DBSJ Japanese Journal, Vol.13-J, No.2
February 2015

GPUを用いたCanopyクラスタリングの高速化
GPU Acceleration of Canopy Clustering

林 史尊♡　小澤 佑介♢
天笠 俊之♠　北川博之♠

Fumitaka HAYASHI　Yusuke KOZAWA
Toshiyuki AMAGASA　Hiroyuki KITAGAWA

大規模なデータに対するクラスタリングは一般に計算コストが高く，その高速化が強く求められている．クラスタリングの高速化を実現する手法の一つにCanopyクラスタリングがあり，これは対象データに事前処理を施すことで，k-means法や凝集法といったアルゴリズムの高速な実行が可能になる．一方，並列計算によってクラスタリングを高速化する手法も数多く提案されており，特に近年ではGPU（Graphics Processing Unit）を利用したアプローチが注目されている．ところが，クラスタリングの前処理として位置付けられるCanopyクラスタリングの高速化についてはほとんど研究されていない．そこで，本研究では，GPUを用いたCanopyクラスタリングの高速化手法を提案する．評価実験により，提案手法の有効性を評価する．

In general, clustering large datasets requires high computational costs, hence there is a strong demand to speedup clustering. Canopy clustering is an approach to speedup clustering algorithms, such as k-means and hierarchical agglomerative clustering, by applying some preprocessings. In the meantime, parallel processing of clustering algorithms has been extensively studied, and in particular GPU (Graphics Processing Unit) has been gaining much attention as a mean to speedup clustering. However, no research has been done to speedup canopy clustering. In this paper, we propose a scheme to accelerate canopy clustering using GPU. We evaluate the effectiveness of the proposed scheme by experimental evaluations.

1. はじめに

近年，データの大規模かが急速に進んでおり，それに従い大規模データの解析に要するコストは肥大化している．そのため，データ処理の効率化が強く求められている．データ・クラスタリング（以降，単に「クラスタリング」と呼ぶ）はデータ解析において頻繁に用いられる手法であるり，データ集合をデータ間の類似度に基づきグループ化する処理である．クラスタリングは一般に計算コストが高いため，大規模データに適用するのは難しく，その高速化が求められている．

♡ 非会員　筑波大学大学院システム情報工学研究科
hayashi@kde.cs.tsukuba.ac.jp
♢ 学生会員　筑波大学大学院システム情報工学研究科
kyusuke@kde.cs.tsukuba.ac.jp
♠ 正会員　筑波大学システム情報系
{amagasa, kitagawa}@cs.tsukuba.ac.jp

クラスタリングの計算高速化方法の一つとして，Canopyクラスタリング [1] が注目を集めている．Canopyクラスタリングは，凝集法やk-means法といったクラスタリング手法の前処理として位置付けられる．Canopyと呼ばれるおおまかなクラスタを事前に生成しておき，クラスタリングにおける距離計算をCanopy同士に対して行うことによって，計算量を大幅に削減することができる．これまで大きな計算時間を要していた大規模データクラスタリングを，実用的なレベルまで高速化できるため，近年注目されている．例えば，Zhi-Gang Fanらは，画像データのクラスタリングにおいて，Canopyクラスタリングを用いた高速化を図った [2]．また，Tuli Deらは，$700,000 \times 1,500$ という大きなサイズの銀河系外スペクトルデータに対し，Canopyクラスタリングを利用してクラスタリング行った [3]．

一方，大規模なデータに対して高速なクラスタリングを行う別のアプローチとして，並列処理が挙げられる．特に，グラフィックス処理に用いられるGPU（Graphics Processing Unit）を一般の処理に用いるGPUコンピューティング [4] が近年注目を集めている．GPUは，小規模なプロセッサを大量を持つという特徴から，単純な計算を高い並列度で実行するような処理に対して高い性能を発揮する．GPUは，クラスタリングにも適用されており，凝集法 [5, 6, 7] やk-means法 [8, 9] などについてはすでにいくつかの手法が提案されている．ところが，CanopyクラスタリングのGPUによる高速化については，これまで議論されていない．

そこで本研究では，GPUを利用したCanopyクラスタリングの高速化手法を提案する．まず，Canopyクラスタリングでは，データ間の距離計算が大量に実行されるため，これをGPUのスレッドを用いて並列に実行する．これに加えて，効率を高めるため，データ空間をセル構造に分割し，距離計算の対象となる候補を絞り込む手法についても議論する．さらに，ナイーブなCanopyクラスタリングおよびセル構造を用いたCanopyクラスタリングそれぞれをCPUおよびGPU上で実装し，提案手法の有効性を実験によって評価した．

2. 基本的事項

2.1 クラスタリング

クラスタリングとは，データ集合をデータ間の類似性に基づきグループ（クラスタ）化する処理である．規則性の発見やデータの分類に利用される．クラスタリングは，その結果が階層構造を持つか否かで，階層型と非階層型の二種類に分けられる．

