グラフィックスプロセッサによる レコメンデーションシステムのアルゴリズム

Algorithm for Recommendation System on Graphics Processor

加　藤　公　一

要　約　レコメンデーションシステムとは，過去の顧客の購買履歴を基に，顧客が買ってくれそうな商品を予測して提示するシステムである．その計算には統計的手法が用いられ，商品数や顧客数が多くなると計算量が莫大になり，計算時間がかかることが従来問題とされてきた．その計算をグラフィックスプロセッサ（GPU）上で効果的に実行するアルゴリズムを提案する．またそれを実装することで，最新 CPU のシングルコア実装と比べて数十倍から数百倍程度の計算速度が達成できることを示す．

Abstract　The recommendation system is a software system which predicts what customers likely to buy according to the history of what they bought. Its computation includes the statistical analysis which needs huge computational efforts when the items and customers increase in number. Consequently, the time complexity has been considered as a major problem. We propose an effective algorithm for the recommendation system which works on a GPU (graphic processing unit). We also show that its implementation works dozens of times to a hundred times faster than a single-core implementation on a CPU.

1. は　じ　め　に

レコメンデーションシステムとは，顧客の購買履歴（あるいは商品評価の履歴）を基に，顧客の嗜好性を予測して提示するシステムである．有名な例としては，ネット通販会社のアマゾンにおいて，商品を選択すると「この商品を買った人はこんな商品も買っています」というタイトルでおすすめ商品のリストが表示される仕組みがある．特に近年，ショッピングサイトのロングテール化が進み，レコメンデーションの重要性が増している．つまり，あまり買う人がいないような物も含めて莫大な種類の商品がウェブで購入できる現状では，顧客が自分の欲しいものにたどり着くための最初のフィルタとしてレコメンデーションシステムの重要性が高まっている．

レコメンデーションシステムでは，「どの商品を薦めるか」の計算が主要部分であり，この計算は統計学的手法に基づく購買行動の類似性の検索にあたる．この計算において，商品数・顧客数が大きくなると計算量が膨大になり非常に時間がかかるというのが従来の問題であった．実用上の問題として，直近の購買行動をシステムに反映するのに数日かかるというケースも知られている．より新鮮なデータを使ったより効果的なマーケティングを実現するという意味で，この計算を高速化するメリットは大きい．

本稿ではグラフィックプロセッサ（GPU）を使ったレコメンデーションシステムの計算高速化を提案する．また，それを実装することにより最新の CPU のシングルコア上の実装と比べて数十倍～数百倍の計算速度が達成可能であることを示す．これにより，例えば従来一日か

かっていた計算が数分〜数十分で終わるようになり，データ更新への追随性の高いシステムの構築が可能になる．

GPU は，本来画面描画などのグラフィックス処理のために専用に作られたプロセッサであるが，近年それを本来の目的以外の一般的計算に利用しようという研究が多くなされている．GPU 上でアプリケーションを実装することの大きなメリットはその計算速度である．最新の GPU には，CPU でいう「コア」にあたるものが 200 個以上入っており，またスレッド生成のコストが非常に小さいため，並列性の高いアプリケーションでは CPU の数百倍程度の大幅なパフォーマンス向上が期待できる．

本稿で説明するレコメンデーションのアルゴリズムは，特異値分解と k- 最近傍問題に分解される．特異値分解のアルゴリズムについては，筆者らが詳細を論文[4]にまとめてある．また，k- 最近傍問題についても，筆者らが論文[3]に詳細をまとめている．本稿は，これらの論文をまとめた解説という位置づけである．

本稿の構成は以下のとおりである．まず 2 章ではレコメンデーションシステムの動作原理を解説し，知られている CPU 上の計算アルゴリズムを説明する．ここで説明するアルゴリズムは，後に説明する GPU 上のアルゴリズムの基礎になるものである．3 章では，GPU 上で動くアプリケーションの開発基盤を説明する．4 章では，GPU 上で高速に動くアルゴリズムを提案する．次に 5 章で計算機による実験結果を示し，CPU と比べて劇的に高速化することを示す．最後に 6 章で結論を述べる．

