時系列データ

時系列データとは，時間によって変化するデータを連続的あるいは周期的に記録して得られたデータ点の系列をいう．代表的な例では音声データや映像データ，気温や湿度など自然現 象の観測データが挙げられる．一般に，時系列データの処理には，その長さに応じた高い計算量が必要となる．

複数の時系列データを分析するにあたって，時系列データ間の類似度を求める処理がよく求められる．これを時系列データの類似検索と呼ぶ．時系列データの類似検索は，特に科学研究においては，観測データを比較することで特定の現象の分析や新たな事実の発見するために用いられている．例えば，我々の研究グループにおいて X 線観測データの類似検索に関する研究が行われた [2] ．図 1 は，この研究において，天体の輝度を継続的に観測することで得られた時系列データ (ライトカーブ) の例である．

2.2 不確実時系列データ

計算機で扱われるデータは正確なものだけでなく，誤差を含む不確実なデータも数多く存在する．不確実なデータには主に科学研究における観測データや，実在の人物に対する調査記録

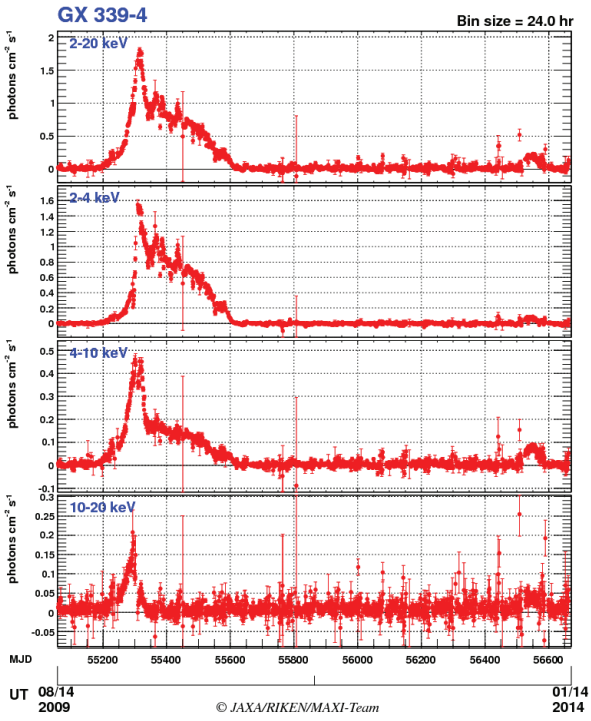


図 1　時系列データの例

や統計等のデータがある [3]．前者の場合，データの不確実性は観測手法や観測装置に由来する．後者の場合，標本調査における標本誤差などの統計的な誤差や，関連する人物のプライバシーを保護する目的で人為的にデータを撹乱している等の理由が考えられる．図 1 に示したライトカーブでは，各時刻のデータがいくつかの観測値の平均と分散によって示されている．このように，多くの観測データでは観測値に誤差が存在し，誤差を考慮した処理が必要となる．

本研究では，このように誤差情報が付加されている不確実データを扱う．また，各データ点が不確実データから構成される時系列データを不確実時系列データと呼ぶ．誤差情報の表現形式には，真の値の候補となる複数回の観測値で表すものや，誤差の確率密度関数で与えられるものがある．一般にデータの誤差は正規分布に従うことが多く，この時，誤差情報として標準偏差の大きさを与えた不確実データが広く利用されている．これらの方式で不確実時系列データを表現した例を図 2 に示す．

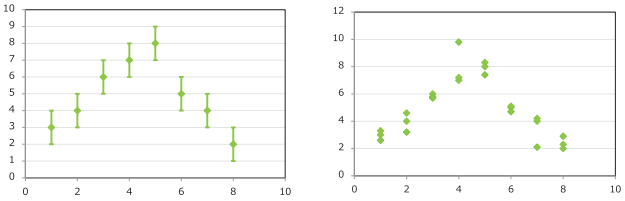
従来の時系列データを対象とした類似検索手法ではデータの不確実性を考慮していなかったため，我々の直感に反する結果を産み，類似検索精度の向上を妨げる要因となっている．不確実時系列データの類似検索ではこの問題を解決する手法を用いる必要がある．このため，昨今では不確実性を考慮した時系列データの類似検索手法がいくつか提案されている．

