GPUを用いた複数の並列化手法による 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　自然科学研究において，計算機によって処理される不確実データは日々増大しており，計算手法の高速化及び効率化が求められている．不確実データからなる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 [5] 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

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．

分野を問わず最も頻繁に用いられるデータ形式の一つに時系列データがある．時系列データは時刻と共に変化する値を記録したものである．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を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．

本研究ではこれまでに，いくつかの手法を組み合わせる事で DUST の高速化を実現し，発表を行ってきた [3] [4]．本稿では，新たに行った高速化及び実験を中心に解説する．具体的には，シンプソン則を用いた積分計算の並列化，複数時系列での類似度計算の並列化，及びこれらの手法を用いた実装の性能評価である．

本稿の構成は以下の通りである．第 2 節では本研究で対象とする不確実時系列データについて述べ，第 3 節で不確実時系列データの類似検索における先行研究について述べる．第 4 節で不確実時系列データの類似検索手法の一つである DUST について説明する．第 5 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 6 節では具体的な高速化手法を説明する．第 7 節で今回行った実験の評価及び考察を行い，最後に第 8 節で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の類似検索

2.1 不確実時系列データ

時系列データとは，時間と共に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るデータの系列をいう．時系列データは気温や湿度など自然現象の観測データとして用いられるだけでなく，各種統計データや，音声や映像等のデータ表現としても用いられる．時系列データに対するマイニング技術は，過去の出来事を分析し，観測対象の異変をいち早く検知したり，将来の傾向を予測するため用いられ，その重要性を増している．

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

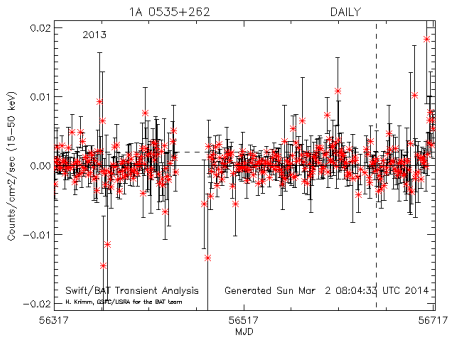


図 1　天体 1A0535-p262 の X 線光量を表す不確実時系列データ [25]

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や統計等のデータがある [6]．図 1 に示したある天体の X 線量を記録した時系列データでは，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．例えば，各時刻のデータが正規分布に従う誤差を持つ不確実時系列データ X は以下のように表される．

$$X = \langle x_0, \ldots, x_n \rangle$$
$$x_i = (\mu_i, \sigma_i)$$

但し，μ_i, σ_i は時刻 i の不確実データに対する誤差の正規分布の平均及び標準偏差を表す．

2.2　時系列データの類似検索

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で用いられる代表的な手法として，ユークリッド距離を類似度として用いる手法や，Dynamic Time Warping [1] などが挙げられる．

時系列間の類似度指標 $d(X, Y)$ が与えられた時、時系列データの集合

$$\mathbf{Y} = \{Y_0, \ldots, Y_m\}$$
$$Y_i = \langle y_{i,0}, \ldots, y_{i,n} \rangle$$

の内，時系列データ $X = \langle x_0, \ldots, x_n \rangle$ に最も類似した時系列を検索することは

$$\arg\min_i d(X, Y_i)$$

を求める事と同義である．

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．この問題に対処するため，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．しかし，このような手法の多くは確率計算など計算負荷の大きい処理を含むため，処理速度が遅く，大量のデータに対する類似検索は難しいという問題があった．

3.　先 行 研 究

3.1　MUNICH

MUNICH [7] 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．

3.2　PROUD

PROUD [8] 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．また，対象とする時系列データが十分な長さを持つ必要があるという問題がある．

3.3　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 [5]．この手法では，不確実時系列データ全体での距離指標だけではなく，二つの不確実データ点間の類似度指標を定義している．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．

4.　DUST

本研究では第 3. 節で挙げた不確実時系列データ類似検索手法の一つである DUST を対象に高速化を行った．本章では DUST について詳細に解説する．

なお，DUST では，不確実時系列データ全体での距離指標 DUST 及び，二つの不確実データ点間の類似度指標 dust を定義している．以下では DUST 及び dust をそれぞれ **DUST 距離** 及び **dust 関数** と呼ぶ．

