Vol.2017-DBS-164 No.7
2017/1/17

GPU向け簡潔データ構造による辞書を利用した効率的な語の重み計算

若月 駿尭[1,a)]　欅 惇志[1,b)]　宮崎 純[1,c)]

概要：GPU 上で効率的に実行可能な，簡潔データ構造を用いたトライ木による語の整数値への変換およびデータ並列プリミティブを用いることで，情報検索で利用される文書中の語の重みである BM25 を効率的に計算する手法について述べる．GPU を用いる際に課題となっていた文字列比較操作の削減やスレッド間の負荷分散を考慮した並列化を行った．比較手法である MapReduce による CPU 上，GPU 上での計算手法と合わせて評価実験を行った結果，辞書とデータ並列プリミティブを活用した提案手法はマルチコア CPU 上で実行した MapReduce による計算手法と比べて約 5 倍の性能向上を達成した．

1. はじめに

文書を高速に処理することが求められる背景として，コンピュータや Web の普及により電子文書が大量に作成，存在するようになったため，それらから現実的な時間で適切な情報を検索し，利用することが困難であるという状況がある．文書を処理し，ユーザの情報要求に合致した文書を提示する，高精度な検索を実現するための方法として，クエリと文書の適合度を測る指標が提案されてきた．

古典的な確率モデルをもとにして単語の出現頻度や文書長を考慮に加えた Okapi BM25[1] や，言語モデルを利用したクエリ尤度モデル [2] が広く利用されている [3]．これらの文書中の語の重み付け手法は，適合度を算出するために各文書中の語の出現頻度 (term frequency, tf) や文書頻度 (document frequency, df) などの統計量を必要とする．高精度な検索を高速に行うためには，これらの索引語重みとその計算に必要な統計量を，大量の文書から効率よく求めなければならない．

さまざまな処理を高速に実現する手段として，一般的な CPU とは異なるアーキテクチャのプロセッサであるが，高い並列性とメモリバンド幅を備えたメニーコアプロセッサである GPU を汎用的な用途に応用することが取り組まれている．例えば高い浮動小数点演算性能が活かされた，数値計算を主とするアプリケーションに対するライブラリが提供されている [*1]．またグラフ処理 [4] といった，非数値計算にも応用が進んでおり GPU 上で多用される効率的な処理が，データ並列プリミティブとして提案されている．また，GPU には不得手とされていた文字列処理についても，データベースにおける文字列検索クエリの高速化 [5] がなされるなど応用範囲が広がっている．

本稿では GPU による効率的な文書の処理を目指して，BM25 の計算をとりあげ，効率的な計算手法について議論する．ボトルネックになりうる文字列操作を，あらかじめ辞書を用いて語を整数値 ID に変換することで回避し，効率的に BM25 の計算を行う手法について述べる．

本稿の構成は以下の通りである．2 節では BM25 の計算に関する問題設定について述べる．3 節では基礎的事項として辞書や GPU のアーキテクチャ，データ並列プリミティブ，MapReduce による BM25 の計算手法について述べる．4 節では GPU によるデータ並列プリミティブを用いた提案手法について述べる．5 節では各手法について評価実験を行い，その結果について議論する．6 節では本稿に関連する研究について述べる．7 節では結論および今後の課題について述べる．

2. 問題設定

文書 d における語 t の BM25 重み w_{td} の計算式を式 1 に示す．

$$w_{td} = \log \frac{N + 0.5}{df_t + 0.5} \cdot \frac{(k_1 + 1) tf_{td}}{k_1((1-b) + b \times (L_d / L_{ave})) + tf_{td}} \quad (1)$$

ここで，N は文書数，df_t は語 t が出現する文書数，tf_{td} は語 t が文書 d 中に出現する回数，L_d は文書 d の文書長，

[1] 東京工業大学情報理工学院情報工学系
Department of Computer Science, School of Computing, Tokyo Institute of Technology
[a)] wakatsuki@lsc.cs.titech.ac.jp
[b)] keyaki@lsc.cs.titech.ac.jp
[c)] miyazaki@cs.titech.ac.jp

