Vol.2014-HPC-146 No.10
2014/10/2

ストリームクラスタリングアルゴリズムのGPUを用いた高速化

奥田 徹[1]　藤本 典幸[1]

概要：データ解析手法の 1 種であるクラスタリングはデータマイニングなどの分野に活用されている．本論文では，クラスタリングの代表的手法である k-means を拡張した，k-means++と k-means#を用いる Monteleoni らのストリームクラスタリングアルゴリズムを，GPU を用い高速化する方法を提案する．2.67GHz Intel Xeon X5550 CPU と NVIDIA GeForce GTX580 GPU を用いて評価した結果, 最大で 52.6 倍の速度向上率となった．

キーワード：クラスタリング，k-means，ストリームアルゴリズム，並列処理，GPGPU

Acceleration of a Streaming Clustering Algorithm for GPUs

Abstract: The clustering is a kind of data analysis techniques and is used in various fields, e.g. data mining. In this paper, we propose a GPU algorithm to accelerate Monteleoni et al's streaming clustering algorithm with k-means++ and k-means# which extend k-means, the representative technique of the clustering. Our experimental results on 2.67GHz Intel Xeon X5550 CPU and NVIDIA GeForce GTX580 GPU show that the proposed GPU algorithm runs a maximum of 52.6 times faster than the corresponding CPU algorithm.

Keywords: clustering, k-means, streaming algorithm, parallel processing, GPGPU

1. はじめに

近年，大規模データ収集技術が広まり，それに伴い大規模データの解析を求められるようになっている．しかし，計算機の主記憶の容量拡大よりもデータの増大が顕著であり，この傾向は広がる一方である．そのため，主記憶に入りきらないサイズのデータを高速に処理する技術が重要になっている．その 1 つとして，限られた容量の主記憶のみで大規模なデータを処理するストリームアルゴリズムがある．また，データ解析手法の 1 つとしてクラスタリング（クラスター分析）という手法がある．これは, 分類対象の集合に対し，共通の特徴を持つものを集めて複数の部分集合（クラスタ）に分類する手法であり，データマイニングなどに利用される．クラスタリングは大きく階層的手法と非階層的手法の 2 種類に分けられ，非階層的手法の代表的手法として k-means アルゴリズム [4] がある．

Monteleoni らは，ワンパスで k-means クラスタリング問題の目的関数を近似的に最適化するクラスタリングアルゴリズムを提案している [1]．そのアルゴリズムは，k-means アルゴリズムを改良した k-means++アルゴリズム [2] とそれを発展させた k-means#アルゴリズム [1] をサブルーチンとして用いる分割統治法であり，k 個のクラスタ中心を出力する．このクラスタ中心は最適解と比較したときに少なくとも $\frac{1}{2}$ の確率で $O(\log k)$ 倍以内の近似解となる．ストリームアルゴリズムは，大規模データを対象とするので，より高速化する必要がある．

そのため本論文では，GPU を用いて Monteleoni らのストリームクラスタリングアルゴリズムを高速化することを考えた．Monteleoni らのアルゴリズムでは，アルゴリズム全体に対する k-means++と k-means#の実行時間の割合が大きいため，この 2 つのアルゴリズムに着目した．また，この 2 つのアルゴリズムは大部分が共通であり，並列化についてもほぼ同様にして行える．NVIDIA GeForce GTX GPU580 で評価実験を行ったところ，2.67GHz Intel Xeon X5550 CPU に対し最大 52.6 倍の速度向上率となった．

[1] 大阪府立大学 大学院理学系研究科
Graduate School of Science, Osaka Prefecture University

本論文の構成は以下の通りである．第 2 章で k-means，k-means++，k-means#，Monteleoni らのストリームクラスタリングアルゴリズムの概要について説明する．第 3 章で提案手法の詳細を示す．第 4 章で評価実験について述べる．第 5 章でまとめと今後の課題について述べる．

2. 準備

2.1 k-means クラスタリング問題

n 個の d 次元ベクトル $\boldsymbol{x}_1, \boldsymbol{x}_2, \cdots, \boldsymbol{x}_n$, 各ベクトルの重み $w(\boldsymbol{x}_i)(i \in \{1, 2, \cdots, n\})$, クラスタの数 k が与えられたとき，目的関数 $\phi_C = \sum_{i=1}^{n} w(\boldsymbol{x}_i) D(\boldsymbol{x}_i, C)^2$ を最小にする k 個のクラスタ中心の集合 $C = \{\boldsymbol{c}_1, \boldsymbol{c}_2, \cdots, \boldsymbol{c}_k\}$ を求める問題を k-means クラスタリング問題という．ここで $D(\boldsymbol{x}, C)$ は C の要素の中で $\boldsymbol{x}$ に最も近いものと $\boldsymbol{x}$ とのユークリッド距離であり，クラスタ中心とは，そのクラスタに分類されているベクトルの平均である．$\boldsymbol{x}_i$ に最も近いクラスタ中心が $\boldsymbol{c}_j$ であるとき，$\boldsymbol{x}_i$ はクラスタ j に分類されていると考える．