4.1　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．なお，以下では時系列データの長さは予め揃えられているものとする．

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．

$$dust(x_i, y_i) = \sqrt{-\log(\phi(|x_i - y_i|)) - k} \quad (1)$$

$$k = -\log(\phi(0)) \quad (2)$$

但し，$\phi(|x_i - y_i|) = Pr(dist(0, |x - y|) \approx 0)$ であり，定数 k は t の値にかかわらず $dust(t, t) \approx 0$ となることを保証する調整項である．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，これらを総合し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．

$$DUST(X, Y) = \sqrt{\sum_{i=0}^{n} dust(x_i, y_i)} \quad (3)$$

実際には， $\phi(|x_i - y_i|)$ 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．

$$Pr(r(x_i) \approx r(y_i)|x_i, y_i) = \int_z Pr(r(x_i) \approx z|x_i) Pr(r(y_i) \approx z|y_i) dz \quad (4)$$

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．

$$Pr(r(x_i) \approx z|x_i) = \frac{Pr(x_i|r(x_i) \approx z) Pr(r(x_i) \approx z)}{Pr(x_i)} = \frac{Pr(x_i|r(x_i) \approx z) Pr(r(x_i) \approx z)}{\int_v Pr(x_i|r(x_i) \approx v) Pr(r(x_i) \approx v) dv} \quad (5)$$

すなわち，式 5 を計算するには 3 つの積分計算を行う必要がある．積分計算の実装については第 6.1 章で述べる．

5. 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 [9], [10].

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 [15] 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．

5.1 GPU のデバイス構成

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 では数百個から数千個搭載されている．実際の計算処理は Streaming Multiprocessor (以下「SM」と呼ぶ) または SMX と呼ばれる単位に分割され，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．

5.2 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．

5.3 CUDA プログラムのアーキテクチャ

本研究では NVIDIA 社の GPGPU プログラム開発環境である CUDA [15] を利用した．これは C/C++ や Fortran 等の言語で 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 ではメインプログラムから呼び出され，GPU 上で並列実行される関数をカーネル関数と呼ぶ．GPU での処理は Grid，Block，Thread という単位で分割され，カーネルはこれらの単位で並列に実行される．

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．カーネル関数 は割り当てられた SP 上で Thread として実行される．各 Thread はプログラムカウンタ，レジスタ，Local Memory を持つ．レジスタは Thread から利用できるメモリの内最も高速にアクセスできる領域である．CUDA プログラムにおいては，カーネル関数及び device 関数内の変数は基本的にレジスタ内に格納される．

32 個の Thread をまとめたものを *Warp* と呼ぶ．Thread の実行スケジューリングは Warp 単位で行われ，また同一 Warp 内の Thread は並列に実行される．

Block は複数の Warp あるいは Thread をまとめたものである．これは SM に対応し，各 Block は対応する SM 上で実行される．SM 上には L1 cache 及び *Shared Memory* の 2 種類のメモリが搭載されており，各 Block は頻繁に用いるデータをこれらの領域に格納することでメモリアクセスを高速化できる．特に Shared Memory はある程度のサイズを持つことから，Global Memory から読みだしたデータをキャッシュし，計算途中の値を保存する等の用途に用いられる．

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 は全体で 1 つのカーネル関数を実行し，Global Memory へのデータの入出力や，依存関係にあるカーネルの同期を行う．複数のカーネルを実行する場合やマルチ GPU プログラミングにおいては，複数の Grid の並列実行を制御する必要がある．

Grid 数，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る．ソースコード内で，Grid は 2 次元，Block 及び Thread 数は 3 次元のベクトルとして管理し，カーネル呼び出し時に指定する．実際に並列に実行される Thread 数及び Block 数は，指定された値を元にデバイスが自動的に決定する．また，処理を並列化する数に制約がない場合，Thread 数や Block 数は実際に使用する GPU の SP あるいは SM の数の倍数とする事が望ましい．これは，指定した値が SM 数や SP 数の倍数となっていない場合，各 SP, SM に均等に処理が割り当てられず，無駄が発生するためである．

