数理解析研究所講究録
第1799巻 2012年 100-102

GPUを用いた簡潔trieの並列探索

田部井 靖生* 田中 秀宗†

1 はじめに

近年，画像処理専用の演算装置であったGPU（Graphics Processing Unit）をより汎用的な計算に利用する動きがある．特に，数値計算や信号処理の分野では，GPUが広く活用され顕著な効果を得ている．本研究では，より汎用的な問題に対してGPUを活用することを究極的な目標とする．そのための1段階として，今回は，ある種の文字列検索問題を取り上げ，この問題に対する並列アルゴリズムをGPU計算のためのプログラミングモデルの1つであるCUDA上で実装し，その高速化効果を確かめる．

2 CUDAプログラミングモデル

CUDAとは，NVIDIA社が開発した汎用GPU計算のためのプログラミングモデルである[1]．CUDAでは，GPU上での処理は**カーネル**と呼ばれ，複数の**スレッド**によってカーネルが実行される．スレッドには階層が存在し，複数のスレッドの集まりを**ブロック**，複数のブロックの集まりを**グリッド**と呼ぶ．ブロック内のスレッドの数，ブロックの数には制限がある．各階層でスレッドがアクセスできるメモリに違いがあり，全てのスレッドからアクセスできる共有メモリを**グローバルメモリ**，同一ブロック内のスレッドからのみアクセスできる共有メモリを**シェアードメモリ**と呼ぶ．それぞれ，グローバルメモリは大容量であるが低速，シェアードメモリは高速であるが小容量であるという特徴を持っている．ここまでで述べたCUDAの概要を図示したものが図2である．この図では，人形でスレッド，外側の大枠でグリッド，内側の枠でブロックを表している．

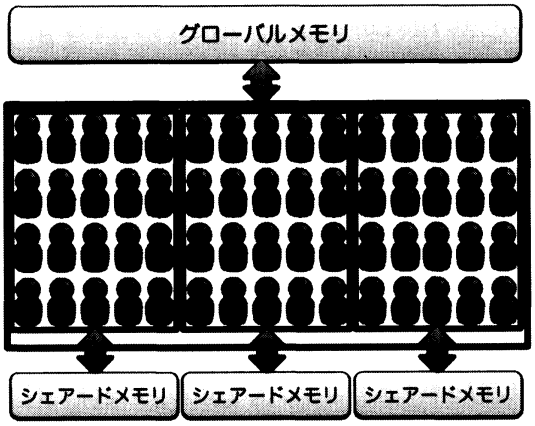

図1: CUDAの概要

3 問題と既存の解法

次のような問題を考える：

入力 文字列（キー）$K_1, K_2, \ldots, K_N$，文字列（クエリ）$Q_1, Q_2, \ldots, Q_n$

質問 各 $i \in \{1, \ldots, n\}$ に対して，キー $K_1, \ldots, K_n$ の中に Q_i と完全に一致する文字列はあるか

例えば，キーとして $K_1 = \mathtt{ACT}$，$K_2 = \mathtt{ACG}$，$K_3 = \mathtt{CTG}$，クエリとして $Q_1 = \mathtt{ACG}$，$Q_2 = \mathtt{ACC}$，$Q_3 = \mathtt{CTG}$ が与えられたとき，$Q_1 \mapsto K_2$，$Q_2 \mapsto \perp$，$Q_3 \mapsto K_3$（$\perp$ はキーの中に一致する文字列が無いことを表す）という対応関係を（例えば表の形で）出力する．$n = 1$ の場合，すなわちクエリが1つだけの場合は，既存研究があり，いくつかの逐次アルゴリズムが知られている．

代表的な手法の1つに，高速にクエリを検索するために，キーを索引化する手法があるが，ここでは索引化したデータを表現するデータ構造としてtrie木を用いる．trie木は辺ラベル付きの木構造で，各辺にアルファベットが割当てられており，頂点から葉までのパスが1つの文字列