2.3　関 連 研 究

不確実時系列データの類似検索に関する関連研究には以下のものがある．

2.3.1　MUNICH

MUNICH [4] は A ßfalg らが 2009 年に提案した手法である．MUNICH では各時刻の不確実データが複数の観測値から構成



(a) 誤差の情報で表現　　(b) 複数の観測値で表現

図 2　不確実時系列データの表現形式

される不確実時系列データを対象とする．全ての観測値の組合せに対し，二つの不確実時系列データ間の距離を計算し，予め定められた閾値よりも距離が小さくなる確率を求め類似度を計算する．MUNICH は不確実データが少なくとも一つの観測値を持っていれば計算可能であるが，精度の向上には各時刻の不確実データが十分に多い観測値を持っている必要があり，計算コストが非常に大きくなる．

2.3.2　PROUD

PROUD [5] は Yeh らが 2009 年に提案した手法である．PROUD は中心極限定理に基づき，対象の時系列が正規分布に従う誤差を持つ不確実時系列データであった場合のユークリッド距離を推定する．各時刻の不確実データについて，誤差情報から値の差を仮定しユークリッド距離を計算すれば，中心極限定理により巨視的には各不確実データの誤差が正規分布であった場合のユークリッド距離に近づいていく．PROUD では元の誤差の分布にかかわらず，正規分布であった場合の類似度を計算するものであるため，誤差の分布が正規分布と大きく異なる場合に適切な結果が得られない問題がある．

3.　DUST

DUST は Sarangi らによって提案された不確実時系列データを比較する手法である [1]．この手法では，不確実時系列データ全体での距離指標 DUST (以下「DUST 距離」と呼ぶ) だけではなく，二つの不確実データ点間の類似度指標 dust (以下「dust 関数」と呼ぶ) を定義している．このため，DUST は不確実時系列データの各時刻で誤差分布が異なる場合に対応しており，他の類似手法に比べて多くの不確実時系列データに適用可能である．また，DUST 自体は二つの不確実時系列データの長さが異なる場合に対応していないが，個別の不確実データ間の距離指標である dust 関数を DTW 法などに適用することで，長さの異なる不確実時系列データの類似度計算に利用できる．

なお，本稿では dust 関数を距離関数ではなく距離指標と呼ぶ．これは，dust 関数がまれに距離の公理を満たさないことがあるためである．また，同様に DUST 距離も厳密には距離と呼べるものではないが，本稿では簡単のため距離という呼称を使用する．

3.1　DUST の計算方法

DUST の計算手順はユークリッド距離の計算手順に類似している．まず各時刻の不確実データ間の dust 関数の値を計算し，その後全体の値を総合する事で，不確実時系列データ全体の DUST 距離を計算する．

二つの不確実時系列データ $x = \langle x_0, x_1, \ldots, x_n \rangle, y = \langle y_0, y_1, \ldots, y_n \rangle$ があるとする．この時，時刻 i での各データ間の dust 関数 $dust(x_i, y_i)$ は次のように定義される．

$$dust(x_i, y_i) = \sqrt{-\log(\phi(|x_i - y_i|)) - k}$$
$$k = -\log(\phi(0))$$

ただし，$\phi(|x_i - y_i|) = Pr(dist(0, |x - y|) \approx 0)$ である．また，定数 k はいかなる値 t に対しても $dust(t, t) \approx 0$ となることを保証する為の調整項である．ここで，$\phi(|x_i - y_i|)$ を求める事は x_i, y_i の真の値 $r(x_i), r(y_i)$ がほぼ等しい確率を計算する事と同義である．