2. レコメンデーションシステムの動作原理

レコメンデーションシステムは端的には購買パターンや評価パターンの類似性を検索するシステムだということができる．そのようなパターンの類似性はベクトルの距離として考えることができる．

例えば，A 〜 E の 5 人が W 〜 Z の四つの映画を観て，そのスコアを 1 〜 5 の点数で図 1 のようにつけたとする．ここで，空欄は人がまだ映画を観ていないことを意味する．この時，映画 X を気に入った人にどの映画を薦めるのが適当であろうか．この場合，空欄に 0 を埋めることで，各列がベクトルだとみなす．そのベクトルを左から v_W，v_X，v_Y，v_Z とすると，v_W，v_Y，v_Z の中で v_X から一番近いものが求めるべき映画に相当する．この場合，

人＼映画	W	X	Y	Z
A	5	4		
B	4	3		3
C				3
D			4	
E		1	5	
	v_W	v_X	v_Y	v_Z

図 1　映画のスコアリングによる例

$$|v_W - v_X| = 1.73\cdots < |v_Z - v_X| = 5.09\cdots < |v_Y - v_X| = 7.54\cdots$$

であるので，映画 X を気に入った人には映画 W を薦めるのが正解となる．実際には，映画 W，X，Y，Z のすべてについて，パターンの近さのランキングを計算してあらかじめデータベースに保存しておき，それを検索時に示すのが通常である．

以上がレコメンデーションシステムの基本的な動作原理であるが，実際には顧客数や商品数が大きなものになるのでさらなる計算上の工夫が必要である．例えば，顧客数と商品数がともに百万とした場合，百万次元ベクトルの距離の計算を百万個のベクトルすべての組み合わせに対して行うのは，実用上計算量が大きすぎる．そこで特異値分解（SVD，Singular Value Decomposition）という一種のデータ圧縮を使い，ベクトルの近い/遠いの関係を保存しながら次元を減らす．

2.1　特異値分解

特異値分解とは図 2 に示すように，与えられた行列をサイズの小さい行列の積で近似する手法である．この場合，行列 A の「どの行とどの行が近いか」という関係が U^T の「どの行とどの行が近いか」という関係に反映され，行列 A の「どの列とどの列が近いか」という関係が V の「どの列とどの列が近いか」という関係に反映される．ここで圧縮後の次元 d は，隠れている説明変数の数だと考えることができる．つまり，起こっている複雑な現象は，少ない説明変数で表現できるという思想に基づくものである．通常 d としては数百位の値が使われることが多い，d があまり小さすぎると「どの行（列）とどの行（列）が近いか」の関係を保存できなくなるので，経験的・実験的に適当な d の値が採用される．

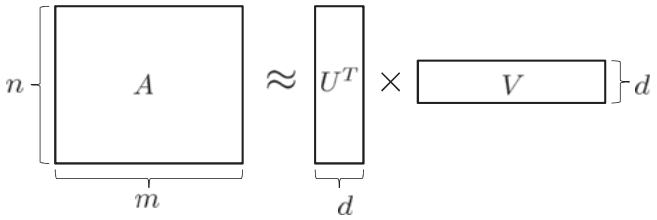

図 2　特異値分解の概念図

この SVD の計算アルゴリズムについては，Webb（Simon Funk というペンネームで知られている）が考えた手法[9]が有名である．Webb のアルゴリズムは Netflix 社という米国の DVD レンタル会社が主催している DVD 嗜好性検索のコンテスト「Netflix Prize」[5]において，高い効率性を示し脚光を浴びた．そのアルゴリズムを以下に説明する．

疎行列 $A=(a_{ij})$ を U^TV で近似したいのだが，問題の性質により，行列 A の要素が 0 であるところは高い近似性が要求されておらず，$a_{ij}\neq 0$ の部分についてよい近似が得られればよい．つまり

$$\left\{E'_{ij}=(a_{ij}-u_i^Tv_j)^2 \mid a_{ij}\neq 0\right\}$$

を同時に最小化することを考えるのが自然である．ただし，ここで $U=(u_1, \ldots, u_n)$，$V=(v_1, \ldots, v_m)$ と列ベクトルにより表しているものとする．しかし，実際にはこれでは変数が大きく動き，数値計算的に安定しないので，$|u_i|^2$ と $|v_j|^2$ もできるだけ小さくなるような修正項を入れた