2.2 k-means アルゴリズム

k-means アルゴリズム（Lloyd のアルゴリズム）は，k-means クラスタリング問題を解くアルゴリズムとしてよく知られている．疑似コードを以下に示す．

入力：n 個の d 次元ベクトル $\boldsymbol{x}_1, \boldsymbol{x}_2, \cdots, \boldsymbol{x}_n$，各 $\boldsymbol{x}_i$ から重みへの関数 w，クラスタ数 k

出力：k 個のクラスタ中心 $\boldsymbol{c}_1, \boldsymbol{c}_2, \cdots, \boldsymbol{c}_k$

(1) $\boldsymbol{x}_1, \boldsymbol{x}_2, \cdots, \boldsymbol{x}_n$ からランダムに k 個選び，$\boldsymbol{c}_1, \boldsymbol{c}_2, \cdots, \boldsymbol{c}_k$ とする．

(2) 以下の（3）（4）を $\boldsymbol{c}_1, \boldsymbol{c}_2, \cdots, \boldsymbol{c}_k$ が変化しなくなるまで繰り返す．

(3) 各ベクトルと各クラスタ中心との距離を求め，各ベクトルを最も近いクラスタ中心に所属させる．

(4) 同じクラスタ中心に所属しているベクトルの平均を求め，各クラスタ中心を求めた平均ベクトルで置き換える．

k-means アルゴリズムでは大域的最適解は必ずしも求まらないが，初期値に対応した局所的最適解が求まることが知られている [3]．疑似コードのステップ (2) における反復回数を r とすると k-means アルゴリズムの時間計算量は $O(rnkd)$ である．n=11，k=3，d=2 の k-means アルゴリズムの動作例を**図 1** に示す．

2.3 k-means++アルゴリズム

k-means++は k-means アルゴリズムの初期値依存を改良したアルゴリズムである．疑似コードを以下に示す．

入力：n 個の d 次元ベクトル $\boldsymbol{x}_1, \boldsymbol{x}_2, \cdots, \boldsymbol{x}_n$，各 $\boldsymbol{x}_i$ から重みへの関数 w，クラスタ数 k

出力：k 個のクラスタ中心 $\boldsymbol{c}_1, \boldsymbol{c}_2, \cdots, \boldsymbol{c}_k$

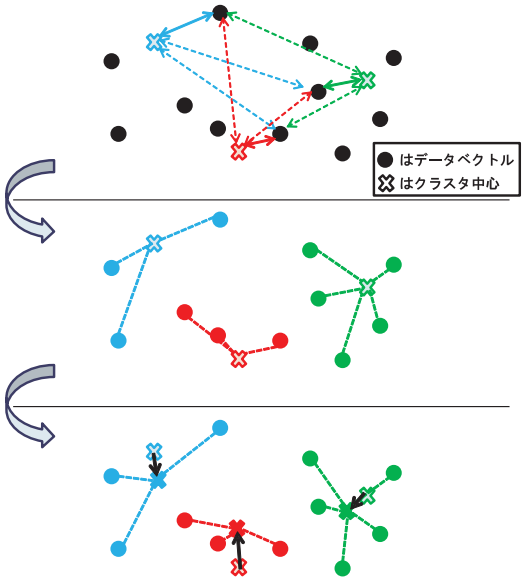

図 1 k-means アルゴリズムの動作例

(1) $\boldsymbol{x}_1, \boldsymbol{x}_2, \cdots, \boldsymbol{x}_n$ からランダムにひとつ選び，クラスタ中心 $\boldsymbol{c}_1$ とし，$C = \{\boldsymbol{c}_1\}$ とする．

