Vol.2014-HPC-143 No.14
2014/3/3

部分一致を含む文字列に対する探索のAVXによる高スループット化

楠堂 航[1]　伊野 文彦[1]　萩原 兼一[1]

概要：本論文では，多数の部分一致を含む文字列に対する高速な探索の実現を目的として，ベクトル命令AVX（Advanced Vector Extensions）によるビット並列アルゴリズムの高スループット化手法を提案する．ビット並列アルゴリズムの特長は，部分一致する箇所の数や長さに実行時間が依存しないことである．提案手法は，探索の高スループット化を図るために，長さの異なる複数のパターンを同時に探索できるよう，ビット並列アルゴリズムを拡張する．具体的には，AVX 命令により CPU コアあたりの探索スループットを高め，OpenMP 指示文によるデータ並列処理を実現する．また，データ構造を工夫することにより，長さの異なる複数のパターンを効率よく同時に処理する．実験の結果，データ構造の工夫により探索スループットをおよそ 2 倍に向上できた．また，ゲノムデータのように多数の部分一致を含み，大きなテキストに対して提案手法が有用であることが分かった．

キーワード：文字列探索，ビット並列アルゴリズム，高速化，AVX

A High-Throughput String Search Using AVX for Partially Matching Data

Ko Kusudo[1]　Fumihiko Ino[1]　Kenichi Hagihara[1]

Abstract: In this paper, we present an AVX (Advanced Vector Extensions) based high-throughput method for a bit-parallel algorithm, aiming at realizing fast string search for partially matching data. An advantage of the bit-parallel algorithm is that, its execution time does not depend on the number and length of partially matching strings. Our method realizes high-throughput string search by extending the bit-parallel algorithm so that it can simultaneously search multiple patterns of different lengths. We use AVX instructions to increase the search throughput per CPU core and employ OpenMP directives to realize data-parallel processing of string search. Furthermore, we improve the data structure so that multiple patterns of different lengths can be efficiently processed at the same time. As a result, we find that our data structure doubles the search throughput. We also find that our method is useful for partially matching large texts such as genomic data.

Keywords: String search, bit-parallel algorithm, acceleration, AVX

1. はじめに

文字列探索とは，文字列からなるテキストおよび 1 個以上のパターンが与えられて，テキスト内でパターンと一致する箇所を特定することである．パケット解析やゲノム解析の分野では，複数のパターンを高速に探索する必要があり，文字列探索の高速化が期待されている．例えば，前者ではネットワーク型侵入検知システムがネットワーク上のパケットを監視し，悪意のあるアクセスやデータを検出している．また，後者では着目したい塩基配列を生物データベースから探し出す際に有用である．

高速な文字列探索手法として，PFAC（Parallel Failureless Aho-Crasick）アルゴリズム [1] が知られている．このアルゴリズムは，AC（Aho Corasick）アルゴリズム [2] を

[1] 大阪大学大学院情報科学研究科コンピュータサイエンス専攻
Department of Computer Science, Graduate School of Information Science and Technology, Osaka University

GPU（Graphics Processing Unit）[3] 向けに拡張して並列化したものである．GPU は，CPU と比べてピークメモリ帯域幅が高くコア数も多いため，高速化を期待できる．しかし，PFAC アルゴリズムは，テキスト内のすべての比較開始点ごとにスレッドを生成し，各々が有限オートマトンを保持する必要がある．各オートマトンの状態遷移数が多い場合，すなわちテキストにおいてパターンと部分一致する箇所が多く，それらが長い場合，探索スループットが低下してしまう．また，ビデオメモリは主記憶よりも容量が小さいため，サイズの大きなテキストに対しては，主記憶・ビデオメモリ間のデータ転送が頻発する．したがって，ゲノムデータのような，多数の部分一致を含みうる大きなテキストに対して探索スループットが低下してしまう．

そこで本研究では，部分一致の有無やテキストサイズの大きさに関わらず高速な文字列探索を実現するために，マルチコア CPU におけるビット並列（BP：Bit-Parallel）アルゴリズム [4] の探索スループットを向上する手法を提案する．BP アルゴリズムは，文字列一致の有無に関わらずメモリ参照量が一定であるため，探索スループットのぶれが小さい．一般に，文字列探索の性能ボトルネックはメモリ参照にあるため，提案手法は複数のパターンを同時に探索することによりパターン間のデータ再利用を図り，高速化を実現する．この際，データ構造を工夫することにより，長さの異なる複数のパターンを効率よく同時に探索する．さらに，ベクトル命令 AVX（Advanced Vector Extensions）2[5] により CPU コアあたりの探索スループットを高め，OpenMP 指示文 [6] を用いたスレッド並列処理により探索のデータ並列性を活用する．