$$E_{ij} = (a_{ij} - u_i^T v_j)^2 + \lambda(|u_i|^2 + |v_j|^2) \tag{1}$$

を E'_{ij} の代わりに用いる．ここで λ は定数で，アプリケーションに応じて経験的によい値を利用するチューニングパラメータである．この目的関数において，最急降下法を各 $(i,\ j)$ に対して逐次的に行うのが Webb の手法である．つまり，$a_{ij} \neq 0$ である (i, j) について

$$u_i \leftarrow u_i - \alpha' \frac{\partial}{\partial u_i} E_{ij}$$

$$v_j \leftarrow v_j - \alpha' \frac{\partial}{\partial v_j} E_{ij} \tag{2}$$

により u_i と v_j を逐次的に更新する．ここでも α' はチューニングパラメータである．ここで，

$$\frac{\partial}{\partial u_i} E_{ij} = -2(a_{ij} - u_i^T v_j) v_j + 2\lambda u_i$$

$$\frac{\partial}{\partial v_i} E_{ij} = -2(a_{ij} - u_i^T v_j) u_j + 2\lambda v_j \tag{3}$$

であるので，一回の計算で u_i と v_j をまとめて更新するとすると，$r_{ij} = (a_{ij} - u_i^T v_j)$ の計算が共通化でき，アルゴリズムは以下のようになる．ただし，$2\alpha' = \alpha$ とおいて，記述してある．

$\{u_i\}_i$ と $\{v_j\}_j$ に乱数を代入
収束条件（$\sum_{ij} r_{ij}^2$ が閾値以下）を満たすまで以下を繰り返す
　$a_{ij} \neq 0$ となるような (i, j) について，以下を繰り返す
　　$r_{ij} \leftarrow (a_{ij} - u_i^T v_j)$
　　$u_i \leftarrow u_i + \alpha(\lambda u_i - r_{ij} v_j)$
　　$v_j \leftarrow v_i + \alpha(\lambda v_i - r_{ij} u_j)$

ここで疑問が残るのは，異なる (i, j) について E_{ij} に最急降下法を適用してよいのかということである．最急降下法は通常一つの目的関数について使われるものであり，この場合 $(i,\ j) \neq (i',\ j')$ ならば，E_{ij} を減少させる変数変化は $E_{i'j'}$ を増加させるかもしれない．この問題については，理論的に裏付けるものはなく，ただこのアルゴリズムで実験的にうまくいく（収束する）から正しい，という以外にない．しかし直感的な解釈を試みると，$(i,\ j)$ に対する変数更新の時は，最急降下により E_{ij} がもっとも激しく降下する方向に進もうとするが，そのときもし $E_{i'j'}$ が上昇するにしても「もっとも激しい上昇」ではないだろうと思うのは自然であり，上昇のしかたはわずかであることが期待される．その後に (i', j') の番が回ってきたときに，いままでわずかに上昇していたとしても，最急降下でそれを十分に打ち消すほど降下するためにこのアルゴリズムが機能するものと思われる．

このように，Webb のアルゴリズムのベースとなる考え方は最急降下法であるが，この考え方を踏襲しつつアレンジを加えて GPU 上で効率的に計算できるようにしたアルゴリズムを 4 章で紹介する．

2. 2　*k*- 最近傍問題

特異値分解が行われた後に解かなければいけない問題が *k*- 最近傍問題である．*k*- 最近傍問

題とは，次のように記述される．

> n 個のベクトルの集合 $\{v_1, v_2, \ldots, v_n\}$ が与えられたとする．その時，それぞれのベクトル $v_i\,(i \leq n)$ に対し，それ以外のベクトル $\{v_j\}_{j \neq i}$ の中から v_i に近い順に上位 k 個を求めよ．

この問題の自明な解法としては，すべての $(i, j)\,(i \neq j)$ の組み合わせについて $\delta(v_i, v_j)$（ただし δ は距離関数）を計算し，その後各 i ごとにソートするというものが考えられる．この自明な解法だと計算量は $O(n^2)$ になり，n が大きくなると実用上問題とされてきた．そこで，ベクトル集合の中からクラスタ（塊）を見つけ，距離計算の対象を絞るという手法[1][2]が用いられてきた．しかし，それは近似計算にすぎず，特にデータに明確なクラスタがない場合には精度がよくない．

k- 最近傍問題について，本稿で提案するアルゴリズムは近似を使わずすべての組み合わせについて距離を計算するものである．GPU で計算することにより十分な計算速度が得られれば，近似計算の必要性はないと考える．

