長大な拡張文字列パターンに対するGPUによる高速な文字列照合

笹川　裕人† 　喜田　拓也† 　有村　博紀†

† 北海道大学大学院情報科学研究科　〒 060–0814 札幌市北区北 14 条西 9 丁目
E-mail: †{sasakawa,kida,arim}@ist.hokudai.ac.jp

あらまし　数千個のパターンを照合する大規模並列文字列照合は，ネットワーク不正侵入検出や生物情報処理など，広い応用分野において重要な基盤技術である．しかし，大規模文字列照合は，計算負荷が非常に大きな処理であり，CPU 上の実時間処理は難しい．そのため，GPU を用いたハードウェアによる高速化が注目されている．そこで本稿では，筆者らが提案した，NFA の階層的なモジュール分割に基づくビット並列照合アルゴリズム RunNFA を GPU 上の並列実装へ拡張する手法を提案する．階層的モジュール分割によるモジュール間の並列実行と，ビット並列手法によるコンパクトな NFA 表現と，高速な遷移計算により，GPU 上での拡張文字列パターンに対する高速な文字列照合を実現する．提案システムの性能評価のため，CUDA による GPU 上での実装を与え，計算機実験によってその有効性を示す．

キーワード　文字列照合，並列処理，GPGPU，拡張文字列パターン，正規表現照合

GPU Acceleration of Faster String Matching for Very Long Extended Patterns

Hirohito SASAKAWA†, Takuya KIDA†, and Hiroki ARIMURA†

† Graduate School of Information Science and Technology, Hokkaido University
Kita 14, Nishi9, Kita–ku, Sapporo, Hokkaido, 060–0814 Japan
E-mail: †{sasakawa,kida,arim}@ist.hokudai.ac.jp

Abstract

Key words　string matching, parallel processing, GPGPU, extended pattern, regular expression matching

1. はじめに

1.1 背　景

数千個のパターンを照合する**大規模並列文字列照合**は，生物情報処理やネットワーク不正侵入検出など，広い応用分野において重要な基盤技術である．特にゲノムやネットワークの分野では，ハードウェアの高性能化により，扱うデータ量が急速に増大しており，データ解析の観点から照合の更なる高速化が求められている．一方で，大規模並列文字列照合は，処理全体の計算負荷が非常に大きく，汎用 CPU 上のソフトウェアによる実時間処理が難しい．そのため，GPU を用いたハードウェアによる高速化が注目を集めている．

GPU (Graphics Processing Unit) とは，画像処理を専門に担当するハードウェアであり，数十から数百の演算コアと，各コア間で共有の高速なメモリを備えている．その高い並列演算能力を画像処理以外の汎用的な計算へ用いる GPGPU (General Purpose computing on GPUs) という手法が近年注目されている [1, 3, 7]．しかし，GPU はメモリサイズや，アクセスパターンに対して，強い制約を持つため，物理シミュレーションや，科学技術計算の分野への応用が中心であり，大規模な離散的計算への応用はまだ少ない．

これまでに著者らは，GPU を用いた大規模並列文字列照合を研究してきた [9, 10]．文献 [9] では，ビット並列手法に基づく Gaps-SHIFT-AND 法 [6] により CBG パターンと呼ばれる，文字クラスと有限長のギャップを許した正規表現の部分クラス上のパターン照合アルゴリズムを与えた．ここでは，パターン長に制限を設け，1 レジスタ内でのみ動作するアルゴリズムの GPU 実装を考え，使用領域が大きい配列による実装に比べて約 60 倍の高速化を実現した．

文献 [10] では，拡張文字列パターンにおけるビット並列照合手法である拡張 SHIFT-AND 法 [5] を拡張し，複数レジスタにまたがる長大なパターンに対して効率良く実行するための NFA の階層的モジュール分解に基づく CPU 上のアルゴリズム RunNFA を与えた．分解した各モジュールの計算順序を工夫し，レジスタ間の通信回数を減らし，複数のレジスタを並べて長いレジスタを模倣する従来型のアルゴリズムと比較して，使