階層型クラスタリングでは，逐次的にクラスタの結合または分割を繰り返すことで，階層的クラスタ構造を得る．凝集法は代表的手法であり，デンドログラムと呼ばれる階層構造が得られる．凝集法は，各クラスタ間の類似度を計算し，類似度が最も高い2個のクラスタを結合することを，クラスタ数が1個になるまで繰り返す（終了条件のクラスタ数を設定することも可能）．凝集法の計算量は，データ数を n とすると $O(n^3)$ と大きい．

非階層型クラスタリングでは，クラスタの分割精度を評価する関数を用意し，評価値が最適なものとなる分割を探索する．代表的な手法にk-means法がある．k-means法は，重心をいくつか定め，各重心について近いデータを求め，各重心を再計算するという処理を，すべての重心の位置が変化しなくなるまで繰り返す．k-means法の計算量は，データ数を n，求めるクラスタ数を k，反復回数を t とすると $O(tkn)$ となる．

これらの手法を単純に処理するのは非効率的である．そのため，アルゴリズムの改良や事前処理などが行われる．前者の例としては，F. Murtaghが最近傍グラフを利用し凝集法の計算量を $O(n^2)$ に抑えている [10]．後者の一例にはCanopyクラスタリング [1] がある．これは，あらかじめ簡単な計算によりおおまかなクラスタを生成しておくことで，後に行う一般的なクラスタリング計算を軽減するという手法である．アルゴリズムの詳細は3節で述べる．

2.2 GPU コンピューティング

GPU（Graphics Processing Unit）は画像処理に特化した演算装置である．近年，GPU を，グラフィックス処理とは異なる用途で利用することが注目されている [4]．このように，GPU を汎用的計算に活用する技術を GPU コンピューティングと呼ぶ．

CPU は大きなプロセッサを数個持つが，GPU は小さなプロセッサを数百個持つ．GPU 内には複数のストリーミングマルチプロセッサ（SM）が存在し，SM はまた複数のストリーミングプロセッサ（SP）から構成される．SM は一つの処理を行う単位となる．

GPU が利用できるメモリもまた，複数の層から構成される．例えば，各 SM が持つオンチップメモリは，対応する SM から高速アクセス可能だが，数十 KB と容量が小さい．デバイスメモリは，CPU 側からアクセス可能で，かつどの SM からもアクセスでき数百 MB〜数 GB と大容量だが，アクセス遅延が大きい（処理実行の単位時間に比べ数十〜数百倍）．GPU での並列処理を行う際は，各メモリの特徴を捉えてデータ分割を行う必要がある．

GPU を用いた処理の基本的な流れは次のようになる．

1. CPU 上で，GPU を使った計算に必要なデータを準備する（GPU 上のメモリ確保等）
2. データを GPU に転送する
3. カーネル（GPU 上の処理を定義した関数）を呼び出す
4. GPU による処理
5. 処理後のデータや結果を必要に応じて CPU に転送する

なお，基本的にカーネル内で動的なメモリ確保は行えない．そのため，どれほどのサイズが必要か分からないデータが存在する場合であっても，カーネル実行前に適当なサイズ分のメモリを確保しなければならない．また，CPU − GPU 間のデータのやり取りは時間がかかるため，データ転送およびカーネル呼び出しの回数はなるべく抑える必要がある．

GPU コンピューティング向けの統合開発環境の一つに，NVIDIA が提供する CUDA（Compute Unified Device Architecture）がある．CUDA は，GPU が行う処理の指示，主記憶と GPU 用メモリ間のデータ転送などをサポートする．プログラミングレベルでは，基本的に C/C++言語のライブラリとして利用する．CUDA プログラミングは，GPU アーキテクチャと対応する階層構造（スレッディングモデル）を基軸として行われる．スレッディングモデルは，スレッド，ブロック（スレッドブロック），グリッド（カーネルグリッド）からなる．スレッド処理は各 SP で実行される，ブロック処理は各 SM で実行される．ブロックの集合をグリッドという．グリッドは GPU 上で行う一連の処理に当たる．通常，1 ブロックでは数百個のスレッドを実行し，1 グリッド内では数百〜数千万のブロックが実行される [1]．実際に使用するスレッド数とブロック数は，処理内容に基づき適切な値をあらかじめ設定しておく必要がある．ただし，現実の計算は物理的構造による制約を受けるため，すべてのスレッドおよびブロックが一度に並列処理されるとは限らない．一度に並列処理しきれない場合は，同じ SM，SP を繰り返し利用する．

スレッドは常に 32 個を同時に計算するが，このまとまりをウォープと呼ぶ．ブロックの処理の中では，1 個のウォープを実行する間に，次のウォープに必要なデータをメモリから得るという流れが繰り返される．

論理的なメモリ構造としては，例えば，ブロック処理中にアクセスできるシェアードメモリや，CPU 側からもアクセス可能なグローバルメモリなどがある．それぞれ，シェアードメモリはオンチップメモリに，グローバルメモリはデバイスメモリに対応する．CPU から転送されたデータは，基本的にグローバルメモリに置かれる．また，先述の通り，グローバルメモリ（デバイスメモリ）はアクセス遅延が大きいため，データの再利用などを行う際にはシェアードメモリを活用する．なお，GPU はグローバルメモリやシェアードメモリ以外にも数種類のメモリを内蔵するが，ここでは説明を割愛する．