3. CUDA について

近年，グラフィックス専用プロセッサであった GPU（Graphic Processor Unit）に通常の数値計算をさせるという技術が注目されており，GPGPU（General Purpose GPU，GPU による一般計算）と呼ばれている．GPGPU の開発環境としては CUDA が広く使われている．CUDA は NVIDIA 社が提供している開発環境で，C 言語の拡張のような言語により比較的直感的なプログラミングができるのが特徴である．CUDA は無償で提供されており，ウェブページ[6]から入手可能である．

本章では CUDA のプログラミングモデルについて概説するが，詳細については NVIDIA 社によるプログラミングガイド[7]を参照されたい．また入門者向けの解説については，書籍[11]を参照されたい．

一般的な並列計算機構と比べたときに，CUDA の特筆すべき点はその並列実行性の高さである．十分なパフォーマンスを得るには千個以上のスレッド数が必要とされ，通常の並列計算よりも問題の細分化が必要である．また，スレッド生成のコストやコンテクスト切り替えのコストが低く，スレッドを増やすことによる計算負荷がほとんど発生しないのも特徴的である．

CUDA では，スレッドはグループ化され，そのグループはスレッドブロックと呼ばれる．スレッドブロック内に属するスレッド数はプログラマが指定できるが，計算速度を考えると最低で 64 個以上必要であり最高で 512 個である．スレッドブロックは同期のための単位でもあり，同一スレッドブロック内のスレッドは同期をとることができる．一方，スレッドブロックをまたがるスレッドの同期は個別にはとれず，GPU から CPU に制御を移すのが唯一の方法であるが，それはすべてのスレッドの動作が終了することを意味する．

CUDA のメモリはグローバルメモリ，共有メモリ，レジスタに分類される．グローバルメモリはすべてのスレッドから参照可能である．一方，共有メモリは各スレッドブロックに属し，属するスレッドブロック内のスレッドからしか参照できない．レジスタは各スレッドに属し，他のスレッドからは参照できない．グローバルメモリはアクセスの速度が遅いので，通常は共有メモリやレジスタにデータを移してから作業することが多い．しかし，共有メモリやレジス

タは，グローバルメモリと比べて容量が小さいので注意が必要である．

メモリアクセスについては，コアレスなメモリアクセス（coalesced memoryaccess，癒着したメモリアクセス）という機構が特徴的である．グローバルメモリ上の一定のメモリブロック（128 バイト）を同一スレッドブロック内の複数のスレッドが同時にアクセスしようとすると，そのアクセスはコアレスになり，高速になる．つまり，通常であればスレッドそれぞれがアクセスをし GPU 内メモリバスの競合が起こるが，コアレスになると必要なデータを一度に取得できる．

4. アルゴリズム

4.1 特異値分解のアルゴリズム

ここでは 2 章で説明した Webb のアルゴリズムをベースにし，GPU 上で効率のよいアルゴリズムを説明する．ここでは概要を説明するのみなので，詳細については参考文献[3]を参照されたい．

まず，Webb のアルゴリズムでは，u_i の更新がそれぞれ独立にできること，同様に v_j の更新もそれぞれ独立にできることに注目する．つまり，$i \neq i'$ とすると，u_i と $u_{i'}$ は更新値の計算がお互いの値に依存していないので，並列に計算できる．同様に $j \neq j'$ ならば v_j と $v_{j'}$ も並列に計算可能である．

u_i と v_j のそれぞれは並列計算可能だが，u_i の計算にはいくつかの v_j の値を使い，v_j の計算にはいくつかの u_i の値を使うので，オリジナルの Webb のアルゴリズムのように u_i と v_j の更新を続けて行うと，並列化が難しくなる．そのため，u_i の値の更新と v_j の値の更新を分離し，すべての 0 でない a_{ij} を見ながら u_i の値を更新し，その後同様に v_j の値を更新することとする．それには共通化していた $r_{ij} = a_{ij} - u_i^T v_j$ の計算を別々に行う必要があるが，並列化によるメリットの方が大きいのであまり問題にならない．こうすることにより，オリジナルの Webb のアルゴリズムと比べて u_i と v_j の更新順序が変わるので，結果として得られる値が異なってしまう．しかし，「変数の一部を変更しながら，複数目的関数の最適化を最急降下法で行う」という考え方は同じであり，実際実験的にも収束することを確かめた．

