GPU 上で辞書を利用した N-gram クエリ尤度の効率的な事前計算

若月 駿尭† 欅 惇志†,†† 宮崎 純†

† 東京工業大学情報理工学院情報工学系 〒 152–8552 東京都目黒区大岡山 2–12–1
†† 国立研究開発法人科学技術振興機構，ACT-I 〒 332–0012 埼玉県川口市本町 4–1–8
E-mail: †{wakatsuki,keyaki}@lsc.cs.titech.ac.jp, ††miyazaki@cs.titech.ac.jp

あらまし 情報検索で利用される文書処理である，N グラムクエリ尤度モデルに基づいた文書中の N グラムのクエリ尤度を，GPU 上で効率的に動作する簡潔データ構造による辞書を用いた語の整数値 ID 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，効率的に計算する手法について述べる．計算に必要となる N グラムの頻度をデータ並列プリミティブを用いてスレッド間の負荷分散を考慮して求めるとともに，ボトルネックとなりうる文字列操作を辞書を導入することで削減する．評価実験では MapReduce フレームワークを利用したマルチコア CPU による並列計算手法と，提案手法を比較し性能の評価を行った．マルチコア CPU による計算と比べて提案手法は性能が優れており，辞書による高速化の効果として，辞書を用いない GPU 上での提案手法と比べて，3 倍から 4 倍程度の性能向上がみられた．

キーワード GPGPU, N-gram, 辞書, データ並列プリミティブ

1. はじめに

情報検索システムにおいて，ユーザの情報要求に的確に応えるために，クエリと文書の適合度を測る指標が研究されてきた．クエリ尤度モデル [1] はこれに対して言語モデルからアプローチする手法である．特に N グラムモデルによる言語モデルを用いる場合には各文書中の N グラムに対して確率値を計算する必要があり，計算に必要な統計量として N グラムの出現頻度等を必要とする．こうした適合度を用いて大量の文書から高速に検索するためには，これらの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．

一方で，昨今のハードウェアの並列性や分散コンピューティングにより大量のデータを高速に処理することが可能になってきている．それを実現する並列プログラミングモデルとして MapReduce [2] が広く用いられており，転置インデックスの構築や，言語モデルの推定に利用されている [3].

GPU は高い並列性と広いメモリバンド幅を備えたメニーコアプロセッサで，これを汎用的な用途に活用することが取り組まれている．数値計算だけでなく，例としてグラフ処理 [4] や文字列マッチング [5]，接尾辞配列構築 [6] といった用途を目的とした研究がなされている．それらを実現するために提案された GPU 上での汎用的で効率的な処理が，データ並列プリミティブとして活用されている．

GPU を用いてクエリと文書の適合度を測る指標の一つである Okapi BM25 [7] を，簡潔データ構造を用いた辞書 [8] による GPU 上での語の ID への変換 [9] を新たなデータ並列プリミティブとして導入し，効率的に GPU 用いて計算する手法 [10] が提案されている．本稿では，これを拡張し N グラムクエリ尤度モデルによる確率値を，効率的に計算する手法を提案する．また辞書により変換され，ID の並びで表される N グラムを prefix-doubling を用いて効率的にソートする手法について述べる．評価実験では MapReduce フレームワークを利用した，マルチコア CPU による並列計算手法と，提案手法を比較し性能を評価する．

2. 関連研究

2.1 クエリ尤度モデル

クエリ尤度モデル (query likelihood model, QLM) [1] は，文書に関する言語モデル θ_d がクエリを生成する確率である $P(Q|\theta_d)$ を適合度として用いて，文書のランキングを行う情報検索手法である [11]．ベクトル空間モデルの TF-IDF や古典的確率モデルである BM25 と比較して，高精度な検索モデルとして広く用いられている [12]．$P(Q|\theta_d)$ を用いる根拠は，文書 d がクエリ Q に適合する確率 $P(d|Q)$ がベイズの定理を用いて式 (1) のように変形できることに基づく．

$$P(d|Q) = \frac{P(Q|d)P(d)}{P(Q)} \propto P(Q|d) \tag{1}$$

ここで，文書 d の生起確率 $P(d)$ は一様と仮定している．また，検索の際に $P(Q)$ は文書によらず一定であるため定数とみなせる．

当初は確率の推定にユニグラム言語モデルを用いていたが，語の並び順に関する依存性をとらえることのできる N グラム言語モデルを用いることが可能である [13]．このときのクエリ尤度は，$n-1$ 個の語の並び t_1^{n-1} の後に語 t_n が現れる条件付き確率 $P(t_n|t_1^{n-1}, \theta_d)$ の積を用いて表される．