次に，これらを総合する事で時系列データ全体の DUST 距離 $DUST(x, y)$ を求める．$DUST(x, y)$ は次のように定義される．

$$DUST(x, y) = \sqrt{\sum_0^n dust(x_i, y_i)}$$

実際の計算では，$\phi(|x_i - y_i|)$ の計算は $Pr(r(x_i) \approx r(y_i)|x_i, y_i)$ の計算と等しいため，以下の計算が行われる．

$$Pr(r(x_i) \approx r(y_i)|x_i, y_i) = \int_z Pr(r(x_i) \approx z|x_i) Pr(r(y_i) \approx z|y_i) dz$$

ここで，式の右辺はベイズの定理から以下のように展開される．

$$Pr(r(x_i) \approx z|x_i) = \frac{Pr(x_i|r(x_i) \approx z) Pr(r(x_i) \approx z)}{Pr(x_i)} = \frac{Pr(x_i|r(x_i) \approx z) Pr(r(x_i) \approx z)}{\int_v Pr(x_i|r(x_i) \approx v) Pr(r(x_i) \approx v) dv}$$

積分計算にモンテカルロ法を用いる場合の手順は以下の通りである．まず，積分範囲内で乱数 z を生成する．次に $Pr(r(x_i) \approx z|x_i), Pr(r(y_i) \approx z|y_i)$ をそれぞれ与えられた誤差分布により計算し，掛け合わせる．これらの手順を繰り返し，得られた値を繰り返し回数で割ることで，近似的な積分値を得る．

DUST 距離を求めるには，比較する時系列データの長さが等しい必要がある．時系列データの長さが異なる場合，事前に長さを揃えておく必要がある．以下では時系列データの長さは等しいものとする．

4. GPGPU

近年，GPU を汎用的に用いることで高速な並列処理を行う GPGPU (General-purpose computing on graphic processing units) と呼ばれる技術が開発され，様々な分野で研究が行われている．GPU は本来グラフィック関係の処理のために設計されたプロセッサであり，単純な処理を行う演算ユニットを多数搭載している．このため単純な並列処理で高い性能を発揮し，グラフィック用途に限らず，並列処理を行う場合には通常の CPU を上回る処理性能を持つことが知られている．GPGPU は科学研究において分子生物学や暗号理論など幅広い分野で利用されている [6], [7].

GPGPU は適切に設計されたプログラムにおいては圧倒的な性能を発揮するが，使用方法によっては性能の向上が見られない場合がある．これは GPGPU による速度向上が主に計算処理の並列化によってもたらされるものであり，GPU に搭載される演算ユニット個々は決して高性能ではない事が原因である．このため，GPGPU を利用し高速化を目指すにあたっては，プログラムの設計が大きな問題となる．

本研究では NVIDIA 社の GPU 及び同社の GPU 開発環境である CUDA [9] を用い GPU プログラミングを行った．次節以降では，GPU とは同社の GPU を指すものとする．以下では GPU の構造と特徴，及び CUDA を利用したプログラムの処理について説明する．

4.1 GPU の構造

GPU のチップは多層的な構成となっている．以下に，NVIDIA Tesla アーキテクチャに於ける GPU のチップ構成について説明する．演算装置の最小単位は Scalar Processor (以下「SP」と呼ぶ) と呼ばれるものであり，一般的な GPU では数百個搭載されている．また，頻繁に用いられる三角関数などの特殊な関数は Special Function Unit (以下「SFU」と呼ぶ) で行われる．SP での処理と SFU での処理は非同期に行われる．実際の計算処理は Streaming Multiprocessor (以下「SM」と呼ぶ) 単位に分割され，SM 内で一斉に実行される．SM は 8 個の SP，いくつかの SFU, 及び後述する Shared Memory などから構成される．

4.2 GPU のメモリ階層

GPU 上で実行されるプログラムは，基本的には予め GPU のメモリに転送されたデータのみを扱うことが出来る．このため，GPU を利用したプログラミングでは，計算に使用するデータを手動で管理し，CPU と GPU との間でのデータ転送を明示的に行う必要がある．データ転送作業は GPU において性能上のボトルネックとなることがあり，この問題を解決するために GPU にはアクセス速度の異なる複数のメモリ階層が存在する．

