

大規模並列文字列照合のGPUによる高速化 GPU Acceleration of Large-scale Parallel String Matching

笹川 裕人*
Hirohito Sasakawa

金田 悠作*
Yusaku Kaneta

有村 博紀*
Hiroki Arimura

概要: 本稿では、多数のパターンを並列に照合する文字列照合システムのGPU上での効率よい実現方法を提案する。

1. はじめに

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、ストリーム処理など、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、データ解析の観点から文字列照合の更なる高速化が求められている。このため、FPGAやGPUなどの並列ハードウェアによる高速化が注目されている。

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るが、2000年代半ばから、開発環境の整備とハードウェアの性能向上により、汎用的な計算のGPUによる高速化が注目されている。しかし、GPUはメモリサイズやアクセスパターンに強い制限を持つため、画像処理や、物理シミュレーションへの応用が中心であり、大規模な離散的計算問題への適用は余り進んでいない。

そこで本稿では、GPUを用いた大規模文字列照合の高速化について考察する。パターンのクラスとして、直線状の正規表現であるCBGパターンのクラスを考える。パターン照合手法には、非決定性有限オートマトン(NFA)の模倣に基づく手法を採用し、次の2つのアルゴリズムを与える。NFA-ARRAYは、NFAを整数配列で表現する素朴な手法である。これに対して、NFA-BPは、NavarroとRaffinot [1]が提案した拡張SHIFT-AND法を用いており、前処理時にNFAを一組のビットマスクを用いてコンパクトに表現し、実行時にビット演算と算術演算を用いてNFAを高速に模倣する。

実験では、提案のNFA-BP手法は素朴な手法に比べて著しく記憶領域を削減でき、CBGパターンに対してPROSITEデータ上で60倍程度の高速化を達成した。

2. 準備

2.1 CBGパターン

CBGパターン(Class of characters and Bounded size Gapsパターン)とは、正規表現の部分クラスの一つであり、文字クラスと、長さ制限ギャップを含むパターンである。これは、 $P_1 = C-[ABC]-x(1,3)-C-B$ のような直線状の正規表現であり、文字クラス[ABC]と長さ制限ギャップ $x(1,3)$ を許す。

2.2 GPUと並列文字列照合

代表的なGPUであるNvidia社のGeforce GTX480では、多数のSM(Stream Multi-processor)を備え、各SMは32個のSIMD演算コアを持つ。各演算コアは、SM内に搭載された高速な共有メモリ(Shared Memory, ユーザ領域16KB, キャッシュ領域48KB)と、GPUボー

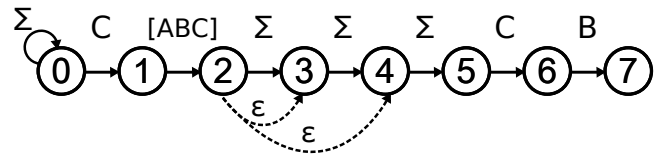


図1: CBGパターン P_1 に対するNFAの例

ドに搭載された、容量の大きなグローバルメモリ(1~数GB)にアクセスし、並列に計算を行うことで、高速な計算を実現する。

我々の並列文字列照合アーキテクチャでは、一つのパターンを一つの演算コアに割り当てる。GPUにおいては、共有メモリのサイズが限られており、例えば、32個の物理スレッドで、16KBの共有メモリを共有すると、演算コア1個当たり500(byte)しか使えない。そのため、SM当たり多数の物理スレッドを同時に実行するには、パターン1個当たりが使用する記憶領域の削減が最も重要である。

3. アルゴリズム

本節では、実験で用いるNFAを用いた照合アルゴリズムを説明する。以下では、 Σ をアルファベットとし、 m をNFAの状態数、 n をテキストの長さ、 w を計算機の語長(bit)とする。