[1] 本研究で用いる fermi アーキテクチャでは，1 ブロック内最大スレッド数は 1024，1 グリッド内最大ブロック数は 65535 × 65535 である．

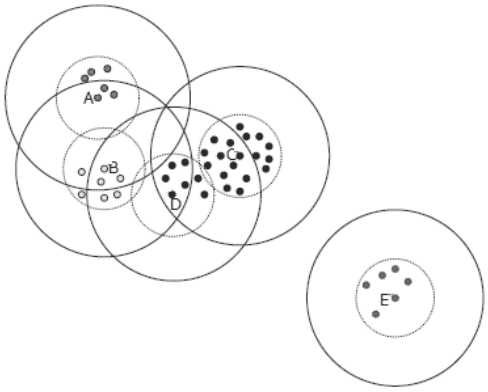

図 1: Canopy の生成．アルファベットのついた点は各 Canopy の中心．実線は中心点からの距離 T_1，破線は中心点からの距離 T_2 を示す．（図は “Efficient Clustering of High Dimensional Data Sets with Application to Reference Matching” [1] Figure 1 を引用．）

3. Canopy クラスタリング

Canopy クラスタリングの基本的なアイデアは，データ集合を Canopy と呼ばれるグループに分割する（1 個のデータが複数の Canopy に属してもよい）．その後，凝集法や k-means 法のような厳密なクラスタリングを行うが，ここでの各データ間の距離計算はそれぞれの Canopy の中のみで行われる．クラスタとなり得るデータをあらかじめ Canopy としてまとめておくことで，精度を大幅に犠牲にすることなく，実際のクラスタリング処理における距離計算を大幅に削減することができる．

Canopy を生成するアルゴリズムの流れは以下のようになる（図 1 を参照）．

1. クラスタリングを行いたいデータセットを用意し，各データのインデックスを中心点候補とする．
2. 中心点候補の中から 1 個のデータをランダムに選択する．
3. 2 で選んだデータと他のすべてのデータについて，各々の距離を計算する．
4. 距離が T_1 以下のデータを Canopy に含める．
5. 距離が $T_2 (< T_1)$ 以下のデータのインデックスを中心点候補から除外する．
6. 中心点候補が 0 個になるまで，2〜5 の処理を繰り返す

生成された Canopy は，以下の性質を持つ．

- 複数の Canopy に，同じデータが所属し得る．
- すべての Canopy の中心点について，互いの距離は T_2 より大きい．

なお， T_1 および T_2 の値は，データの性質や距離尺度等をもとに設定する．T_1 は各 Canopy が含むデータ数に関わり， T_2 は Canopy 生成数に関わる．生成した Canopy の大きさや数は，この後に行われる本格的なクラスタリングの計算結果の妥当性，および計算時間に影響する．なお，Canopy クラスタリング後の処理については，本論文では議論しない．

4. GPU 上での並列処理

ここでは Canopy クラスタリングを GPU 上で並列処理する手法について説明する．前節で述べた Canopy クラスタリングの手順のうち，「2. 中心点候補の中から 1 個のデータをランダムに選

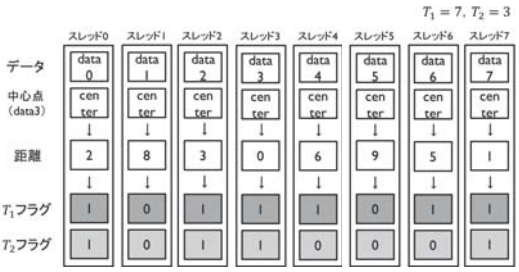

図 2: 距離計算の並列化．各スレッドは，スレッド番号に対応するインデックスのデータと，中心点となるデータを読込む．(この例では，data3 が中心点として選ばれている．) 中心点データと各データで距離計算を行い，その結果によって，T_1 および T_2 フラグの値を決定する．

択する．」から「5. 距離が $T_2(< T_1)$ 以下のデータのインデックスを中心点候補から除外する．」は GPU 上で実行することが可能である．以下では，ページ数の都合から，3, 4, 5 についてその詳細を説明する．

前提として，データセットは事前に GPU 上のグローバルメモリ上に配置しておく．また，Canopy クラスタリングは他の一般的なクラスタリング手法の事前処理として行われるため，後の処理を GPU 上で行うことを前提として実装する．具体的には，各 Canopy にどのデータが所属しているかという情報を GPU 上に残す．

4.1 中心点の選択と距離計算

処理の簡単のため，新しく選択する Canopy の中心は，常に中心点候補の先頭要素とする．アルゴリズムの性質上，どのデータが中心として選択されたとしても，T_2 近傍にあるデータは必ず中心点候補から削除される．そのため，中心点選択をランダムに行わないことの影響は小さいと考えられる．

続いて，中心点となったデータと各データとの距離計算を，各スレッド上で並列に行う．1 個のスレッドで中心点データと比較対象のデータ 1 個を読込み，その距離を算出する．さらに，同一スレッド上では，算出距離と T_1 および T_2 値との比較を行い，T_1 より距離が小さい場合は T_1 フラグとして 1 を，T_2 より距離が小さい場合は T_2 フラグとして 1 を，それぞれ T_1 フラグ列，T_2 フラグ列に記録する（1 でない場合は 0 を記録する）．

