Vol.2012-ARC-201 No.23
2012/8/2

Content-Centric Networking における GPGPU を用いた 文字列探索システムの提案と評価

池内一将[†1]　石田慎一[†2]　西宏章[†2]

近年，様々なインターネットサービスの多様化により，インターネットはより人々の生活の中心的位置を占めつつある．そして，ネットワークセキュリティ等のコンテンツベースのサービスを提供する Content-Centric Networking (CCN) の必要性も高まっている. ネットワークの中継点として存在するルーターは, 高機能化や Deep Packet Inspection (DPI) 技術の進歩により，単なるパケット中継器に留まらず，パケットペイロードを解析することにより CCN のようなサービス指向型ネットワークの実現に寄与できる．我々はこのようなサービスを提供するルーターとして，サービス指向型ルーター(SoR)を提案している．一方で，演算集約性と並列性の高いデータ構造に対して，安価なハードウェア価格で高速な処理を実現できる技術として General-Purpose computing on Graphics Processing Unit (GPGPU) が注目されており，また，ルーター専用のパーツではなく，汎用的な製品を利用したルーターも登場していることから，将来的にルーターに GPGPU を搭載する事も考えられる．また，ルーターには多数のユーザーのパケットストリームが流れることから，それぞれのユーザーに対してサービスを提供できる必要がある．このような背景から，本研究では，CCN における GPGPU を利用した文字列探索システムを構築するための予備評価として，複数のユーザーのトラフィックデータへの処理を実現するシステムの提案と評価を行う．

Implementation and Evaluation of A GPU-based String Matching System on Content-Centric Networking

Kazumasa Ikeuchi[†1]　Shinichi Ishida[†1]　Hiroaki Nishi[†2]

In recent years, demands for Content-Centric Networking (CCN) to provide a content-based service, such as a network security service, with many users are increasing in the situation that Internet traffic are more and more growing. Although, routers on CCN are required to process for large amount string information of traffic data, in general, a network processor which can achieve such a fast processing is very expensive compared with a general purpose processer. And nowadays, General Purpose on Graphics Processing Unit (GPGPU) attracts our attention due to its merits of low-cost and strong calculation power for a data which has highly calculation intensive and parallelized structure. In such a background, it is meaningful that we utilize GPGPU as a network processor for string matching on a router to achieve effective and low-cost system. This paper implements and evaluates a new string matching system which achieves a scalable processing for traffic data of multiple users on CCN.

1. はじめに

近年，動画や音楽配信，クラウドや twitter 等の豊かなサービスの増加につれて，インターネットは我々の生活基盤の中心的位置を占めつつある．このような動向は，ネットワークにおけるトラヒック量のみならず，ユーザーのコンテンツに対する要求の高まりという変化ももたらした．このような要求に対する新しいネットワークアーキテクチャとして，Content-Centric Networking (CCN) (1)が提案されている．CCN は，既存の対話型ネットワーク構造を変えることなく、ユーザーの要求するコンテンツに応じて柔軟にルーティングを決定し，ユーザーの嗜好に合った，より高度で豊かなコンテンツやサービスを提供できる可能性がある．

トラヒック情報を把握する手段として，ネットワーク中継点に存在するルーターは非常に優れた位置に存在する．近年，ルーターは高い計算能力とパケット交換能力の獲得と, Deep Packet Inspection (DPI) (2)について多くの研究がなされていることから, 単なる IP パケットの中継機器に留まらず，様々なレイヤーに及ぶパケット解析が可能となっている．従来エンドホストのみで収集，解析が行われていたネットワーク情報を，ルーターが収集，解析することにより，ルーターが様々なサービス提供する事が可能となる．このようなルーターとして，我々はサービス指向型ルーター (SoR) (3)を提案している．SoR は CCN を実現する手段の一つである．

その一方で，General-Purpose Computing on Graphics Processing Unit (GPGPU) (4) は, 高い並列処理能力を持つ計算手法として注目されている．Graphics Processing Unit (GPU)は，画像処理のための行列演算等，並列性と演算集約性の高い処理を高速に行うプロセッサである．また，ハードウェア価格が比較的安価でありホストと PCI Express インターフェースを用いて容易に接続することができる．この GPU を，より汎用的な学術的計算に応用する技術を GPGPU と呼び，近年，多くの研究がなされている．同様に，GPGPU を利用した Parallel Failure-less Aho-Corasick (PFAC) (5)アルゴリズム等の文字列探索アルゴリズムの研究も盛んに行われており，高速な文字列探索における

