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

若月 駿尭† 欅 惇志†† 宮崎 純††

† 東京工業大学工学部情報工学科 〒152–8550 東京都目黒区大岡山 2-12-1
†† 東京工業大学大学院情報理工学研究科 〒152–8550 東京都目黒区大岡山 2-12-1
E-mail: †{wakatsuki,keyaki}@lsc.cs.titech.ac.jp, ††miyazaki@cs.titech.ac.jp

あらまし 大量の文書に対するテキスト処理を GPU を用いて効率的に行うために，GPU の特性を考慮した省メモリかつメモリへのランダムアクセスの少ない辞書データ構造の実装方式の提案を行う．GPU を用いたテキスト処理を効率的に行うためには，単語を ID に変換する大規模な語彙集合を扱う辞書データ構造が必要である．提案手法は状態遷移表 (STT) によるトライ木と比較して 40 分の 1 以下のメモリ使用量であるとともに，辞書に含まれる語彙数が多い状況では STT より実行時間が短くなることが判明した．

キーワード 辞書，記号化，GPGPU，情報検索

1. はじめに

コンピュータの普及や Web の隆盛により，大量の電子文書が日々作成されている．これら大量の文書を利用するためには，文書中のテキストを高速に処理する必要がある．しかし，近年シングルコアプロセッサの性能向上は鈍化しており，逐次実行による大幅な高速化は期待できない．そのため多数のコアを一つのチップに搭載することにより性能向上を目指す方向に進んでいる．処理の並列化を行い，これら多数のコアを搭載するハードウェアを活用することで，高速なテキスト処理を実現することができると期待されている．

多数のコアを搭載したプロセッサは，メニーコアプロセッサと呼ばれている．GPU は 3D グラフィックスの描写を目的としたメニーコアプロセッサであり，安価であるため広く普及している．この GPU を汎用的な用途に応用する GPGPU が研究開発されている．GPU はスーパーコンピュータのアクセラレータとして採用される (注1) など科学技術計算の応用が目立っているが，データ処理に重点をおくデータインテンシブな分野においても GPU を利用した研究が行われている [1,2].

効率よく GPU を活用しテキスト処理を行うためには，以下のような課題がある．例えば高精度な情報検索で用いられる統計量である各文書中の単語の出現頻度 (term frequency, TF) を求める際には，各スレッドが担当範囲の文書中での単語の出現を求めた後に，単語をキーとしてデータを配分し各単語の統計量を求める．このデータ配分のために単語をキーとしたソートが用いられる．しかし GPU に適した高速な並列基数ソートは，単語などの可変長キーによるソートに対応できず，他の可変長キーソート手法では各スレッドに不規則な負荷が生じるなど [3] 効率的な文字列ソートは困難であり整数値ソートに性能が劣っている．

(注1)：http://www.top500.org/

そこで，自然言語文書における最小解釈単位は単語であるため，単語の語彙集合を扱うデータ構造としての辞書を用い，単語を一意な整数値の ID に変換することで，代わりに整数値ソートで済むようになる．GPU を用いた整数値ソートについては数多くの研究がなされており，広く使われている高速なライブラリ [4] が存在する．

GPU が搭載するメモリは一般的にデータ処理に用いられるサーバが搭載するメモリと比べると少量である．また Web 文書には固有名詞，人名，URL，メールアドレス等が含まれるため，大規模な語彙集合を，省メモリで格納可能な辞書データ構造が求められる．しかし GPU 上で用いることのできる辞書データ構造，特に簡潔データ構造を用いるものについては，ほとんど研究されておらず効率的な手法は明らかでない．

本稿では GPU 上で効率的に利用できる辞書データ構造について議論する．簡潔データ構造を用いるトライ木の参照アルゴリズム [5] を GPU が不得手なメモリへのランダムアクセスが少なくなるよう効率的に実装する手法を提案する．評価実験により，非圧縮なアルゴリズムと比べて辞書のメモリ使用量が少なく，語彙数が多い状況では，より高速に単語から ID への変換が可能であることを明らかにする．

本稿の構成は以下の通りである．2. 節では用語の定義および本研究の基礎をなす技術について述べる．3. 節では GPU 上で簡潔データ構造を用いたトライ木による辞書の効率的な実装手法を提案する．4. 節では提案手法を一般的な手法と比較する実験を行うことにより評価する．5. 節では本研究に関連する研究について述べる．6. 節では結論および今後の課題について述べる．

2. 基礎的事項

2.1 用語の定義

本稿で用いる用語と記号の定義を行う．

長さ n のテキスト T を文字 c の並び $T = c_1c_2c_3 \ldots c_n$ と定義

する．c は有限な集合であるアルファベット Σ の要素であり，Σ の要素数 $|\Sigma|$ を σ で表す．英単語を考えたとき $\Sigma = \{a, b, \ldots, z\}$, $\sigma = 26$ である．また，自然言語で書かれた文書は有限の語彙から単語を並べることによって構成されている．語彙 L を定義し単語 w は語彙 L の要素 $w \in L$ であると定義すると，単語数 m のテキスト T は単語 w の並び $T = w_1 w_2 w_3 \ldots w_m$ と表すことができる．

2.2 辞　　書

辞書はキーと呼ぶ文字列の集合を扱うデータ構造である．キーの集合とその文字列に関する付帯情報 (ID や説明など) をあらかじめ辞書に登録する．その後この辞書を利用して，あるキーが辞書に含まれているか否かを判定することができ，含まれているのであればその付帯情報を取り出すことができる．

巨大な語彙を扱う必要のある分野，または使用できるメモリの限られた環境下では，低速な二次記憶ではなくメインメモリ，さらにはより高速なプロセッサのキャッシュに収めることができるような，メモリ使用量が少なく，かつ高速な操作が行える辞書が理想である．Martínez-Prieto らは理論上だけではなく実用的な実際のメモリ使用量と実行時間に着目し，さまざまな圧縮辞書データ構造の比較実験を行っている [6].