以降では，まず 2 章で関連研究を紹介し，3 章で BP アルゴリズムについてまとめる．次に，4 章で提案手法を示し，5 章で評価実験の結果を示す．最後に 6 章で本稿をまとめる．

2. 関連研究

文字列探索は，計算集中型というよりはメモリ集中型の問題である．したがって，メモリ参照量を最小化するために，テキストから読み込んだ文字を可能な限り再利用する試みがなされている．例えば，AC アルゴリズム [2] は，複数のパターンを同時に探索することにより，テキストの読み込みをパターン間で共有し，メモリ参照量を削減する．具体的には，あらかじめ複数のパターンを 1 つのトライ木として表現し，トライ木を決定性有限オートマトンとみなして，テキストの文字を順に入力していく．入力文字に対して適切な遷移がない場合，遷移に失敗したものとみなし，比較開始点を変更した場合の一致可能性を調べるために，オートマトンをバックトラッキングする．

PFAC アルゴリズム [1] は，GPU の持つ多数のコアを活用できるよう，AC アルゴリズムを拡張し並列化している．

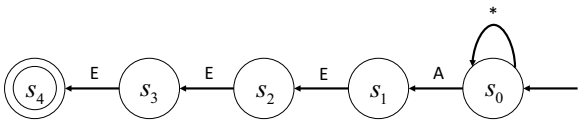


図 1 パターン AEEE に対する非決定性オートマトン（*は任意の文字を表す）

拡張点は，バックトラッキングを除去したことである．替わりに，すべての比較開始点に対してスレッドを 1 つずつ生成し，遷移に失敗したスレッドを順次消滅させている．彼らは，GeForce GTX 580 を用いて 15 Gbps もの探索スループットを達成しているが，テキストにおいてパターンと部分一致する箇所およびその長さが増大するにつれ，スレッドを早期に消滅できないがゆえに，探索スループットが 3 Gbps まで低下してしまう．

Tumeo ら [7] は，PC クラスタ上の並列処理により AC アルゴリズムを高速化している．彼らは，テキストを分割して探索のデータ並列性を活用する．ただし，分割領域間をまたぐ文字列一致を正しく認識するために，この手法は境界に袖領域を必要とする．この袖領域はスレッド間で重複して読み込まれるため，スレッド数の増大とともに，並列化効率が低下してしまう．また，パターンの一致が頻繁である場合，キャッシュミスが多発し，単一ノードにおける探索スループットがおよそ 2 Gbps まで低下してしまう．

Prasad ら [8] は，遺伝子配列を対象として複数のパターンを同時に探索できるよう，BP アルゴリズム [4] を拡張している．彼らは，すべてのパターン長が同一であり，かつそれらの和がワード長を超えないことを前提としている．現行の x64 アーキテクチャは 64 ビットのワードを採用しているため，パターン長の和が 64 文字を超える場合，ビット並列処理による高速化は望めない．一方，提案手法は異なる長さのパターンを許容し，AVX2 を用いることによりワード長を 64 ビットから 256 ビットまで増大させている．さらに，OpenMP 指示文によるデータ並列処理を実現している．

3. ビット並列アルゴリズム

BP アルゴリズム [4] は，複数の比較開始点を同時に探索するためにビット演算の並列性を活用し，単一パターンに対する文字列探索を高速化する．パターン長がワード長を超えない場合，高速な探索を期待できる．受理状態を 0 もしくは 1 のいずれかで表すかに応じて，それぞれ Shift-or 型および Shift-and 型の種類がある．提案手法は，ビット演算の数が少ない Shift-or 型を採用している．以降，テキスト t およびパターン p の長さをそれぞれ n および m とする．また，それぞれの i (≥ 1) 番目の文字を t_i および p_i で表す．

BP アルゴリズムは，前処理としてパターンから非決定性有限オートマトンを生成する（**図 1**）．このオートマト

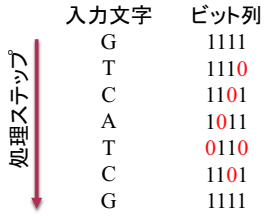


図 2 テキスト GTCATCG およびパターン TCAT に対する BP アルゴリズムの振る舞い

ンは，初期状態 s_0 に加えて m 個の状態 $s_1, s_2, \ldots, s_m$ を持ち，状態 s_i $(1 \leq i \leq m)$ はパターンを構成する文字 p_i に対応する．つまり，オートマトンの現在状態が s_i にあるとき，パターンにおける i 番目までの文字が一致していることを意味する．現在状態が状態 s_i であることを $s_i = 0$（でないことを $s_i = 1$）で表せば，テキスト t から文字を 1 つ読み込むたびにビット列 $S = s_m s_{m-1} \ldots s_1$ を更新することにより文字列探索を実現できる．更新のために必要なビット演算は次式で表せる．