また，大域的な統計量として文書コレクション C に関する言語モデル θ_C を用いることにより，クエリ語が頻度ゼロの文書の適合度がゼロになることを回避する効果や IDF と同様の効果をもたらす，スムージング手法が広く用いられている [11]．線形補間法によるスムージングを施した条件付き確率は式 (2) を用いて推定される．

$$P(t_n|t_1^{n-1}, \hat{\theta}_d, \lambda) = \lambda \frac{f(t_1^n, d)}{f(t_1^{n-1}, d)} + (1-\lambda)\frac{f(t_1^n, C)}{f(t_1^{n-1}, C)} \tag{2}$$

ここで $f(t_1^n, d)$ は文書 d 中に現れる t_1^n の個数とする．

2.2 Suffix-σ

語数が σ 以下の N グラムの頻度を一度の MapReduce ジョブのみで効率的に求める手法として Suffix-σ [14] が提案されている．基本的なアイデアとして以下の三つが挙げられており，過度なデータの転送やメモリの消費を抑える工夫がなされている．

（1） Map ステップにおいて suffix (ある語から末尾までの語の並び) を出力する．出力する suffix は前から σ 語までで打ち切ってよい．

（2） 各 suffix を最初の語を用いて Reducer に分配する．一度の MapReduce ジョブで数えるために必須である．

（3） suffix を辞書順の逆順にソートする．Reduce ステップで順次 suffix を読む際に，確定した N グラムの頻度を残りの suffix を見る前に出力できる．

Suffix-σ はナイーブな手法や，アプリオリ法 [15] に基づく手法と比べて同等以上の性能を示している．

図 1 は $\sigma = 3$ のときの例を表している．各 Mapper にはそれぞれ文書が割り当てられている．たとえば d_2 の 2 語目からの suffix `a x b x` を考えると，2 語目から始まるすべての N グラム `a`，`a x`，`a x b`，`a x b x` が prefix に現れている．Reduce ステップではこれを考慮して N グラムの頻度を数えるため，Map ステップでは suffix のみを出力すればよい．また `a x b x` 等の $\sigma = 3$ 語より多い N グラムの頻度は数えないため，最長 $\sigma = 3$ 語に切り詰め `a x b` を出力する．Map ステップで出力された suffix は辞書順の逆にソートされ，最初の語により Reducer に分配される．Reducer は逐次的に suffix を読みながら，スタックを用いて prefix で表されている N グラムの頻度を記録し，頻度が確定した N グラムを順次出力する．

2.3 GPU による BM25 計算の高速化手法

GPU 上で効率的に実行可能な，簡潔データ構造を用いた辞書 [9] およびデータ並列プリミティブを用いて BM25 を求める手法 [10] が提案されている．データ並列プリミティブとはアルゴリズムの構成要素となる，効率的で汎用的な並列処理のことを指す．この手法は以下の大きく三つのステップに分けられ，それぞれがデータ並列プリミティブの組み合わせにより実装されている．

（1） 各文書から語を抽出する．

（2） 語をソートすることで集約する．

（3） 隣接する語の個数から出現頻度を求め，BM25 の計算を行う．

辞書を用いる場合には，辞書を用いて (2) のソートを行う前に語を ID に変換し，以降の文字列比較コストを削減している．

ソート済みの語のリストからそれぞれの語の出現頻度を求める count 操作の例を次に示す．

```
{a a b b b x} *0
↓ 前の要素と異なる要素を compact 操作により抽出.
{(a,0) (b,2) (x,5)} *1
↓ 次の要素との距離を計算.
{(a,2) (b,3) (x,1)} *2
```