CUDA のスレッド階層構造への割り当ては，各スレッドブロックに u_i と v_j の計算を担当させることにした．その計算の中で，内積 $u_i^T v_j$ の計算は要素ごとの並列の掛け算と縮約計算による足し算で計算可能で，u_i（または v_j）への代入は各座標値の計算を各スレッドが受け持てばよい．

スレッドブロック間の負荷のバランスが問題になる．最初のフェーズで，i 番目のスレッドブロックは $a_{ij} \neq 0$ であるような j について，v_j の値を利用して計算する．実データでは，$a_{ij} \neq 0$ であるようなものの数は i によってかなりの偏りがある．これは，例えば人気のある商品と人気のない商品の差が激しいということは現実にありうることから明らかであろう．そのため，特定のマルチプロセッサに負荷が集中しないように工夫する必要がある．CUDA では若い番号のスレッドブロックからマルチプロセッサに割り当てられるので，単純に負荷の量でソートして，非ゼロ要素の数が多い順にスレッドブロックに割り当てることとする．

以上の説明をまとめて，アルゴリズムを記述する．まず，全体の流れは以下のようになる．

a_{ij} について非ゼロ要素が多い行が上にくるようにソートしたものを b_{ij} とし，

非ゼロ要素が多い列が左にくるようにソートしたものを c_{ij} とする．
このときの行の入れ替えの逆写像と列の入れ替えの逆写像をそれぞれ p，q とする．
（つまり $a_{p(i),j} = b_{ij}$, $a_{i,q(j)} = c_{ij}$）

収束条件を満たすまで以下を繰り返す
u_i の値の更新を行う（GPU 上での計算）
スレッドブロック間の同期（一度制御が CPU にもどる）
v_j の値の更新を行う（GPU 上での計算）

u_i の計算は以下のようになる．ただし，ここで *bid* はスレッドブロックの番号（0 から振られる）で，*tid* はスレッドの番号（スレッドブロックごとに 0 から振られる）である．また各スレッドブロックあたりのスレッドの数は BSIZE とする．また，ベクトル u_i の k 番目の要素を $u_i^{(k)}$ などと表すこととする．

$b_{bid,j} \neq 0$ となるようなすべての j について以下を行う
For $k := tid$ to $d-1$ step BSIZE
$r_k \leftarrow r_k + u_{p(i)}^{(tid)} v_j^{(tid)}$
縮約計算により $r = b_{ij} - \sum_k r_k$ を計算
スレッド間同期
For $k := tid$ to $d-1$ step BSIZE
$u_{p(bid)}^{(k)} \leftarrow u_{p(bid),j}^{(k)} + \alpha(\lambda u_{p(bid),j}^{(k)} - r v_j^{(k)})$

このアルゴリズムで，二つの For 文の部分は d 個の座標値をスレッドで分担するためのものであり，BSIZE ずつ飛ばして読むことで複数のスレッドが連続領域を同時に読むようになり，コアレスなメモリアクセスになる．また，縮約計算とは結合法則が成り立つ演算子についての集約計算のフレームワークを利用したものであり，詳細は省略する．ここで計算される r の値は $b_{ij} - u_i^T v_j$ に等しい．
V の更新計算についてもほぼ手順は同様であるので説明は省略する．

4.2　*k*- 最近傍問題のアルゴリズム

この章ではベクトル $v_0, v_1, \ldots, v_{n-1}$ に対して k- 最近傍問題の GPU 上のアルゴリズムを示す．本章においては，アルゴリズムの概略のみを説明することとする．詳細については，筆者らによる論文[4]を参照されたい．また，その論文では，複数 GPU による並列計算のアルゴリズムも説明しているが，本稿では省略する．
k- 最近傍問題の計算手順は次の二つのステップに分けられる．