$$S_i = (S_{i-1} \ll 1) \mid B[t_i] \tag{1}$$

ここで，演算子 $\ll$ および $\mid$ はそれぞれビットごとの左論理シフトおよび論理和を表し，S_i はテキストにおける i 番目の文字 t_i を読み終えたときのビット列であり，$S_0 = 11 \ldots 1$ である．また，B は表であり，テキストからの入力文字の種類ごとに適切なビットマスクを格納している．入力文字 x に対するビットマスク $B[x] = b_m b_{m-1} \ldots b_1$ は次式で与える．

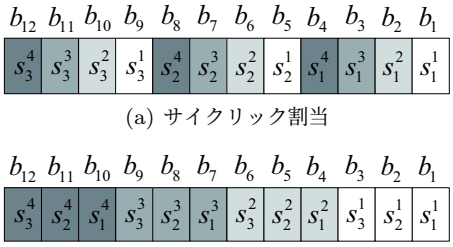
$$b_i[x] = \begin{cases} 0, & \text{if } x = p_i, \\ 1, & \text{otherwise.} \end{cases} \tag{2}$$

このように，シフト演算がオートマトンの状態遷移を実現し，ビットマスク B が入力文字 t_i に応じてその状態遷移を有効化あるいは無効化している（**図 2**）．

3.1 複数パターンの同時探索

Prasad ら [8] は，同一の長さ m を持つ複数のパターンを同時探索できるよう，BP アルゴリズムを拡張している．q (≥ 2) 個のパターンに対し，彼らは q 個のオートマトンを生成し，オートマトンの状態を表すビットをサイクリックに並べてビット列を構成している（**図 3**(a)）．つまり，j (≥ 1) 番目のパターンから生成したオートマトンの i 番目の状態を s_i^j とすると，ビット列は $s_m^q s_m^{q-1} \ldots s_m^1 s_{m-1}^q s_{m-1}^{q-1} \ldots s_1^1$ で表せる．このサイクリック割当の利点は，更新のために必要なビット演算を簡潔に表現できることである．q ビットごとに同一オートマトンの状態を割り当てているため，必要なビット演算は次式で与えられる．

$$S_i = (S^{i-1} \ll q) \mid B[t_j] \tag{3}$$



(a) サイクリック割当

(b) ブロック割当

図 3 複数のパターンを同時探索するためのビット割当 [8]（$q = 4$ かつ $m = 3$）

なお，状態を表すビットの並べ方としては，ブロック割当も考えられる（図 3(b)）．この場合，ビット列は $s_m^q s_{m-1}^q \ldots s_1^q s_m^{q-1} s_{m-1}^{q-1} \ldots s_1^1$ で表せる．しかし，この割当はベクトルの更新を複雑にしてしまう．具体的には，ビット列のうち $s_1^q s_m^{q-1}$ のような，異なるオートマトンの状態を表すビットが隣接する部分はシフト演算してはならない．したがって，ワード内にシフトすべきビットとすべきでないビットが混在してしまい，ビット演算の適用が複雑になる．

4. 提案手法

提案手法は，長さの異なる複数パターンを同時探索できるよう，BP アルゴリズムに対して以下の 3 点を拡張する．

(1) ベクトル命令 AVX2[5] による複数パターンの同時探索
(2) 一致パターンを高速に特定できるデータ構造
(3) OpenMP 指示文 [6] によるデータ並列処理

提案手法の入力は，テキスト t，q 個のパターン $p_1, p_2, \ldots, p_q$，および表 B である．出力は，探索結果を格納する数列 R である．数列 R は，0〜q の値を n 個だけ並べたものである．パターン p_j $(1 \leq j \leq q)$ がテキスト内で一致していて，その最後の文字がテキストにおいて i 番目である場合，$R_i = j$ である．ここで，R_i は数列 R における i 番目の数を表す．一方，いずれのパターンとも一致しない場合，$R_i = 0$ である．以降では，j $(\leq q)$ 番目のパターンの長さを m_j とする．

なお，数列 R のデータ構造は，複数のパターンが同一箇所で一致することを許容しないことに注意されたい．このような探索結果を排除するために，提案手法は互いに共通の接尾辞を持つパターンを同時に探索しないことを前提にする．したがって，共通の接尾辞を持つパターンに対しては，それらを区別して探索を繰り返す必要がある．