[*1] https://developer.nvidia.com/gpu-accelerated-libraries

L_{ave} は全文書の平均文書長である．なお k_1 および b はパラメータである．

本稿では，文書群 C 中のすべての文書 $d \in C$ について，文書 d とそれに含まれるすべての語 t の組 $\langle t, d\rangle$ に対応する w_{td} を計算し，$\langle t, d, w_{td}\rangle$ タプルを出力することを考える．

入力となる文書群は，1 行 1 単語の形式かつ文書と文書の区切りは空行で示されたデータとなっていることを仮定する．出力となるタプルの並び順や格納形式は任意とする．

GPU を用いて BM25 の計算を行う際の課題として，各語に対する w_{td} の計算に必要な df_t が，全文書を先に処理しなければ求まらない点や，ハッシュを用いた効率的な集約が困難である点があげられる．また，ソートを用いて集約を行う場合についても，文字列をキーとした効率的なソートは，不規則な負荷やメモリアクセスなどの面から困難である．

3. 基礎的事項

3.1 辞書

辞書は文字列キーの集合に対する効率的な参照が可能なデータ構造である．辞書を実現する方法としてハッシュテーブルやトライ木が用いられるが，GPU 上において効率的に動作させる手法は自明ではない．

辞書により全ての語を整数値 ID に変換することを考えると，辞書に含まれていない語については情報が失われてしまうため，できる限り多くの語を格納できる省メモリな辞書が求められる．

本稿では省メモリな簡潔データ構造を用いたトライ木による Prefix Matching アルゴリズム [6] を GPU 向けに最適化した実装 [7] を利用する．

3.2 GPU

本稿では GPU として Maxwell アーキテクチャの NVIDIA GeForce GTX 970 を用い，GPGPU コンピューティングフレームワークとして CUDA を用いた．

GPU において実行される各スレッドは 32 スレッドごとにワープという単位でまとめられ，32 コアにより実行される．ワープ内のすべてのスレッドは同じ命令が実行されるため，プログラム中の分岐により異なる実行パスをたどる際には，各スレッドは自らの実行パス以外による結果を破棄しつつ，順次すべてのパスが実行される．これを warp divergence と呼び，性能低下の原因となる．

また，DRAM へのアクセスは基本的に 128 バイトのメモリトランザクションにより行われる．ワープ内の 32 スレッドが連続する 128 バイトの領域内のデータにアクセスするとき，これをコアレスアクセスと呼び，効率的なメモリアクセスのために重要である．

3.3 データ並列プリミティブ

GPU におけるデータ並列プリミティブは，アルゴリズムの構成要素として用いられる汎用的な処理であり，scan[8]，merge[9] といった効率的なアルゴリズムが提案されている．また，これらのアルゴリズムが実装されたライブラリ [10] が存在する．本稿では主に moderngpu[10] ライブラリを使用し，用いたデータ並列プリミティブについて以下に述べる．図 1 に各データ並列プリミティブの実行例を示す．

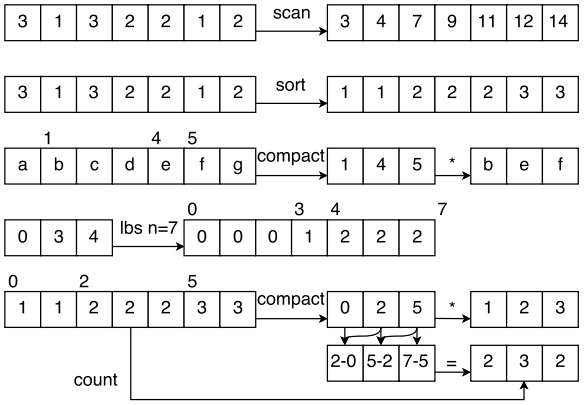

図 1 各データ並列プリミティブの実行例

3.3.1 scan

scan または prefix-sum と呼ばれる操作は，入力として結合則を満たす二項演算子 $\oplus$ と要素数 n の配列 $[a_0, a_1, a_2, ..., a_{n-1}]$ を受け取り，配列 $[a_0, (a_0 \oplus a_1), ..., (a_0 \oplus a_1 \oplus ... \oplus a_{n-1})]$ を出力する．このように i 番目の出力に i 番目の入力が含まれているものを inclusive な scan と呼び，含まれないものを exclusive な scan と呼ぶ．