最もよく用いられるのは Global Memory と呼ばれるメモリである．一般的な GPU プログラミングでは，使用するデータはまずこの領域に転送される．Global Memory はどの SP からもアクセス可能であるが，次に述べる Shared Memory に比べるとアクセス速度は遅くなる．

Global Memory に次いで用いられるのが Shared Memory である．これは SM 上に存在し，同じ SM 上の SP からのみアクセスできる．この様に Global Memory に比べ利用範囲は限られるが，より高速なアクセスが可能である．

SM 上には Shared Memory の他に Register と呼ばれるメモリ領域が存在する．これは最も高速なアクセスが可能であるが，それぞれの SP からのみアクセス可能となっている．

GPU 上にはその他，定数が格納される Constant Memory, Texture Memory が存在する．これらの領域は CPU からのみ変更でき，GPU 上のプログラムからは読み込み専用となる．

4.3 CUDA

本研究では NVIDIA 社の GPU 開発環境である CUDA [9] を利用した．これは C 言語で GPGPU プログラムを開発するためのフレームワークであり，GPU プログラミングにおいて事実上の標準となっている．CUDA では，GPU 上で実行する処理を Kernel という関数として定義し，GPU での処理単位を Grid，Block，Thread という単位で分割して考える．

処理の最小単位は Thread と呼ばれる．これはチップでの演算装置の最小単位である SP に対応している．Kernel は割り当てられた SP 上で Thread として実行される．

Block は複数の Thread をまとめたものであり，SM に対応する．Shared Memory や SFU は SM 上に存在するため，プログラムにおいては Block 毎に管理される．

Grid はさらに複数の Block から成り立っており，CUDA における処理の最大単位である．Grid に対する Block 数，Block に対する Thread 数は利用者が設定することができ，3 次元のベクトルとして管理される．並列実行されるプログラムは，与えられた設定値，及び実際に使用するデバイスの SM, SP 数に応じ，実行時に自動的に割り当てられる．

また，処理を並列化する数に制約がない場合，SM 及び SP 数の倍数とする事が望ましい．これは Thread, Block の割り当てが最適に行われる為である．

5. 提 案 手 法

DUST は計算コストが大きく，多量のデータの解析には膨大な時間が掛かる．主に dust 関数内の確率密度関数及びその内部の積分計算が実行時間の大部分を占めており，これらの計算効率を改善することが性能向上において重要であると考えられる．また，時系列データ内の各時刻での計算を並列処理することで更なる高速化が期待できる．

比較のため，Dallachiesa ら [8] による CPU での実装 [16] を参考にし，複数の手法を組み合わせることで段階的に高速化を行う．以下では具体的な高速化手法について述べる．

5.1 モンテカルロ積分の並列化

Dallachiesa らの実装 [16] においては，積分計算に GNU Scientific Library [13], 確率密度関数の計算に Boost C++ Libraries [12] を用いている．積分計算はモンテカルロ法による積分を行なっている．モンテカルロ法とは，積分範囲内で擬似乱数を生成し，これを入力として被積分関数の値を求める試行計算を繰り返し行うことで積分計算の近似解を求める手法である．CPU での実装では各積分計算に対し 50,000 回の試行を行っている．

本手法では，これらの計算を GPU 上で行う事で dust 関数計算の高速化を実現する．具体的には，Boost C++ Libraries のソースコードを参考に GPU 上で確率密度関数を計算するプログラムを作成し，続いて GNU Scientific Library のソースコードを参考にモンテカルロ積分を行うプログラムを作成した．

モンテカルロ積分における試行回数は 49,152 回とした，これは [8] のプログラムでの 50,000 回の試行と近い精度が得られ，かつ処理の割り当てが効率的に行われるよう，GPU の SM 数を考慮した結果である．