同様に，同時に探索できるパターンの長さや数に関して 2 つの前提があり，それらは以下の式で表せる（後述）．

$$q \leq 32 \tag{4}$$

$$(M - 1) \cdot q \leq 224 \tag{5}$$

ここで，$M = \max_{1 \leq j \leq q} m_j$ は q 個のパターンにおける最大の長さを表す．

4.1 AVX2 による複数パターンの同時探索

より長く，より多くのパターンを同時に探索するために，提案手法は AVX2 命令 [5] によるベクトル処理を実現する．AVX2 とは，インテル社の Haswell アーキテクチャ以降で利用可能な SIMD（Single Instruction Multiple Data）命令セットである．この命令セットは，256 ビットデータに対する浮動小数点演算や整数演算を提供する．したがって，x64 アーキテクチャのワード長 64 ビットよりも，およそ 4 倍ほど長いパターンを同時に探索できる．

提案手法は，ベクトル演算の適用を簡潔にするために，Prasad ら [8] のサイクリック割当を用いる．ただし，長さの異なるパターンを効率よく探索するために，この割当を拡張する（後述）．式 (3) で必要な命令のうち，シフト演算は `_mm256_slli_epi32()` で実現し，論理和演算は `_mm256_or_si256()` で実現できる．

4.2 一致パターンを高速に特定できるデータ構造

長さの異なるパターンを同時に探索するために解決すべき課題は以下の 2 点である．

(1) サイクリック割当への適応
(2) 一致パターンの高速な特定

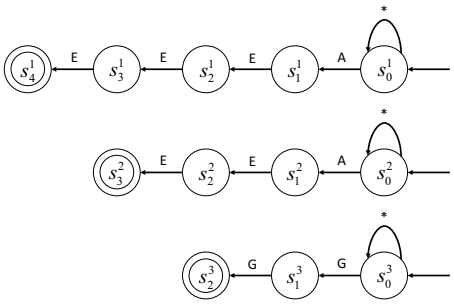
まず，長いパターンと比較して，短いパターンはオートマトンの状態数が少ない．したがって，ビット数の不揃いが原因でオートマトンの状態を表すビットをサイクリックに並べることはできない．仮に，不揃い部分を詰めて並べたとしても，同一オートマトンの状態を q ビットごとに割り当てていないため，式 (3) のような簡潔な更新を実現できない．

提案手法は，式 (3) の簡潔さを維持するために，不揃い部分に無効なダミービットを並べる（**図 4**）．さらに，無効なビット（状態）に対しては，ビットマスク B を工夫して，オートマトンを正しく解釈する．具体的には，無効な状態への遷移（無効なビットへのシフト）が，入力文字に関わらず正しく実行されるよう，ビットマスク B を指定する．例えば，i ビット目が無効である場合，入力文字に関わらず $b_i = 0$ とする．

$$b_{(i-1)q+j} = 0, \quad \text{for all } 1 \le j \le q \quad \text{and for all } 1 \le i \le M - m_j \qquad (6)$$

同様に，初期状態を表すビット列 S_0 に対しても，該当する無効ビットを 0 で初期化する必要がある．

次に，Prasad ら [8] のサイクリック割当において一致パターン p_j $(1 \le j \le q)$ を特定するためには，$s_i = 0$ を満たす i（受理状態のうち現在状態にあるもの）を検出し，$j = i \mod p$ を計算（その受理状態を持つオートマトン，すなわちパターンを特定）すればよい．しかし，上記のダミービットを用いる提案手法では，ダミービットを受理状態にあるものとして誤検出してしまう．この誤検出を避けるためには，受理状態に対応するビットの値を（B とは別の）ビットマスクで調べ，その値が 0 となるビットの位置を基にパターンを特定すればよいが，そのための処理がやや複雑になってしまう．



(a) 3 種類のパターンに対応するオートマトン

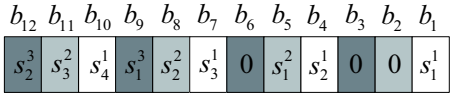

(b) パディング済みのビット列（長さ 12 の場合）

図 4 パターンの長さを揃えるためのパディング（b_2，b_3 および b_6 がダミービット）

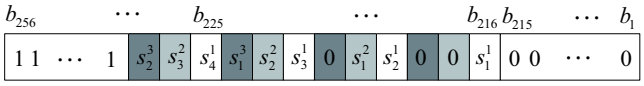


図 5 提案手法のデータ構造

上記の問題を解決するために，提案手法は q 個のオートマトンの受理状態を，256 ビットのうちの最上位 32 ビット $b_{256}b_{255}\ldots b_{225}$ のみに割り当てる（**図 5**）．つまり，ダミービットを上位桁から詰めるのではなく，下位桁から詰める．さらに $q < 32$ の場合，最上位 32 ビットのうち，該当するオートマトンのないビットを無効化するために，それらの値が常に 1 となるようにビットマスク B を指定する．つまり，受理状態に対応するビットから無効ビットへのシフトが無効となるように，次式のようにビットマスク B の最上位 $32 - q$ ビット $b_{256}b_{255}\ldots b_{225+q}$ を与える．