以降では辞書を単語の集合を扱うデータ構造として扱い，付帯情報は単語の ID とする．

2.2.1 辞書符号化

語彙 L を辞書を用いて表現し，各単語 w に一意な整数値の単語 ID を割り当てる変換 $f_L : w \to i$ を考える．テキスト T を文字 c の並びとして表現するのではなく，f_L を用いて単語 w の ID i の並び $T = i_1 i_2 i_3 \ldots i_m$ として表現する．そうすると二つの単語の等価を確認する処理では，低速な文字列同士の比較ではなく，高速な整数値同士の比較として処理できる．また，テキストが極端に短い単語で構成されていない限り，テキストを格納するために必要なメモリ領域は少なくなる．これを辞書符号化と呼ぶ．

森谷らが行った GPU を用いた MapReduce による索引語重み付けにおいて，Shuffle ステップ時の語のソートにおける文字列比較が，処理時間全体の大部分を占めていることが報告されている [7] ことからも，辞書符号化を行うことで実行時間を減少させることが可能であると期待される．Bazoobandi らは RDF に対する辞書符号化のための，動的に辞書が更新される状況下での，メモリ使用量の少ないコンパクトなトライ木を提案している [8].

2.2.2 STT を用いたトライ木

トライ木 [9] は可変長キーを用いて情報を取り出すためのデータ構造である．トライ木はエッジに文字 c のラベルが付与されている木構造である．キーを一文字ずつ取り出し，木のルートからその文字に対応するラベルを持つエッジをたどりノードを移動することにより，目的のキーの情報を取り出すことができる．例として aba, ba, bb, cb, cc をキーとしたトライ木の具体例を図 1 に，状態遷移表 (state transition table, STT) を表 1 に示す．

この STT によるトライ木の実装は最も高速 [8] であるが，表

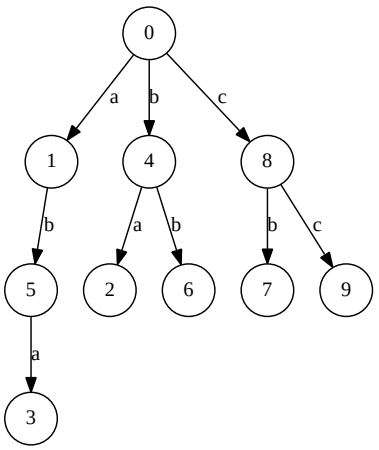

図 1　トライ木

表 1　状態遷移表

N	a	b	c
0	1	4	8
1	-1	5	-1
2	-1	-1	-1
3	-1	-1	-1
4	2	6	-1
5	3	-1	-1
6	-1	-1	-1
7	-1	-1	-1
8	-1	7	9
9	-1	-1	-1

1 にも現れているように遷移先が -1，すなわち遷移できないことを示すセルが占める割合がかなり多く，必要なメモリ領域が大きいという課題がある．STT に必要なメモリ領域はトライ木のノード数を t，ノード番号を 4 バイト整数で表すとすると $4\sigma t$ バイトとなる．

2.2.3 簡潔データ構造を用いたトライ木

本研究ではメモリ使用量と実行速度のバランス，GPU への適合性を考慮し Hon らによる Prefix Matching アルゴリズム [5] をもとにすることにした．以降では当アルゴリズムについて述べる．

当アルゴリズムは簡潔データ構造を用いたトライ木により Prefix Matching を行う．トライ木のノードは，対応する接頭辞を逆順に並び替えた文字列 (reverse prefix) の，辞書順を用いてルートを 0 として順序付けられている．この順序をノードを表す番号とし，各接頭辞の ID としても用いる．図 1 に各ノードに前述の方法で番号を付けた aba, ba, bb, cb, cc を持つトライ木の図を示す．

各文字ごとに，あるノードにおいてその文字で次のノードに遷移可能な場合は 1，そうでない場合は 0 をノードの番号順に並べたビット列を構築する．これらのビット列を利用してノード x から文字 c を用いて遷移した先の子ノード x' を求める方法について説明する．

まずノード x から文字 c で遷移可能であるか否かは c のビット列 (B_c) の x 番目のビットを見ることで判断する．ノード x に対応する接尾辞を P とすると，P の後に文字 c をつなげてできる文字列は Pc と表せる．Pc が辞書に含まれているならば，接尾辞 Pc に対応するノード x' が存在する．ここで各ノードは reverse prefix の辞書順に並んでいるため，ノード x は文字 c で遷移できるノードの中で，何番目であるか B_c を調べることで求めることができる．この関数を $\mathrm{rank}(x, B_c)$ と表す．すなわち文字 c で始まるノードの中で何番目か分かり，P を逆順に並び替えた文字列を Q とすると，求めたいノードの reverse prefix は cQ であるので，文字 c で始まる最初のノードの番号 $C[c]$ を予め記憶しておくことで，cQ のノード番号は $x' = \mathrm{rank}(x, B_c) + C[c]$ と求められる．各ノードから遷移する