(2) 以下を $j = 2 \sim k$ まで繰り返す．

(3) 確率 $\frac{w(\boldsymbol{x}')D(\boldsymbol{x}', C)^2}{\sum_{i=1}^{n} w(\boldsymbol{x}_i)D(\boldsymbol{x}_i, C)^2}$ で新たなクラスタ中心 $\boldsymbol{c}_j = \boldsymbol{x}' \in \{\boldsymbol{x}_1, \boldsymbol{x}_2, \cdots, \boldsymbol{x}_n\}$ を選び，$C = C \cup \{\boldsymbol{c}_j\}$ とする．

k-means++は平均的に $\Theta(\log k)$-近似アルゴリズムであり，時間計算量は $O(nkd)$ である．

2.4 k-means#アルゴリズム

k-means#は k-means++を発展させたアルゴリズムである．疑似コードを以下に示す．

入力：n 個の d 次元ベクトル $\boldsymbol{x}_1, \boldsymbol{x}_2, \cdots, \boldsymbol{x}_n$，各 $\boldsymbol{x}_i$ から重みへの関数 w，クラスタ数 $3k \log k$

出力：$3k \log k$ 個のクラスタ中心 $\boldsymbol{c}_1, \boldsymbol{c}_2, \cdots, \boldsymbol{c}_{3k \log k}$

(1) $\boldsymbol{x}_1, \boldsymbol{x}_2, \cdots, \boldsymbol{x}_n$ からランダムに $3 \log k$ 個選び，クラスタ中心 $\boldsymbol{c}_1, \cdots, \boldsymbol{c}_{3 \log k}$ とし，$C = \{\boldsymbol{c}_1, \boldsymbol{c}_2, \cdots, \boldsymbol{c}_{3 \log k}\}$ とする．

(2) 以下を $j = 2 \sim k$ まで繰り返す．

(3) 確率 $\frac{w(\boldsymbol{x}')D(\boldsymbol{x}', C)^2}{\sum_{i=1}^{n} w(\boldsymbol{x}_i)D(\boldsymbol{x}_i, C)^2}$ で新たなクラスタ中心 $\boldsymbol{c}_i = \boldsymbol{x}' \in \{\boldsymbol{x}_1, \boldsymbol{x}_2, \cdots, \boldsymbol{x}_n\}$ を選ぶことを $i = (j-1) \times 3 \log k + 1 \sim j \times 3 \log k$ まで繰り返し，$C = C \cup \{\boldsymbol{c}_{(j-1)\times 3 \log k + 1}, \cdots, \boldsymbol{c}_{j \times 3 \log k}\}$ とする．

k-means#は少なくとも $\frac{1}{4}$ の確率で $O(1)$-近似アルゴリズムであり，時間計算量は $O(nkd \log k)$ である．k-means や k-means++とは異なり，このアルゴリズムは $3k \log k$ 個のクラスタ中心を出力する．

2.5 Monteleoni らのストリームクラスタリングアルゴリズム

Monteleoni らのアルゴリズムは分割統治法に基づいて k-means クラスタリング問題を解くストリームアルゴリズ

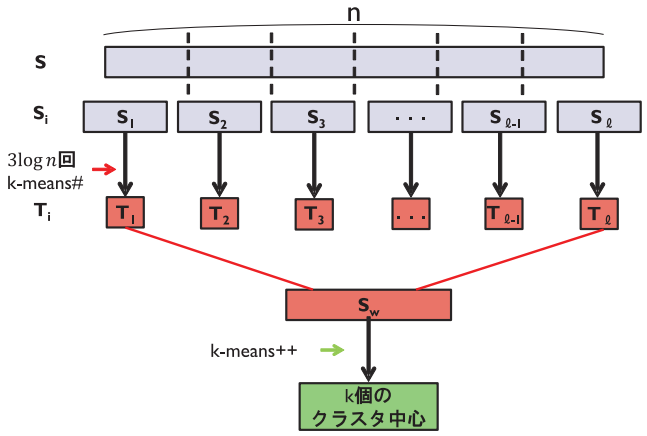

図 2 Monteleoni らのストリームクラスタリングアルゴリズム

ムである．疑似コードを以下に示す．

入力：n 個の d 次元ベクトルの集合 $S = \{\boldsymbol{x}_1, \boldsymbol{x}_2, \cdots, \boldsymbol{x}_n\}$，$S$ の要素の重み，クラスタ数 k

出力：k 個のクラスタ中心 $\boldsymbol{c}_1, \boldsymbol{c}_2, \cdots, \boldsymbol{c}_k$

(1) 集合 S を S_i のサイズが $\sqrt{nk}$ となるように，$S_1, S_2, ..., S_r$ に分割する．

(2) 各分割 S_i，S_i の要素の重み，クラスタ数 $3k \log k$ に対し，$3 \log n$ 回独立に k-means#を実行し，その中で目的関数が最小のクラスタ中心の集合を $T_i = \{\boldsymbol{t}_{i1}, \boldsymbol{t}_{i2}, \cdots, \boldsymbol{t}_{i3k \log k}\}$ とする．ここで $\boldsymbol{t}_{ij}$ に属するベクトルの集合を S_{ij} とする．