6. 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，参考実装を元に，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．

6.1 dust 関数内の積分計算の並列化

我々はまず，dust 関数内部の積分計算の高速化を試みた．第 4.1 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，これらの積分計算は並列に処理可能である．

次に，数値積分手法自体の高速化を試みた．本研究では積分手法にモンテカルロ法による積分及びシンプソン則を用いた積分を採用し，GPU を用いてこれらの処理の高速化を行った．以下の節では各手法に対する高速化について説明する．

6.1.1 モンテカルロ法における試行の並列化

本節では，モンテカルロ積分を用いた DUST 実装の高速化について説明する．参考実装では GNU Scientific Library [19] を利用し，モンテカルロ法による dust 関数内部の積分計算を行っている．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．モンテカルロ法は実装方法が単純であり，不連続点の多い関数など他の数値積分法では計算が難しい関数も積分可能である事から，物理シミュレーションやコンピュータグラフィックスなど様々な分野で用いられている．

モンテカルロ法での具体的な計算方法，及びその並列化手法については [3] に省略する．

6.1.2 シンプソン則における積分区間の分割及び並列処理

モンテカルロ法による積分計算は大量の試行を必要とし，推定積分値の誤差も大きい．また，試行に用いる乱数の為に大量の記憶領域が必要となる．そのため，我々はシンプソン則を利用した積分を採用し，必要な場合のみモンテカルロ法を用いるようにした．簡単のため，以下ではシンプソン則を利用した積分手法をシンプソン法と呼ぶ．

シンプソン法はニュートン・コーツの公式において次数が 2 の場合の数値積分手法である．この手法では，積分区間の両端及び中点での関数値から被積分関数を 2 次関数で近似し積分値を計算する．シンプソン法は計算が比較的単純であり，区分求積法や台形法よりも精度が優れている．

シンプソン法で積分を行う具体的な流れは以下のとおりである． まず，積分区間を微小区間に分割する．分割された区間が小さいほど積分の誤差は小さくなる．次に，各微小区間ごとに推定積分値を求める．微小区間の両端を x_0, x_2, 区間の中点を x_1 とし，また $h = (x_2 - x_0)/2$ をおくと，関数 $f(x)$ の推定積分値は次の式で求められる．

$$\int_{x_0}^{x_2} f(x)dx = \frac{1}{3}h(f(x_0) + 4f(x_1) + f(x_2)) \qquad (6)$$

微小区間ごとに得られた積分値を合計することで，積分区間全体の推定積分値が得られる．

シンプソン則を利用した積分値の推定では，十分な精度を得るには積分区間の分割数を調整する必要がある．我々は積分区間の分割数を決定する為，正規分布関数を対象に調査を行った．調査対象を正規分布関数としたのは，不確実データの誤差情報として最も一般的に用いられる為である．正規分布関数の標準偏差は $\sigma = 1.0$ とした．

調査は以下の手順で行った．まず，幅 1.0 の区間の窓を想定する．この窓内で，初期分割数 $n = 2$ として積分値を推定する．次に分割数 $2n$ での積分値を推定する．2 つの推定値の差

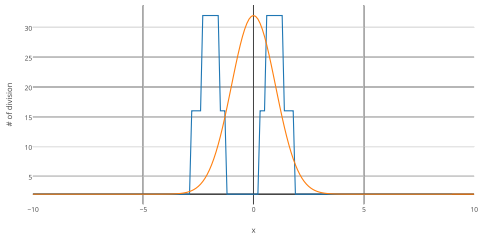


図 2　許容誤差 10^{-8} の時，正規分布関数の各部位で必要な分割数

が予め定めた許容誤差よりも小さければ十分な精度が得られているものとし，そうでなければ n を 2 倍し，同様の処理を繰り返す．窓を [-10.0, 10.0] 内で動かし，正規分布関数の各部位で必要な分割数を求める．

許容誤差を 10^{-8} とした時に各地点で必要な分割数は図 2 の通りであった．図より，シンプソン法では積分区間を (区間の幅) $\times$ 32 分割すれば十分な精度が得られるものとし，本研究に反映した．