用レジスタ数が 10 のパターンの時で 20%高速に動作する事を確認した．

本稿では，RunNFA [10] を GPU 実装への拡張し，長大かつ大量の拡張文字列パターンに対する高速な照合を行う方法を提案する．RunNFA はパターンを上位モジュールと，下位モジュールの 2 階層で構成される階層型 NFA に変換し，各モジュールの遷移をビット並列手法を用いて模倣する方法である．階層型 NFA は，各モジュールが独立して動作できることが特徴であり，並列実行に向いている．また，ビット並列手法によるコンパクトな NFA 表現により，GPU のメモリ制約下でも高速な遷移計算を実現した．

計算機実験では，100 個のパターンについてアミノ酸データ上で照合を行い，提案システムは，同アルゴリズムの CPU 実装に対して，2 倍程度高速であった．

1.2 関 連 研 究

大規模文字列照合に関する関連研究について説明する．Lin, Tsai ら [2] は，Aho-Corasick 法と呼ばれる，決定性有限オートマトンを用いた照合アルゴリズムの GPU 上の実装を与えている．Wu, Diao, Rizvi ら [8] は，高速ストリームデータ上における複合イベント処理システムのための系列イベントクエリに対する照合アルゴリズムを与えている．これに関連して，Margara, Cugola ら [4] は，GPU 上の複合イベント処理システムを与えている．

2. 準　　備

2.1 拡張文字列照合問題

文字集合 Σ をアルファベットとする．**テキスト**は，長さ n の文字列 $T = t_1 \cdots t_n \in \Sigma^*$ である．

[**定義** 1] **拡張文字列パターン**: Σ 上の正規表現 $P = a_1 \ldots a_m (m >= 0)$ が**拡張文字列パターン**であるとは，任意の $i = 1, \ldots, m$ に対して，a_i が下記の定義の (1)-(6) のいずれかであることをいう．なおここで，$\equiv$ は構文糖衣を表し，$\cdot$ と，*, | は言語の連結と，繰り返し，選言をそれぞれ表す．

（1） 定数文字：$a \in \Sigma$. $L(a) = a$.
（2） ワイルドカード：“.”. $L(.) = \Sigma$.
（3） 文字種：$\beta = [abc\cdots] \subseteqq \Sigma$. $L(\beta) = \beta$.
（4） オプション文字：$\beta? \equiv (\beta|\varepsilon)$. $L(\beta?) = L(\beta) \cup \{\varepsilon\}$.
（5） 限定繰り返し：$\beta\{x, y\} \equiv (\beta?)^{y-x}\beta^x$.
（6） 0 個以上の非限定繰り返し：β^*. $L(\beta^*) = L(\beta)^*$.
（7） 1 個以上の非限定繰り返し：$\beta^+ \equiv \beta\beta^*$.

ここで, 拡張文字列パターン $P = a_1, \cdots, a_m$ の**サイズ**を P の中の文字と演算子の個数の総和とし，その**言語**を, $L(P) = L(a_1) \cdots L(a_m)$ と定義する．

[**定義** 2] **パターンの出現**: パターン P が T の**部分系列として位置** $1 \leqq q \leqq n$ **に出現する**とは，長さ 0 以上の文字列 $u, v, w \in \Sigma^*$ が存在して, (i) $T = uvw$ かつ (ii) $v \in L(P)$, (iii) $q = |uv| + 1$ となることをいう. すなわち, q は, パターン P の後端の出現位置である．

[**例** 1] アルファベット $\Sigma = \{A, B, C\}$ に対して，$P_1 = AB^+A?B?C?CB?C?A?$ は拡張文字列パターンの例である．

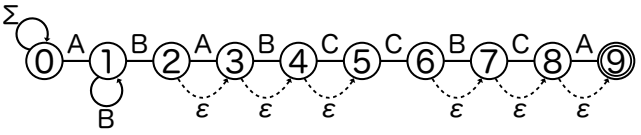

図 1　例 1 のパターン P_1 に対応する非決定性有限オートマトン (NFA)