またリスト中の各語について求めた頻度を用いて計算を行いたいときに，`*2` を用いると 3 スレッドがそれぞれ 2 回，3 回，1 回と計算を異なる回数繰り返すことになり負荷の偏りが生じる．これは load balancing search (lbs) [16] 操作により回避することができ，以下のように各語に対応する頻度の配列を生成することで，元の語数である 6 スレッドで等量の計算を行うようにできる．

```
{0 2 5} *1 のセグメント
↓ lbs 操作によりセグメントから配列を生成.
{0 0 1 1 1 2}
↓ {2 3 1} *2 を参照
{2 2 3 3 3 1}
```

このようにして，GPU 上で辞書とデータ並列プリミティブを用いて，各文書中の語の出現頻度や語の出現する文書数等を効率的に求め，BM25 の計算を行う手法となっている．

3. GPU を用いた確率値の推定

本稿では 2.3 節で述べた手法 [10] を拡張し N グラムの統計量算出に対応することでクエリ尤度の推定を行うアルゴリズムを提案する．提案手法のフローを図 2 に示す．各文書中の N グラム t_1^n について，線形補間法によるスムージングが施された式 (2) による条件付き確率を推定するアルゴリズムを考える．

入力となる文書コレクションは前処理が済んでおり，文書と文書の区切りは空行で示され，1 行 1 単語の形式となっているデータを仮定する．

3.1 ステップ 1

Suffix-σ のアイデアから，すべての suffix の開始位置が分かっていれば，すべての N グラムの頻度を数えることができる．これはすべての語の開始位置と一致するため変更の必要はない．

3.2 ステップ 2

N グラムとして比較を行いソートする必要がある．すなわち語の開始位置から最大 σ 個の語を比較する．そのためソートのコストは σ に応じて増加する．また，文書をまたぐ N グラムを compact [16] 操作により除去する．

3.3 ステップ 3

ソート済みの N グラムと文書 ID から，与式 (2) を計算するために必要な 4 種類の統計量を，それぞれ以下の条件による count 操作により求める．図 3 に $\sigma = 2$ のときの実行例を示す．

- $f(t_1^n, d)$:σ 語一致かつ文書 ID 一致
- $f(t_1^n, C)$:σ 語一致
- $f(t_1^{n-1}, C)$:$\sigma - 1$ 語一致
- $f(t_1^{n-1}, d)$:$\sigma - 1$ 語一致かつ文書 ID 一致

$f(t_1^{n-1}, d)$ については $\sigma - 1$ 語一致かつ文書 ID 一致の N グラムを数える必要があるが，t_1^{n-1} で始まる N グラム全体を見たときに文書 ID がソートされていないため，count 操作を適用できない (表 1)．そのため $f(t_1^{n-1}, C)$ を求める際に用いたセグメントごとに文書 ID をソートしてから count 操作を適用す