我々はシンプソン法における各微小区間の計算を並列に行う事で高速化を行った．本研究での DUST の実装は，Dallachiesa らの参考実装に基づき，dust 内部の積分計算では積分区間を [-16, 16] としている．積分区間は [-16, 16] であるため，微小区間の総数は $32 \times 32 = 1024$ となる．すなわち，一度の積分計算で実行される Thread 数は 1024 となる．

6.2　Parallel Reduction

モンテカルロ法，シンプソン法共に，各スレッドで並列に計算した値を合計する処理が必要となる．モンテカルロ法では各試行の結果，シンプソン法では各微小区間の推定積分値である．これを GPU 上で並列に計算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集約処理を 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．OpenMP などの並列計算環境では予め Reduction のための命令が用意されているものも存在する．並列に Reduction を行う事を以下では Parallel Reduction と呼ぶ．

本研究では NVIDIA によるサンプル実装 [17] を参考に，GPU 上で Parallel Reduction を行うプログラムを実装した．具体的な処理の流れは以下のとおりである．まず，Reduction 対象の全ての値を Shared Memory に移動する．その後，GPU 上の全スレッドを利用して並列に Reduction を行う．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毎に Reduction を行う．

なお，各ブロックが処理すべきデータ数 N に対し，同時実行スレッド数を T としたとき，各スレッドはループにより N/T 個のデータを処理する．今回 Reduction の対象となる処理はモンテカルロ法，シンプソン法共に，各データに対する処理結果を加算する処理である．すなわち，各スレッドは全ての処理の結果を保存しておく必要はなく，予めスレッド毎に処理結果を加算して保存する．そのため，実際に Reduction を行うデータ数は T となる．

我々は GPU 上での Reduction 処理を最適化は divergent branch の解消，バンクコンフリクトの解消，ループの展開という順番で行った．本稿では細かい最適化手法についての詳細な説明は省略する．

6.3　時間毎の計算の並列化

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．すなわち，時系列データの次元数だけカーネル関数を呼び出しており，カーネル呼び出しのコストや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時刻の dust 関数を求める事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データを GPU 上の Global Memory にロードする．また，モンテカルロ法を用いる場合は，各試行に必要な乱数を GPU 上で生成する．次に，時系列データの各時刻での dust 関数処理を計算する．時系列データの長さと同数の Block を生成し，各 Block で対応する時刻の dust 関数を計算する．各 Block は対応する時刻の時系列のデータ，及びモンテカルロ法を用いる場合は試行に必要な乱数を Register にロードし，dust 関数を計算する．全 Block で dust 関数の計算が終了したら，結果を計算機のメインメモリに移動し，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．

6.4　複数の時系列データに対する DUST 計算の並列化

時系列データの類似検索において，最近傍検索など，ある時系列データと複数の時系列データとの組合せで類似度を求める処理が頻繁に用いられる．そのため，我々はある時系列データと複数の時系列データ処理間での DUST 距離を求める処理を GPU 上で並列に処理し，高速化を測った．

具体的な処理の流れは以下のとおりである．まず，検索対象となるの時系列データ及び比較する時系列データ全てを GPU の Global Memory に転送する．次に，前節までの実装と同様，時系列データの各時刻での dust 関数処理を計算する．時系列データの長さと同数の Block を生成し，各 Block で対応する時刻の dust 関数を計算する．この時，一度の dust 関数の計算に用いる Thread 数を使用する GPU の SM 中の SP 数の倍数とすることで，全ての Thread に均等に処理を割り当て，処理効率を向上できる．この実装では前節までの実装と比べ計算の並列度は向上しないが，GPU へのデータ転送のコスト及びカーネル関数の呼び出しコストを軽減できる．