3.1 配列を用いた基本アルゴリズム

アルゴリズムNFA-ARRAYは、よく知られたNFA模倣に基づく正規表現照合アルゴリズムを、CBGパターンに適用したものである。前処理で、入力されたパターン P を受理する図1のようなNFAを構築する。図で、文字集合 $A \subseteq \Sigma$ をラベルに持つ実線矢印は、 A の任意の文字による遷移を表しており、点線矢印は空遷移を表す。パターンと空遷移、新旧のNFA状態は4本の整数配列で表現する。実行時には、与えられたテキストから文字を1字読むごとにNFAを遷移させ、NFAが受理状態に到達したとき、パターンの出現を報告する。

3.2 ビット並列手法を用いた照合アルゴリズム

アルゴリズムNFA-BPは、ビット演算と数値演算レジスタ内の並列性を利用し、計算を高速化する手法である**ビット並列手法**(bit-parallel method)に基づいて、3.1節のNFA-Arrayの記憶領域と計算時間を削減したものである。ここでは、Navarroらによる拡張SHIFT-AND法 [1]を用いる。図2にアルゴリズムの疑似コードを示す。詳細は文献 [1]を参照されたい。

前処理時には、手続きPREPROCESSが、パターン P を解析し、NFAを表現する長さ m (bit)のビットマスク $(Ch[c])_{c \in \Sigma}, Ebeg, Eend$ を構築する。マスク $Ch[c] \in \{0, 1\}^m$ は文字 c による遷移を表しており、各位置 $1 \leq i \leq m$ に対して、 $Ch[c][i]$ は $c \in P[i]$ の時は1であり、そ

*北海道大学大学院情報科学研究科, Hokkaido University

Algorithm 1 Gaps-SHIFT-AND

```

1: procedure PREPROCESS( $P$ )
2:   本文を参照.

3: procedure SEARCH( $Ch, Ebeg, Eend, T$ )
4:    $n \leftarrow \text{length}(T)$ ;
5:    $S \leftarrow 0^L$ ;
6:   for  $i \leftarrow 1$  to  $n$  do
7:      $S \leftarrow ((S \ll 1) \mid 0^{L-1}1) \& Ch[T[i]]$ ;
8:      $S \leftarrow S \ll ((Eend - (S \& Ebeg)) \& \sim Eend)$ ;
9:     if  $S \& 10^{L-1} \neq 0^L$  then
10:      report an occurrence at  $i$ ;

```

図2: Gaps-SHIFT-AND アルゴリズム

れ以外の時は0である。マスク $Ebeg$ と $Eend \in \{0, 1\}^m$ は空遷移を表し、NFA 中の空遷移が連続する場所 (図1の状態2から4) の先頭状態 (同, 状態2) と終端状態 (同, 状態4) に対応する位置にそれぞれビット1を立て、それ以外の位置にビット0を立てている。

実行時には、手続き SEARCH でテキスト T の各文字を走査しながら、 $Ch, Ebeg, Eend$ を用いて、パターン照合を行う NFA を模倣する。図2の21行目では、SHIFT と AND 演算を用いて文字遷移を模倣し、22行目では、減算によるビット伝搬を用いて空遷移を模倣する。

4. 理論的解析

本節では、前節の2つの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。

記憶領域: ビット並列手法は、パターン1個当たりが使用する記憶領域の削減に有効である。NFA-ARRAY は、パターンを表す最大長さ $\ell \leq |\Sigma|m$ の文字配列1本と、空遷移と新旧の NFA 状態を表す長さ m の整数配列3本を用いるので、合計 $S = 3m + \ell/b$ (word) の記憶領域を使用する。ここに b は1文字のビット数である。NFA-BP は、 $|\Sigma|$ 個の文字遷移用マスクと2個の空遷移用マスクを用いるので、合計 $S = (|\Sigma| + 3) \lceil \frac{m}{w} \rceil$ (word) の記憶領域を使用する。