(3) $S_w \leftarrow T_1 \cup T_2 \cup \cdots \cup T_r$，$\boldsymbol{t}_{ij}$ の重みを $w(\boldsymbol{t}_{ij}) = |S_{ij}|$ とし，クラスタ中心を合併する．

(4) S_w，重み $w(\boldsymbol{t}_{ij})$，クラスタ数 k に対し k-means++を実行し，その結果を $\boldsymbol{c}_1, \boldsymbol{c}_2, \cdots, \boldsymbol{c}_k$ とする．

各分割 S_i のサイズは $O(\sqrt{nk})$ であるため，要求メモリが $O(d\sqrt{nk} \log k)$ で済む．これは全データの要求メモリ $O(nd)$ に比べ少ない．また，このアルゴリズムは少なくとも $\frac{1}{2}$ の確率で $O(\log k)$-近似アルゴリズムであり，時間計算量は $O(nkd \log n \log k)$ である．**図 2** はこのアルゴリズムを図示したものである．

3. 提案手法

3.1 提案手法の概要

Monteleoni らのストリームクラスタリングアルゴリズムの並列化するにあたり，最も時間計算量の多い k-means#に関する部分を中心に，2.5 節の疑似コードで 3 つの部分に着目した．

1 つ目は，k-means++と k-means#はともに距離計算 $D(\boldsymbol{x}', C)^2$ を全てのベクトル $\boldsymbol{x}' \in \{\boldsymbol{x}_1, \boldsymbol{x}_2 \cdots, \boldsymbol{x}_n\}$ に対し k 回行う点である．距離計算の時間計算量は 1 回あたり，k-means++で $O(nd)$，k-means#で $O(nd \log k)$ であり，アルゴリズム内で処理に最も時間がかかる部分である．

2 つ目は，k-means++と k-means#はともに確率 $\frac{w(\boldsymbol{x}')D(\boldsymbol{x}', C)^2}{\sum_{i=1}^{n} w(\boldsymbol{x}_i)D(\boldsymbol{x}_i, C)^2}$ で新たなクラスタ中心を選ぶ点である．この時間計算量は 1 回あたり，k-means++で $O(n)$，k-means#で $O(n \log k)$ であり距離計算より少ないが，距離計算の結果が必要となるため CPU で行うと，CPU-GPU 間の通信により時間がかかってしまう．また，並列探索法が知られているので活用する．

3 つ目は，k-means#の出力するクラスタ中心の集合 T_i の要素の重みづけ $w(\boldsymbol{t}_{ij}) = |S_{ij}|$ である．この時間計算量は $O(n)$ であるが，各ベクトルの所属クラスタ中心の情報は距離計算を行う際に更新されるため，こちらも CPU で行うと CPU-GPU 間の通信に時間がかかるので GPU で並列化する．