7.　評 価 実 験

7.1　実 験 環 境

我々は提案手法の性能を評価するため，いくつかの実験を行った．実験に用いた計算機環境を表 1 に示す．

7.1.1　データセット

実験に当たり，UCR dataset [20] を利用した．UCR dataset

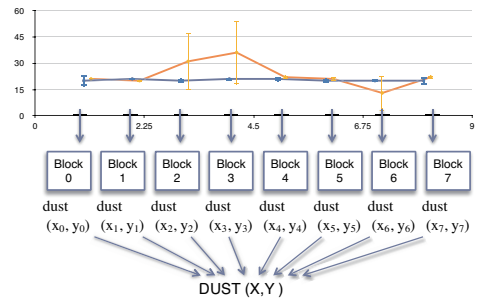


図 3　時刻毎の処理の並列化

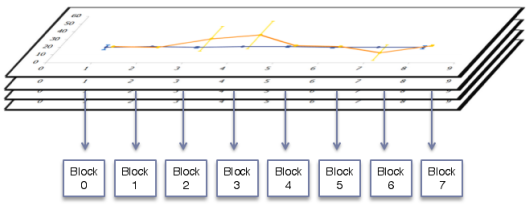


図 4　複数の時系列データに対する DUST 計算の並列化

表 1　Environment for experiments

CPU	Intel® Xeon® CPU E5-1650
CPU cores	6 (12 threads with Hyper-Threading)
CPU clock	3.50 GHz
Main memory	32 GB
GPU	NVIDIA® Tesla K20Xm
Scalar processors	2688
Processor clock	732 MHz
Global memory	6 GB

は様々な分野で得られた 47 種類の時系列データセットの集合であり，時系列に関する分類やクラスタリングなどの研究で用いられている．

我々は UCR データセット内の Gun-Point データセットを利用した．Gun-Point データセットは 150 次元の時系列データ 200 個からなるデータセットである．分類の研究に用いる場合は，データセット中 50 のデータを学習に用い，残り 150 の時系列を使って評価を行う．本研究では各時系列間の類似度が計算できれば十分である為，学習用データを区別せずに扱っている．

Gun Point データセットに含まれる時系列データは誤差情報を持たない．そのため，我々は正規分布に従って各時刻の値を撹乱した時系列データを作成し，実験に使用した．正規分布は標準偏差 $\sigma = 0.2, 0.4, ..., 2.0$ を用い，10 種類のデータセットを生成した．

7.1.2　実験に用いた実装

実装は C++ 言語で行い，GCC 4.6.3, nvcc 5.5 及び 6.5 をコンパイラに使用した．確率計算及びモンテカルロ法の実装に際し，Boost C++ Library [18] 及び GNU Scientific Library [19] を参考にした．

本研究の実装は，まず積分にモンテカルロ法を用いた場合の実装の高速化を行い，次にシンプソン法による積分を採用するという手順で行った．そのため，本章ではまずモンテカルロ法を用いた実装の性能評価について述べ，次にシンプソン法による積分を含めた性能評価について説明する．以下では，提案実装のうちモンテカルロ法を用いる物を *GPU Monte Carlo*, シンプソン法を用いる物を *GPU Simpson* と呼ぶ．

実験では比較のため，GPU Monte Carlo, GPU Simpson それぞれに対し，GPU を用いずに OpenMP [23] による並列計算を行う実装を別途作成した．これらの実装を *CPU Monte Carlo*, *CPU Simpson* と呼ぶ．

7.2　モンテカルロ法の並列化

7.2.1　モンテカルロ法の試行回数

モンテカルロ積分における試行回数を変化させた時，CPU Monte Carlo と GPU Monte Carlo 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．実験に使用した GPU の SP を考慮し，各 SP の負荷が均等になるよう，試行回数は $12288n\,(n = 1, 2, ..., 10)$ となるよう変化させた．次に，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 5(a) に示す．図に示す曲線の差が大きいため，縦軸は対数軸を用いている．どちらの実装においても，試行回数の増加に対し，処理時間はほぼ比例して増加している．

2 つの実装の処理時間の比を図 5(b) に示す．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 20 倍でほぼ一定となっている．ただし，試行回数が小さい時は，両実装の速度比はわずかに小さくなる．これは実行時間がそれほど長くないために，GPU でのカーネル関数の呼び出しのコストが相対的に大きくなる事が原因だと考えられる．