距離計算および T_1, T_2 フラグを求める処理について，簡単な図を用いて説明する．2 は，8 個のデータについて処理を行おうとするものである．ここで，$T_1 = 7, T_2 = 4$ と設定し，中心点には data3 が選択されたものとする．各スレッドは，まず，スレッド番号に対応するインデックスのデータ（data0〜data7）と，中心点となるデータ（center，この例では data3）を読込む．その後，距離計算を行ったところ，それぞれのスレッドで 3 段目のような結果が得られた．各スレッドは自身の中で求められた距離と T_1, T_2 の値を比較して，割当てられたデータが中心点に対しどのような位置にあるかを，T_1 フラグ，T_2 フラグとして求める．例えば data0 ならば，中心点から距離が 2 であり，これは T_1 および T_2 以下であるため，それぞれのフラグに 1 が記録される．

4.2 Canopy の更新

距離計算処理によって得られた T_1 フラグ列を用いて，選択した中心点に対応する Canopy の所属データ（Canopy メンバ）を記録する．このとき，Canopy メンバを更新する場合に必要な情報は，計算対象とする Canopy にどのインデックスのデータが所属しているかである．これは，T_1 フラグ列中で 1 の値を持つインデックスを集めることで得られる．フラグの立ったインデックスを抽出するには，Inclusive Scan [11] と呼ばれる処理を行う．Inclusive Scan では，配列 $Q_{in} = [q_0, q_1, q_2, ..., q_n]$ から配列 $Q_{out} = [q_0, q_0 + q_1, q_0 + q_1 + q_2, ..., q_0 + q_1 + q_2 + ... + q_n]$ が得られる．

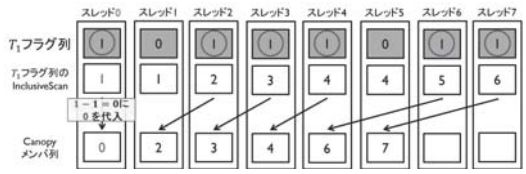

図 3: Canopy の更新．T_1 フラグ列（1 段目）と Inclusive Scan 後の T_1 フラグ列（2 段目）を用意する．T_1 フラグ列中で値が 1 であるインデックスについて，Inclusive Scan 後の T_1 フラグ列の値を確認する．インデックス値を，Canopy メンバ列中の確認した値の位置に格納する．なお，2 回目の Canopy 生成処理以降は，既存の Canopy メンバ数 + Inclusive Scan 後の T_1 フラグ列の値の位置にインデックス値を格納すればよい．

Inclusive Scan 後の T_1 フラグ列は，T_1 フラグが立っているデータの数を先頭から順に記録しているものとみなせる．この特徴を利用することで，T_1 フラグ列と Inclusive Scan 後の T_1 フラグ列から，T_1 フラグの立ったインデックスを抽出した配列が得られる．まず，T_1 フラグ列において値が 1 であるインデックスの，Inclusive Scan 後の値を得る．抽出配列では，この値の位置にインデックス値が格納される（図 3 参照．ただし，配列のインデックスは 0 から数えるため，実際に格納される位置は，Inclusive Scan 後の値 −1 となる）．

この抽出配列は計算対象である Canopy のメンバであるため，これを Canopy メンバ列に追加し，その時点での Canopy メンバ列の要素数を Canopy 境界列に記録することで，Canopy の情報が更新される．

4.3 中心点候補の更新

続いて，T_2 フラグ列を用いて，中心点候補の除去を行う．その時点で残っている中心点候補それぞれの T_2 フラグ列を調べ，値が 1 であるものを除去，あるいは値が 0 であるものを抽出すればよい．これは，以下に示す 3 段階の処理によって実現される．

1. 中心点候補に残っているデータのインデックスについてのみ，T_2 フラグ列から値を抽出する．また，3 の処理のため，抽出時にフラグの値（0, 1）を反転させる．
2. 抽出 T_2 フラグ列の Inclusive Scan を計算する．
3. その時点での中心点候補，抽出 T_2 フラグ列，Inclusive Scan 後の抽出 T_2 フラグ列から，中心点候補として残るデータのインデックスを求める．

1 の処理は単純に，中心点候補であるデータのインデックスそれぞれについて T_2 フラグ列の値を確認し，抽出 T_2 フラグ列に格納する（図 4 参照）．このとき，3 の処理で T_2 フラグが 0 であるものを抽出するため，T_2 フラグの値を反転させる．

3 の処理の基本的な流れは，Canopy 更新時の処理と同様である．抽出 T_2 フラグ列中で値が 1 であるインデックスについて，Inclusive Scan 後の抽出 T_2 フラグ列の値を確認する．中心点候補の同インデックスに格納されている値を，新しい配列の確認した値の位置に格納し，これを更新後の中心点候補とする（図 4 参照）．なお，抽出 T_2 フラグ列の末尾の値は，更新後の中心点候補の総数を意味する．この情報は CPU 側に転送され，Canopy クラスタリングの終了条件判定に用いられる（値が 0 であれば Canopy クラスタリングを終了する）．