以降の 3.2 節，3.3 節，3.4 節で着目した 3 つの部分のそれぞれの並列化の詳細について述べる．

3.2 距離計算の並列化

距離計算は，d 次元ベクトル $\boldsymbol{x}_i = (x_{i1}, x_{i2}, \cdots, x_{id})$ とクラスタ中心 $\boldsymbol{c}_j = (c_{j1}, c_{j2}, \cdots, c_{jd})$ に対し，$D(\boldsymbol{x}_i, \boldsymbol{c}_j)^2 = (x_{i1} - c_{j1})^2 + (x_{i2} - c_{j2})^2 + \cdots + (x_{id} - c_{jd})^2$ を求める計算である．これを n 個の d 次元ベクトル $\boldsymbol{x}_1, \boldsymbol{x}_2, \cdots, \boldsymbol{x}_n$ がそれぞれ，k-means++では 1 つの新たなクラスタ中心 $\boldsymbol{c}_j$ と，k-means#では $3 \log k$ 個の新たなクラスタ中心 $\boldsymbol{c}_{(j-1)\times 3\log k+1}, \boldsymbol{c}_{(j-1)\times 3\log k+2}, \cdots, \boldsymbol{c}_{j\times 3\log k}$ と行う必要がある．まず n 個の d 次元ベクトルに対し，各ベクトルをそれぞれ 1 スレッドに割り当てることで n-並列化する．そして各スレッドで，割り当てられたベクトル $\boldsymbol{x}_i$ と新たなクラスタ中心 $\boldsymbol{c}_j$ または $\boldsymbol{c}_{(j-1)\times 3\log k+1} \sim \boldsymbol{c}_{j\times 3\log k}$ との距離計算を行う．現在の所属クラスタ中心を $\boldsymbol{c}'$ とし，割り当てられたベクトルと現在の所属クラスタ中心 $\boldsymbol{c}'$ の距離 $D(\boldsymbol{x}_i, \boldsymbol{c}')^2$ と，先ほど求めた $D(\boldsymbol{x}_i, \boldsymbol{c}_j)^2$ または $D(\boldsymbol{x}_i, \boldsymbol{c}_{(j-1)\times 3\log k+1})^2 \sim D(\boldsymbol{x}_i, \boldsymbol{c}_{j\times 3\log k})^2$ を大小比較する．そして後者の方が小さければ現在の所属クラスタ中心 $\boldsymbol{c}'$ を新たなクラスタ中心に変更する．この方法では各スレッドで k-means++なら $O(d)$，k-means#なら $O(d \log k)$ の計算を行っている．また，コアレスアクセスにするために，n 個の d 次元ベクトルを n 行 $\times d$ 列の行列とするのではなく，d 行 $\times n$ 列の転置行列としてグローバルメモリに記憶する．距離計算は k-means++，k-means#ともに k 回行われるので，この並列実行も k 回行われる．評価実験ではブロックサイズを $\frac{n}{1024}$，スレッドサイズを 1024 とした．

3.3 ルーレット選択の並列化

ベクトル $\boldsymbol{x}_j$ が新たなクラスタ中心に選ばれる確率を p_j とすると，$p_j = \frac{w(\boldsymbol{x}_j)D(\boldsymbol{x}_j, C)^2}{\sum_{i=1}^{n} w(\boldsymbol{x}_i)D(\boldsymbol{x}_i, C)^2}$ となる．これより，$0 \leq p \leq 1$ となる p を一様乱数で生成し，$\sum_{j=1}^{h-1} p_j < p \leq \sum_{j=1}^{h} p_j$ となる $h (1 \leq h \leq n)$ を見つけることで，新たなクラスタ中心となるベクトル $\boldsymbol{x}_h$ を求める．この方法はルーレット選択 (Roulette Wheel Selection) とも呼ばれる．

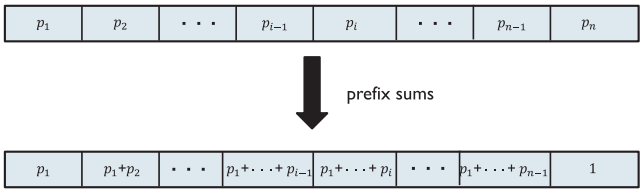

図 3 ルーレット選択における prefix sums の利用

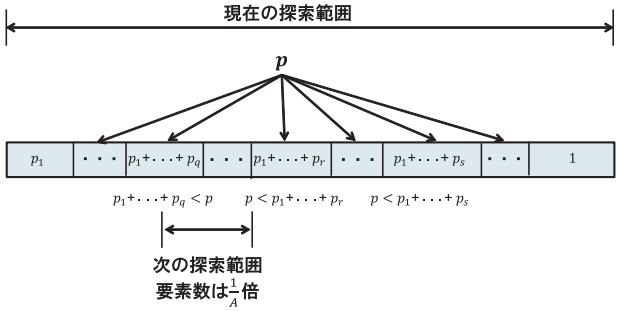

図 4 ルーレット選択における A 分探索の利用

ルーレット選択の GPU 実装として，prefix sums を用いた後に A 分探索を用いる手法が提案されている [10]．我々の実装はこの手法に基づいている．

まず，全ての確率 $p_i(i \in \{1,2,\cdots,n\})$ に対し，**図 3** のように prefix sums を実行する．prefix sums とは $\langle a_1,a_2,\cdots,a_n\rangle$ を入力として与えると，$\langle a_1,a_1+a_2,\cdots,a_1+a_2+\cdots+a_n\rangle$ を出力として返す計算である．prefix sums については高速な CUDA 実装 [8][9] が知られているので，我々の実装ではこれを用いた．