7.2.2　時系列の長さ

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．

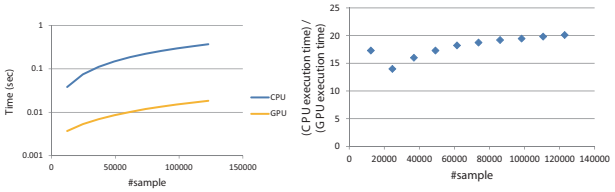
両実装の処理時間を図 6(a) に示す．図に示す曲線の差が大きいため，縦軸は対数軸を用いている．どちらの実装においても，時系列が長くなるにつれ，処理時間はほぼ比例して増加している．

2 つの実装の処理時間の比を図 6(b) に示す．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 20〜25 倍で大きい変化はない．時系列が短い時に両実装の速度比が小さくなるのは前節と同じくカーネル関数の呼び出しコストが原因だと思われる．

7.2.3　各高速化手法の効果

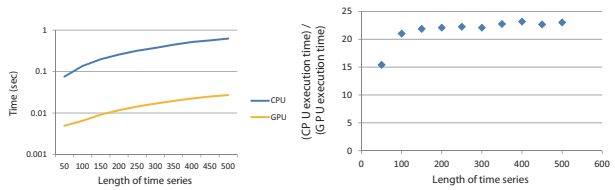
この実験では，第 6. 章で説明した高速化手法のうち，モンテカルロ積分の並列化，Parallel Reduction 及び時間毎の計算の並列化が DUST 距離の計算時間にどの程度影響するかを観察した．

比較のため，3 種類の実装を新たに用意した．GPU 1 はモン



(a) 実行時間　(b) CPU Monte Carlo と GPU Monte Carlo の速度の比

図 5　モンテカルロ積分の試行回数を変化させた時の DUST 距離の計算時間



(a) 実行時間　(b) CPU Monte Carlo と GPU Monte Carlo の速度の比

図 6　時系列の長さを変化させた時の DUST 距離の計算時間

表 2　Comparison of different implementations.

	実行時間 (sec)	参考実装との速度比
CPU 参考実装	14.03	1.0
CPU Monte Carlo	0.1675	83.7612
GPU 1	0.194	72.3196
GPU 2	0.07655	183.279
GPU 3	0.0911	1539.56

テカルロ積分の並列化のみ行った実装を，GPU 2 はモンテカルロ積分の並列化と Parallel Reduction を行った実装を，GPU 3 はモンテカルロ積分の並列化と Parallel Reduction 及び時間毎の計算の並列化を行った実装である．これらの実装に対し，データセット中からランダムに 100 組の時系列の組合せを選び，DUST 距離の計算にかかる時間の平均を測定した．測定結果を表 2 に示す．

図より，モンテカルロ積分の並列化のみを行った実装 GPU 1 の実行速度は CPU Monte Carlo よりも遅いことがわかる．近年では CPU のマルチコア化が進み，CPU もある程度の並列計算能力を備えており，シングルスレッドでの実行を想定した実装を単純に並列化するだけでは GPU の性能を活かし切れず，CPU と比べても利益は得られない．しかし，GPU 2 は GPU 1 に比べ 2 倍以上速く，GPU 3 は更に 8 倍ほど速くなっており，最終的に CPU Monte Carlo と比較しても 18 倍の計算速度を実現した．これらの事から，GPU プログラミングでは GPU の特性を考慮した設計が重要であると言える．

7.3　シンプソン則による積分の並列化

7.3.1　2 時系列データ間での DUST 距離の計算速度

シンプソン法を用い積分計算を行った場合，GPU による並列化でどの程度高速化が可能なのか調べるため，実験を行った．本実験で比較した実装は CPU Monte Carlo, CPU Simpson, GPU Monte Carlo, GPU Simpson の 4 つである．標準偏差 $\sigma = 0.2, 0.4, \ldots, 2.0$ の正規分布で撹乱したデータセットそれぞれからランダムに 100 通りの時系列データの組合せを選び，DUST 距離を計算した．この時，各実装での実行時間の平均を記録した．