3.3.2 sort

GPU 上で高速に動作するソートアルゴリズムとしては基数ソート [11] が最も高速であるとされている．しかしながら基数ソートはソートできる値の性質に制約がある．比較ソートでは，効率的な merge[9] をもとにしたマージソート [10] が高速である．本稿ではマージソートを使用する．

3.3.3 compact

compact または filter は n 個の要素から k 個 $(k \leq n)$ を出力する操作である．各要素が条件を満たしているかどうかを判定し，条件を満たす要素の個数 k を計算するステップと，条件を満たす要素の位置から出力を行うステップからなる．

3.3.4 load balancing search

要素数 n の配列を分割する位置を，前から順に並べた要素数 k の配列を入力として，要素数 n の配列の各要素の値が対応する要素数 k の配列のインデックスとなっている配列を出力する操作である [10]．

3.3.5 count

要素数 n の配列を入力とし，それぞれの値に対して，同じ値を持つ要素の数を出力する．配列が値によりソートされていない場合にはソートを行う．次にソートされた配列から，値の境界位置を抽出する．これは配列の各要素に対して，一つ前の要素と値が等しいか否かによる compact 操

作により実現できる．次に抽出された位置の配列の各要素に対して，一つ後の要素からの差を計算することで，同じ値を持つ要素の数が得られる．最後の要素については n との差を計算する．得られた要素数に対応する値は，境界位置の配列が示す位置の値となる．同様の手法が GPU を用いたラベル伝搬法で用いられている [12].

3.4 MapReduce による計算手法

本節では既存手法として森谷らによる MapReduce を用いた BM25 の計算手法 [13] と CPU, GPU 上における MapReduce フレームワークについて述べる．

MapReduce[14] は大規模なコンピュータクラスタ上でアプリケーションを動作させることを想定して作られたプログラミングモデルである．そのシンプルさ，プログラミングの容易さから，クラスタ [*2]，共有メモリマルチプロセッサ [15]，GPU[16] などの環境を想定した，さまざまなフレームワークが提案されている．

本稿では MapReduce による BM25 の計算手法を，CPU 上で動作する Phoenix++[15] フレームワークおよび GPU 上で動作する Mars[16] を拡張したフレームワーク [13] をもとに，Shuffle ステップの際のソートを Baxter のマージソート [10] に差し替えたフレームワークを使用する．

以降では MapReduce により BM25 の計算を行う手順について述べる．なお MapReduce による計算手法では簡単のために平均文書長 L_{ave} は既知とする．

3.4.1 Map

入力文書群は文書ごとに分割され，文書単位で並列に Map 処理を行う．各 Map 処理では文書 d から語を切り出し，語 t を key，文書 ID d を $value$ とした $\langle key, value\rangle$ ペアを出力する．同じ語が複数回出現している場合では，複数回 $\langle key, value\rangle$ ペアが出力される．また各 Map 処理において，出力した $\langle key, value\rangle$ ペアの個数を文書長 L_d として配列に記録する．

3.4.2 Reduce

語 t を持つ $\langle key, value\rangle$ ペア単位で Reduce 処理を行う．集約された $value$ から t が出現した文書 ID d のリストが得られる．同じ文書中に複数回出現している場合にはリスト中に同じ文書 ID が出現回数個含まれている．すなわち文書 ID d における t の 出現頻度 tf_{td} はリスト中に含まれる d の個数として求めることができる．また，t の文書頻度 df_t は リスト中に含まれるユニークな d の個数である．

今回はソートされた文書 ID リストを用いることで，これらの値を求める方針をとる．文書 ID リストを先頭から順に走査し，異なる値の数，すなわち df_t を求める．再び前から順に走査し，同じ値 d の数を数え，文書 d における tf_{td} を求めていく．tf_{td} が求まると，文書 d における語 t の BM25 w_{td} が計算できるため，$\langle t, d, w_{td}\rangle$ を出力する．