この prefix sums の結果を探索範囲とするが，探索範囲は n 要素あり広いため，探索範囲を A 個の均等な大きさの範囲に分割する．そして $A-1$ 個のスレッドを用いて，j 番目 $(j \in \{1,2,\cdots,A-1\})$ のスレッドが $j+1$ 個目の分割の最初の要素をサンプリングする．サンプリングした要素と p を大小比較することで大小関係が入れ替わる分割を見つけ出し，p が含まれる分割を次の探索範囲とする．次の探索範囲にも同様に繰り返し行い，探索範囲を狭めていく．**図 4** は，$\sum_{j=1}^{q} p_j < p$ かつ $p < \sum_{j=1}^{r} p_j$ であるため，次の探索範囲が $\sum_{j=1}^{q} p_j \sim \sum_{j=1}^{r-1} p_j$ となり，この範囲に対して同様に繰り返していく例である．この繰り返し 1 回につき探索範囲が $\frac{1}{A}$ 倍となるので，高々 $O(\log_A n)$ 回の繰り返しで $\boldsymbol{x}_h$ が見つかる．k-means++は 1 度に 1 個の新たなクラスタ中心を，k-means#は 1 度に $3\log k$ 個の新たなクラスタ中心を選択するので，この新たなクラスタ中心数をブロックサイズに，スレッドサイズを $A-1$ として並列化を行った．また，評価実験では $A=1024$ とした．

3.4 重み計算の並列化

各ベクトルの所属クラスタ中心は距離計算時に判明しているため，これを利用し各クラスタ中心に属すベクトル数を求める．そのため，Sanders らのヒストグラムの計算 [7] の考えを利用する．n 個のベクトルをそれぞれ 1 スレッドに割り当て，atomicAdd 関数 [6] を用いることでクラスタ中心の頻度数，つまり重みを求める．しかし，n 個のスレッドで同時に atomicAdd 関数を用いると，全てのスレッドが一握りのメモリへアクセスを試みるためパフォーマンスが著しく低下してしまう．これを改善するために，atomicAdd 関数を 2 段階で用いることにする．まず，各ブロックのスレッド数を 1024 とする．1 段階目では，各ブロックでクラスタ数 $3k\log k$ をサイズとする共有メモリを用意し，1024 個のスレッドで同時に atomicAdd 関数を用いて，共有メモリに各クラスタ中心 $\boldsymbol{c}_i(i \in \{1,2,\cdots,3k\log k\})$ の頻度数を記憶させる．これにより，各ブロックでベクトル 1024 個分の頻度数が求まる．2 段階目では，各ブロックの共有メモリに記憶された頻度数に対し，$3k\log k$ 個のスレッドで同時に atomicAdd 関数を用いて，グローバルメモリに各ブロックで求まった頻度数をまとめる．この結果，パフォーマンスの低下を回避しつつ，重みであるクラスタ中心の頻度数を求めることができる．

4. 評価実験

4.1 実験環境

データベクトル数 n，クラスタ数 k，次元数 d を変化させ，Monteleoni らのストリームクラスタリングアルゴリズムの速度向上率を求めた．CPU プログラムは 2.67GHz Intel Xeon X5550，Linux 2.6.27.29 (Fedora10 x86 64) の環境で実行し，GPU プログラムは 2.67GHz Intel Core i7-920，NVIDIA GeForce GTX 580, 64bit Windows 7 Professional の環境で実行した．CPU プログラムのコンパイルには gcc 4.3.2 を，GPU プログラムのコンパイルには Visual Studio 2008 Professional と CUDA 4.2 を用いた．n の大きさは 200 万〜1 億 2800 万の間における 7 種，k と d の大きさは 2〜64 の間における 6 種を対象とした合計 252 通りで評価実験を行った．64bit 版 Mersenne Twister[5] を用い，データベクトルの float 型の各要素を [0,1) の範囲の一様乱数で生成した．また各ベクトルの重みは 1 としている．

4.2 CPU プログラムと GPU プログラムの比較

252 通りのデータで評価実験を行っているが，入力データが巨大なため最初に一括して主記憶上に生成できない．そのため S_i 毎に入力データを生成し，処理が終わるとそのデータを破棄している．また，実行時間は S_1 のデータの生成開始から k 個のクラスタ中心を出力するまで測定しているため，CPU-GPU 間のデータの転送時間も含んでいる．

表 1 は $d=8$ の場合の提案手法の速度向上率である．n，k が小さいと各分割 S_i のサイズ $\sqrt{nk}$ が小さくなり，並列性が少なくなることから十分な速度向上を得られない．また，例えば $n=3200$ 万,$k=8$ と $n=1600$ 万,$k=16$ の速度向上率に注目すると，S_i のサイズが同じでも k が大きい