1. 距離計算：すべての (i, j) の組み合わせに対して距離 $\delta(v_i, v_j)$ を計算する
2. 部分ソート：計算された距離について，i ごとに昇順ソートした上位 k 個を得る

距離計算については，N 体問題の知られているアルゴリズムのアイデアを応用して新たなア

ルゴリズムを提案する．だがそれ以前に，問題のサイズが大きいために，GPU のメモリに入るように問題の分割をしなければならない．その分割を示したのが図 3 である．ここで，x 軸と y 軸はベクトルのインデックスを表す．つまり，大きな四角の中の点 (x, y) は，v_x と v_y の距離の計算に対応する．ここで特に距離関数 δ が対称的であるとき（つまり $\delta(u, v) = \delta(v, u)$ であるとき）は $x > y$ の部分のみを計算すればよく，図 3 の灰色部分は計算する必要がない．以下，特に計算不要部分については明記せずに説明を進める．

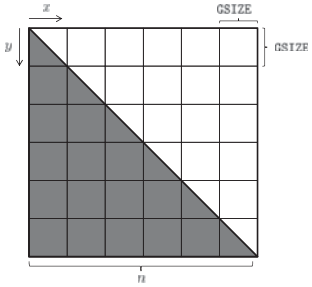

図 3　トップレベルの問題の分解

こうして分割されたグリッドに対する解法として提案するのは，既に知られている N 体問題の Nyland らによる解法[8]を応用したものである．Nyland らのアルゴリズムでは，一つのスレッドブロックに均等に複数の行が割り当てられ，そのブロックがさらに決まった列数で区切られ，区切られた部分がスレッドで同時実行される．図 4 がそれを説明している．この場合，同一スレッドブロック内の 16 スレッドが同時に小さい四角内の距離計算をし，終わったら右の四角の計算に移る．16 スレッドが計算を始める前には共有メモリに一度必要なデータを読み込み，計算結果も共有メモリに格納する．次の四角に移る前に，計算結果はグローバルメモリにコピーされる．こうすることで，共有メモリを有効に使い，コアレスなアクセスを最大限に引き出せる．

一般に N 体問題では 3 次元や 6 次元などを扱うことが多いが，本稿における問題では次元が大きいために図 4 でいう小さい四角に当たるデータは共有メモリには入りきらない．そのため，次元方向にも分割する．つまり，決められた数の座標値を読み込んでは計算するといういうことを繰り返す．図 5 がこの手順を説明している．

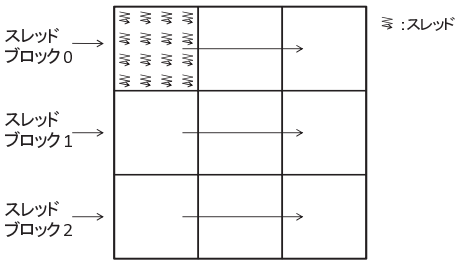

図 4　Nyland らによる N 体問題アルゴリズム

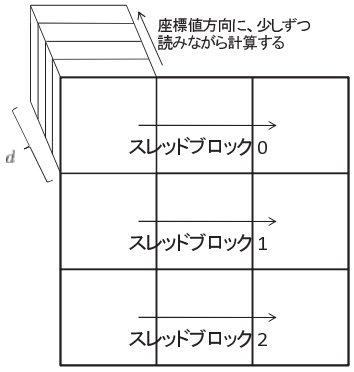

図 5　提案アルゴリズムの概念図

次に部分ソートの計算をする．ここで，k は行列 A のサイズ（行数，列数）に比べて十分小さいと仮定する．各スレッドブロックは，インデックス対集合 $(x,\ y)$ の「y = 一定」の部分それぞれに割り当てられる．つまり，スレッドブロック番号 y のスレッドブロックは $\{\delta(v_x,\ v_y)\}_{0 \le x \le n-1}$ の部分ソートを行う．スレッドブロック内のスレッドの数を BSIZE とすると，計算のメカニズムは以下のようになる．