[*2] http://hadoop.apache.org/

4. 提案手法

4.1 データ並列プリミティブによる手法

以下の 8 ステップによって BM25 の計算を行う．図 2 に全体の計算手順を示す．

4.1.1 語の切り出し

BM25 は語ごとに計算されるため，文書から語を切り出す必要がある．文書が前処理されているとの仮定から，以下が成り立つ．

(1) ある文字がアルファベットかつ一つ前が改行であれば，その文字は語の先頭である．
(2) 語の先頭かつ二つ前が改行であれば，その文字は文書の先頭である．

(1) の条件による compact 操作により，語の先頭である文字の位置を並べた配列を出力する．加えて，出力される各要素について，(2) の条件より，語が文書の先頭であれば 1, 先頭でなければ 0 を並べた配列を出力する．ただし，最初の文書の先頭は 0 とする．

4.1.2 文書 ID 割当

前のステップにおいて語の切り出しが完了した．しかし語がどの文書に属しているかという情報はないため，これを求める必要がある．そこで語が文書の先頭であるかどうかを記録した配列に対して，和の inclusive scan を適用することで，同じ文書中の語は同じ値となる単調増加の配列を得る．これにより求まった値を文書 ID とみなす．

4.1.3 文書長計算

BM25 の計算には各文書の長さが必要である．文書 ID の配列を用いてこれを求める．文書 ID d の文書長は配列中の d である要素数となる．文書 ID の配列は既にソートされた状態であるので，count 操作により文書長の配列が得られる．

4.1.4 ソート

語の位置を key，文書 ID を $value$ としたソートを行い，同じ語の要素をまとめる．比較関数として，語の位置から文字列を読み，文字列比較を行うものを用いる．ここでは安定なソートを用いているため，等しい key を持つ要素の範囲では，$value$ である文書 ID の大小関係も保たれる．

4.1.5 tf_{td} 計算

語 t と 文書 ID d が等しい要素の数が tf_{td} となる．前のステップにより t に関してソートされた状態であるので，count 操作により tf_{td} の値が得られる．加えて，tf_{td} に対応する語の位置と文書 ID d を並べた配列を出力する．

4.1.6 df_t 計算

前のステップによりユニークな語 t と文書 ID d の組を要素として持つ新たな配列が得られた．この配列のうち，語 t が等しい要素の数が df_t となる．tf_{td} の場合と同様に，t に関してソートされた状態であるので，count 操作により df_t の値が得られる．

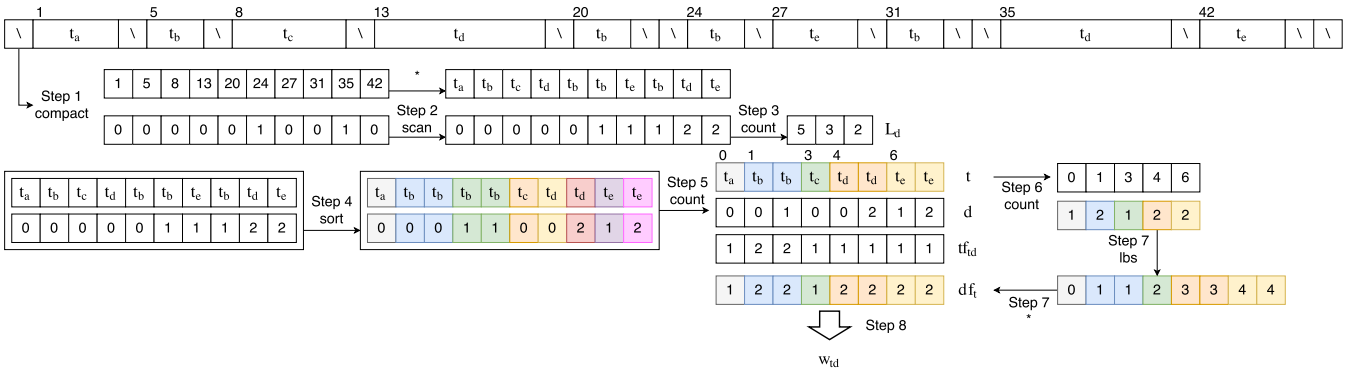

図 2　提案手法の実行例：データ並列プリミティブを用いる 8 つのステップから構成される．辞書を用いる場合ではソートの前で $t_a, ..., t_e$ を ID に変換し，その後のステップにて用いる．