表 1　$d = 8$ の場合の提案手法の速度向上率

k＼n	200 万	400 万	800 万	1600 万	3200 万	6400 万	12800 万
2	0.384	0.555	0.764	1.138	1.529	2.019	2.773
4	1.601	2.233	3.159	4.377	5.977	7.451	9.353
8	4.263	6.147	8.345	11.715	12.829	15.803	18.339
16	9.168	12.607	17.85	18.114	22.181	24.042	32.931
32	18.699	25.902	36.04	31.343	43.439	44.66	50.262
64	28.455	39.762	33.233	45.762	46.931	52.683	51.281

表 2　$n = 6400$ 万の場合の提案手法の速度向上率

d＼k	2	4	8	16	32	64
2	1.232	4.836	12.229	19.48	38.128	46.992
4	1.475	5.744	13.245	21.323	41.28	49.703
8	2.019	7.451	15.803	24.042	44.66	52.683
16	2.937	9.908	19.237	25.381	43.892	48.251
32	4.439	11.845	19.862	26.418	43.891	47.628
64	6.445	12.55	23.521	27.78	43.931	47.47

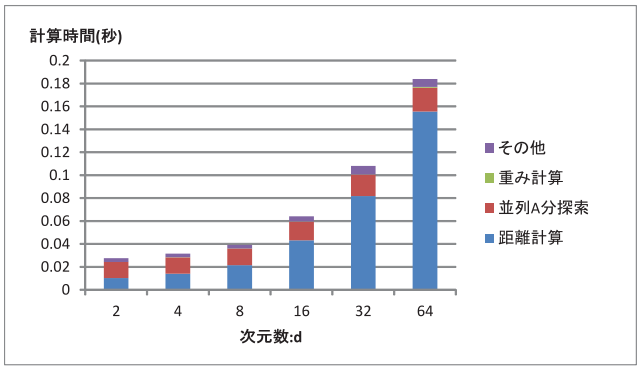

図 5　$|S_i| = 64000$，$k = 64$ の場合の k-means#1 回の計算時間の内訳

場合の方が速度向上率が高くなっている．これは k が大きいほど，並列 A 分探索での並列性が高くなるためである。

表 2 は $n = 6400$ 万の場合の提案手法の速度向上率である．クラスタ数が多くなると次元数の増加による速度向上がほぼみられないが，これは次元数に対しての並列化を行っていないことが原因である．次元数が最も影響するのは，距離計算であるため並列化の改良が必要である．我々の評価実験では $n = 6400$ 万，$k = 64$，$d = 8$ の時，最大の速度向上率 52.68 倍となった．

4.3　実行時間の内訳

Monteleoni らのアルゴリズムでは，$3 \log n \times \sqrt{\frac{n}{k}}$ 回実行される k-means#の実行時間が全体実行時間の大部分を占めている．そこで，k-means#1 回の計算時間全体に占める距離計算，並列 A 分探索，重み計算，その他の部分（グローバルメモリの確保・解放および GPU-GPU 間の転送など）の計算時間の割合を調べた．重み計算は k, d の値に関係なくほぼ一定であるが，並列 A 分探索は n, k に影響され，距離計算は n, k, d の全てに大きく影響される．例えば，4.2 節の評価実験で最大の速度向上率となった $n = 6400$ 万，$k = 64$

```
1 for (int i = 0; i < n; i++) {
2   for (int j = 0; j < k; j++) {
3     for(int h = 0; h < d; h++) {
4       diff = X[i][h] - C[j][h];
5       D[i][j] += diff * diff;
6 }}}
```

図 6　距離計算の疑似コード

```
1 for (int i = 0; i < n; i++) {
2   for (int j = 0; j < k; j++) {
3     for(int h = 0; h < d; h++) {
4       AB[i][j] += A[i][h] * B[h][j];
5 }}}
```