5. セル構造を用いた Canopy クラスタリング

5.1 基本的アイデアとアルゴリズム

前節で述べた GPU 化の手法は，オリジナルの Canopy クラスタリングのアルゴリズムの基本的な構成には手を加えず，各ステップの処理を並列化したものだった．ここでは更なる高速化を試みる．具体的なアイデアは，一定の以上離れた Canopy 同士は，そ

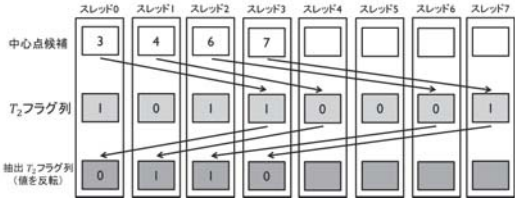

図 4: 中心点候補の更新 (1)．更新前の中心点候補（1 段目）それぞれの T_2 フラグの値を，抽出 T_2 フラグ列として得る．更新後は T_2 フラグの値が 0 であるもの（中心点から距離 T_2 以内に存在しないデータ）を残したいため，後の処理に向けて T_2 フラグの値を反転させておく．

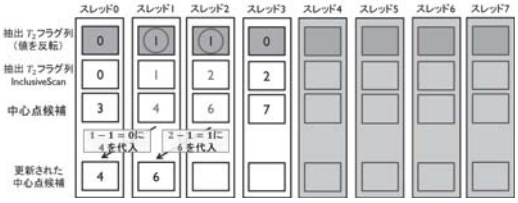

図 5: 中心点候補の更新 (2)．抽出 T_2 フラグ列と Inclusive Scan 後の抽出 T_2 フラグ列から，中心点候補が更新後，どの位置に格納されるかが分かる．ここから，T_2 フラグが 1 になっている中心点候補を抜き出せる．

れが含むデータに重複がないため，並列に処理できるというものである．

これを実現するため，本研究ではセル構造を用いる．データ空間を間隔 T_1 のメッシュ状に分割し，分割された個々の領域をセルと呼ぶ．このとき，すべてのデータは必ずいずれか 1 個のセルに含まれる（計算上は，各データに対し所属セルを求める）．ここで，ある 1 個のデータが Canopy の中心点に選ばれたとする．この中心点に対する計算範囲は，中心点が所属するセルと，それに隣接するセルになる．セルのサイズから，その Canopy に含まれる（中心点から距離 T_1 以内にある）データは，中心点の所属セルおよび隣接セル内にしか存在し得ないためである．同様に，中心点と計算範囲が重複しないように，異なる中心点を可能な限り選択する（図 6 参照）．選択後，すべての計算範囲内のデータについて，距離計算，Canopy および中心点候補の更新を行う．

提案手法のアルゴリズムは以下の通りである．

1. クラスタリングを行いたいデータセットを用意し，各データのインデックスを中心点候補とする．
2. 各データの所属セルを計算し，それぞれの所属セルに対応する隣接セルを求める．(各所属セルはセル番号を与えて管理する)．具体的には，全データについて以下の処理を行う．
   (a) 各次元の値と T_1 との商（小数点以下切り捨て）を求め，所属セルの位置情報とする．
   (b) 同じ位置情報を持つセルがそれまでに現われていなければ，その所属セルに新しいセル番号（直前の番号 +1）を与える．
   (c) 新しいセル番号が現われたとき，これまでに現れたセルの位置情報と比較して，すべての次元について差が 1 以下であるセルを，その所属セルの隣接セルとする．
3. 中心点候補の中から，計算範囲が重複しないように複数の中心点を選択する．具体的には，以下の処理を行う．
   (a) この時点での中心点候補と同一の配列を用意し，これを cp 中心点候補とする．
   (b) 選択した中心点を格納する配列を用意し，これを中心点列とする．
   (c) cp 中心点候補の先頭を選択して取り出し，中心点列に格納する．
   (d) 選択した中心点の所属セル，またその隣接セルに所属するデータを，cp 中心点候補から除去する．
   (e) cp 中心点候補が 0 個になるまで，c，d の処理を繰り返す．
4. 各中心点に対応する計算範囲内のデータを抽出する．具体的には，以下の処理を行う．
   (a) 各中心点に対応する計算範囲内のデータのインデックスを格納する配列を用意し，これを各中心点対応データメンバ列とする．
   (b) 各中心点について，その所属セルおよび隣接セルに所属するデータを，各中心点対応データメンバ列に加える．
5. 3 で選んだ複数の中心点と，それぞれの中心点に対応する計算範囲内のデータについて，各々の距離を計算する．
6. 距離が T_1 以下のデータを Canopy メンバに追加する．
7. 距離が $T_2 (T_1 > T_2)$ 以下のデータのインデックスを中心点候補から除外する．
8. 中心点候補が 0 個になるまで，3〜7 の処理を繰り返す