†1 慶應義塾大学理工学研究科
Graduate School of Science and Technology, Keio University
†2 慶應義塾大学理工学研究科
Graduate School of Science and Technology, Keio University
†3 慶應義塾大学システムデザイン工学科
Dept. of System design Engineering, Keio University

GPGPU の有効性が示されている．

従来，専用のハードウェアで構成されていたルーターであるが，近年，ハードウェアコストの抑制や，より広範なアプリケーションへの適用を考え，汎用のハードウェアを利用したルーターも登場している．一般的な PC と同様の拡張スロットを有し，GPGPU を搭載可能なモデルも存在している．この流れは今後加速していくと考えられると共に，GPGPU により DPI を高速化したルーターの登場も想定可能となってきた．そこで，我々はネットワークパケットストリームに対する DPI を，GPGPU を用いて高速化する手法について検討し，予備評価を得ること目的とした研究を行っている．

SoR に搭載される Smart Linux Interface Monitor (SLIM) は多数のユーザーのパケットが含まれるトラヒックから，特定ユーザーのコネクション毎の L7 情報を抽出する．SoR では，抽出されたコネクション毎の L7 情報に，対応する異なるパターンセットを用いた探索処理を行うため，ユーザー数が増加してもスループットを維持できるスケーラビリティを保つことは重要な問題である．複数のコネクションの L7 情報に対して，それぞれ異なるパターンセットの探索処理を行う新しい実装として，マルチユーザーサポートを提案する．

以上より，本論文では，ネットワークパケットストリームに対して高速な文字列探索システムを行うプロセッサとして GPGPU を，文字列探索アルゴリズムとして PFAC アルゴリズムを応用し，複数のユーザーに対してスケーラブルな探索を実現するシステムの提案と評価を行う．

2. GPGPU・CUDA

2.1 GPGPU

GPU を本体の目的である画像処理や動画のレンダリングではなく，より一般的な演算用途に利用する技術が GPGPU である．GPU は複数のスレッドが一つの命令を並列実行する Single Instruction Multi Thread (SIMT) 型プロセッサで，行列演算などの並列計算に対する処理性能が高い特徴がある．

2.2 メモリ構造

図 1 に GPU コンピューティングの概略を示す．GPU (デバイス)は PC の PCI Express(PCIe)のスロットに装着し CPU(ホスト)に接続される．また，表 1 に示すように，デバイスはホストから転送されてきたデータを保持するための各種のメモリを持ち，デバイスによる処理を行う際は，ホストメモリ - デバイスメモリ間で必ずデータのロード，ストア処理が発生する．グローバルメモリはデバイスメモリとも呼ばれ，オフチップで最も容量が大きく，すべての入出力データはデバイスメモリを経由し処理を行う．また，32bit 幅×32768 本のレジスタと，64KB のシェアードメモリがオンチップ実装され，GPU コアクロックと同じ速度で動作している．また，読み取り専用メモリとしてテクスチャメモリが存在する．テクスチャメモリは，アクセス速度は低速である一方，空間的局所性に対するキャッシュ機構を持っている．

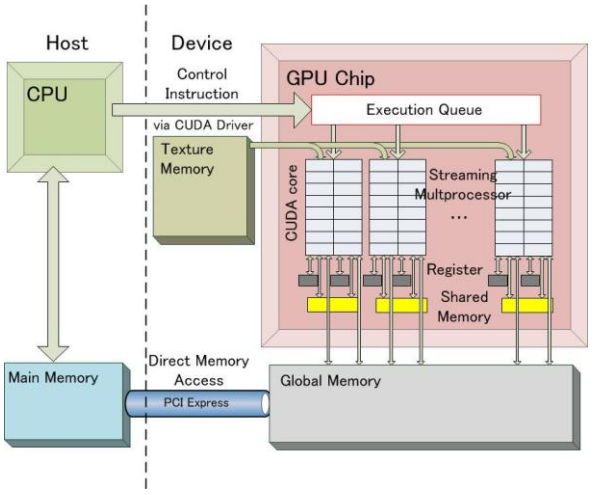

図 1　GPU コンピューティングの概略

表 1　GPU メモリ

Memory	Location	Cached	Access
Register	On	n/a	R/W
Shared	On	n/a	R/W
Global	Off	Yes	R/W
Texture	Off	Yes	Read only

2.3 CUDA

Compute Unified Device Architecture (CUDA) (6)は，NVIDIA 社が提供する CPU･GPU 統合型プラットホームで，OpenCL や Cg 等の高度なグラフィック処理言語ではなく，拡張 C 言語のみで GPU で動作するプログラムの記述が可能である．