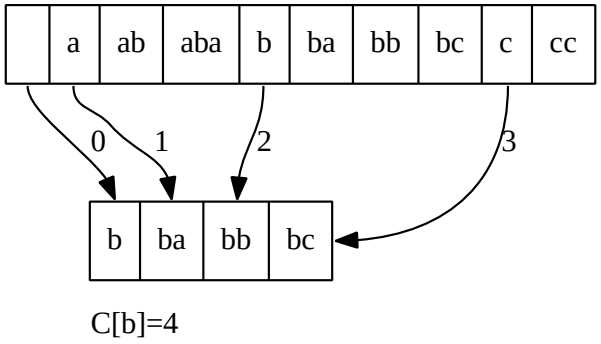

図 2　各ノードから b で遷移する際の例

様子を図 2 に示す．以下に遷移先ノード番号の計算例を示す．

- ルートノード 0 から文字 b で遷移した先の子ノード

$x' = \mathrm{rank}(0, B_b) + C[b] = 0 + 4 = 4$

- ノード 8 から文字 b で遷移した先の子ノード

$x' = \mathrm{rank}(8, B_b) + C[b] = 3 + 4 = 7$

rank 関数をサポートするビット列のデータ構造として RRR [10] と呼ばれる簡潔データ構造が用いられている．この手法は $O(1)$ 時間で rank 関数が実行でき，$\log \binom{t}{n_c} + o(n_c)$ ビットのメモリ領域で格納できる．ここで t は全てのノード数, n_c は文字 c で遷移可能なノード数である．

この結果を用いることで簡潔データ構造によるトライ木は，ある文字列 P に対して文字列から ID を求める操作を $O(|P|)$ 時間，トライ木を $tH_0(X) + O(t)$ ビットのメモリ領域で格納できることが示されている [5]．ここで $H_0(s)$ は文字列 s の 0 次経験エントロピー，X はトライ木の XBW string である．

2.2.4　rank 関数をサポートするビット列

RRR はまずビット列を長さ t のブロックに分割し，各ブロックはブロック内の 1 の数，クラス k により分類される．各クラス k に含まれる要素は $\binom{t}{k}$ 個である．各ブロックはクラス k と要素の番号 r の組 (k, r) で表現される．組 (k, r) と元のビット列間の変換はテーブルを用いる方法と，二項係数を用いて計算する方法 [11] がある．k は $\lceil \log t + 1 \rceil$ ビットで配列 K に，r は $\lceil \log \binom{t}{k} \rceil$ ビットで配列 R に格納する．k は固定長なので k を格納する配列 K を用いて任意のブロックの k をただちに取得することができるが，r は各ブロックの k により長さが定まる可変長なので配列 K から目的のブロックまで k を取得しポインタを計算する必要がある．加えて，ビット列の先頭からのランクとポインタを持つスーパーブロックを s ビットごとに一つ用意する．このデータ構造を用いて $\mathrm{rank}(x, B_c)$ を求めるアルゴリズムを Algorithm 1 に示す．

Algorithm 1 $\mathrm{rank}(x, B_c)$ を求めるアルゴリズム

```
1: function RANK(x, B_c)
2:     u ← x/s                          ▷ x の属するスーパーブロック
3:     o ← rank[B_c][u]                 ▷ ビット列の先頭からのランク
4:     p ← pointer[B_c][u]              ▷ ポインタ
5:     v ← x/t                          ▷ x の属するブロック
6:     for i = u × s/t to v − 1 do
7:         o ← o + K[i]                 ▷ ランクを加算
8:         p ← p + ⌈log (t choose K[i])⌉   ▷ ポインタを加算
9:     end for
10:    k ← K[v]                         ▷ ブロック v の組 (k, r) を取得
11:    r ← R(p, ⌈log (t choose K[v])⌉)  ▷ p ビット目から ⌈log (t choose K[v])⌉ ビット読み出し
12:    b ← Decode(k, r)                 ▷ 組 (k, r) を元のビット列に復元
13:    o ← o + Popcount(b, x mod t)     ▷ x までの 1 の数を加算
14:    return o
15: end function
```

2.3　GPGPU

3D グラフィックスの描写を目的として浮動小数点演算性能とメモリバンド幅を向上させている GPU (graphics processing unit) [12] を，3D グラフィックス以外のこれまで CPU が使用されてきた用途に利用する GPGPU (General-purpose computing on GPU) が研究されている．

GPGPU のための主なプログラミングモデルとして，標準規格である OpenCL と NVIDIA による独自規格の CUDA [12] がある．本稿では NVIDIA 製の GPU を使用し，GPU のアーキテクチャを考慮した実装を試みるため CUDA を使用する．実験で使用する NVIDIA GeForce GTX 970 を例として GPU のアーキテクチャについて述べる．

2.3.1　GPU のアーキテクチャ

GPU は ストリーミングマルチプロセッサ (SM) を複数束ねた構成になっており，例えば本研究で使用する GTX 970 には 13 基の SM が搭載されている．各 SM には 128 基の CUDA Core と レジスタ，キャッシュ，ワープスケジューラ等が含まれる．GTX 970 のコア数は合計で 1664 となる．GPU で実行されるプログラムであるカーネルは CPU から起動され複数の SM において並列に実行される．複数のスレッドはブロックという単位でまとめられ，ブロック内のスレッド数とブロック数を指定して起動される．各スレッド，ブロックには ID が割り当てられ，カーネル内からこの値を用いることができる．

2.3.2　GPU のメモリ階層

GTX 970 には GPU が利用できるメモリ領域としてボード上に 4GB の DRAM が搭載され，理論上では 224GB/s という高いバンド幅を持つ．しかし，理論値に近いスループットが達成できるのはある程度長い範囲をシーケンシャルアクセスする場合のみであり，細かい粒度でのランダムアクセスではスループットが低下する．ランダムアクセスの際のスループットは，一度のアクセスの際のデータサイズが小さいほど，ランダムアクセスの範囲となるメモリ領域が広いほど低下する [13]．