また，第 3.1 節で示したとおり，dust 関数内には 3 箇所の積分計算が存在する．各積分計算の間に依存関係が無いことから，これらの積分計算を並列に処理する．

具体的な処理の流れを以下に示す．

まず，3 箇所の積分計算に対し各 49,152 個の乱数を生成する．乱数の生成には cuRand ライブラリ [10] を使用し，乱数の生成も GPU 上で並列に行う．生成した乱数は GPU の Global Memory に格納する．

次に，生成した乱数に対して被積分関数の値を計算し，Thread 毎に合計する．全ての乱数に対し計算が終了したら，各 Thread の合計値を Shared Memory に移動する．最後に，Shared Memory 内の値を合計し，試行回数で割ることで，積分値の近似値を得る．

5.2 Parallel Reduction

前節で説明した手法では Shared Memory に格納した値を合計する必要がある．これを GPU 上で並列処理することで高速化が可能となる．

ある配列の全ての要素に対し総和などを求める処理を Reduction と呼ぶ． Reduction は様々な場面で頻出する処理であり，GPGPU 上で効率よく並列処理するための研究がなされている．本研究では NVIDIA によるサンプル実装 [11] を参考にし，各積分計算の試行結果を GPU 上で合計する処理を実装した．具体的には，生成した乱数全てに対して試行を行い，値を Shared Memory に移動する．その後，GPU 上の全スレッド，あるいは次節で説明する時間方向の並列化を行った場合はブロック内の全スレッドを利用して Reduction を行う．

Reduction の最適化の流れは以下の通りである．

まず，バンクコンフリクトの発生を防ぐためメモリアクセスを調整する． Shared Memory は 16 個あるいは 32 個のバンクから構成されている．複数の Thread が同時に 1 つのバンクにアクセスすることをバンクコンフリクトと呼ぶ．バンクコンフリクトが発生した場合，各 Thread は他の Thread のアクセスが終了するまで待機する事になり，遅延が発生する．その為，メモリアクセスの順番を調整しバンクコンフリクトを防ぐ必要がある．

次に divergent branch を回避する．CUDA プログラムの実行の最小単位は Thread であるが，実際には 32 個の Thread 単位で SIMD で処理を行う．これを Warp と呼ぶ．CUDA では分岐処理において Warp 単位で分岐条件の判定を行う．その為，Warp 内の全ての Thread で条件の値が同じでない時，その Warp は真偽両方の場合の結果を計算する必要があり，高速化においてボトルネックとなりうる．このような分岐を divergent branch と呼ぶ．その為，CUDA では分岐が単純になるようプログラムを設計する必要がある．

最後にループの展開を行う．プログラムが並列実行される SM, SP 数が決定するのは実行時だが，割り当てる Thread, Block 数はプログラムのコンパイル時に決定される．また，割り当てが可能な Thread, Block 数には，使用するデバイスによって上限が存在する．よって，Block 内で全 Thread に対し