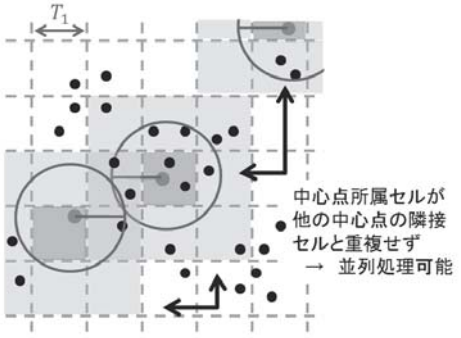

図 6: セル構造を用いた複数 Canopy の並列生成．この例は，2 次元データ空間にセルを生成したものである．中心点（緑）を，計算範囲（赤および橙のセル）が重複しないように選択する．これにより，複数の Canopy を並列して生成することが可能となる．

セル構造を用いない Canopy クラスタリングと比較して，2〜4 の処理が追加されている．後の基本的な流れは元手法と変わらない．

2 については，処理が逐次的で GPU 上での並列処理に不向きであるため，CPU 上で行うものとする．ここでは，全データについて，各次元の値からそのデータがどのセルに所属しているかを求める．実際には，それぞれのデータに対し，各次元の値とセルのサイズの商（小数点以下切り捨て）をそのデータが所属するセルの座標とする．セルにはセル番号が与えられ，既存しない座標のセルが現われた場合には，そのセルに新しい番号を与える．データとセルは，データ番号およびセル番号を用いて互いの所属情報を持つ．ここで，セルのサイズを一辺 T_1 とすることにより，あるデータが Canopy 中心点として選択されたとき，その Canopy のメンバとなり得るデータは，その中心点の所属セルとその隣接セルのみであると判定できる．Canopy 計算時には，各中心点に対して所属・隣接セル内のデータを求めることで，距離計算の回数を削減することができる．

ある $C_n(c_{n,0}, c_{n,1}, ..., c_{n,d})$ において，その隣接するセルの座標は，C_n との各次元の差が ±1 であるという条件を満たす．これを利用することで，隣接セルを以下のように求められる．

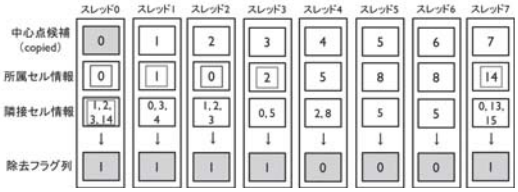

図 7: 複数中心点の選択．1 段目は cp 中心点候補，2 段目は各候補の所属セル番号，3 段目は所属セルの隣接セルのセル番号である．cp 中心点候補には所属・隣接セルに所属しないデータを残すため，これらに残存フラグを立てる．その後は同様に処理を行ない，cp 中心点候補を更新する．

1. 既存セルを，小さい次元を優先して昇順ソートする（辞書順ソート）．
2. k の初期値を 0 として，以下の処理を再帰的に繰り返す．
   (a) $c_{n,k}$ と各既存セルの k 次元目の座標を比較する．
   (b) 初めて $c_{n,k} - 1 ¡ c_{i_{-1},k}$ となる位置 i_{-1}，初めて $c_{n,k}$ ¿ $c_{j_{-1},k}$ となる位置 j_{-1} を求める．
   (c) 初めて $c_{n,k} - 1 ¡ c_{i_{+1},k}$ となる位置 i_{+1}，初めて $c_{n,k}$ ¿ $c_{j_{+1},k}$ となる位置 j_{+1} を求める．
   (d) $[i_{-1}$〜$j_{-1}], (j_{-1}$〜$i_{+1}), [i_{+1}$〜$j_{+1}]$ の範囲内それぞれで，$k+1$ 次元目の座標を比較する．$k = d$ のとき，範囲内にある $c_{n,d} \pm 1$ と等しい座標を持つセルを，C_n の隣接データとする．

5.2 GPU 上での並列処理

ここでは，セル構造を用いた手法の詳細を述べる．

5.2.1 複数中心点の選択

中心点を，その計算範囲（所属セル＋隣接セル）が重複しないように複数選択する．そのためには，新しく中心点を選択する度に，所属・隣接セル内のデータを候補から除去していけばよい．GPU では，以下のような処理を行う．

(a) この時点での中心点候補と同一の配列を用意し，これを cp(copied) 中心点候補とする．
(b) 選択した中心点を格納する配列を用意し，これを中心点列とする．
(c) cp 中心点候補の先頭を選択して取り出し，中心点列に格納する．
(d) 選択した中心点の所属セル，またその隣接セルに所属しないデータについて残存フラグを立てる（図 7 参照）．
(e) 残存フラグ列の Inclusive Scan を計算する．
(f) cp 中心点候補，残存フラグ列，Inclusive Scan 後の残存フラグ列を利用して，cp 中心点候補を更新する．
(g) cp 中心点候補が 0 個になるまで，(c)〜(f) の処理を繰り返す．

5.2.2 各中心点に対応する計算範囲内のデータ列抽出

この処理は，複数中心点の選択と同一のカーネルで行うことができる．各中心点に対応するデータメンバ列として抽出するデータは，中心点の所属セルとその隣接セル中にあるため，計算内容は複数中心点の選択とほぼ変わらない．