$$b_i = 1, \quad \text{for all } i > 224 + q \qquad (7)$$

最上位 32 ビットに受理状態を割り当てることにより，AVX2 の tzcnt 命令を用いて一致パターンを高速に特定できる．この命令は符号なし 32 ビット整数をオペランドで指定でき，連続する 0 の数を最下位ビットから数えて返す．つまり，値が 1 となるビットの桁数を迅速に取得できる．したがって，現在のビット列 S_i のうち上位 32 ビットをビット反転し，その値を tzcnt 命令に渡せば，どのパターンと一致したのかを特定できる．なお，AVX2 を使用できない Ivy Bridge 以前のアーキテクチャでは，SSE（Streaming SIMD Extensions）命令セット [9] の popcnt 命令を用いて tzcnt 命令を実現できる．具体的には，`_tzcnt_u32(x)` は `_mm_popcnt_u32((~x)&(x-1))` で代用できる．

上記で示したデータ構造は，同時に探索できるパターン

の長さや数に関して制約がある．まず上位 32 ビットを用いて一致パターンを特定しているため，式 (4) を満たす必要がある．次に，q 個のオートマトンの状態のうち，受理状態を除いたものをパディングしてビット列として並べているため，必要とするワード長は $(M-1)\cdot q$ となる．このワードを 224 ビットのベクトルに格納しているため，式 (5) を満たす必要がある．

4.3 OpenMP 指示文によるデータ並列処理

CPU 上のすべてのコアを用いるために，提案手法は OpenMP 指示文によるデータ並列処理を実現している．コア数を c とすると，テキスト t を c 個の部分テキストに分割し，各々をいずれかのコアに割り当てる．この際，Tumeo ら [7] と同様，分割領域をまたぐ文字列一致を正しく認識するために袖領域を設ける．提案手法は，複数のパターンを同時に探索しているため，それらのうちの最大の長さ M が袖領域の大きさ $M-1$ を決める．

5. 評価実験

提案手法の性能を評価するために，探索スループット $\rho = n/T$ を計測し，PFAC アルゴリズム [1] の GPU 実装 PFAC-GPU および CPU 実装 PFAC-CPU と比較した．ここで，T は探索に要した実行時間を表す．なお，実行時間は主記憶へのテキストの読み込み時間や各パターンに対するビットマスクの生成時間を含まない．また，GPU 版の実行時間は CPU・GPU 間のデータ転送時間を含めている．PFAC-CPU は，PFAC アルゴリズムを基に，テキストを分割することによるデータ並列性を活用したものである．

表 1 に，実験に用いた実行環境を示す．なお，最新の CUDA 5.5[10] は PFAC-GPU のコンパイルに失敗したため，そのコンパイルには CUDA 4.2 を用いた．一方，GeForce GTX 780 は最新のドライバを必要としたため，CUDA 5.5 のドライバを用いて実行した．また，gcc 4.8.1 では tzcnt 命令を使用できなかったため，popcnt 命令で代用した．なお，CPU の Hyper-Threading 技術は有効にしていて，CPU 版の実装はいずれも 8 スレッドを用いて実行した．

5.1 探索スループットの特性

提案手法の探索スループットを PFAC アルゴリズムと比較するために，テキストにおいて部分一致する文字列の長さ l やその一致する割合 x を変えながら探索スループット ρ を計測した．l や x を網羅的に調べるために，テキストとしてダミーデータを用いた．このダミーデータは，基礎文字列 abcdefghij の反復で構成されていて，そのデータサイズは 512 MB である．一方，同時に探索するパターンとして $q = 10$ 個を用意し，各々の長さ m_j $(1 \le j \le q)$ は 20 文字とした．l を制御するために，基礎文字列の一部を大文字に

表 1 実験環境

項目	仕様
OS	Ubuntu 12.04.3
CPU	Intel Core i7 4770K（4 コア）
主記憶	DDR3-1600 16 GB
GPU	NVIDIA GeForce GTX 780
ビデオメモリ	GDDR5 3 GB
I/O バス	PCIe 3.0 x16
コンパイラ	gcc 4.8.1（提案手法）
	gcc 4.6.4（PFAC アルゴリズム）
CUDA Driver	5.5
CUDA Runtime	4.2