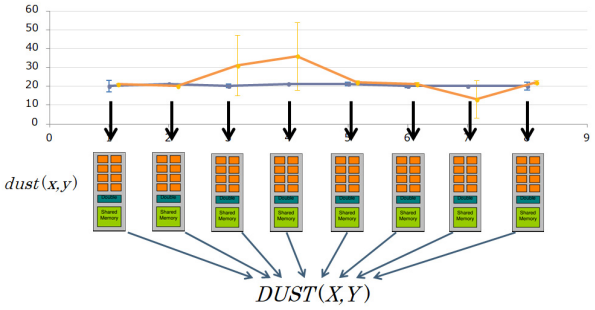


図 3　時刻毎の処理の並列化

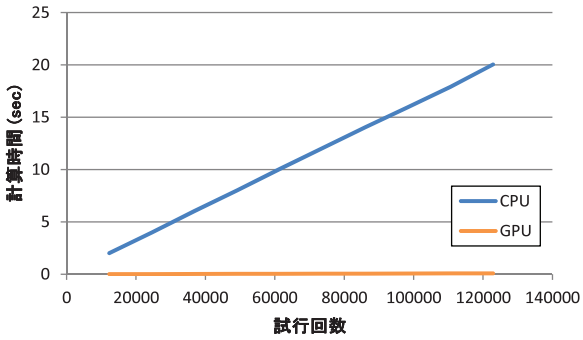


図 4　実験 1:各実装の計算時間

ループを行うようなプログラムでは，Block 毎の Thread 数をコンパイル時定数として与えることでループを展開することが出来る．

5.3　時刻毎の計算の並列化

これまで述べてきた手法は，ある時刻の dust 関数の計算を並列化するものであり，時系列データ内の各時刻に対する dust 関数の計算は逐次的に行っていた．また，時系列データの次元数だけ GPU での処理を呼び出すため，呼び出し及びデータ移動によるオーバーヘッドが発生していた．ここでは全時刻に対する dust 関数の計算を並列化し，一度のカーネル呼び出しで全ての処理を行うことで全体の処理を高速化する．

具体的な処理の流れを以下に示す．まず，dust 関数の計算に必要な 49152 ×(対象となる時系列データの次元数) 個の乱数を GPU 上で生成する．次に，1 時刻を 1Block に割り当て，時刻のデータ及び乱数を Block に読み込む．続いて，1 時刻の dust 関数の計算を全て 1Block 内で行う．全ての時刻に対してこの処理を並列に行い，求めた値を用いて CPU 上で DUST 距離を計算する．

ここでの処理のイメージを図 3 に示す．

6.　評 価 実 験

提案手法と既存手法の比較のため，3 種類の評価実験を行った．実験に用いた計算機環境は以下のとおりである．

- CPU 名：Intel(R) Xeon(R) CPU E5630
- CPU クロック周波数：2.53GHz
- コア数：4
- メモリ容量：4GB
- GPU 名：NVIDIA(R) Quadro(R) FX 4800
- GPU コアクロック周波数：600MHz
- GPU コア数：192
- GPU メモリ容量：1.5GB

プログラムの実装は C/C++ で行い，コンパイラには gcc 4.6.3 及び nvcc 4.2 を利用した．また，GPU を用いない積分計算及び確率計算において Boost C++ Library [12] ，及び GNU Scientific Library [13] を利用した．実験に用いるデータには，UCR データセット [14] の Gun Point データセットを撹乱し，誤差情報を付与したものを使用した．

以下，DUST 距離を計算するプログラムの実装について，Dallachiesa らによる従来の実装 [16] を「CPU 実装」，提案手法を用いた実装を「GPU 実装」と呼ぶ．

6.1　実験 1:モンテカルロ積分の試行回数の処理時間への影響

モンテカルロ積分における試行回数を変化させた時，CPU 実装と GPU 実装において処理時間がどのように変化するかを観察した．データセット内の 150 次元の時系列データを対象に，全ての組み合わせで DUST 距離を計算し，平均処理時間を計測した．両実装の処理時間を図 4 ，処理時間の比を図 5 に示す．図 4 より，試行回数の増加に対し，各処理時間はほぼ比例して増加することがわかる．また，図 5 より，試行回数にかかわらず，CPU 実装と GPU 実装の処理時間の比は 210〜230 倍でほぼ一定であることがわかる．

6.2　実験 2:不確実時系列データの次元数の処理時間への影響

不確実時系列データの次元数を変化させた時，CPU 実装と GPU 実装において処理時間がどのように変化するかを観察した．まず，データセット内の時系列データを 50 次元に揃え，線形補完によって $50n\,(n = 1, 2, ..., 10)$ 次元の時系列データをそれぞれ 10 個ずつ作成した，次に，次元の同じ時系列データに対し全ての組み合わせで DUST 距離を計算し，平均処理時間を計測した．両実装の処理時間を図 6 ，処理時間の比を図 7 に示す．図 6 より，時系列の次元の増加に対し，各処理時間はほぼ比例して増加することがわかる．また，図 7 より，時系列データの長さにかかわらず，CPU 実装と GPU 実装の処理時間の比は 230〜250 倍でほぼ一定であることがわかる．