4.1.7　df_t 割当

前のステップで得られた df_t の配列は語に対して一つの要素となるため tf_{td} の配列とは要素数が異なり対応していない．各 df_t の要素に対してスレッドを割り当て BM25 の計算を行う方法では，各スレッドが計算，出力する BM25 の数に偏りが生じ，warp divergence によりコアの稼働率が低下する．これを避け，各スレッドが均等に BM25 を計算できるように，df_t の配列から tf_{td} の配列に対応するように df_t を並べた配列を求める．

ある t について，df_t に対応する tf_{td} の要素数は df_t である．前ステップの count 操作時に作成される語 t の境界位置の配列を再利用する．境界位置の配列を入力とする load balancing search 操作により，各 tf_{td} の要素に対応する df_t 配列のインデックスが求まるため，インデックスを用いて df_t の値を読み取り出力する．

4.1.8　BM25 計算

以上のステップにより計算に必要な要素が求まったため，BM25 w_{td} の計算を行い出力する．文書長は文書 ID d を用いて取得する．平均文書長は総単語数を文書数で割った値である．

4.2　辞書を用いる手法

3.1 節にて述べた辞書 [7] を用いる．辞書の構築はあらかじめ行われており，CPU のメモリ上にあるとする．GPU に入力を転送するとき，辞書を同時に転送する．

ソートステップの前に，語を辞書により一意な整数値に変換するステップを設ける．語の位置の配列を入力とし，語の ID となる整数値の配列を出力する．以後のソート，tf_{td} 計算，df_t 計算ステップでは，この整数値の配列を用いて語の比較を行う．

文書中に出現した語が辞書に含まれていない場合は，未知語としてすべて同じ ID が割り振られる．そのため未知語は全て同一の語として BM25 の値が算出される．ただし，未知語の存在によって辞書に含まれる語の BM25 算出には悪影響を及ぼすことはない．

5.　評価実験

5.1　実験準備

本節では次の 4 種類の手法について比較し評価を行う．

MapReduce Phoenix++ (MRP)
Phoenix++によるマルチコア CPU 上で実行する手法．スレッド数は実験に用いる CPU が持つ論理コア数の 8 とする．

MapReduce Mars revised (MRM)
Mars ベースのフレームワークによる GPU 上で実行する手法．

Parallel Primitives (PP)
データ並列プリミティブ による GPU 上で実行する手法．

Parallel Primitives with Dictionary (PPD)
PP に辞書による変換処理を加えた手法．

実行時間の計測範囲は，文書が CPU のメモリ上に読み込まれている状態から，BM25 の値が CPU のメモリ上に書き込まれるまでとした．そのため GPU とのデータの転送時間を含む．

TREC ClueWeb09 Category B[*3] に含まれる英文 Web 文書約 5,000 万ページから，大文字小文字を区別せずに英文字のみで構成される単語の出現頻度を調べた．そのうち出現頻度が高い上位 1,000 万種類の単語を語彙として用いる．

辞書を用いる手法と用いない手法について同条件で比較するために人工的な文書を用いる．実際の文書を模すため，以下の条件により人工的に文書を生成する．各文書の文書長は $\mu = 6.0, \sigma = 1.1$ の対数正規分布に基づく乱数により決める．文書中の各単語は語彙から出現頻度に基づく離散分布によりランダムに抽出する．この方法を用いて，ファイルサイズが合計 100MB, 200MB, 300MB, 400MB, 500MB となる 5 個のデータセットを作成した．各データセットに対応する単語数 n_w，文書数 n_{dl}，tf_{td} の個数 n_{tf}，