- 各スレッドはローカルに一定の大きさ D の浮動小数点型配列をもつ．
- スレッドブロック内の共有メモリに「現在の上位 k 個」を保存するための降順ヒープ構造をもつ．つまり，現時点で一番小さいものから k 番目に大きいものまでをヒープ構造で保持し，k 番目に大きいものが瞬時に（計算量 $O(1)$ で）参照できるようにする．
- スレッド番号 tid のスレッドは $x = tid$ から始めて，BSIZE ずつ飛ばしながら配列の要素を読み込む．
- 読み込んだ要素が，現在 k 番目に小さいもの（上記のヒープにより瞬時に参照可能）よりも小さければ，それはヒープに追加する候補になるので，ローカルの配列に保存する．そうでなければいらないので捨てる．
- 上記のような，読み込み・比較を D 回繰り返すたびに，ローカル配列がいっぱいになる可能性があるので，排他制御をしながらローカル配列内の数値をヒープに追加する．このとき追加前の k 番目の要素は捨てる．

ここでヒープ構造とは，ソートされている状況を維持するための一般的な構造であり，詳細は教科書[10]などを参照されたい．ここでのアイデアは，k が十分に小さいことによりほとんどの数値が捨てられることによる．つまり，一般的に排他制御をしながらヒープに追加する作業は，スレッドの待ち状態が発生するためスループットが悪くなるが，そこに至るまでのふるい落としにより，ほとんどの数値は捨てられてしまっているため，排他制御によるパフォーマンス損失を軽減している．

5.　実験と考察

ここでは，特異値分解と k- 最近傍問題のそれぞれについて，提案アルゴリズムを実装したもののベンチマークを行う．比較の対象となる CPU は，Intel Core i7 920（2.67GHz）であり，

CPU 上の実装はいずれもシングルコアでのみ動かしたものである．また，GPU としては，NVIDIA GTX280 を用いた．

特異値分解の計算実験の結果を表 1 に示す．データはサイズ $n \times n$ の行列の 1%の要素をランダムに選び，その要素に乱数を入れることで用意した．また，圧縮後のサイズは 256 である．

表 1　特異値分解の計算の所要時間（1 イテレーションあたり，単位：秒）

n	10000	20000	40000
GTX280 (a)	0.2	1.0	4.1
i7 920 (CPU) (b)	6.2	25.0	100.0
(b)/(a)	31.0	25.0	24.3

ここで示される計算時間は 1 イテレーションあたりである．つまり，すべての (i, j)（ただし $a_{ij} \neq 0$）に対して，U と V を更新する時間を示している．収束までの時間で比較しなかったのは，アルゴリズムの変更により結果の数値が異なるため，収束までの繰り返し回数が異なってくるからである．また，収束までの繰り返し回数は，各種パラメータの設定により敏感に変化し，CPU に有利なパラメータ設定もあれば GPU に有利なパラメータ設定もある．そのため，純粋にアルゴリズムの性能を評価するには 1 イテレーションあたりで比較するのが妥当だと判断した．

表 1 では，問題のサイズが大きくなるほど GPU の CPU に対する優位性が低くなる傾向が読み取れ，実用上大きなデータを扱うことを考えるとこれは悪い傾向であると言える．原因はまだ究明できていないが，サイズが大きくなるほど負荷バランスの偏りが大きくなるのが原因だと予想される．とはいっても $n = 20000$ から $n = 40000$ へのパフォーマンス比の落ち込みはさほど大きくなく，実用上あまり重要な問題ではないと考える．

次に，k- 最近傍問題についての計算実験の結果を表 2 と表 3 に示す．表 2 は Hellinger 距離に関する実験であり，表 3 は Euclid 距離に関するものである．ここで，ベクトルの次元 $d = 256$ とし，$k = 100$ として計算した．表中の n はベクトルの数を示している．

表 2　k- 最近傍問題の計算の所要時間（Hellinger 距離の場合，単位：秒）

n	10000	20000	40000	80000
GTX280 (a)	2.7	8.6	34.1	131.8
i7 920 (CPU) (b)	354.2	1419.0	5680.7	22756.9
(b)/(a)	131.1	173.3	166.5	172.6

表 3　k- 最近傍問題の計算の所要時間（Euclid 距離の場合，単位：秒）

n	10000	20000	40000	80000
GTX280 (a)	2.6	8.2	32.1	124.8
i7 920 (CPU) (b)	124.8	503.0	2010.0	8041.4
(b)/(a)	48.0	61.3	62.6	64.4