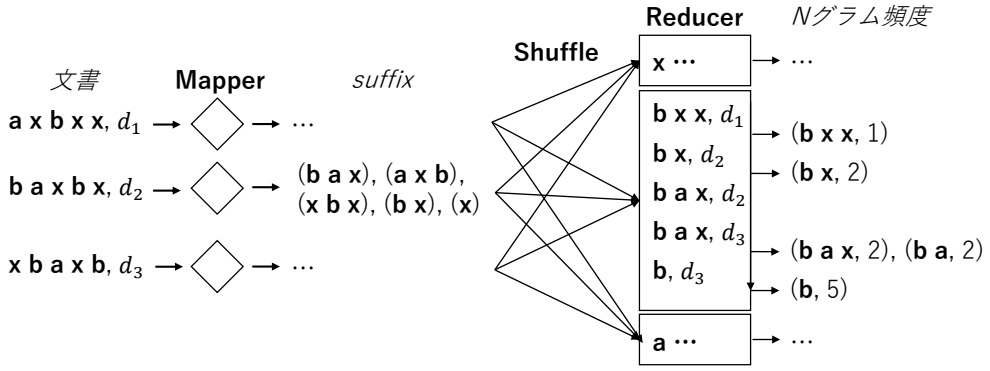

図 1　$\sigma = 3$ のときの SUFFIX-σ による頻度計算の例

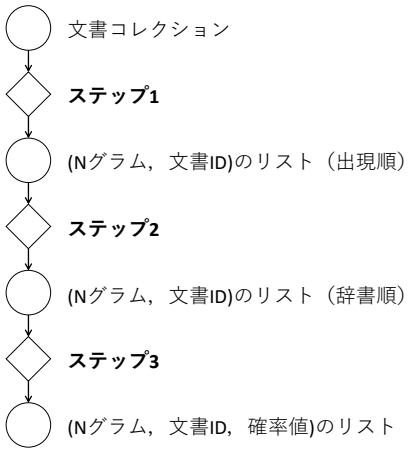

図 2　GPU を用いた確率値の推定フロー

る．このソートには複数のセグメントについて，それぞれソートを行う segmented sort [6] を用いる．count 操作適用後に求めた頻度をソートする前の順に並べ替える．

なお，このステップでは繰り返し同じ N グラム同士の単純な比較を行うことを防ぐために，あらかじめ隣接する N グラムについて，先頭から一致する語の数を記録した配列を用意する．この配列を利用して，隣接する N グラム同士が各 count 操作の条件を満たしているかどうか判定している．

表 1　N グラムの頻度を求める例 ($\sigma = 3$).
d' はセグメントごとにソートされた d,
a，b 等は一つの語を示す.

w_1^n	d	$f(w_1^n, d)$	$f(w_1^n, C)$	$f(w_1^{n-1}, C)$	d'	$f(w_1^{n-1}, d')$
b w z	7	…	…	…	…	…
b x a	2	2	3	6	2	3
b x a	2				2	
b x a	5	1			2	
b x b	7	1	1		5	2
b x c	2	1	2		5	
b x c	5	1			7	1
b y a	3	…	…	…	…	…

以上により計算に必要な統計量が求まったため，条件付確率を計算する式 (2) の計算を行う．各統計量は lbs 操作により対応する N グラムへ分配され，各 N グラムと文書 ID の組について並列に計算される．

4. 辞書による高速化

従来手法では辞書により語を整数値 ID に変換することで，高速な整数値ソートを用い文字列比較コストを削減し高速化を達成していた．N グラムクエリ尤度の計算においても同様に辞書による変換を行うことで高速化が見込まれるが，N グラムとしてソートする必要があるため σ が 2 以上の場合では，すべての N グラムの長さは同一ではあるものの，文字列ソートと同様の課題が生じる．

そこで，ソート対象が suffix であり，ソートに用いる prefix 長が σ と一定であることに着目すると，接尾辞配列構築に用いられてきた prefix-doubling を適用することができる．Karp ら [17] により考案され，Manber ら [18] により接尾辞配列構築に導入された prefix-doubling は，suffix の prefix を反復毎に prefix 長を 2 倍にしながらソートする手法である．原理としては，suffix が長さ k の prefix c_1^k で既にソートされているとき，同じ prefix c_1^k を持つ suffix のなかでの，長さ $2k$ の prefix c_1^{2k} を用いた順序関係は，それぞれの suffix の開始位置から k 文字後の suffix が既に長さ k の prefix c_{k+1}^{2k} を用いてソート済みであることから，これらの suffix の順序関係から得られることに基づいている．なお通常の接尾辞配列構築では文字が最小単位であるが，ここでは整数値 ID の語を最小単位としている．

本研究では，GPU 上で高速に接尾辞配列を構築するアルゴリズム [6] の一部である segmented sort による prefix-doubling アルゴリズムを用いる．完全な接尾辞配列の構築はせず，ソート済み prefix 長が σ に達した時点でソートステップを完了する．擬似コードをアルゴリズム 1 に示す．T は語の ID の並びで表された文書コレクションとなり，出力として T 上での整数値座標で示された σ グラムが辞書順に並んだ V が得られる．

辞書に含まれていない語には未知語として共通の ID を与える．未知語を含む N グラムについては意味をなさない値が出力されるが，他の N グラムの値には影響を及ぼすことはない．