[*3] http://trec.nist.gov/

表 1　データセットの詳細

	n_w	n_{dl}	n_{tf}	n_{df}
100MB	17,881,505	24,411	11,029,756	447,663
200MB	35,767,202	48,725	22,074,419	687,255
300MB	53,651,412	73,163	33,119,396	882,477
400MB	71,532,437	97,683	44,123,823	1,051,534
500MB	89,409,102	122,178	55,165,752	1,203,716

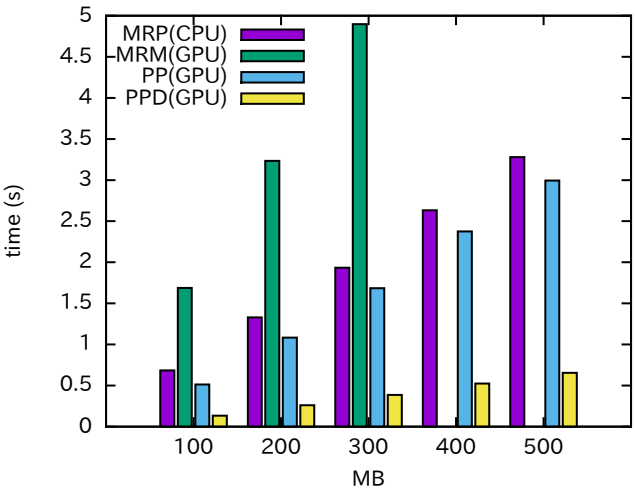

図 3　各手法の実行時間（秒）
MRM(GPU) 400MB 以上はメモリ不足により実行不可.

df_t の個数 n_{df} を表 1 に示す.

以降の実験は Intel Core i7-6700K (4.0GHz, 4 コア), DDR4 16GB, NVIDIA GeForce GTX 970 (1.05GHz, 1664CUDA コア, 4GB), Ubuntu 14.04, CUDA 7.5. を用いて行った.

5.2　実験結果

図 3 に各手法の実行時間を示す．CPU による計算と比べて，データ並列プリミティブを用いた GPU による計算が高速となる結果となった．特に辞書を用いた PPD 手法では，マルチコア CPU 上で実行している MRP 手法と比べて，5.0 倍から 5.1 倍の性能となっている．辞書を用いることの効果としては，PPD 手法は PP 手法と比べて，3.8 倍から 4.5 倍の性能向上を達成することができている．

一方 GPU を用いた MapReduce による手法は，400MB 以上のデータセットについては GPU 上のメモリ不足により実行できず，すべてのデータセットで CPU 以上に実行時間が長い結果となった．

また，CPU による計算手法の実行時間について，森谷らはスレッドプログラミングによるナイーブな実装を用いて，250MB のデータセットを処理した際の実行時間を 254 秒と報告 [13] しており，MapReduce を用いることで，容易かつ効率的に BM25 の計算が実現できていると考えられる.

CPU, GPU による MRP, MRM 手法について各ステップの実行時間の内訳を図 4 に示した．shuffle ステップは

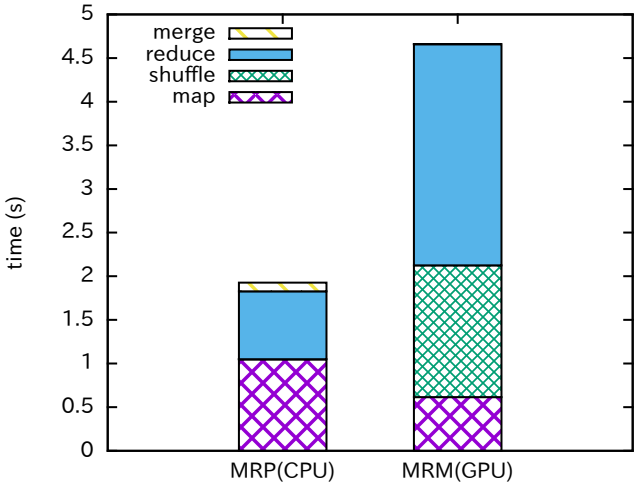

図 4　MapReduce 手法の実行時間内訳 (秒) (300MB)