例として、アミノ酸配列のように $|\Sigma| = 20$ で、パターン長が $m = 32$ 、文字集合の平均サイズが1.5である場合、すなわち $\ell = 1.5 \times 32 = 48$ の場合、NFA-Array と NFA-BP の使用記憶領域は、それぞれ102 (byte) と23 (byte) である。この場合に、NFA-BP は NFA-ARRAY の1/4前後の記憶領域しか使わないことがわかる。

計算時間: ビット並列手法は、スレッド1個当たりの計算速度の向上にも有効である。NFA-ARRAY は、入力文字ごとに長さ m の配列を走査するので、その計算時間は、 $T = O(\ell n) = O(mn)$ 時間を要する。NFA-BP は、長さ $\lceil \frac{m}{w} \rceil$ (word) のビットマスクに対して入力文字ごとに1 word 当たり定数回の演算を行う。従って、前処理に $P = O(|\Sigma| \lceil \frac{m}{w} \rceil)$ 時間かかり、照合に $T = O(n \lceil \frac{m}{w} \rceil)$ 時間である。結果として、NFA-BP は NFA-ARRAY の $O(w)$ 倍程度の高速化が期待できる。

表1: アルゴリズムに対する総計算時間 (sec)

Algorithm	GPU CPU	q	Number of Patterns p				
			100	200	400	800	1600
NFA-AR	CPU		144	279	543	NA	NA
NFA-AR	GPU	4	341	457	464	NA	NA
NFA-BP	GPU	14	6.93	6.98	7.24	12.1	19.7

表2: SM 毎の物理スレッド数に対する総計算時間 (sec)

Algo	NFA-AR	NFA-BPL	NFA-BP	NFA-BP
Threads q	4	14	14	32
Time (sec)	464	7.24	8.38	12.1

5. 実験

前節で説明した2つの照合アルゴリズム NFA-ARRAY (以下では NFA-AR) と NFA-BP を実装し、計算機実験を行った。データとして ExPASy の PROSITE データベース (<http://expasy.org/prosite/>) から1600個のCBGパターンと最大180MBのテキスト ($|\Sigma| = 20$) を取得し、GPU (NVIDIA Geforce GTX480, SM15機, 1.5GBメモリ, 480コア) を備えたPC (Corei7-930 2.8GHz, 12GBメモリ) 上で実験を行った。

データのユーザ用共有メモリ (16KB) への明示的な割り付けは行わず、キャッシュ用共有メモリ (48KB) への自動割り付けを利用した。入力テキストはサイズ $B = 1MB$ ずつ、GPU のグローバルメモリに置いた。各 SM には、全ての物理スレッドが共有メモリに収まるよう事前に計算した定数 q 個の物理スレッドを割り当て、総パターン数 p 個の論理スレッドを実行した。

表1に、物理スレッド数 q とパターン数 p を変化させた時の計算時間を示す。表より GPU 版の NFA-AR は CPU 版に比べ同程度か2倍程度遅く、GPU 版の NFA-BP は GPU 版の NFA-AR より約60倍高速であった。NFA-AR が遅い理由としては、GPU のメモリアクセス制約による速度低下が考えられる。また、NFA-BP が高速であったのは、NFA-AR に比べ約3.5倍の物理スレッドを同時実行でき、個々のスレッドもより高速に実行できるためと考えられる。

表2では、比較のため、データを明示的に共有メモリに割り付ける NFA-BPL を用意し、自動的な共有メモリの割り当てを用いる NFA-BP と比べた。その結果、自動的な共有メモリ割り付けを用いても、1割程度の速度低下しかなかった。これより、本稿の問題では、メモリ使用量の節減が、細かなメモリ割り付けより重要であった。

6. 今後の課題

複数 SIMD コアを用いたパターン照合の高速化と、より複雑なパターンへの拡張は興味深い問題である。

参考文献

- [1] Gonzalo Navarro and Mathieu Raffinot, Extended string matching, Chapter 4, *Flexible Pattern Matching in Strings* Cambridge, 2002.
- [2] NVIDIA, *NVIDIA CUDA C Programming Guide Version 3.2*, 2010.
<http://developer.nvidia.com/>