チップ内にあるメモリ領域として L2 キャッシュ，Unified L1/Texture キャッシュのほか，同一ブロック内のスレッドはそれらの間で共有されるメモリ領域であるシェアードメモリを利用できる．これらのメモリ領域は高いバンド幅と低いレイテンシを持つが容量は限られている．

2.3.3　ワ　ー　プ

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ており，ワープ内のすべてのスレッドは同じ命令が実行さ

Algorithm 2 単語単位で並列にトライ木をたどるアルゴリズム

```
1: i ← blockIdx.x × blockDim.x + threadIdx.x
2: while i < m do
3:     b ← begin[i]
4:     l ← length[i]
5:     node ← 0
6:     for k = 0 to l − 1 do
7:         c ← string[b + k]
8:         node ← ChildNode(node, c)
9:     end for
10:    result[i] ← node
11:    i ← i + blockDim.x × gridDim.x
12: end while
```

れる．そのためプログラム中の分岐により各スレッドが異なる実行パスをたどっている場合，各スレッドは自らの実行パス以外の命令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とになる．これを warp divergence と呼び，性能低下の原因となる．

DRAM へのアクセスは 32, 64, 128 バイト のメモリトランザクションにより行われる．ワープ内のスレッドのメモリアクセスが上記のメモリトランザクション内に収まるとき，メモリアクセスはコアレスされると呼び，最も効率が良いのはワープ内の各スレッドの 4 バイトアクセスが一つの 128 バイトメモリトランザクションで行われる場合である．

3. 提 案 手 法

3.1 テキストの表現

本稿では辞書符号化を行うテキストはあらかじめ前処理等が行われ，各単語の位置と長さが判明していると想定する．具体的には図 3 に示すように string 配列に処理するテキスト全体，begin 配列に各単語の位置，length 配列に各単語の長さがそれぞれ記録されているとする．

3.2 並列化手法

GPU の利用にあたっては GPU の持つ 多数のコアを用いて並列に処理を行うことによって性能を引き出す必要がある．トライ木を用いて ID への変換をする際には単語の始めの文字から 1 文字ずつ遷移していくことで ID が求まる．それぞれの単語は独立に辞書を用いて ID への変換が可能である．ゆえに各スレッドが 1 単語ずつ変換処理を行うことにより多数のコアを有効活用することができる．

単語単位で並列に処理を行うアルゴリズムを Algorithm 2 に示す．全スレッド数 N 並列に単語の処理を行う．各スレッド i は k を自然数として，入力テキスト全体の単語数以下の $i+kN$ 番目の単語の処理を担当する．隣り合うスレッドは隣り合う単語を処理するため，配列 begin, length, result へのアクセスは必ずコアレスアクセスになり効率的である．

3.3 遷 移 関 数

トライ木の実装にあたっては， Algorithm 2 に含まれる，あるノード x から入力の文字 c による遷移先の子ノード x' を求める遷移関数 ChildNode をどのように実現するかにより，性能やメモリ使用量は大きく変わる．GPU 上で動作させる遷移関数として求められる点を以下にまとめる．

Algorithm 3 提案手法による遷移関数

```
1: function ChildNode_Proposed(x, c)
2:     if getbit(x, B_c) = 1 then
3:         x' ← rank(x, B_c) + C[c]
4:     else
5:         x' ← −1
6:     end if
7:     return x'
8: end function
```

- 比較的少量なメモリで利用可能であること
- ランダムアクセスの少ない手法であること
- キャッシュを有効活用できる手法であること
- 実行パスの分岐が少ないアルゴリズムであること

ゆえに本研究では省メモリな遷移関数を実現する 2.2.3 節にて述べたアルゴリズムを採用し，新たな遷移関数 Algorithm 3 を提案する．

3.4 実　　装

2.2.4 節で述べたビット列の実装として CPU 上で動作するライブラリでは sdsl [14] などいくつか存在するが，GPU 上で利用できるライブラリは存在しない．ブロックサイズなどの各パラメータは理論上ビット列の長さ n に依存して決まるが，実装するうえで可変なパラメータで効率的な実装をすることは難しいため，固定したパラメータが用いられる．まずブロックサイズ t は 15 ビットとする．これによりブロックのクラス k がとりうる値の範囲 $[0, \ldots, 15]$ は 4 ビット整数を用いて表現できるため無駄が生じない．スーパーブロックは 16 ブロックごとに一つ置く．このときランク，ポインタ，ブロックのクラス 16 個の合計が 16 バイトとなる．これらを一つのサイズ 16 バイトの構造体としてまとめて格納することで，CUDA のベクトルデータ型による一度の 16 バイトアクセスで必要なデータをレジスタに取得できるため効率的である．

符号列から元のビット列の復号に関してはテーブルを用いて行う．このテーブルはエントリ数 2^{15}，エントリサイズ 2 バイトのため合計で 64 キロバイトである．

これらのデータは繰り返し用いられるため Unified L1/Texture キャッシュにキャッシュするよう設定する．一方入力となる begin, length, string 配列は一度アクセスした後に再びアクセスされることはないのでこれらのデータはキャッシュしないよう設定し，辞書データがキャッシュから追い出されることを防ぐ．

各クラスの r が要するビット数，テーブルにアクセスする際のオフセットは定数としてシェアードメモリにキャッシュする．それぞれ要素数は 16 なのでバンクコンフリクトは生じない．