結果を図 7 に示す．図より，CPU Simpson は GPU Monte Carlo よりも速いことがわかる．これは，本研究ではモンテカルロ法で使用するスレッド数が 49152, シンプソン法での積分に必要なスレッド数が 1024 と大幅に異なっていることが大きな要因となっている．さらに，モンテカルロ法の実行では乱数生成部分の処理コストが大きいことも一因となっている．

どの実装でも，実行時間はデータの標準偏差の大きさにかかわらずほぼ一定である．GPU Simpson は CPU Simpson より約 11〜12 倍高速であることがわかる．

7.3.2　複数の時系列データの DUST 計算の並列化

我々は複数の時系列データの DUST 計算を並列化した際の性能評価のための実験を行った．本実験で比較した実装は CPU Monte Carlo, CPU Simpson, GPU Monte Carlo, GPU Simpson の 4 つである．

実験は以下の手順で行った．Gun Point データセットから，時系列の数を $1, 2, \ldots, 200$ と変更した 200 個のデータセットを作成する．作成した各データセットごとに，各時系列データと他の全ての時系列データとの間で DUST 距離を計算し，プログラムの実行時間の平均を記録する．

なお，この実験では GPU Monte Carlo についてはデータセット内の時系列の個数が 67 個までの結果しか記録していない．GPU Monte Carlo はメモリ使用量が大きく，67 個以上の時系列データについて DUST を計算しようとすると，今回の実験で用いた GPU のメモリサイズを超過してしまうため，結果の計測は不可能である．実用的な実装では，大きなデータセットを用いた類似検索を行うために，データを分割して処理する等の工夫が必要となる．このような処理は実装自体の性能評価の妨げとなり，本研究の主目的から外れるため，本実験では GPU Monte Carlo については 67 個以下の時系列データを用いた結果のみを計測した．

結果を図 8 に示す．CPU Simpson と GPU Simpson の計算時間の比を観察すると，10 個の時系列データについて計算した時は 20 倍程度だが，200 個の時系列データについて計算した時は 60 倍を超え，比較する時系列データの数が多いほど速度比が大きくなることがわかる．これは，時系列データの数が増えるほど，比較に用いるデータの転送コストやカーネル関数の呼び出しコストが実行時間全体に対し小さくなる為と思われる．また，全体的に CPU Simpson と GPU Simpson の計算時間の比は前節での実験よりも大きくなっている．これは第 6.4 節で述べた通り，複数時系列での DUST 距離計算を一度のカーネル呼び出しで行うことで，カーネル呼び出しやデータ転送のコストが抑えられるためであると考えられる．

8.　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

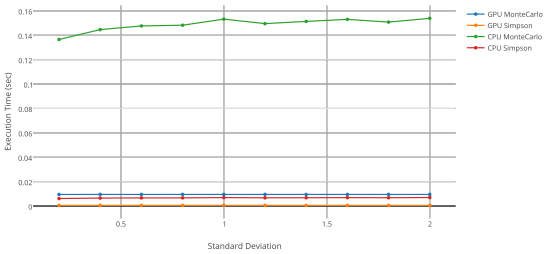


図 7　2 時系列データでの DUST の計算時間

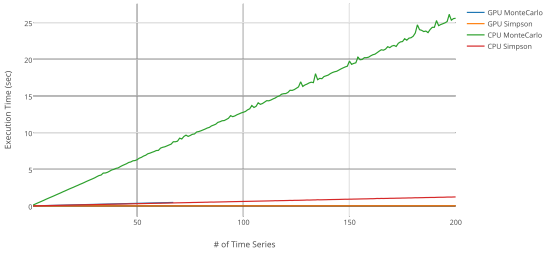


図 8　複数の時系列データについて DUST を並列計算した時の計算時間

る 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 内部の積分計算の並列化，Parallel Reduction，時刻毎の計算の並列化によって処理時間の高速化，複数の時系列に対する DUST 計算の並列化を行った．積分計算にはモンテカルロ法及びシンプソン法の 2 種類を用い，それぞれ評価を行った．最終的に，モンテカルロ法による実装では GPU を用いない実装に対し約 20 倍の高速化を，シンプソン法による実装では GPU を用いない実装に対し約 12 倍の高速化を実現した．今後の展望としては，複数の GPU を用いる実装についての研究が考えられる．また，対象とする時系列データの種類を限定することで，その特徴を考慮したより特化した手法を開発できると思われる．