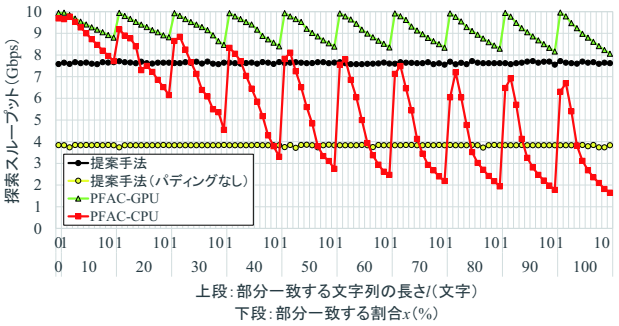


図 6 探索スループットの特性

置換したものをパターンとして用いている．例えば，$l = 3$ に対しては，パターンとして abcDEFGHIJABCDEFGHIJ を用いる．また，x を制御するために，基礎文字列に巡回置換を適用したものを組み合わせて同時に探索した．例えば，$l = 3$ かつ $x = 20\%$ に対しては，10 個のパターンのうち部分一致するものとして abcDEFGHIJABCDEFGHIJ および bcdEF...A を用い（... は省略部分），部分一致しないものとして CD...B，DE...C，EF...D，FG...E，GH...F，HI...G，IJ...H および JA...I を用いた．なお，提案手法において $q = 10$ 個のパターンのために必要なビットマスクの生成時間は 0.15 ミリ秒程度であり，性能ボトルネックではない．

図 6 に，提案手法，パディングなしの提案手法，PFAC-GPU および PFAC-CPU の探索スループットを示す．ここで，パディングなしの提案手法とは，ビット列の最下位ビット b_1 からオートマトンの状態をサイクリックに割り当てたものである．この場合，長さの異なるパターンに対しては，受理状態に対応するビットが互いに隣接せずに散在するため，ビット演算を用いて現在状態にある受理状態を検出している．

図 6 より，部分一致する文字列の長さ l およびその一致する割合 x に関わらず，提案手法の探索スループット ρ はほぼ一定である．また，パディングを施すことにより，ρ は 3.8 Gbps から 7.6 Gbps に倍増できている．この理由は，受理状態の検出処理が簡素になり，処理すべき命令の数を削減できたことによる．したがって，l や x に依存するこ

となく，探索スループットを向上できた．

一方，PFAC アルゴリズムは GPU 版であれ CPU 版であれ，探索スループット ρ が l に依存していて，パターンと部分一致する文字列が長いほど，ρ が低下している．この理由は，各スレッドが保持するオートマトンが多くの状態遷移を必要としていて，スレッドあたりの計算量が l とともに増大するためである．さらに，PFAC-CPU の探索スループット ρ は x にも依存している．x の増大は，多くの状態遷移を必要とするオートマトンが増えることを意味する．そのような多数のオートマトンを数万オーダの並列性で処理できる GPU は，x の増大に対してさほど ρ を低下させていないが，8 並列程度の PFAC-CPU は x に起因する低下が顕著である．

次に，l および x を固定させたうえで，同時に探索するパターンの長さ m_j $(1 \leq j \leq q)$ を変動させたときの探索スループットを計測した（**図 7**）．計測時には，$q = 3$ 個のパターンを用い，いずれも 1 番目から $m_j - 1$ 番目の文字がテキストにおいて部分一致するようにした．パターンの長さの総和 $\sum_j m_j$ は 103 文字であるため，3 個のパターンのうち 100 文字がテキストと部分一致する．なお，$m_2 = 34$ として，m_1 および m_3 を変動させている．

図 7 より，提案手法はパターンの長さの組み合わせに関わらず探索スループット ρ が一定である．一方，PFAC-GPU はパターンの長さが揃うにつれ，ρ が 4.2 Gbps から 5.7 Gbps まで向上している．この向上は，スレッド間の計算負荷が分散できたことによる．$m_1 = 68$ および $m_3 = 1$ の場合，スレッドあたりの状態遷移は 0 回，33 回もしくは 67 回であるのに対し，$m_1 = 35$ および $m_3 = 34$ の場合，スレッドあたりの状態遷移は 34 回もしくは 33 回である．GPU は，32 個のスレッド（ワープ）ごとに命令を実行しているため，後者の場合，同一ワープに所属する 32 個のスレッドがほぼ同じ計算負荷を持ち，前者よりも ρ が向上した．

なお，PFAC-CPU の探索スループット ρ は，提案手法と同様にほぼ一定だが，1.3 Gbps に留まる．PFAC-CPU は，部分一致する文字列の長さだけテキスト内の文字を繰り返し読み込むため，その長さを 100 文字に固定した条件下では，ρ が一定となる．なお，PFAC-CPU はテキストを分割してデータ並列処理を施しているため，PFAC-GPU における負荷分散の問題は生じない．