6.3　実験 3:各高速化手法の比較

5 節で説明した各高速化手法が計算時間に与える影響を比較する．モンテカルロ積分の並列化，Parallel Reduction，時刻毎の並列化を段階に実装したプログラムを使用し，それぞれの処理時間を比較した．それぞれの処理時間，及び CPU 実装と比べ何倍速いかを 表 8 に示す．積分計算を並列化しただけで CPU 実装と比べ約 30 倍の高速化が可能であるが，Parallel Reduction により更に約 1.7 倍の高速化，時間毎の並列化を行うことで更に約 4 倍の高速化を実現し，最終的には約 230 倍の処理効率を実現した．

7.　結　　論

本稿では，不確実時系列データの類似検索手法の一つである

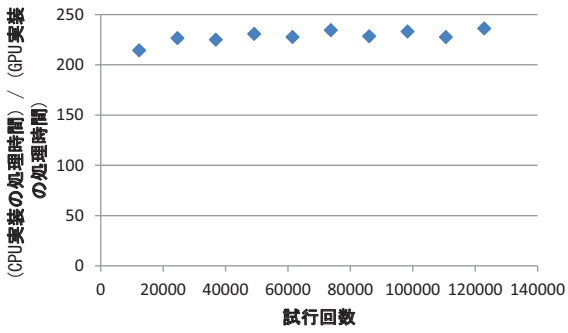


図 5　実験 1:CPU 実装と GPU 実装での計算時間の比

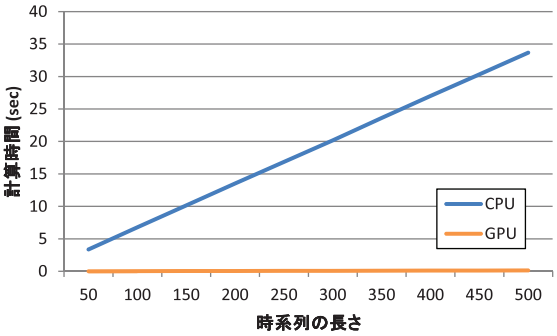


図 6　実験 2:各実装の計算時間

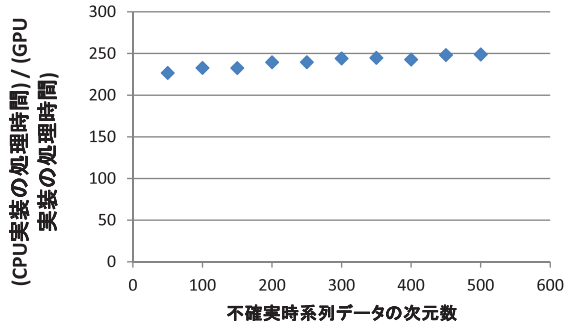


図 7　実験 2:CPU 実装と GPU 実装での計算時間の比

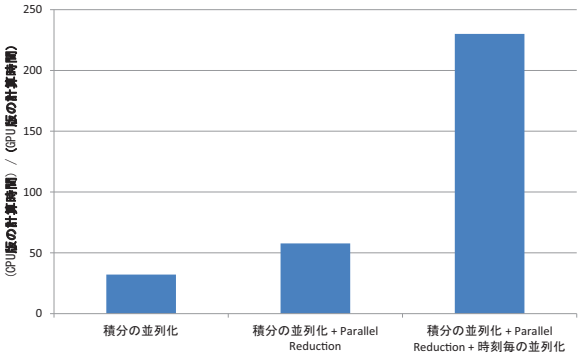


図 8　各高速化手法の速度比較

DUST を GPGPU を用いて高速化する手法を紹介し，またその効果を実験によって示した．並列計算を得意とする GPU の特性を考慮し，DUST 内部の計算をモンテカルロ積分の並列化，Parallel Reduction，時刻毎の計算の並列化によって処理時間の高速化を行い，従来の実装に対し約 230〜250 倍の高速化を実現した．今後は，より詳細な実験を行い，また今回紹介した以外の関連手法を調査する予定である．GPU を用いない並列処理技術に対し今回の提案手法が適用できるか検証し，適用可能ならば性能比較を行う．また，不確実時系列データの類似検索に用いられる他の手法との精度及び速度の比較を行い，より優れた類似検索手法の開発を目指す．