これまでのクエリ尤度モデルについての提案手法をまとめた擬似コードをアルゴリズム 2 に示す．

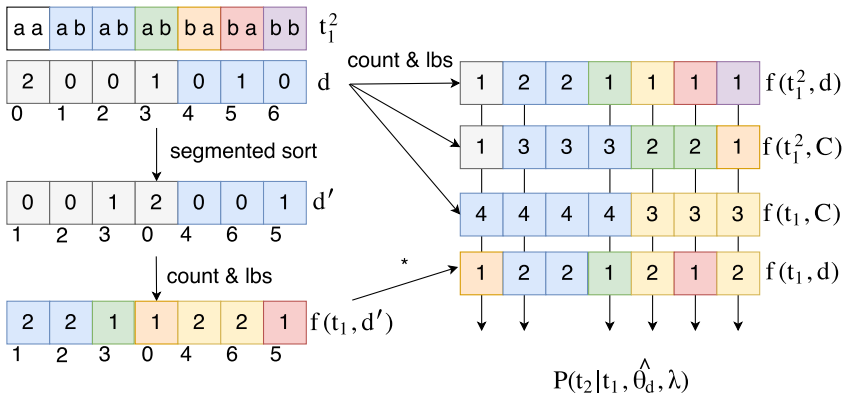

図 3　$\sigma = 2$ のときのクエリ尤度モデル，ステップ 3 の実行例．
文書は `d0: a b a b b, d1: a b a, d2: a a`

Algorithm 1 prefix-doubling アルゴリズム

Require: T, σ
  $K \leftarrow T$
  $n \leftarrow K.size()$
  $V \leftarrow [0, 1, 2, \cdots]$
  sort(K, V)
  $k \leftarrow 1$
  $p \leftarrow 1$ // ソート済み prefix 長
  while $p < \sigma$ **do**
    for $i = 1$ to $n - 1$ **do**
      if $K[i-1] \neq K[i]$ **then**
        $F[i] \leftarrow 1$
      else
        $F[i] \leftarrow 0$
      end if
    end for
    $S \leftarrow$ segment(K)
    $G \leftarrow$ prefix_sum(F)
    for $i = 0$ to $n - 1$ **do**
      $I[V[i]] \leftarrow G[i]$
    end for
    for $i = 0$ to $n - 1$ **do**
      $K[i] \leftarrow I[V[i] + k]$
    end for
    segmented_sort(K, V, S)
    $p \leftarrow p + k$
    $k \leftarrow k \times 2$
    // ちょうど σ 分でソートされた結果にするため，次の比較に用いる差分 k を補正する
    if $p + k > \sigma$ **then**
      $k \leftarrow \sigma - p$
    end if
  end while
  return V

Algorithm 2 提案手法の擬似コード

Require: T, σ
  // ステップ 1
  $W, D_{head} \leftarrow$ compact(T) // 語の先頭の位置を W に抽出，文書の最初の語は D_{head} に 1 を立てる
  $D \leftarrow$ prefix_sum(D_{head})
  // ステップ 2
  if 辞書を用いる **then**
    $W \leftarrow$ dict(W)
    $SA \leftarrow$ prefix-doubling(W, σ)
  else
    sort(W, D) // W から σ 個語を比較
    $SA \leftarrow W$
  end if
  $S \leftarrow$ compact(SA) // 異なる文書 ID を含む N グラムを除去
  // ステップ 3
  $X \leftarrow$ lbs(count(S, D)) // $f(t_1^n, d)$
  $Z \leftarrow$ lbs(count(S)) // $f(t_1^n, C)$
  $W \leftarrow$ lbs(count(S)) // $f(t_1^{n-1}, C)$
  $D' \leftarrow D$
  $P \leftarrow [0, 1, 2, \cdots]$
  segmented_sort(D', P, segment(W))
  $Y' \leftarrow$ lbs(count(S, D')) // $f(t_1^{n-1}, d')$
  for $i = 0$ to $Y'.size() - 1$ **do**
    $Y[P[i]] \leftarrow Y'[i]$
  end for
  $V \leftarrow$ QLM(X, Y, Z, W)
  return (S, D, V)