拡張文字列照合問題とは，与えられたパターンとテキストに対し，パターンが出現するテキスト中の位置を列挙する問題である．正確には，次のように定義する．

[**定義** 3] **拡張文字列照合問題**: テキスト T と拡張文字列パターン P を入力として受け取り，P の T 中の出現位置を全て求める．

2.2 拡張 SHIFT-AND 法

ビット並列手法とは，ビット演算と，数値演算レジスタ内並列性を利用し，計算を高速化する手法である．拡張 SHIFT-AND 法は，Navarro ら [5,6] による，ビット並列手法を用いた拡張文字列パターンに対する照合アルゴリズムである．このアルゴリズムでは，前処理時に，与えられた拡張文字列パターンに対応する非決定性有限オートマトン (NFA) を構築し，照合時に，この NFA の遷移をビット演算と，整数の加算を用いて模倣し照合を行う．例えば，例 1 のパターンに対しては，図 1 のような NFA が構築される．ここで，テキスト長を n，NFA の状態数を ℓ とする．拡張 SHIFT-AND 法は，前処理に $O(m + |\Sigma|)$ 時間，照合に $O(n\lceil \ell/w \rceil)$ 時間それぞれかかり，アルゴリズム全体で $(2\Sigma + 4)\lceil \ell/w \rceil$ の領域を使用する．

2.3 計算モデル

計算モデルとして，レジスタ長 w の RAM モデルを用いる．ここで，レジスタ内の AND 演算 “&”，OR 演算 “|”，NOT 演算 “∼”，XOR 演算 “∧”，左シフト演算 “<<”，右シフト演算 “>>”，整数の加減算を $O(1)$ 時間で実行できると仮定する．ビットマスク (マスク) は，幅 w のビット列である．また，各 $0 \leqq j \leqq w - 1$ に対して，$Bit(j)$ は，j 番目のビットのみに 1 が立ったビットマスクを表す．

2.4 GPU と GPGPU

Graphics Proccesing Unit (GPU) とは，画像処理専用のハードウェアである．GPU は，複数の **Stream Multiprocessor (SM)** を備え，各 SM は 32 個の SIMD 演算コアを持つ．各演算コアは，SM 内に搭載された 64KB の高速な共有メモリと，GPU ボード上に搭載された，低速であるが，大容量（数 GB）のグローバルメモリにアクセスし，並列に計算を行うことで，高速な画像処理を実現する．このような多数の演算コアと，高速なメモリを利用した高い演算能力を，画像処理以外の汎用的な計算に用いる手法を **GPGPU (General Purpose computing on GPUs)** と呼び，近年注目を集めている [1,3,7].

3. GPU 照合システム

本節では，アルゴリズム RunNFA を GPU 実装へ拡張する GPU 上の照合システムについて説明する．以下では，GPU 上で並列に実行される計算を**スレッド**と呼ぶ．

3.1 システムのアーキテクチャ

提案システムは，大きく分けて CPU 側の計算（**ホスト**）と GPU 側の計算（**デバイス**）からなる．システムは任意のパターン集合 $Pat = \{P_1, P_2, \ldots, P_k\}$ に対して以下の様に動作する．

（1） Pat の各パターンに対し，対応するビットマスク集合を計算する（ホスト側）．

（2） 計算したビットマスク集合をデバイス側へ転送する（ホスト，デバイス間通信）．

（3） ビットマスク集合を用いて各パターンの照合を並列に実行する（デバイス側）．

（4） 照合結果をホスト側へ報告する（ホスト，デバイス間通信）．

3.2 RunNFA アルゴリズム

パターン集合 Pat における各々のパターンの照合について説明する．基本的な照合アルゴリズムは，著者らが提案したビット並列手法に基づく RunNFA アルゴリズムを用いる．RunNFA では，まず，前処理として，入力パターン P を階層型 NFA と呼ぶ，NFA の集合に変換する．階層型 NFA は 1 つの**上位モジュール**（upper module）UM と，k 個の**下位モジュール**（lower module）LM で構成される．i 番目のモジュールを M_i と書く．また，モジュール M が持つ変数 X を表す時は，$M.X$ と書く．例として，図 1 の NFA を，$w = 4$ のとき，階層型 NFA に変換したものを図 2 に示す．次に，変換された階層型 NFA に対応するビットマスク集合を計算する．各モジュールが保持するビットマスク集合は以下の通りである．

- S：状態集合を表すビットマスク.
- $Ebeg$：ε 遷移の開始位置を示すビットマスク.
- $Eend$：ε 遷移の終了位置を示すビットマスク.
- $Eblk$：ε 遷移が 1 回以上連続する位置を示すビットマスク.
- $Ch[c]$：文字 $c \in \Sigma$ によって文字遷移が行われる位置を示す.
- $Rep[c]$：文字 $c \in \Sigma$ によって文字繰り返し行われる位置を示す.