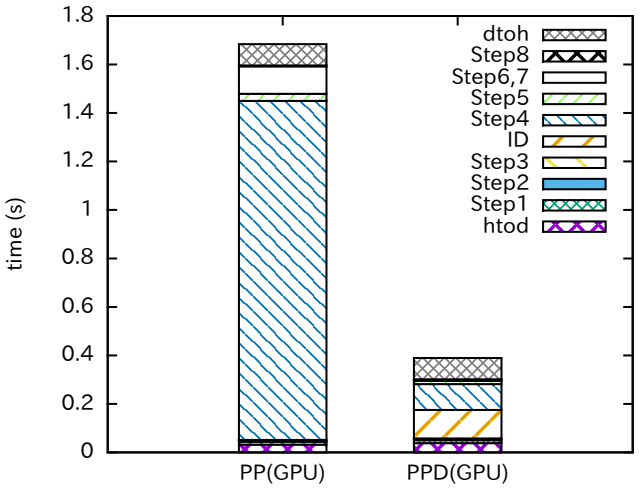

図 5　データ並列プリミティブ手法の実行時間内訳 (秒) (300MB)
dtoh: 結果のデータ転送，htod: 入力のデータ転送，ID: 辞書による変換，各 Step は 4 節に対応する.

$\langle key, value \rangle$ ペアを集約するステップであり，Mars ではソートを行うことにより実現されている．Phoenix++ではハッシュにより実現されており，map, reduce ステップに含まれている．merge ステップは各スレッドが Reduce ステップで出力した $\langle key, value \rangle$ ペアのリストを一つのリストにマージするステップである．Mars を用いた MRM 手法の Reduce ステップでは文字列操作が行われないにもかかわらず実行時間の割合が大きいが，これは各スレッドの負荷の偏りが原因と考えられる.

データ並列プリミティブを用いた手法について，各ステップの実行時間の内訳を図 5 に示した．辞書を用いない PP 手法では，文字列のまま語を扱うため，不規則な長さを持つ語を扱う必要のあるステップであるソート (4)，TF 計算 (5)，DF 計算 (6,7) が占める実行時間の割合が大きくなっている．対して PPD 手法では，辞書を用いて語を ID に変換する ID ステップを設けることで，その後のステッ

プにおいて文字列の代わりに整数値 ID を用いることで，それらのステップの実行時間が削減できている．

PPD 手法では辞書を用いることで高速化を達成した．その対価として，辞書による変換に要する時間の発生，辞書に含まれない語に対する計算ができなくなる点があげられる．特にデータ転送を除いた実行時間のうち，辞書による変換が占める割合は約 45%であり，高速かつコンパクトな辞書を利用することは，情報検索のように大量の語を扱う上で重要である．

6. 関連研究

Sitaridi らは GPU 上における文字列検索の SQL クエリを想定した文字列の逐次検索手法として，最適化が可能な規則的なメモリアクセスパターンを持つ KMP 法をベースとした手法，また GPU に適した文字列の格納形式である pivot レイアウトを提案，評価している [5]．文字列検索の SQL クエリはクエリに一致する文書が列挙されるブーリアンモデルであり，より高精度な検索のためには，クエリと文書の適合度を測る必要がある．

森谷らは GPU 上の MapReduce フレームワークである Mars[16] を拡張し，BM25 の計算を高速に行う手法を提案している [13]．しかしながら Map および Reduce ステップにおける各スレッドの負荷の偏り，また文字列比較によるソートが原因となる性能の劣化が課題となっていた．本稿では，データ並列プリミティブ，辞書を用いることで，これらの課題の解消を試みた．

7. おわりに

本稿では BM25 を効率的に GPU を用いて計算する手法を提案した．評価実験の結果からデータ並列プリミティブを組み合わせることで，GPU 上で効率的に文字列処理を行うことが可能であることが判明した．さらに簡潔データ構造による辞書を用いて，GPU 上での手法においてボトルネックとなっていた不規則な処理が発生する文字列比較を避けることにより，マルチコア CPU による計算と比べて最大で 5.1 倍，辞書を用いない GPU 上での手法と比べて最大で 4.5 倍の性能向上を達成した．辞書を用いることは文字列処理を GPU 上で効率的に行う際に有効であることが判明し，コンパクトかつ高速な辞書の重要性が明らかとなった．