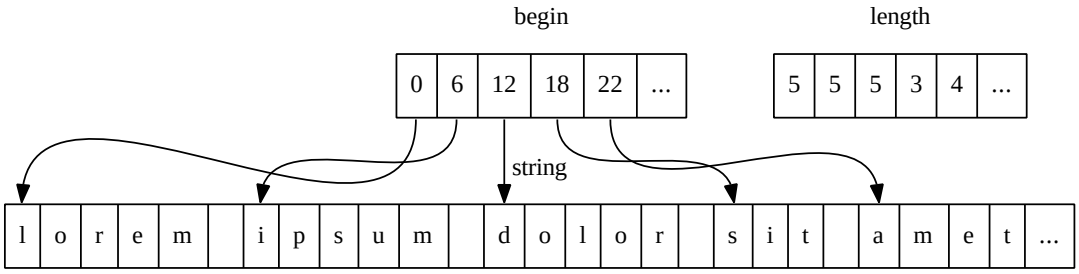

図 3　テキストの表現

Algorithm 4 状態遷移表 (STT) による遷移関数

```
1: function ChildNode_STT(x, c)
2:     x' ← STT[x][c]
3:     return x'
4: end function
```

4. 評価実験

4.1 実験準備

実験に用いる語彙は TREC の公開する ClueWeb09 Category B に含まれる英文 Web 文書 5,000 万ページから，ノイズとなる単語を除外して，4,000 万種類の単語とその出現頻度を抽出することで用意した．なお，単語は全て小文字化処理されている．すなわち $\Sigma = \{a, b, \ldots, z\}$，$\sigma = 26$ である．

各実験において k 単語を辞書として使用する際には 4,000 万種類の単語からなる語彙集合 L から出現頻度の高い上位 k 単語を取り出した部分集合 $L_k \subset L$ を使用する．

入力として与える単語数 m のテキストとして以下の二種類を使用する．

- 出現頻度による合成テキスト T_{freq}
- 一様分布による合成テキスト T_{unif}

T_{freq} は出現頻度に基づく離散分布により L_k から m 回復元抽出，T_{unif} は一様分布により L_k から m 回復元抽出して得た合成テキストである．

本稿では GPU を用いたテキスト処理の一部として組み込むことを想定し処理対象のテキストおよび辞書はあらかじめ GPU のメモリに配置されているとする．そのため以降の実験結果における実行時間は CPU ↔ GPU 間のデータ転送時間を含まない．また，辞書として使用する単語は頻繁に更新されないことを想定する．従って，一度構築した辞書は再構築しない．

状態遷移表 (STT) を用いた遷移関数の実現法 (Algorithm 4) と，提案手法による遷移関数の実現法 (Algorithm 3) を比較する．並列化手法は Algorithm 2 を双方に用い，遷移関数のみそれぞれの手法に変更した．

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．

4.2 メモリ使用量

図 4 に辞書データ構造のみのメモリ使用量を示す．辞書のメモリ使用量について提案手法は STT と比べて大幅に省メモリである．辞書に含まれる単語の語彙数が 1,000 万，語彙サイズ 100MB のとき，辞書のメモリ使用量は STT は 2.9GB であるのに対して，提案手法では 58MB である．

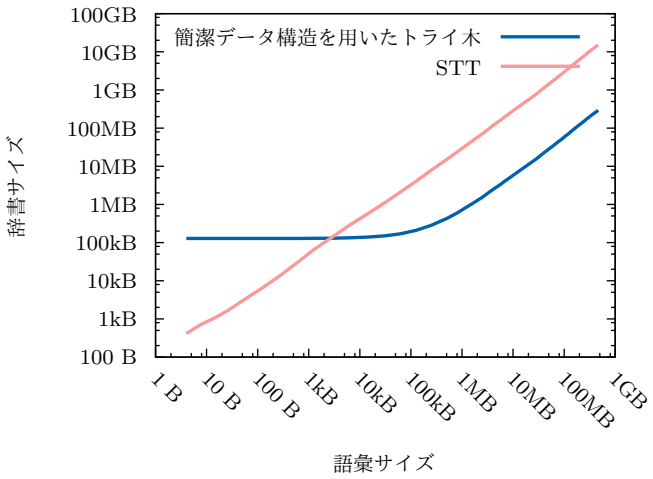

図 4　辞書データ構造のメモリ使用量

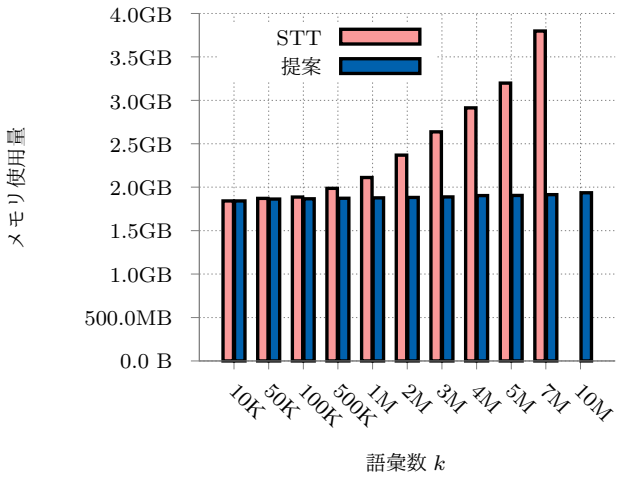

図 5　全体のメモリ使用量，T_{freq}，$m = 100$M

図 5 に辞書やテキスト等，評価実験において使用された合計の GPU メモリ使用量を示す．入力として単語数一億のテキスト T_{freq} を与えており，入出力の占めるメモリ領域は約 1.7