5. 評価実験

5.1 実験準備

本節では次の 3 種類の手法について比較し評価を行う．次の提案手法を PP, PPD と表記する．

Parallel Primitives (PP)

データ並列プリミティブによる GPU 上で実行する提案手法．

Parallel Primitives with Dictionary (PPD)

PP に辞書による変換処理を加え，prefix-doubling によるソートを行う手法．

また，次の比較手法を MRP と表記する．

MapReduce Phoenix++ (MRP)

マルチコア CPU 向けの MapReduce フレームワークである Phoenix++ [19] 上での MapReduce による並列計算手法．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．

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，計算結果が CPU のメモリ上に書き込まれるまでとした．そのためデータの転送時間 (HtoD: CPU から GPU, DtoH: GPU から CPU) を含む．辞書データ構造は実験の前にあらかじめ構築されており，文書と共に GPU へ転送する．

実験に用いる語彙と文書コレクションは次の方法で用意した．英文 Web 文書約 5,000 万ページからなる TREC ClueWeb09 Category B[(注1)]から，辞書に登録する 1,000 万種類の語彙として英文字のみで構成される単語から出現頻度が高いものを採用した．入力となる文書コレクションは英文字のみで構成される単語を抽出した文書を，それぞれ 100MB，200MB，300MB に達するまでランダムに集め，三つの文書コレクション作成した．そのため文書中には辞書に登録されていない未知語が含まれている．未知語を含む t_1^σ の割合を表 2 に示す．

表 2　未知語を含む t_1^σ の割合

σ	1	2	4	8
100MB	0.57%	1.40%	4.48%	15.8%
200MB	0.51%	1.30%	4.26%	15.0%
300MB	0.55%	1.36%	4.38%	15.5%

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．segmented sort などのソートアルゴリズムや，compact 操作などのデータ並列プリミティブは moderngpu 2.0 [16] に実装されているものを用いた．

5.2 実験結果

$\sigma = 2$，$\sigma = 8$ のときの各手法の実行時間を図 4 に示す．また図 5 に 300MB を入力として，σ を変化させたときの実行時間を示す．今回の条件では σ によらず，辞書を用いた GPU による PPD，辞書を用いない GPU による PP，CPU 上の MapReduce による MRP の順に実行時間が短い結果となった．