各種ビットマスクの計算方法の詳細については [10] を参照されたい．ビットマスク集合の計算後，RunNFA は図 3 の実行時アルゴリズムを用いて照合を行う．図 2 の実行時アルゴリズムでは，まず，1〜4 行目までで階層型 NFA の初期化を行う．次に，テキストを 1 文字ずつ読み込みながら，以下の手順を繰り返し，階層型 NFA の遷移を模倣する．

Step1　UM から LM へ遷移情報を伝える (6 行目).
Step2　LM の遷移を行う (7 行目).
Step3　LM から UM へ遷移情報を伝える (8 行目).
Step4　UM の遷移を行う (10 行目).
Step5　UM が終了状態に達したかどうかをチェックし，達していれば報告する (11 行目).

3.3 GPU 実装

GPU 上での実装の詳細について説明する．GPU 上では，階層型 NFA の各モジュールに対して，1 ずつスレッドを割り振り，遷移の計算を並列に行う．

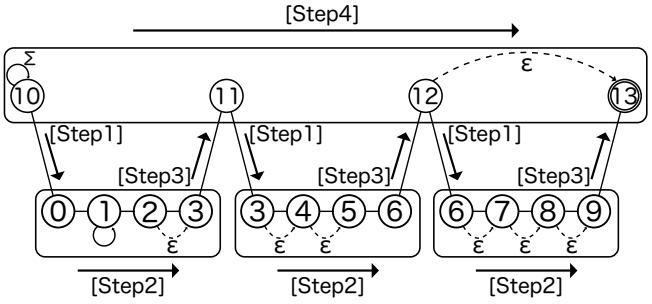

図 2　階層型 NFA の動き.

Algorithm 1 RunNFA($UM, (LM_j)_{j=0}^{k-1}$: module, T: text)

```
1:  procedure SEARCH
2:      n ← length(T);
3:      UM.I ← 0^{w-1}1;
4:      EpsClo(UM);
5:      for i = 0 to n do
6:          for j = 0 to k − 1 do UpperToLower(UM, LM_j);
7:          par j = 0 to k − 1 do RunExShiftAnd(LM_j, T[i]);
8:          for j = 0 to k − 1 do LowerToUpper(LM_j, UM);
9:          Syncthreads();
10:         EpsClo(UM)
11:         if UM.S & Bit(w) ≠ 0 then
12:             report an occurrence at i;
```

図 3　階層型 NFA による照合アルゴリズム.

Algorithm 2 EpsClo(M: module)

```
1: M.S ← M.S | M.I
2: Z ← M.S | M.Eend
3: M.S ← M.S | (M.Eblk & ((∼ (Z − M.Ebeg)) ∧ Z))
```

図 4　手続き EpsClo.

Algorithm 3 RunExShiftAnd(LM_j: module, $T[i]$: letter)

```
1: LM.S ← LM.S | LM.I;
2: EpsClo(LM);
3: LM.S ← (((LM.S << 1) | LM.I) & LM.Ch[T[i]]) |
   (LM.S & LM.Rep[T[i]]);
4: EpsClo(LM);
```

図 5　手続き RunExShiftAnd.

Algorithm 4 UpperToLower(UM, LM: modules)

```
1: if UM.S & Bit(j) ≠ 0
2: then LM.I ← 1;
3: else LM.I ← 0;
```

図 6　手続き UpperTolower.

手続き UpperToLower（図 6）は，上位モジュールのビット状態を，下位モジュールの開始状態に対してコピーする手続き

Algorithm 5 LowerToUpper(LM, UM: module)

1: **if** $LM.S \,\&\, Bit(w) \neq 0$
2: **then** $UM.S \leftarrow UM.S \mid Bit(j)$; ▷ j ビット目を 1 にする
3: **else** $UM.S \leftarrow UM.S \,\&\, (\sim Bit(j))$; ▷ j ビット目を 0 にする

図 7　手続き LowerToUpper.