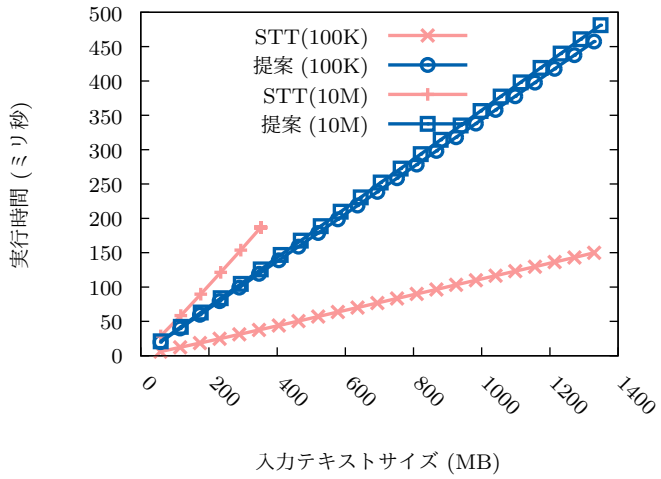

図 6　入力テキストサイズによる性能特性

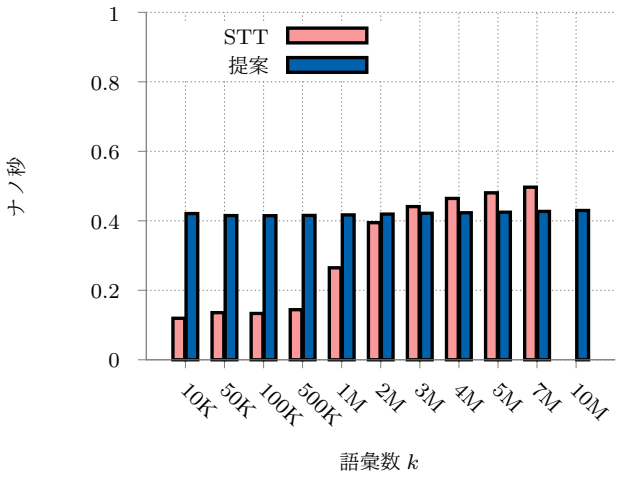

図 7　一文字あたりの実行時間，T_{freq}, $m = 100$M

GB である．実験に使用した GPU の搭載するメモリ容量は 4GB であり，語彙数 10M 以降では STT は GPU メモリの不足により実行できない．一方提案手法は語彙数 10M の場合でも 58MB のメモリ使用量であるため，より多くの語彙数にも対応でき，残りのメモリ領域をより多くのテキストを格納するためにも使用することができる．

4.3　入力テキストサイズによる性能特性

次に実行時間について示す．ブロック数，スレッド数については複数の組み合わせで実行し最短の実行時間を採用した．図 6 は語彙数 100K (100,000) と 10M (10,000,000) の場合について，入力テキストサイズを約 50MB ずつ増加させたときの STT と提案手法の実行時間をグラフに示したものである．各系列の右端の点は GPU メモリを超過しない最大の入力テキストサイズを処理したときの実行時間となっている．

いずれの場合についても実行時間は入力テキストサイズに比例している．STT は語彙数によって実行時間の変化が大きいが，提案手法は語彙数による実行時間の変化が小さい．また，語彙数 10M の場合の STT は辞書の占めるメモリ使用量が大きいため処理できる入力テキストサイズは最大で約 350MB となっている．

4.4　語彙数の大小による性能特性

辞書に載せる語彙数を変化させたときの実行時間について示す．実行時間は処理する文字数に依存するため一文字あたりの実行時間を示している．入力として単語の出現頻度を考慮し実際のテキストを模した単語数一億のテキスト T_{freq} を与え，ID に変換したときの実行時間を図 7 に示す．語彙数が 3M より少ないときは STT が提案手法より高速であるが，語彙数 3M の時点で逆転し提案手法の実行時間の方が短くなる．今回実験した語彙数の範囲では STT は語彙数が増えるに従って実行時間が増大していくが，提案手法はゆるやかな上昇にとどまっている．

4.5　テキスト特性による性能特性

入力として一様分布で作られた単語数一億のテキスト T_{unif} を与え， ID に変換したときの実行時間を図 8 に示す．STT，

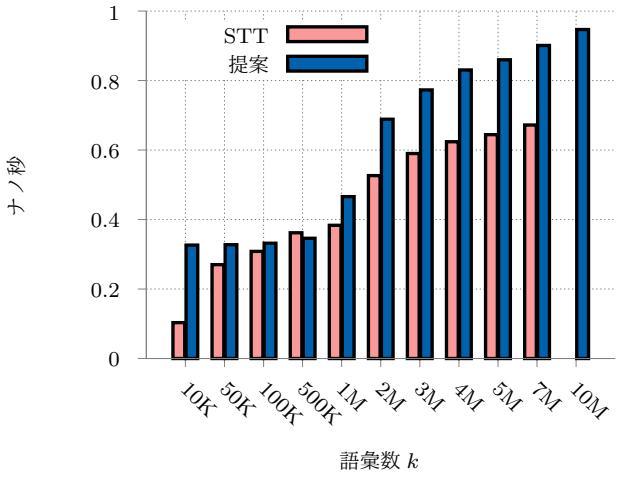

図 8　一文字あたりの実行時間，T_{unif}, $m = 100$M

提案手法ともに語彙数が増加すると実行時間が長くなり，増加率は T_{freq} と比べて大きい．この差はテキストの性質の差によるものと考えられる．一様分布では語彙に含まれる出現頻度の低い単語も全て同じ確率で抽出される．出現頻度の低い単語は単語長が長いことが多く，結果としてテキストに含まれる単語長のばらつきが大きくなる．テキスト中の単語長の平均および分散を表 2 に示す．T_{unif} は語彙数が増加すると単語長の分散が増加する．