(注1)：http://trec.nist.gov/

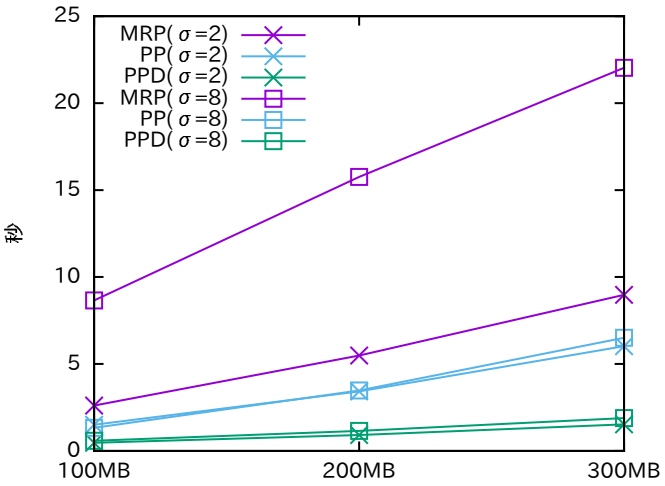

図 4　各手法の実行時間．

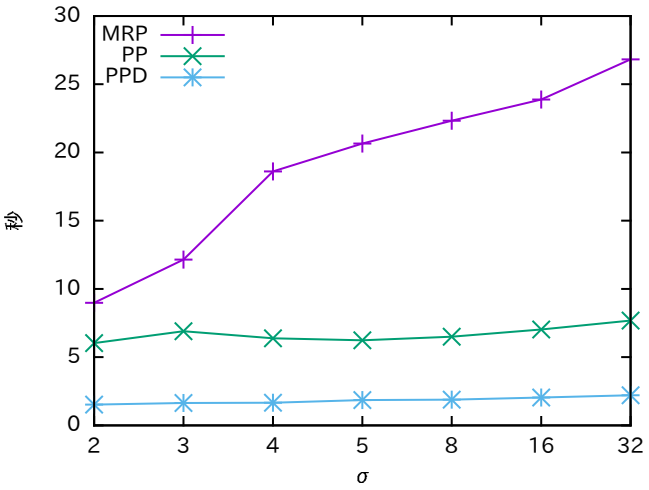

図 5　σ を変化させたときの各手法の実行時間 (300MB)．

入力となる文書コレクションのサイズによる変化としては，サイズが大きいほど PP および PPD は MRP に対する実行時間の差が縮まり，PP と PPD の間ではサイズが大きいほど実行時間の差が広がる傾向が見られる．MRP では σ が大きくなるほど実行時間が増加している．PP と PPD に関しても実行時間への影響は小さいが同様に増加傾向にある．PPD については表 2 に示したように，σ が大きいときには未知語を含む N グラムの割合が大きくなっていることに注意する必要がある．

PP と PPD のステップごとに分けた実行時間を図 6 に示す．図中の Dict は，辞書により語を ID に変換するために要した時間に対応する．ステップ 2 は主にソートが占めており，ステップ 3 には隣接する N グラム間の比較が含まれている．PPD ではそれら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，実行時間が削減されていることが見て取れる．

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，未知語を含む N グラムについての計算ができなくなる点，辞書による変換に要する処理時間が発生する点があげられるが，高速な辞書を用いたことで，全体の実行時間が大幅

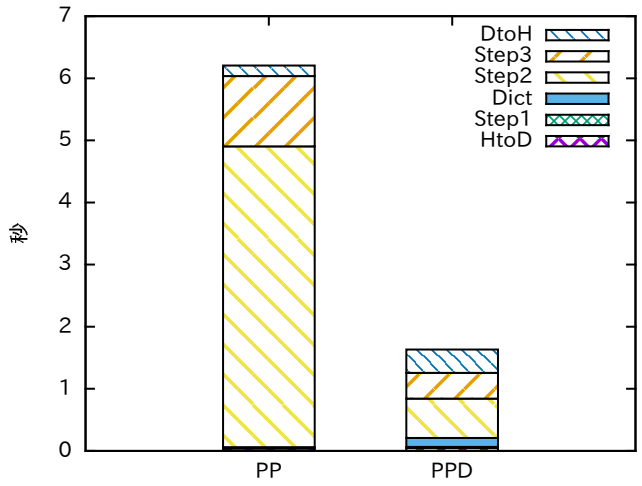

図 6　実行時間の内訳 (300MB, $\sigma = 4$).

に削減にされ，辞書による変換に必要なコストの増加よりも，ソートや文字列比較にかかるコストの減少の効果の方が高くなっている．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と言える．

6. おわりに

本稿では N グラムクエリ尤度を GPU を用いて効率的に事前計算する手法を提案した．GPU 上でデータ並列プリミティブを組み合わせた提案手法は，マルチコア CPU による計算と比べて高い性能であることが判明した．$\sigma = 2$ の場合では，マルチコア CPU による計算と比べて，辞書を用いた手法が 5.5 倍から 6.0 倍，辞書を用いない手法でも 1.4 倍から 1.7 倍高速という結果となった．

GPU 上で簡潔データ構造による辞書を新たなデータ並列プリミティブとして導入することで，ネックとなる文字列比較の削減やソートコストの prefix-doubling による削減といった効果により辞書を用いない GPU 上での手法と比べて 3 倍から 4 倍程度の性能向上を達成した．

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 上で効率的にクエリ尤度モデルに基づく語の重み計算といった文書処理を行うことが可能であることが判明した．辞書は文書処理を GPU 上で効率的に行うためのデータ並列プリミティブとして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，GPU のメモリにすべてが載りきらない大規模な文書コレクションに対する計算法の考案や，より精緻な言語モデルの推定法に基づく計算アルゴリズムなどの考案が考えられる．

謝　辞

本研究の一部は，JSPS 科研費 (JP15H02701，JP16H02908，JP15K20990，JP17K12684)，JST ACT-I の助成を受けたものである．ここに記して謝意を表す．

文　献

[1] Jay M. Ponte and W. Bruce Croft. A language modeling approach to information retrieval. In *Proceedings of the 21st Annual International ACM SIGIR Conference on Research and Development in Information Retrieval*, pp. 275–281, 1998.