である．1 行目の if 文により，コピー先の下位モジュールに該当する状態のビットを確認し，その状態に応じて，1 または，0 のビットを下位モジュールの開始状態にコピーする．手続き LowerToUpper（図 7）は，下位モジュールの終了状態を，上位モジュールの該当位置にコピーする手続きである．手続きの内容は，UpperToLower とほぼ同様である．

UpperToLower と LowerToUpper では，上位モジュールの状態マスク S を保持する 1 つのレジスタに対して，全ての下位モジュールのスレッドが同時にアクセスするため，モジュール同士のアクセス競合が起こる．提案手法では，特に競合の解決は行わず，書き込み時にロックを行いながら変更を行う Atomic 命令を使用し，照合の正しさを保証している．この対処方法では，下位モジュール数 k に対して，$O(k)$ 回のアクセスが必要になる．提案システムでは採用していないが，平衡二分木スキームを利用することで，このアクセス回数を $O(\log k)$ 回のアクセスまで減らすことができる．また，上位モジュールへの書き込みを伴う LowerToUpper においては，全てのモジュールの書き込みが終了した上で次の操作を行う必要があるため，手続き LowerToUpper のあとで全スレッドの同期手続き Syncthreads() を行う（10 行目）．

手続き RunExShiftAnd（図 5）は，下位モジュールの NFA の遷移を計算する手続きである．これは，Navarro と Raffinot [5] によって提案された Extended-SHIFT-AND 法そのものである．ビット演算と，整数の加算を用いて，NFA の文字遷移と文字繰り返し（3 行目）と，ε 閉包の計算（2,4 行目）を各モジュールについて，1 文字あたり $O(1)$ で計算する．RunExShiftAnd では，モジュールごとに動作が独立しているため，並列に計算を実行することができる．また，手続き UpperToLower により，上位モジュールの状態をコピーした際，初期状態に 0 がコピーされた場合については，遷移計算を行う必要がないため，この場合，スレッドを休ませることで高速化をはかることができる．

なお，上位モジュールと，下位モジュールは同時に遷移計算を行うことはないので，先頭の下位モジュール LM_0 の遷移計算を担当するスレッドが，上位モジュールの遷移計算も兼任することで，パターン 1 つあたりのスレッドを 1 つ減らすことができる．よって，1 つのパターンの照合に必要なスレッド数は下位モジュール数分の k 個である．

4. 計算量の解析

本節では，RunNFA アルゴリズムについて，記憶領域と計算時間について詳しく解析する．

4.1 記憶領域

上位モジュール数は 1 であり，下位モジュール数は $k = \lceil \ell/w \rceil$ 個である．上位モジュール一つ当たり，NFA の状態集合を表すビットを 1 個と，ε 遷移のためのビットマスクを 3 個用いるので，合計で $S_u = 4$ (語) の領域を使用する．下位モジュール一つ当たり，NFA の状態集合を表すビットを 1 個と，ε 遷移のためのマスクビットを 3 個，文字遷移と文字繰り返しのためのマスクビットをそれぞれ Σ 個用いるので，合計で $S_l = (2|\Sigma| + 4)\lceil \ell/w \rceil$ (語) の領域を使用する．以上から，RunNFA は合計で，

$$\begin{aligned} S_{all} &= (2|\Sigma| + 4)\lceil \ell/w \rceil + 4 \\ &= O(|\Sigma|\lceil \ell/w \rceil)\,(\text{語}) \end{aligned}$$

の領域を使用する (表 1).

マルチストリームマッチングは，s 個の入力ストリームに対して，一つのパターンの照合を同時に行う．この場合，前処理で構築するマスクを一組だけ保持して共有し，状態マスクのみを s 個保持することで単純に s 台のパターン照合アルゴリズムを走らせることができる．このとき，領域は

$$\begin{aligned} S_{all} &= s(\lceil \ell/w \rceil + 1) + (2|\Sigma| + 3)\lceil \ell/w \rceil + 3 \\ &= O((s + |\Sigma|)\lceil \ell/w \rceil)\,(\text{語}) \end{aligned}$$

となり，入力ストリーム数 s が非常に大きいときにメモリ削減に有効である．

表 1　記憶領域の解析.