*JST ERATO 湊離散構造処理系プロジェクト，mail:yasuo.tabei@gmail.com
†東京工業大学 情報理工学研究科，mail:tanaka7@is.titech.ac.jp
[1]このプログラミングモデルを実現するためのツール群が，Nvidia社によって提供されている（http://www.nvidia.com/object/cuda_home_new.html）．

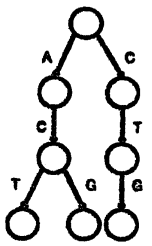

図 2: trie の例

に対応している．例えば，キーが $K_1 = \mathtt{ACT}$，$K_2 = \mathtt{ACG}$，$K_3 = \mathtt{CTG}$ のときの trie は図 3 のようになる．

実用的には，この trie を格納するための領域を節約するため，簡潔データ構造が用いられることが多い．簡潔データ構造とは，表現するデータを極限まで圧縮した上で，少しの保持情報を付け加えることで，有用な問合せにも答えられるデータ構造である．例えば，この問題を解くためのライブラリ群である tx-trie [?] は，木構造の簡潔データ構造である LOUDS [?, ?] を利用している．

4 並列化

前章で述べた trie を用いたアルゴリズムの並列化を行った．以降ではまず，ブロック数を 1 つとしたときの並列化手法について述べる．

まず，各クエリを各スレッドに割当て，別々のクエリに対して trie を並列に探索するという並列化手法が考えられる．この場合，trie とクエリをグローバルメモリに置き，各スレッドは割当てられたクエリの文字列を探索し，その結果をグローバルメモリに書き出す．以後この手法を**クエリ並列**と呼ぶこととする．

次に，trie の各段を各スレッドに割当て，1 つのクエリに対して trie を並列に探索するという並列化手法を提案する．複数のクエリに対しては，パイプライン的に各クエリに対する処理を重ね合わせることで高速化を図る．この場合，trie とクエリをグローバルメモリに，スレッド数と同じ長さの配列をシェアードメモリに用意する．この配列には，各スレッドが次に探索を開始する trie の頂点番号が格納される．各ステップで，各スレッドはシェアードメモリの配列から頂点番号を，グローバルメモリから担当するクエリの文字を読み，1 段だけ探索し，その結果をシェアードメモリの配列の次スレッドの担当部に書き込む．全てのクエリの処理が終わるまでこの処理を繰返す．以後この手法を *trie* **分割**と呼ぶこととする．

ブロック数を複数使う場合は，クエリをブロック数で等分割し，各ブロックで上で述べた並列化手法を並列に行う．例えば，クエリ数が $n = 1000$，ブロック数が 100 の場合，各ブロックには 10 個のクエリが割当てられる．

表 1: 実験結果（1 ブロック，クエリ数 $n = 100,000$）

クエリ長 m	87	174	348	696
逐次	3.76	9.96	30.18	78.02
クエリ並列	1.86	2.27	3.39	8.84
	(2.02 倍)	(4.38 倍)	(8.91 倍)	(8.83 倍)
trie 分割	9.65	11.10	15.09	22.89
	(0.39 倍)	(0.90 倍)	(2.00 倍)	(3.41 倍)

(単位は秒，括弧内は対逐次の高速化効果)

5 実験

前章で述べた 2 つの手法と逐次アルゴリズムを実装し，実行時間の比較を行った．これらの手法は，tx-trie [?] を基に実装した．入力には，m 文字のゲノムのデータ（アルファベット $\{\mathtt{A},\mathtt{C},\mathtt{G},\mathtt{T},\mathtt{N}\}$，N は不明を表す）を n 本用意し，キーとクエリの両方に利用した．実験環境は次の通りである：

- CPU：AMD Phenom X4 9850 (2.5GHz × 4 cores)
- GPU：Tesla C2070 (1.15GHz × 448 cores)
- CUDA 4.0

trie 分割の場合，ブロック当たりのスレッド数はクエリ長 m となるが，比較のため，クエリ並列の場合でも，ブロック当たりのスレッド数をクエリ長 m で設定する．