Hellinger 距離とは，二つの確率分布間の距離として統計的評価でよく用いられるもので，実際レコメンデーションの計算でもよく用いられる．Hellinger 距離は以下の式で定義される．

$$\delta_{\mathrm{Hellinger}}(u, v) = \sum_l \left(\sqrt{u^{(l)}} - \sqrt{v^{(l)}}\right)^2 \tag{4}$$

ただし確率分布であることにより

$$\sum_l u^{(l)} = 1, \quad \sum_l v^{(l)} = 1 \tag{5}$$

を前提とする．一方 Euclid 距離は，我々が通常「距離」と呼んでいるもので，

$$\delta_{\mathrm{Euclid}}(u, v) = \sqrt{\sum_l (u^{(l)} - v^{(l)})^2} \tag{6}$$

と定義される．ただし，この実験では外側の根号は大小の比較に関して本質的ではないので省いて計算している．つまり，$\delta^2_{\mathrm{Euclid}}$ による k- 最近似問題を計算している．

ここで，CPU による計算時間が Hellinger 距離と Euclid 距離で大きく異なるのは，Hellinger 距離の計算に多発する平方根の計算のコストに由来するものである．一方 GPU では，Hellinger 距離と Euclid 距離で計算時間にあまり大きな違いが見られないのは，並列性の高さにより平方根の計算コストを吸収したからである．結果として，Euclid 距離の方が GPU による速度的優位性が低くなっている．

6. お わ り に

GPU によるレコメンデーションシステムのアルゴリズムを考案し，実験により提案アルゴリズムが CPU と比べて飛躍的な計算速度向上につながることを示した．特に特異値分解の計算は CPU の 30 倍程度の計算速度を達成し，k- 最近傍問題では 170 倍程度（Hellinger 距離の場合）を達成した．これにより，大規模なデータに対しても短時間で計算することが可能になり，顧客の購買行動により素早く対応できるマーケティングツールの開発につながると考える．

参考文献
［1］ A. S. Das, M. Datar, A. Garg, and S. Rajaram. Google news personalization: scalable online collaborative filtering. In *WWW '07: Proceedings of the 16th international conference on World Wide Web*, pp. 271-280, New York, NY, USA, 2007. ACM.
［2］ P. Indyk and W. Miller. Approximate nearest neighbors: towards removing the curse of dimensionality. In *Proceedings of the 1998 Symposyum on Theory of Computing*, 1998.
［3］ K. Kato and T. Hosino. Singular value decomposition for collaborative filtering on a gpu. In *Proceedings of 9th World Congress on Computational Mechanics and 4th Asian pacific Congress on Computational Mechanics (WCCM/APCOM2010)*, 2010.（掲載予定）.
［4］ K. Kato and T. Hosino. Solving k-nearest neighbor problem on multiple graphics processors. In *Proceedings of The 10th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGrid2010)*, pp. 769-773, 2010.
［5］ Netflix Prize. http://www.netflixprize.com/
［6］ NVIDIA. CUDA Zone. http://www.nvidia.com/object/cuda_home.html
［7］ NVIDIA. CUDA programming guide. http://www.nvidia.com/object/cuda_develop.html
［8］ L. Nyland, M. Harris, and J. Prins. Fast *n*-body simulation with CUDA. In *GPU Gems III*, pp. 677-695. NVIDIA, 2007.

［9］ Simon Funk (Bradyn Webb). Try this at home.
http://sifter.org/~simon/journal/20061211.html
［10］ エイホ，ホップクロフト，ウルマン．データ構造とアルゴリズム．培風館，1987.
［11］ 青木，額田．はじめての CUDA プログラミング．工学社，2009.
上記参考文献内の URL は 2010 年 6 月 10 日時点での存在を確認.

執筆者紹介 加 藤 公 一（Kimikazu Kato）
1998 年日本ユニシス（株）入社．2008 年に東京大学大学院情報理工学系研究科で博士号取得．情報理工学博士．現在，先端技術部にて，計算幾何，離散最適化などの研究に従事．電子情報通信学会，IEEE 会員.