複数中心点の選択時，選択した中心点に対応する計算範囲内のデータは，候補から除去するデータと同一であるため，これを利用して各中心点に対応する計算範囲内のデータを抽出することができる（図 8 参照）．また，Inclusive Scan 後の残存フラグ列の末尾の値は，加えた対応データメンバ列の総数と一致するため，これを対応データ境界列に格納しておく．

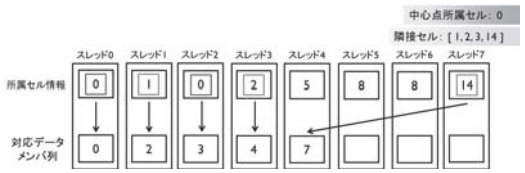

図 8: 各中心点に対応する計算範囲内のデータ列抽出．この例では，注目する中心点の所属セルのセル番号を 0，その隣接セルのセル番号を 1，2，3，14 とした場合を考える．これらのセルに含まれるデータは，複数中心点の選択時にその候補から除去するデータと同一であるため，これをそのとき注目している中心点に対応するデータメンバとして得る．その後，別の中心点に対して同じ処理を繰り返す場合は，新しく得られるデータ列を対応データメンバ列の末尾に加えていく．

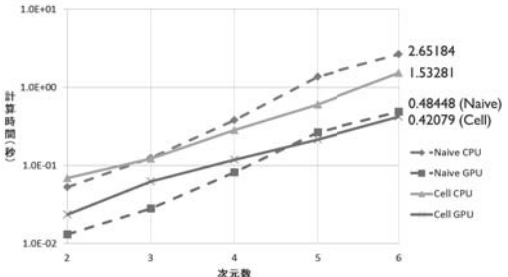

図 9: 実験結果（次元数を変化）

6. 評価実験

提案手法の有効性を評価するため，評価実験を行った．

6.1 実験環境

実験に用いた計算機環境について述べる．CPU は Intel ®Xeon(R) E5630（2.53 GHz x 4 core），OS は Ubuntu 12.04 LTS（64 bit），メモリは 4.0 GB である．また，GPU は NVIDIA GeForce GTX 460（1,350 MHz x 336 core, 767 MB Memory）を用いた．

データセットは，人工的に生成した 2〜6 次元のベクトル集合である．データ数は 100〜1,000,000 個まで変化させ，各ベクトルはサイズ $(\text{データ数} \times 10)^2$ の空間中で正規分布するように生成した．

データ間の距離にはユークリッド距離を用いた．また，T_1, T_2 の値は実験ごとに固定とし $T_2 = 0.7T_1$ とした．計算時間の測定については，Canopy クラスタリングの処理部分のみを対象とする．CPU–GPU 間のデータ転送時間もこれに含めるものとする．

6.2 実験結果

以下の 4 手法について，データ数を固定し次元数を変化させる場合と，次元数を固定しデータ数を変化させる場合の 2 通りについて，計算時間を比較する実験を行った．

- CPU 上でのナイーブな Canopy クラスタリング（Naive CPU）
- GPU 上でのナイーブな Canopy クラスタリング（Naive GPU）
- CPU 上でのセル構造を用いた Canopy クラスタリング（Cell CPU）
- GPU 上でのセル構造を用いた Canopy クラスタリング（Cell GPU）

6.2.1 実験 1：次元数の変化による計算時間の変化

データ数を 100,000 個に固定し，次元数を 2〜6 に変化させて，4 手法の計算時間を比較した．また，$T_1 = 100,000, T_2 = 70,000$ に固定した．結果のグラフを図 9 に示す．

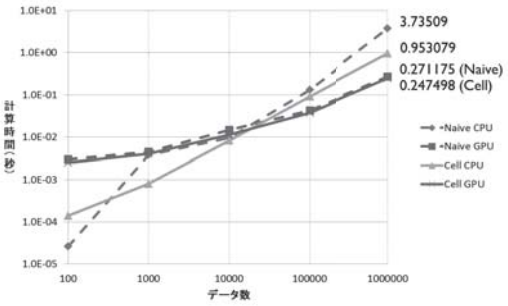

図 10: 実験結果（データ数を変化）

いずれの手法においても，次元数に比例して計算時間が増加している．CPU 同士で比較すると，次元数が大きい場合，セル構造を用いた手法が計算高速化に貢献していることが分かる．次元数が大きくなるほどセルの数が増加し，各々の中心点に対応する計算範囲内のデータが減少するためであると考えられる．GPU 上の手法では，両者に大きな差は見られなかった．

CPU と GPU で比較すると，GPU 上での計算によって大きく高速化できていることがわかる．

6.2.2 実験 2：データ数の変化による計算時間の変化

次元数を 2 に固定し，データ数を 100〜1,000,000 に変化させて，4 手法の計算時間を比較した．また，$T_1 = 0.5 * データ数$,$T_2 = 0.35 * データ数$ に固定した．結果のグラフを図 10 に示す．

データ数が小さいときに GPU 上の手法の計算時間が大きくなっているのは，CPU–GPU 間のデータ転送のオーバヘッドが大きいためである．また，Cell CPU もデータ数が小さい場合に計算時間が Naive CPU より大きくなっているが，これは事前のセル生成処理による計算回数削減の影響が小さいためである．次元数 2，データ数 1,000,000 のとき，Canopy GPU は Cell CPU に対して約 15 倍の性能を達成した．