5.2 実データによる性能比較

現実的なテキストデータとして，生物学分野におけるゲノムデータ [11] および自然言語処理のためのコーパスデータ [12] を用い，探索スループットを計測した．このゲノムデータは，ヒト完全長相補的 DNA を収集したものである．ゲノムデータにおいて出現する文字の種類（A，T，G および C）は少ないため，パターンとの部分一致を多く含み

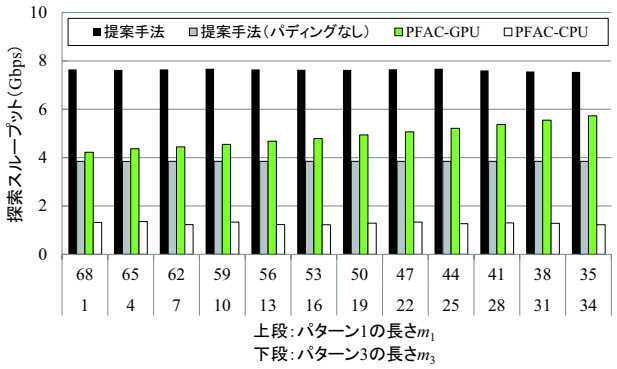
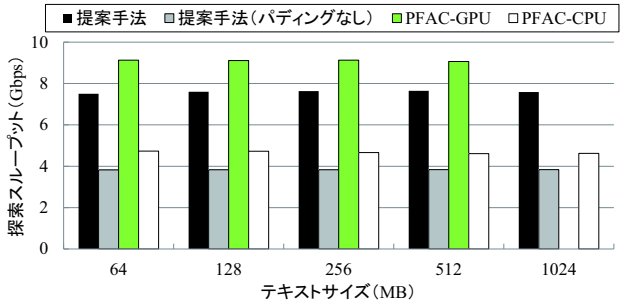
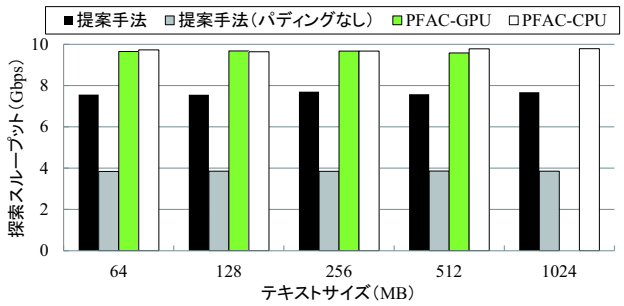


図 7 同時に探索するパターンの長さ m_1〜m_3 を変動させたときの探索スループット（$m_2 = 34$）



(a) ゲノムデータ



(b) コーパスデータ

図 8 現実的なデータに対する探索スループットの比較

うる．一方，コーパスデータはアルファベットの小文字や大文字，数字や改行コードなどの 239 種類の文字を含むため，ゲノムデータと比べて部分一致は少ない．なお，ゲノムデータのうち，ヘッダ部分は除去している．また，コーパスデータはアスキー文字のみを含むよう，ギリシャ文字などを除去している．

図 8 に，テキストサイズを 64〜1024 MB に変動させたときの提案手法および PFAC アルゴリズムの探索スループットを示す．いずれのデータに対しても $q = 8$ 個のパターンを用意した．ゲノムデータに対して用いた 8 個のパターンは，H. sapiens|ENST00000408996 から抜き出した文字列であり，それらの長さは 26〜28 文字である．一方，コーパスデータには，長さが 5〜10 文字の director, Americans, adventure, Bolshevism, class, ridden, think および workingmen を用いた．

図 8 より，提案手法は現実的なデータに対してもパディ

ングにより探索スループットを倍増できている．しかし，PFAC-GPU の探索スループット（ゲノムデータに対して 9.7 Gbps，コーパスデータに対して 8.8 Gbps）と比べて，提案手法の探索スループットは 7.6 Gbps に留まっている．ただし，ビデオメモリの枯渇が原因で，PFAC-GPU はテキストサイズが 512 MB を超えるとプログラム実行に失敗する．一方，提案手法は主記憶が枯渇しない限り，大きなテキストを探索できる．

PFAC-CPU はコーパスデータに対しては PFAC-GPU を少し上回る探索スループット 9.8 Gbps を達成しているが，ゲノムデータに対する探索スループットは提案手法を下回る（5.3 Gbps）．ゲノムデータは文字の種類が少ないため，パターンと部分一致する頻度が高く，探索スループットが低下している．この結果は，5.1 節で示した探索スループットの特性と合っている．