単語長のばらつきはスレッド間の負荷のばらつきを生じさせる．特にワープ内のスレッドは同期して実行されるため，ワープ内のすべてのスレッドはワープ内でもっとも長い単語を処理しているスレッドが終了するまで待機することとなる．そのため単語長のばらつきが大きいほど待機状態となるスレッドが増加し実行効率が低下する．メモリアクセスのみで遷移可能な STT と比べて提案手法はランク計算に必要な演算量が多いため実行効率低下の影響がより大きく，実行時間が増加したと考えられる．

表 2　テキスト中の単語長の平均，分散

語彙数		10K	100K	1M	10M
T_{freq}	平均	4.55	4.78	4.84	4.86
	分散	6.34	6.91	7.09	7.24
T_{unif}	平均	6.66	7.12	7.84	9.44
	分散	6.42	7.04	8.32	17.2

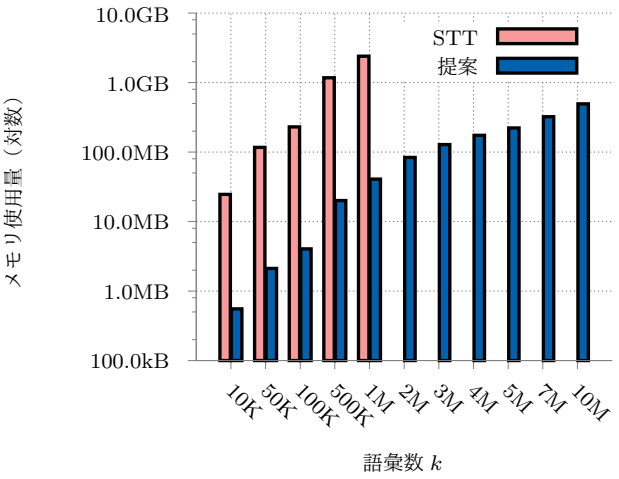

図 9　辞書データ構造のメモリ使用量，$\sigma = 256$

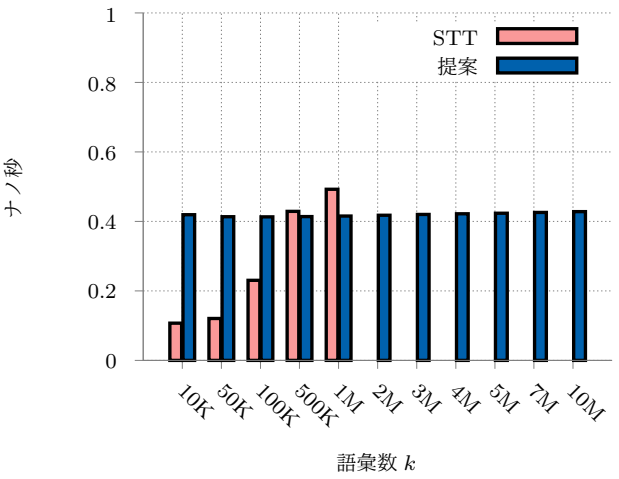

図 10　一文字あたりの実行時間，T_{freq}, $m = 100$M, $\sigma = 256$

4.6　文字種が 256 種類の場合

これまで文字種は $\sigma = 26$ 種類として実験を行ってきたが，数字や記号を含む単語も語彙として扱いたい場合や英語以外の単語を語彙として扱いたい場合には任意の 1 バイトを文字として処理する方法が考えられる．1 バイトで表すことのできる文字種は最大で 256 個のため $\sigma = 256$ とした場合の実験を行った．ただし実験に用いたデータはこれまでと同様に英単語のみである．図 9 にメモリ使用量，図 10 に実行時間を示す．σ が 26 から 256 に増加したためメモリ使用量はどちらの手法も約 10 倍に増加している．提案手法と STT の実行時間が逆転する語彙数は 500K となり STT の実行時間はメモリ使用量に依存していることが分かる．

5.　関 連 研 究

田部井らは木構造の簡潔データ構造である LOUDS を利用したトライ木を用いて GPU 上での並列探索を試みている [15]．並列化手法として各クエリを並列に処理するクエリ並列と，トライ木の各深さにスレッドを割り当て，パイプライン的に並列化するトライ分割の二種類を提案している．

ゲノムデータを対象とした評価を行っており，$\sigma = 5$ で辞書に使用するゲノムパターン数は最大で 1M まで実験が行われている．実験結果としてクエリ並列がトライ分割よりも高速であり，逐次アルゴリズムと比べて最大で 35 倍程度の高速化を達成している．しかし GPU を用いてどのような手法を用いると高速化が達成できるのか明らかではなく，GPU の理論的計算モデルや GPU アルゴリズムの理論的解析が求められると結論付けている．

Chacon らはゲノム配列を対象として FM-Index と呼ばれる全文索引アルゴリズムの GPU 上での高速化法を提案している [13]．FM-Index は本稿で用いた 2. 2. 3 節で述べたアルゴリズムに類似した原理を用いるアルゴリズムであるが，この研究では簡潔データ構造を用いていない．ゲノムデータを対象とした評価を行っており，少ない文字種であることを利用したアルゴリズムの改善を行った．また FM-Index はランダムアクセスが必要なアルゴリズムであるので，GPU が不得手とするランダムアクセスを減少させ，かつワープ内の複数のスレッドが協調してメモリアクセスを行うことでコアレスアクセスとなるメモリアクセスにする工夫を行っている．実験結果としてマルチコア CPU と比べて 8 倍，NVIDIA 社による GPU 上で動作するバイオインフォマティクスライブラリである NVBIO に含まれる FM-Index の実装と比べて 3 倍から 5 倍の高速化を達成している．

Lin らはパケット中から攻撃パターンを検出する侵入検知システムを対象として GPU を用いたマルチパターン完全一致マッチングを行う状態遷移表によるアルゴリズム PFAC を提案している [16]．あくまで特異なパターン検出が主目的であり，本研究の目的であるすべての単語を ID に変換するような用途は考慮されていない．また実験に使われているパターン数は数万にとどまっている．