7. まとめ

本研究では，大規模データを対象とするクラスタリングを高速化するため，Canopy クラスタリングの GPU を利用した並列計算を試みた．また，セル構造を用いてさらに並列度を高める手法を提案した．データ数，次元数を変化させ実験を行った結果，提案手法がナイーブな Canopy クラスタリングと比較し，次元数 2，データ数 1,000,000 であるとき，およそ 15 倍高速であることが分かった．

今後解決すべき課題として，GPU のメモリ容量に関する問題がある．GPU のメモリ容量は CPU に比べ小さく，データサイズによってはすべてのデータを GPU に転送できない場合がある．この問題の対策として，複数の GPU を用いる研究 [12] を参考に，CPU 側でデータセットを複数に分割し，分割されたサブデータセットごとに GPU へ転送し処理するという方法が考えられる．

[謝辞]

本研究は，JSPS 科研費（26280037）および「エクサスケール計算技術開拓による先端学際計算科学教育研究拠点の充実」事業の支援によるものである．

[文献]

[1] A. McCallum, K. Nigam, L. H. Ungar. Efficient Clustering of High Dimensional Data Sets with Application to Reference Matching. In *Proceedings of the sixth ACM SIGKDD international conference on Knowledge discovery and data mining*, 2000.

[2] Zhi-Gang Fan, Yadong Wu, Bo Wu. Maximum Normalized Spacing for Efficient Visual Clustering. In *Proceedings of 19th ACM International Conference on Information and Knowledge Management (CIKM 2010)*, 2010.

[3] Tuli De, Didier Fraix Burnet, Asis Kumar Chattopadhyay. Clustering large number of extragalactic spectra of galaxies and quasars through canopies. In *Communication in Statistics - Theory and Methods (2013) 000*, 2013.

[4] Owens, J.D. and Houston, M. and Luebke, D. and Green, S. and Stone, J.E. and Phillips, J.C. GPU Computing. *Proceedings of the IEEE*, Vol. 96, No. 5, pp. 879–899, 2008.

[5] Dar-Jen Chang, Mehmed Kantardzic, Ming Ouyang. Hierarchical clustering with CUDA/GPU. In *ISCA PDCCS*, 2009.

[6] S. A. Arul Shalom and Manoranjan Dash. Efficient Partitioning Based Hierarchical Agglomerative Clustering Using Graphics Accelerators With CUDA. *IJAIA*, Vol. 4, No. 2, 2013.

[7] S. A. Arul Shalom, Manoranjan Dash, Minh Tue. An Approach for Fast Hierarchical Agglomerative Clustering Using Graphics Processors with CUDA. In *PAKDD 2010, Part II*, pp. 35–42, 2010.

[8] Mario Zechner, Michael Granitzer. Accelerating K-Means on the Graphics Processor via CUDA. In *Intensive Applications and Sercices, 2009. INTENSIVE '09. First International Conference on*, 2009.

[9] BAI Hong-tao, HE Li-li, OUYANG Dan-tong, LI Zhanshan, LI He. K-Means on commodity GPUs with CUDA. In *2009 World Congress on Computer Science and Information Engineering*, 2009.

[10] F. Murtagh. A Survey of Recent Advances in Hierarchical Clustering Algorithms. In *The Computer Journal*, Vol. 26, 1983.

[11] Hubert Nguyen. *GPU Gems 3*, chapter 39. Addison-Wesley Professional, 2007.

[12] Mohiuddin K. Wasif, P. J. Narayanan. Scalable clustering using multiple GPUs. In *HIPC '11 Proceedings of the 2011 18th Intarnational Conference on High Performance Conputing*, 2011.

林 史尊 Fumitaka HAYASHI
平成 24 年 3 月筑波大学情報学群情報科学類卒業．平成 26 年 3 月筑波大学大学院システム情報工学研究科博士前期課程修了．修士（工学）．平成 26 年 4 月より株式会社アイ・エル・シー勤務．

小澤 佑介 Yusuke KOZAWA
筑波大学大学院システム情報工学研究科博士後期課程在学中．2013 筑波大学大学院システム情報工学研究科博士前期課程修了，修士（工学）．GPU を用いた並列データ処理技術の研究開発に従事．データベース学会学生会員．

天笠 俊之 Toshiyuki AMAGASA
筑波大学システム情報系准教授．データ工学，データベース，Web マイニング等の研究に従事．日本データベース学会，情報処理学会，ACM 各会員．電子情報通信学会，IEEE 各シニア会員．

北川 博之 Hiroyuki KITAGAWA
1978 年東京大学理学部物理学科卒業．1980 年同大学理学系研究科修士課程修了．日本電気（株）勤務の後，筑波大学講師，助教授を経て，現在，筑波大学システム情報系教授，同計算科学研究センター教授併任．理学博士（東京大学）．データベース，データマイニング，情報検索等の研究に従事．本会会長，情報処理学会フェロー，電子情報通信学会フェロー，ACM，IEEE-CS，日本ソフトウェア科学会，各会員．