今後の課題として，他の文書処理における辞書の有効性の検証や GPU 上のメモリはサイズが限られているため，GPU 上のメモリにすべてが載らない文書群に対する計算手法の考案が考えられる．

謝辞 本研究の一部は，JSPS 科研費 (JP26280115, JP15H02701, JP16H02908, JP15K20990) の助成による．ここに記して謝意を表す．

参考文献

[1] Robertson, S. E., Walker, S., Jones, S., Hancock-Beaulieu, M. and Gatford, M.: Okapi at TREC-3, *Proceedings of The 3rd Text REtrieval Conference*, pp. 109–126 (1994).

[2] Ponte, J. M. and Croft, W. B.: A Language Modeling Approach to Information Retrieval, *Proceedings of the 21st Annual International ACM SIGIR Conference on Research and Development in Information Retrieval*, SIGIR '98, pp. 275–281 (1998).

[3] Manning, C. D., Raghavan, P. and Schütze, H.: *Introduction to Information Retrieval*, Cambridge University Press (2008).

[4] Wang, Y., Davidson, A., Pan, Y., Wu, Y., Riffel, A. and Owens, J. D.: Gunrock: A High-performance Graph Processing Library on the GPU, *Proceedings of the 21st ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '16, ACM, pp. 11:1–11:12 (2016).

[5] Sitaridi, E. A. and Ross, K. A.: GPU-accelerated string matching for database applications, *The VLDB Journal*, Vol. 25, No. 5, pp. 719–740 (2016).

[6] Hon, W., Ku, T., Shah, R., Thankachan, S. V. and Vitter, J. S.: Faster compressed dictionary matching, *Theoretical Computer Science*, Vol. 475, pp. 113 – 119 (2013).

[7] 若月駿尭， 欅惇志， 宮崎純：効率的なテキスト処理を目指した簡潔データ構造を用いるトライ木の GPU 上での実装，第 8 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 2016), C6-5 (2016).

[8] Harris, M., Sengupta, S. and Owens, J. D.: Parallel Prefix Sum (Scan) with CUDA, *GPU Gems 3* (Nguyen, H., ed.), Addison Wesley (2007).

[9] Green, O., McColl, R. and Bader, D. A.: GPU Merge Path: A GPU Merging Algorithm, *Proceedings of the 26th ACM International Conference on Supercomputing*, ICS '12, ACM, pp. 331–340 (2012).

[10] Baxter, S.: moderngpu 2.0, (online), available from ⟨https://github.com/moderngpu/moderngpu/⟩ (accessed 2016-12-05).

[11] Merrill, D. and Grimshaw, A.: High Performance and Scalable Radix Sorting: a Case Study of Implementing Dynamic Parallelism for GPU Computing, *Parallel Processing Letters*, Vol. 21, No. 02, pp. 245–272 (2011).

[12] 小澤佑介，天笠俊之，北川博之：GPU を用いたラベル伝搬法によるグラフクラスタリングの高速化，第 8 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 2016), A8-3 (2016).

[13] 森谷祐介， 欅惇志， 宮崎純：GPU を用いた MapReduce による高精度検索のための高速な重み計算，第 7 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 2015), G3-6 (2015).

[14] Dean, J. and Ghemawat, S.: MapReduce: Simplified Data Processing on Large Clusters, *Proceedings of the 6th Conference on Symposium on Opearting Systems Design & Implementation - Volume 6*, OSDI'04, USENIX Association, pp. 10–10 (2004).

[15] Talbot, J., Yoo, R. M. and Kozyrakis, C.: Phoenix++: Modular MapReduce for Shared-memory Systems, *Proceedings of the Second International Workshop on MapReduce and Its Applications*, MapReduce '11, ACM, pp. 9–16 (2011).

[16] Fang, W., He, B., Luo, Q. and Govindaraju, N. K.: Mars: Accelerating MapReduce with Graphics Processors, *IEEE Transactions on Parallel and Distributed Systems*, Vol. 22, No. 4, pp. 608–620 (2011).