本提案は文字種，語彙数ともに多い辞書を使用することを想定し，簡潔データ構造を用いたトライ木 [5] を， GPU 上で効率的に動作させるという点で，これらの研究とは異なる．

6.　お わ り に

本稿では大量の文書に対するテキスト処理を GPU を用いて効率的に行うために必要となる，単語を ID に変換する辞書として，簡潔データを用いた省メモリなトライ木の GPU 上での実装方法を提案し，評価実験を行った．

その結果，提案手法は STT と比べて 40 分の 1 以下のメモリ使用量となった．単語を ID に変換する処理速度については，語彙数が少ない場合には 提案手法 は STT よりも低速であるが，語彙数が多い場合については提案手法は STT よりも高速

となった．語彙数が 700 万単語のとき提案手法は STT と比べて 16%高速であった．また，処理するテキストの特性として単語長の分散が処理速度に大きく影響し，分散が大きい場合には大きく性能が低下することが分かった．

今後の課題として，本稿では最も単純な並列化手法を用いたが，各単語の処理時間は単語長に依存するため，各タスクの処理時間を考慮してスレッド間での負荷分散を行うスケジューリング手法を用いることで性能向上の余地があると考えられる．

辞書データ構造の構築については辞書が頻繁に更新されるものではないとして本稿の範囲には含めていないが，効率的な構築法は応用先を広げるためには重要だと考えられる．

今回は合成したテキストを利用したが，実際のテキストでの実験も行い，索引語重み付け計算 [7] を高速化するために利用したい．

謝　　辞

本研究の一部は，科研費基盤研究 (B)(課題番号:15H02701)，挑戦的萌芽研究 (課題番号:26540042) の支援による．ここに記して謝意を表す．

文　　献

[1] Yongpeng Zhang, Frank Mueller, Xiaohui Cui, and Thomas Potok. Data-intensive document clustering on graphics processing unit (GPU) clusters. *Journal of Parallel and Distributed Computing*, Vol. 71, No. 2, pp. 211 – 224, 2011.

[2] Wenbin Fang, Bingsheng He, Qiong Luo, and Naga K. Govindaraju. Mars: Accelerating mapreduce with graphics processors. *IEEE Transactions on Parallel and Distributed Systems*, Vol. 22, No. 4, pp. 608–620, 2011.

[3] Andrew Davidson, David Tarjan, Michael Garland, and John D. Owens. Efficient parallel merge sort for fixed and variable length keys. In *Proceedings of Innovative Parallel Computing (InPar '12)*, pp. 1–9, May 2012.

[4] Nathan Bell and Jared Hoberock. Thrust: A productivity-oriented library for cuda. In Wen-mei W. Hwu, editor, *GPU Computing Gems Jade Edition*, chapter 26, pp. 359–372. Morgan Kaufmann Publishers Inc., Oct 2011.

[5] Wing-Kai Hon, Tsung-Han Ku, Rahul Shah, Sharma V. Thankachan, and Jeffrey Scott Vitter. Faster compressed dictionary matching. *Theoretical Computer Science*, Vol. 475, pp. 113 – 119, 2013.

[6] Miguel A. Martínez-Prieto, Nieves Brisaboa, Rodrigo Cnovas, Francisco Claude, and Gonzalo Navarro. Practical compressed string dictionaries. *Information Systems*, Vol. 56, pp. 73 – 108, 2016.

[7] 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), 2015. Article No. G3-6.

[8] HamidR. Bazoobandi, Steven de Rooij, Jacopo Urbani, Annette ten Teije, Frank van Harmelen, and Henri Bal. A compact in-memory dictionary for rdf data. In *Proceedings of the 12th European Semantic Web Conference (ESWC2015)*, pp. 205–220, May 2015.

[9] Edward Fredkin. Trie Memory. *Communications of the ACM*, Vol. 3, No. 9, pp. 490–499, Sept 1960.

[10] Rajeev Raman, Venkatesh Raman, and Srinivasa Rao Satti. Succinct indexable dictionaries with applications to encoding k-ary trees, prefix sums and multisets. *ACM Transactions on Algorithms*, Vol. 3, No. 4, Nov 2007. Article No. 43.

[11] Gonzalo Navarro and Eliana Providel. Fast, small, simple rank/select on bitmaps. In *Proceedings of the 11th International Conference on Experimental Algorithms*, SEA'12, pp. 295–306, Berlin, Heidelberg, 2012. Springer-Verlag.

[12] NVIDIA. CUDA Toolkit Documentation. `http://docs.nvidia.com/cuda/`.

[13] A. Chacon, S. Marco-Sola, A. Espinosa, P. Ribeca, and J.C. Moure. Boosting the FM-Index on the GPU: Effective techniques to mitigate random memory access. *IEEE/ACM Transactions on Computational Biology and Bioinformatics*, Vol. 12, No. 5, pp. 1048–1059, Sept 2015.

[14] Simon Gog and Matthias Petri. Optimized succinct data structures for massive data. *Software: Practice and Experience*, Vol. 44, No. 11, pp. 1287–1314, 2014.

[15] 田部井靖生, 田中秀宗. GPU を用いた簡潔 trie の並列探索. 数理解析研究所講究録, Vol. 1799, pp. 100–102, Jan 2012.

[16] Cheng-Hung Lin, Chen-Hsiung Liu, Lung-Sheng Chien, and Shih-Chieh Chang. Accelerating pattern matching using a novel parallel algorithm on GPUs. *IEEE Transactions on Computers*, Vol. 62, No. 10, pp. 1906–1916, Oct 2013.