図 7　行列積の疑似コード

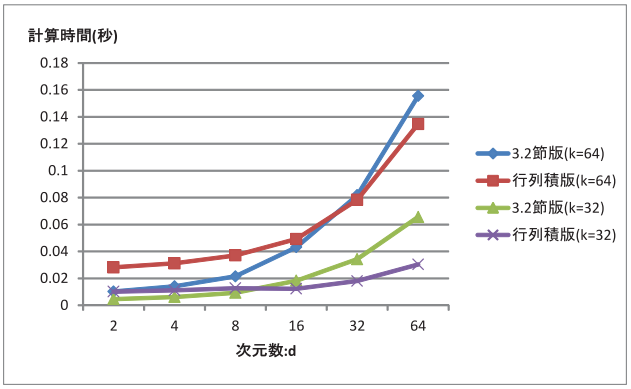

図 8　$n = 64000$ の場合の行列積版距離計算の性能

の場合の次元数 d の変化による k-means#の計算時間の変化を調べたものが**図 5** である．このとき k-means#が実行される S_i のサイズ $\sqrt{nk}$ は 64000 である．d が大きくなるにつれ距離計算の比重が重くなっており，$d = 64$ の時には並列 A 分探索の約 7.5 倍の計算時間であり，k-means#全体の計算時間の約 85%を占める．これより，さらなる高速化のためには距離計算で次元数に関しても並列化を行う必要がある．

4.4　k-means# の距離計算の行列積版の試作

4.3 節から k-means#の距離計算の改良の必要性がある．この改良として行列積に注目した．**図 6** は n 個の d 次元ベクトルが配列 X，k 個の d 次元ベクトルが配列 C に与えられたときの距離計算の疑似コードである．配列 D はベクトル $\boldsymbol{x}_i$ と $\boldsymbol{c}_j$ の距離の 2 乗を $D[i][j]$ に格納している．**図 7** は行列 A と B の積を求める疑似コードである．これらの疑似コードから行列積と距離計算の計算パターンは類似していることが分かる．行列積については，高速な CUDA 実装がいくつか知られているので，その中で比較的単純な実装 [7] を距離計算に適用することを試みた．

図 8 は n=64000，k=32，64 の場合の k-means#1 回の，

3.2 節の距離計算の並列化と行列積版の距離計算の並列化の計算速度の比較である．行列積版の距離計算の並列化は次元数が大きいほど高速化しているが，逆に次元数が小さいほど遅くなっており，n，k，d の値によっては 3.2 節の距離計算の並列化より遅くなることも多い．このため試作した行列積版距離計算の性能は不十分であり，さらなる改良が必要である．なお，この節の内容は最終的な提案手法に含まれておらず，4.2 節および表 1，表 2 に示した速度向上率とは関係がない．

5. まとめと今後の課題

Monteleoni らのストリームクラスタリングアルゴリズムを GPU を用いて並列化して最大 52.68 倍の速度向上率を得た．

さらに速度向上率を上げるためには 4.3 節，4.4 節から距離計算のさらなる並列化が課題となる．また，Monteleoni らは分割統治アルゴリズムで S_i のサイズを利用可能なメモリサイズ $M = n^{\alpha}(0 < \alpha < 1)$ とし，複数階層化するアルゴリズムも提案しているので，このアルゴリズムの並列化も課題である．

参考文献

[1] Ailon, N., Jaiswal, R., and Monteleoni, C.: *Streaming k-means Approximation*, Procs. of Neural Information Processing Systems (NIPS), pp.10–18 (2009).
[2] Arthur, D. and Vassilvitskii, S.: *k-means++: the Advantages of Careful Seeding*, Procs. of ACM-SIAM Symposium on Discrete Algorithms (SODA), pp.1027–1035 (2007).
[3] 加藤直樹，羽室行信，矢田勝俊: データマイニングとその応用, 朝倉書店 (2008).
[4] Lloyd, S. P.: *Least Squares Quantization in Pcm*, IEEE Transactions on Information Theory, Vol.28, No.2, pp.129–136 (1982).
[5] 松本眞: Mersennne Twister Home Page, 入手先 ⟨http://www.math.sci.hiroshima-u.ac.jp/˜m-mat/MT/mt64.html⟩ (2014.09.02).
[6] NVIDIA Corp.: CUDA Programming Guide, 入手先 ⟨http://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html⟩ (2014.09.02).
[7] Sanders, J. and Kandrot, E.: *CUDA by Example: an Introduction to General-Purpose GPU Programming*, Addison-Wesley Professional (2010).
[8] Sengupta, S., Harris, M., and Garland, M.: *Efficient Parallel Scan Algorithms for GPUs*, NVIDIA Technical Report NVR-2008-003 (2008).
[9] Sengupta, S., Harris, M., Zhang, Y., and Owens, J. D. : *Scan Primitives for GPU Computing*, Proc. of Graphics Hardware, pp.97–106 (2007).
[10] Uchida, A., Ito, Y., and Nakano, K.: *An Efficient GPU Implementation of Ant Colony Optimization for the Traveling Salesman Problem*, Proc. of International Conference on Networking and Computing (ICNC), pp94–102 (2012).