Module	ε 遷移	文字遷移	状態	Module 数	合計
Upper	3	0	1	1	4
Lower	3	2Σ	1	$\lceil \ell/w \rceil$	$(4 + 2\Sigma)\lceil \ell/w \rceil$

4.2 計算時間

NFA サイズが ℓ のパターンの前処理時間は，$O(\ell)$ 時間である．図 3 の手続き RunNFA は，テキスト T の文字を 1 文字読むごとに，for ループ内 (5〜12 行目) の手続きを行う．6〜8 行目の UpperToLower (図 6)，RunExShidtAnd (図 5)，LowerToUpper (図 7) の 3 つの手続きは，すべての下位モジュール $LM_j (0 \leqq j \leqq k-1)$ について行われ，それぞれに対して $O(1)$ 時間かかるので，合計で $O(k) = O(\lceil \ell/w \rceil)$ 時間かかる．また，10 行目の上位モジュールにおける EpsClo は，UM_0 について実行され，$O(1)$ 時間かかる．以上から，for ループ内の処理には，

$$O(\lceil \ell/w \rceil) + O(1) = O(\lceil \ell/w \rceil)\,(\text{時間}) \qquad (1)$$

かかる．

以上から，逐次計算量について，次の定理を示すことが出来る．

［**定理** 1］ 提案アルゴリズム RunNFA は，$\ell \leqq w(w-1)$ を満たす拡張文字列パターン P と入力テキスト T に対して，前処理時間 $O(\ell)$ と領域 $O(|\Sigma|\lceil \ell/w \rceil)$ 語を使い，計算時間 $O(n\lceil \ell/w \rceil)$ で P の T 中のすべての出現を見つける．ここに，n はテキスト長であり，ℓ はパターンに対応する NFA の総状態数である．

GPU などの並列ハードウェアの理論的モデルである並列 RAM (PRAM) 上の並列計算量について，次の結果を示すことが出来る．

［**定理** 2］ 提案アルゴリズム RunNFA は，$\ell \leqq w(w-1)$ を満たす拡張文字列パターン P と入力テキスト T に対して，前処理時間 $O(\ell)$ と，一台当たり $O(|\Sigma|)$ 語の領域使用する $O(\lceil \ell/w \rceil)$ 台のプロセッサを用いて，並列計算時間 $O(n \log \lceil \ell/w \rceil)$ で P の T 中のすべての出現を見つける．ここに，n はテキスト長であり，ℓ はパターンに対応する NFA の総状態数である．

NFA サイズ ℓ が非常に大きなパターンに対して，多くのプロセッサを用いることで高速化できる．

5. 実　　験

本論文で提案したシステムの有用性を検証するため計算機実験を行った．実験では，提案システムである RunNFA アルゴリズムの CUDA による GPU 実装と，比較対象として，同アルゴリズムの C++言語で実装した CPU 実装と，パターンの NFA を配列で表現し，その遷移を模倣する ArrayNFA アルゴリズムの CUDA による GPU 実装を用いた．なお，ArrayNFA は，文字遷移，ε 遷移，NFA の新旧の状態集合を長さ ℓ の 4 本の配列を用いて表現し，遷移の計算は，状態集合をスキャンしながら，一文字あたり $O(\ell)$ 時間で計算するアルゴリズムである．

5.1 実験環境とデータ

実験は，以下の計算機上で行った．

- CPU : Intel Core i7 processor 2.8GHz
- メモリ : 12GB
- GPU : NVIDIA GeForce GTX480
  - 演算コア数 : 480 個
  - メモリ : 1.5GB
- OS : Debian GNU/Linux 6.0.5
- CPU コンパイラ : g++ 4.6.3
- CUDA コンパイラ : nvcc 5.0

以下では，パターンに含まれる $\{*, +, ?\}$ の文字を特殊文字と呼ぶことにする．また，ℓ をパターンの NFA サイズとする．

実験で使用したデータは，ExPASy : SIB Bioinformatics Resource Portal (注1) から取得した 10MB のアミノ酸配列データである．アミノ酸配列データは，$|\Sigma| = 20$ のテキストデータである．パターンは，ランダムに生成した特殊文字を 60% 含んだ拡張文字列パターンを使用した．