[2] Jeffrey Dean and Sanjay Ghemawat. MapReduce: Simplified data processing on large clusters. *Communications of the ACM*, Vol. 51, No. 1, pp. 107–113, 2008.

[3] Jimmy Lin and Chris Dyer. *Data-Intensive Text Processing with MapReduce*. Morgan and Claypool Publishers, 2010.

[4] Yangzihao Wang, Andrew Davidson, Yuechao Pan, Yuduo Wu, Andy Riffel, and John D. Owens. Gunrock: A high-performance graph processing library on the GPU. In *Proceedings of the 21st ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '16, pp. 11:1–11:12, 2016.

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

[6] Leyuan Wang, Sean Baxter, and John D. Owens. Fast parallel suffix array on the GPU. In *Proceedings of the Euro-Par 2015: Parallel Processing - 21st International Conference on Parallel and Distributed Computing*, pp. 573–587, 2015.

[7] Stephen E. Robertson, Steve Walker, Susan Jones, Micheline Hancock-Beaulieu, and Mike Gatford. Okapi at TREC-3. In *Proceedings of The 3rd Text REtrieval Conference*, pp. 109–126, 1994.

[8] WingKai Hon, TsungHan Ku, Rahul Shah, Sharma V. Thankachan, and Jeffrey Scott Vitter. Faster compressed dictionary matching. *Theoretical Computer Science*, Vol. 475, pp. 113 – 119, 2013.

[9] 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.

[10] Toshiaki Wakatsuki, Atsushi Keyaki, and Jun Miyazaki. A case for term weighting using a dictionary on GPUs. In *Proceedings of the Database and Expert Systems Applications: 28th International Conference*, pp. 103–117, 2017.

[11] 江口浩二. 情報検索のための確率的言語モデルに関する動向と課題. 電子情報通信学会論文誌, Vol. J93-D, pp. 157–169, Mar 2010.

[12] Christopher D. Manning, Prabhakar Raghavan, and Hinrich Schütze. *Introduction to Information Retrieval*. Cambridge University Press, 2008.

[13] Fei Song and W. Bruce Croft. A general language model for information retrieval. In *Proceedings of the 1999 ACM CIKM International Conference on Information and Knowledge Management*, pp. 316–321, 1999.

[14] Klaus Berberich and Srikanta Bedathur. Computing n-gram statistics in MapReduce. In *Proceedings of the 16th International Conference on Extending Database Technology*, pp. 101–112, 2013.

[15] Rakesh Agrawal and Ramakrishnan Srikant. Fast algorithms for mining association rules in large databases. In *Proceedings of the 20th International Conference on Very Large Data Bases*, VLDB '94, pp. 487–499, 1994.

[16] Sean Baxter. moderngpu 2.0. `https://github.com/moderngpu/moderngpu/`.

[17] Richard M. Karp, Raymond E. Miller, and Arnold L. Rosenberg. Rapid identification of repeated patterns in strings, trees and arrays. In *Proceedings of the Fourth Annual ACM Symposium on Theory of Computing*, STOC '72, pp. 125–136, 1972.

[18] Udi Manber and Gene Myers. Suffix arrays: A new method for on-line string searches. In *Proceedings of the First An-*

nual ACM-SIAM Symposium on Discrete Algorithms, pp. 319–327, 1990.

[19] Justin Talbot, Richard M. Yoo, and Christos Kozyrakis. Phoenix++: Modular mapreduce for shared-memory systems. In *Proceedings of the Second International Workshop on MapReduce and Its Applications*, MapReduce '11, pp. 9–16. ACM, 2011.