6. まとめ

本稿では，多数の部分一致を含む文字列に対する高速な探索を目的として，BP アルゴリズムの高スループット化手法を提案した．提案手法は，AVX2 により CPU コアあたりの探索スループットを高め，OpenMP 指示文によりデータ並列性を活用した並列探索を実現する．また，長さの異なるパターンを効率よく同時に探索できるよう，データ構造を工夫する．効率を高めるアイデアは，ダミービットのパディングである．受理状態にあるオートマトンを 1 命令で検出するために，下位桁へのパディングを採用している．

実験の結果，パディングなしの実装と比べて，提案手法はおよそ 2 倍の速度向上を得た．また，パターンやテキストの内容や長さに依存することなく，ほぼ一定の探索スループットを実現した．CPU 版の PFAC アルゴリズムと比較して，部分一致を多く含むゲノムデータに対しては 1.4 倍の高スループット化を果たしたが，部分一致をさほど含まないコーパスデータに対しては探索スループットが 20%低かった．また，GPU 版の PFAC アルゴリズムに対しては，いずれのデータに対しても探索スループットがおよそ 20%低かった．しかし，ビデオメモリ容量を超える大きなテキストに対して，提案手法は有用である．

今後の課題としては，似た文字列を探し出す近似文字列探索 [13] への対応が挙げられる．近似文字列を探索できることは，BP アルゴリズムに固有の特長である．このような曖昧な探索は，生物学分野において求められており，検索対象のテキストが塩基配列の読み取りエラーを大量に含む場合に有用である．

謝辞 本研究の一部は，JST CREST「進化的アプローチによる超並列複合システム向け開発環境の創出」，科研費 23300007 および 23700057 の補助による．本研究に関して議論して頂きました日本電気株式会社の柏木岳彦氏に感謝いたします．

参考文献

[1] Lin, C.-H., Liu, C.-H., Chien, L.-S. and Chang, S.-C.: Accelerating Pattern Matching Using a Novel Parallel Algorithm on GPUs, *IEEE Trans. Computers*, Vol. 62, No. 10, pp. 1906–1916 (2013). http://code.google.com/p/pfac/.

[2] Aho, A. V. and Corasick, M. J.: Efficient String Matching: An Aid to Bibliographic Search, *Communications of the ACM*, Vol. 18, No. 6, pp. 333–340 (1975).

[3] Lindholm, E., Nickolls, J., Oberman, S. and Montrym, J.: NVIDIA Tesla: A Unified Graphics and Computing Architecture, *IEEE Micro*, Vol. 28, No. 2, pp. 39–55 (2008).

[4] Baeza-Yates, R. and Gonnet, G. H.: A New Approach to Text Searching, *Communications of the ACM*, Vol. 35, No. 10, pp. 74–82 (1992).

[5] Intel Corporation: Intel Architecture Instruction Set Extensions Programming Reference (2013). http://download-software.intel.com/sites/default/files/managed/71/2e/319433-017.pdf.

[6] Chandra, R., Dagum, L., Kohr, D., Maydan, D., McDonald, J. and Menon, R.: *Parallel Programming in OpenMP*, Morgan Kaufmann, San Mateo, CA (2000).

[7] Tumeo, A., Villa, O. and Chavarría-Miranda, D. G.: Aho-Corasick String Matching on Shared and Distributed-Memory Parallel Architectures, *IEEE Trans. Parallel and Distributed Systems*, Vol. 23, No. 3, pp. 436–443 (2012).

[8] Prasad, R., Agarwal, S., Yadav, I. and Singh, B.: A Fast Bit-parallel Multi-patterns String Matching Algorithm for Biological Sequences, *Proc. Int'l Symp. Biocomputing (ISB'10)*, No. 46 (2010). 4 pages.

[9] Klimovitski, A.: Using SSE and SSE2: Misconceptions and Reality, *Intel Developer Update Magazine* (2001).

[10] NVIDIA Corporation: CUDA C Programming Guide Version 5.5 (2013). http://docs.nvidia.com/cuda/pdf/CUDA_C_Programming_Guide.pdf.

[11] Yamashita, R., Wakaguri, H., Sugano, S., Suzuki, Y. and Nakai, K.: DBTSS provides a tissue specific dynamic view of Transcription Start Sites, *Nucleic Acid Research*, Vol. 38, pp. D98–104 (2010). http://dbtss.hgc.jp/.

[12] Ferragina, P. and Navarro, G.: Pizza&Chili Corpus Compressed Indexes and their Testbeds (2005). http://pizzachili.dcc.uchile.cl/.

[13] Tran, T. T., Giraud, M. and Varré, J.-S.: Bit-Parallel Multiple Pattern Matching, *Proc. 9th Int'l Conf. Parallel Processing and Applied Mathematics (PPAM'11), Part II*, pp. 292–301 (2011).