5.2 GPU 上のアルゴリズムの比較

GPU 上のアルゴリズム，RunNFA と，ArrayNFA に対して，$\ell = 130$ のパターンを用意し，パターン数を 20 個〜100 個まで 20 個ずつ変化させながら照合時間を計測した．表 2 に実験結果を示す．ここで，t は SM あたりの同時に起動可能なスレッド数を表す．全てのパターン数において，RunNFA の実行時間は，ArrayNFA に対して 20 倍前後高速である．これは，ArrayNFA の使用領域が非常に大きく，GPU の共有メモリを圧迫し，同時に起動可能なスレッド数が 4 つに制限されるのに対し，RunNFA は NFA をレジスタに圧縮してコンパクトに表現するため，14 スレッドを並列実行可能であることが原因と考えられる．また，ArrayNFA は，状態遷移の計算に一文字あたり $O(\ell)$ 時間かかり，RunNFA は，各モジュールに対して定数時間であることも高速化の一因であると考えられる．

(注1) : ExPASy : SIB Bioinformatics Resource Portal- http://expasy.org/

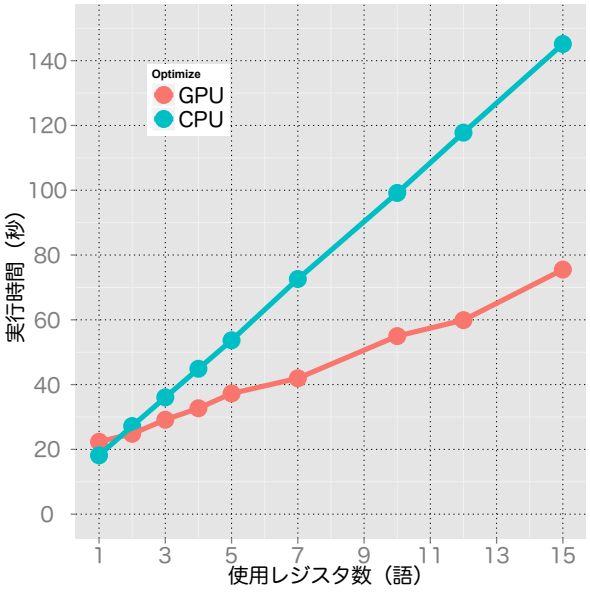

図 8　パターンのワード数と実行時間．

表 2　アルゴリズムに対する総計算時間（秒）．

Algorithm	t	Number of Patterns p				
		20	40	60	80	100
ArrayNFA	4	568.2	569.1	570.1	571.6	628.5
RunNFA	14	20.9	23.8	29.3	32.8	32.6

5.3 パターンの使用するワード数による比較実験

RunNFA の GPU 実装と CPU 実装に対して，100 個のパターンの照合時間を，パターンの使用するワード数 $\lceil \ell/w \rceil$ を変化させながら計測し，比較した．図 8 に実験結果を示す．ワード数が 1 個の場合は，CPU 実装が GPU 実装に比べ実行時間が短いが，ワード数が 2 個以上になると GPU 実装が上回っている．CPU 実装は，ワード数に対して線形に実行時間が伸びているのに対して，GPU 実装の実行時間は，伸びが緩やかである．使用レジスタ数が 12 の時，GPU 実装が CPU 実装に対して 2 倍高速である．これは，GPU の並列性により，パターン数の増加に対してスケールしているためであると考えられる．

5.4 パターン数に関する実験

RunNFA の GPU 実装に対して，使用ワード数が 4 ワードとなるパターンを用意し，パターン数を 10 個〜200 個まで 10 個ずつ変化させながら照合時間を計測した．図 9 に実験結果を示す．パターン数 120 個と 130 個の間で照合時間に大きな差が出ているのがわかる．これは，パターン一つ当たりの使用ワード数が 4 ワードのパターンに対して，今回実験に使用した GPU が持つコア数が 480 個であるため，120 パターンを照合する時，