謝辞　本研究の一部は，平成 25 年度共同研究（富士通研究所 CPE25149）によるものである．

文　献

[1] Smruti R. Sarangi and Karin Murthy, “DUST: A Generalized Notion of Similarity Between Uncertain Time Series,” Proceedings of the 16th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, pp. 383-392, 2010.

[2] 林史尊, 天笠俊之, 北川博之, 海老沢研, 中平聡志, “動的タイムワーピング距離を用いた X 線天文データの類似検索”, 宇宙航空研究開発機構研究開発報告, vol. 12, pp. 19-27, 2013.

[3] Charu C. Aggarwal and Philip S. Yu, “A Survey of Uncertain Data Algorithms and Applications” IEEE Trans. on Knowl. and Data Eng. 21, 5, pp. 609-623, 2009.

[4] Johannes Aßfalg, Hans-Peter Kriegel, Peer Kröger, Matthias Renz, “Probabilistic Similarity Search for Uncertain Time Series”, Proceedings of the 21st International Conference on Scientific and Statistical Database Management, pp. 435-443, 2009.

[5] Mi-Yen Yeh, Kun-Lung Wu, Philip S. Yu, Ming-Syan Chen, “PROUD: A Probabilistic Approach to Processing Similarity Queries over Uncertain Data Streams”, Proceedings of the 12th International Conference on Extending Database Technology: Advances in Database Technology, pp. 684-695, 2009.

[6] Weiguo Liu, Bertil Schmidt, Gerrit Voss, Andre Schroder, Wolfgang Muller-Wittig, “Bio-sequence Database Scanning on a GPU”, Proceedings of the 20th International Conference on Parallel and Distributed Processing, p. 251, 2006.

[7] 岩井啓輔, 黒川恭一, 西川尚紀, “GPU の汎用計算環境 CUDA による暗号アルゴリズムに対するキークラックの高速化 (アクセラレーションと回路設計,2009 年並列/分散/協調処理に関する『仙台』サマー・ワークショップ (SWoPP 仙台 2009))”, 電子情報通信学会技術研究報告. CPSY, コンピュータシステム, vol. 109, no. 168, pp. 49-54, 2009.

[8] Michele Dallachiesa, Besmira Nushi, Katsiaryna Mirylenka, Themis Palpanas, “Uncertain Time-series Similarity: Return to the Basics”, Proc. VLDB Endow., vol. 5, no. 11, pp. 1662–1673, 2012.

[9] Parallel Programming and Computing Platform — CUDA — NVIDIA. [Online]. http://www.nvidia.com/cuda

[10] cuRAND — NVIDIA Developer Zone [Online]. https://developer.nvidia.com/cuRAND

[11] Mark Harris, “Optimizing Parallel Reduction in CUDA” [Online]. http://developer.download.nvidia.com/compute/cuda/1.1-Beta/x86_website/projects/reduction/doc/reduction.pdf

[12] Boost C++ Library. [Online]. http://www.boost.org/

[13] GSL - The GNU Operating System. [Online]. http://www.gnu.org/software/gsl/

[14] Welcome to the UCR Time Series Classification/Clustering Page. [Online]. http://www.cs.ucr.edu/~eamonn/time_series_data/

[15] CUDA Memory Architecture. [Online]. http://www.cvg.ethz.ch/teaching/2011spring/gpgpu/cuda_memory.pdf

[16] Michele Dallachiesa, Besmira Nushi, Katsiaryna Mirylenka, Themis Palpanas. "Uncertain Time-Series Similarity: Return to the Basics" [Online]. http://disi.unitn.it/~dallachiesa/uncertaintssimilarity/