ブロックが 1 つの場合　クエリ数 n を 100,000 で固定し，クエリ長 m を変化させたときの実行結果を表 5 に示す．クエリ数 m を 348 で固定し，クエリ長 n を変化させたときの実行結果を表 5 に示す．

どの場合でも，クエリ並列が trie 分割より高い高速化効果を得た．しかし，入力サイズを大きくしていったとき，クエリ並列の高速化効果が頭打ちになっているのに対し，trie 分割の高速化効果は高まっている．これにより，大き

表 2: 実験結果（1 ブロック，クエリ長 $m = 348$）

クエリ数 n	1,000	10,000	100,000	1,000,000
逐次	0.30	3.02	30.18	370.34
クエリ並列	0.03	0.3	3.39	66.48
	(9.15 倍)	(9.03 倍)	(8.91 倍)	(5.57 倍)
trie 分割	0.20	1.55	15.09	162.91
	(1.56 倍)	(1.95 倍)	(2.00 倍)	(2.27 倍)

(単位は秒，括弧内は対逐次の高速化効果)

表 3: 実験結果（複数ブロック，ブロック数 32）

クエリ長 m	87	174	348
逐次	52.02	145.09	370.34
クエリ並列	1.48	4.41	13.67
	(35.25 倍)	(32.93 倍)	(27.10 倍)
trie 分割	4.17	6.30	16.24
	(12.47 倍)	(23.02 倍)	(22.80 倍)

(単位は秒，括弧内は対逐次の高速化効果)

なサイズの入力に対しては，trie 分割が効果を発揮すると考えられる．

ブロックが複数の場合 ここでは，クエリ数 n を 1,000,000 で固定する．ブロック数を 32 で固定し，クエリ長 m を変化させたときの実行結果を表 ?? に示す．また，クエリ長 m を 348 で固定し，ブロック数を変化させたときの実行結果を表 5 に示す．

クエリ長 87，ブロック数 32 のとき，クエリ並列で 35.25 倍の高速化効果が得られた．どの場合でも，trie 分割よりクエリ並列の方が高い高速化効果が得られている．2 つの手法とも，クエリ長を伸ばすと高速化効果が頭打ちになり，ブロック数にはあまり影響を受けないという共通した特徴が見られる．これは，前段階で行った，ブロックが 1 つの場合のクエリ並列の特徴を受け継いだことによるものだと考えられる．

全ての実験でクエリ並列が trie 分割を上回る結果となったが，これは trie 分割のメモリへの不連続アクセスによるものだと考えられる．trie の異なったレベルにある頂点は，LOUDS の場合，メモリ上の不連続な領域に格納される．CUDA では，グローバルメモリへの連続した領域へのアクセスが高速にできるという特徴があるが，これを活かせなかったことが，trie 分割が威力を発揮できなかった原因だと考えられる．

表 4: 実験結果（1 ブロック，クエリ長 $m = 348$）

ブロック数	32	64	128	256
逐次	370.34	370.34	370.34	370.34
クエリ並列	13.67	13.84	13.89	14.08
	(27.10 倍)	(26.76 倍)	(26.66 倍)	(26.30 倍)
trie 分割	16.24	17.72	17.32	17.82
	(22.80 倍)	(20.90 倍)	(21.39 倍)	(20.78 倍)

(単位は秒，括弧内は対逐次の高速化効果)

6 結論

本研究では，ある文字列検索問題に対して GPU 上での並列化手法を提案し，計算機実験で 35 倍程度の高速化効果を実現した．この研究を通しても，GPU を用いるとどのような手法を用いれば高速化できるのかは明らかになっていないのが現状である．今後，高速化できる手法を明らかにすることで，GPU の理論的計算モデルの構築，および GPU アルゴリズムの理論的解析など研究課題が生まれることを期待する．