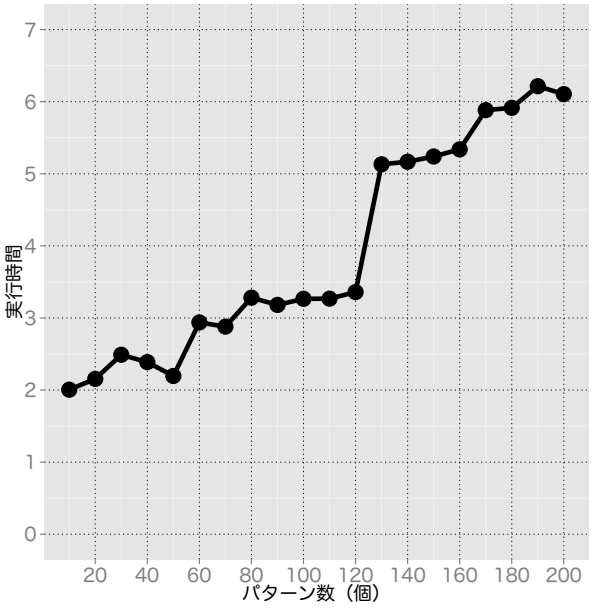

図 9　パターン数と実行時間の変化.

全ての演算コアが使い切られ，130 パターン以降では，スレッドが逐次的に動作する事になるためこのような結果になったと考えられる.

6. おわりに

本稿では，大規模文字列照合の GPU を用いた高速化を実現する方法について考察を行った．提案システムでは，NFA の階層型モジュール分解に基づくビット並列照合アルゴリズム RunNFA を GPU 上の実装へ拡張する手法を与え，計算機実験において，同アルゴリズムの CPU 実装に比して，パターン数 100 のとき，約 2 倍高速に照合できることを確認した．階層型 NFA の各下位モジュールにおいて，照合の計算が並列に実行できることに加え，ビット並列手法を用いたことによる NFA のコンパクトな表現により，NFA を配列実装した場合にくらべ，SM あたり 3.5 倍のスレッドが並列実行できることにより，高速な照合を実現した．今後の課題としては，本稿では，3.3 節で検討した，上位モジュールでの競合に対して，平衡二分木を使う方法など手法の改良により競合回数を減らすことによる，パフォーマンスの改善があげられる．また，階層型 NFA を多階層の構成へ一般化し，より複雑な正規表現の照合を実現することを考えている.

文　献

[1] N. Bell and M. Garland. Implementing sparse matrix-vector multiplication on throughput-oriented processors. In *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*, p. 18. ACM, 2009.

[2] Cheng-Hung Lin, Sheng-Yu Tsai, Chen-Hsiung Liu, Shih-Chieh Chang, and Jyuo-Min Shyu. Accelerating string matching using multi-threaded algorithm on GPU. In *the 2010 IEEE Global Telecommunications Conference (GLOBECOM'10)*, pp. 1 –5, dec. 2010.

[3] S.A. Manavski and G. Valle. Cuda compatible gpu cards as efficient hardware accelerators for smith-waterman sequence alignment. *BMC bioinformatics*, Vol. 9, No. Suppl 2, p. S10, 2008.

[4] A. Margara and G. Cugola. High performance content-based matching using gpus. In *Proceedings of the 5th ACM international conference on Distributed event-based system*, pp. 183–194. ACM, 2011.

[5] Gonzalo Navarro and Mathieu Raffinot. Fast and simple character classes and bounded gaps pattern matching, with application to protein searching. In *Proceedings of the fifth annual international conference on Computational biology*, RECOMB '01, pp. 231–240, New York, NY, USA, 2001. ACM.

[6] Gonzalo Navarro and Mathieu Raffinot. *Flexible Pattern Matching in Strings: Practical on-line search algorithms for texts and biological sequences*. Cambridge University Press, 2002. ISBN 0-521-81307-7. 280 pages.

[7] P.D. Vouzis and N.V. Sahinidis. GPU-BLAST: using graphics processors to accelerate protein sequence alignment. *Bioinformatics*, Vol. 27, No. 2, pp. 182–188, 2011.

[8] Eugene Wu, Yanlei Diao, and Shariq Rizvi. High-performance complex event processing over streams. In *Proceedings of the ACM SIGMOD International Conference on Management of Data (SIGMOD'06)*, pp. 407–418. ACM, 2006.

[9] 笹川裕人, 金田悠作, 有村博紀. 大規模並列文字列照合の gpu による高速化. 第 10 回情報科学技術フォーラム, pp. D–009, 2011.

[10] 笹川裕人, 金田悠作, 有村博紀. 長大な拡張文字列パターンに対する大規模文字列照合の高速化. 第 4 回データ工学と情報マネジメントに関するフォーラム, pp. D7–1, 2012.