謝　　辞

本研究の一部は，平成 26 年度共同研究 (産総研基盤 A 24240015A) によるものである．

文　　献

[1] H. Sakoe et al., “Dynamic Programming Algorithm Optimization for Spoken Word Recognition,” IEEE Trans. on Acoustics, Speech and Signal Processing, Vol. ASSP-26, No. 1, pp. 383-392, 1978

[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, pp. 19-27, 2013

[3] 黄 峻, 小澤 佑介, 天笠 俊之, 北川 博之,“GPGPU を用いた不確実時系列データ類似検索の高速化,” 第 6 回データ工学と情報マネジメントに関するフォーラム (DEIM 2014), D5-1, 2014

[4] Jun Hwang, Yusuke Kozawa, Toshiyuki Amagasa and Hiroyuki Kitagawa, “GPU Acceleration of Similarity Search for Uncertain Time Series,” The 3rd Workshop on Advances in Data Engineering and Mobile Computing (DEMoC-2014) in conjunction with NBiS-2014, pp.627-632, Sept. 10-12, 2014.

[5] S. Sarangi et al., “DUST: A Generalized Notion of Similarity Between Uncertain Time Series,” Proc. 16th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD ’10), pp. 383–392, 2010

[6] C. Aggarwal and P. Yu, “A Survey of Uncertain Data Algorithms and Applications,” IEEE Trans. on Knowledge and Data Engineering, vol. 21, no. 5, pp. 609-623, 2009

[7] J. Aßfalg et al., “Probabilistic Similarity Search for Uncertain Time Series,” Proc. 21st International Conference on Scientific and Statistical Database Management (SSDBM 2009), pp. 435-443, 2009

[8] M. Yeh et al., “PROUD: A Probabilistic Approach to Processing Similarity Queries over Uncertain Data Streams,” Proc. 12th International Conference on Extending Database Technology: Advances in Database Technology (EDBT ’09), pp. 684-695, 2009

[9] W. Liu et al., “Bio-sequence Database Scanning on a GPU,” Proc. 20th International Conference on Parallel and Distributed Processing (IPDPS 2006), pp. 251, 2006

[10] S. Manavski., “CUDA Compatible GPU as an Efficient Hardware Accelerator for AES Cryptography,” IEEE International Conference on Signal Processing and Communications, 2007 (ICSPC 2007). pp. 65-68, 2007

[11] J. Owens et al., “GPU Computing,” Proc. of the IEEE, vol. 96, No. 5, pp. 879-899, 2008

[12] V. Lee et al., “Debunking the 100X GPU vs. CPU Myth: An Evaluation of Throughput Computing on CPU and GPU,” Proc. of the 37th Annual International Symposium on Computer Architecture, pp. 451-460, 2010

[13] Simpson’s Rule – from Wolfram MathWorld
http://mathworld.wolfram.com/SimpsonsRule.html

[14] M. Dallachiesa et al., “Uncertain Time-series Similarity: Return to the Basics,” Proc. VLDB Endowment, vol. 5, no. 11, pp. 1662-1673, 2012

[15] Parallel Programming and Computing Platform — CUDA — NVIDIA
www.nvidia.com/cuda

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

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

[18] Boost C++ Library
www.boost.org/

[19] GSL - The GNU Operating System
www.gnu.org/software/gsl/

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

[21] CUDA Memory Architecture.
www.cvg.ethz.ch/teaching/2011spring/gpgpu/cuda_memory.pdf

[22] M. Dallachiesa et al., “Uncertain Time-Series Similarity: Return to the Basics,” http://disi.unitn.it/~dallachiesa/uncertaintssimilarity/

[23] OpenMP.org
http://openmp.org/wp/

[24] Intel Instruction Set Architecture Extensions
http://software.intel.com/en-us/intel-isa-extensions/

[25] MAXI — RIKEN
http://maxi.riken.jp/top/index.php