一般的に，CPU ではコアの数と同数のスレッドを用いて処理を行うが，GPU における CUDA プログラミングでは 10^3~10^5個に及ぶ多数のスレッドを用いて並列処理を行う．CPU プログラミングで一般的な POSIX スレッドと比較して，CUDA スレッドの生成コストは極めて小さく，軽量である．デバイスで実行するプログラムをカーネルと呼び，スレッドはカーネル上で動作する最小単位のプロセスである．実際には，32 個のスレッドを 1 ウォープという単位でまとめ処理の並列化とスケジューリングのパイプライン化が行われる．一つのウォープ内では常に単一の命令が発行されるため，フローコントロール命令等で，ウォープ内における異なるスレッドに異なる命令を実行しようとする場合，それぞれの命令を対応するスレッドに 1 度ずつ実行するため，このような実装はスレッドの並列度を低下させ，

従ってパフォーマンスの大幅な低下をもたらす．これをウォープ・ダイバージェンスと呼ぶ．

3. Parallel Failure-less Aho-Corasick algorithm

3.1 Aho-Corasick algorithm

Aho-Corasick algorithm(AC 法) (7)は，Knuth-Morris Pratt algorithm (8)を複数のパターンに拡張した，複数文字列探索アルゴリズムである．探索されるパターンセットから決定性有限オートマトンである AC オートマトンを構築し，テキストデータの入力に対して状態を遷移する事により探索処理を行う．

AC オートマトンは 3 つの関数から構成される．1 つは，それぞれの状態に対する入力文字に応じた状態遷移を定義する goto 関数，2 つ目は入力文字に有効な状態遷移が定義されていない時の状態遷移を定義する failure 関数，最後に特定状態において検出した文字列を出力する output 関数である．例として，図 2 にパターンセット{“he”, “hers”, “his”, “she”}からなる AC オートマトンを示す．各ノードの数字は状態番号を，文字は入力文字に対応する状態遷移を表す．また，実線矢印は照合がヒットした際の goto 関数による状態遷移を，破線矢印は照合がミスヒットした際の failure 関数による状態遷移を，そして二重丸で表現されるノード番号 2，4，6，9 は出力すべき output 関数が定義される状態である事を示す．

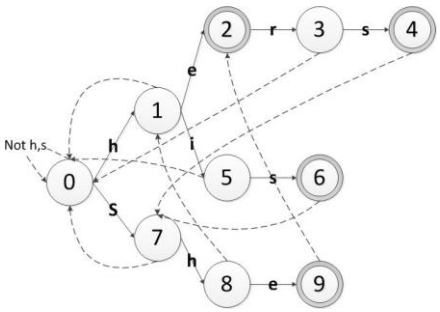

図 2　AC オートマトン

3.2 Parallel Failure-less Aho-Corasick algorithm

Parallel Failure-less Aho-Corasick algorithm (PFAC 法) は，AC 法を基に CUDA アーキテクチャに応用したアルゴリズムである．PFAC 法におけるオートマトンを PFAC オートマトンと呼び，図 3 に示す．PFAC オートマトンでは，ウォープ・ダイバージェンスを回避するため，AC オートマトンの 3 つの制御構文の内，failure 関数を排除する．しかし，failure 関数を排除し，goto 関数による有効遷移のみを記述する場合，テキストデータの特定の位置を起点とする探索処理のみしか検出できない問題がある．

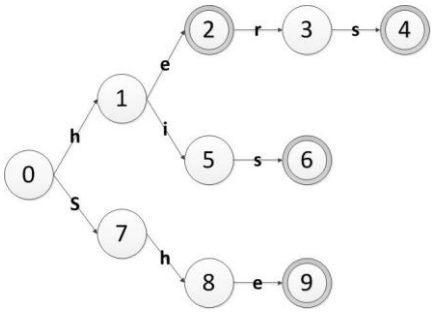

図 3　PFAC オートマトン

これを解決するため，PFAC 法ではテキストデータの全ての位置をスレッドの検索開始位置として割り当てる．図 4 に PFAC 法における各スレッドの文字列探索処理の様子を示す．Failure 関数を記述する必要がないため，オートマトンを記述する状態遷移表のサイズも縮小でき，メモリ使用量を節約できる．また，PFAC 法では，各スレッドは有効遷移がなくなった時点で探索処理を終了するため，スレッドの寿命を短縮でき，リソースを有効に活用することが出来る．

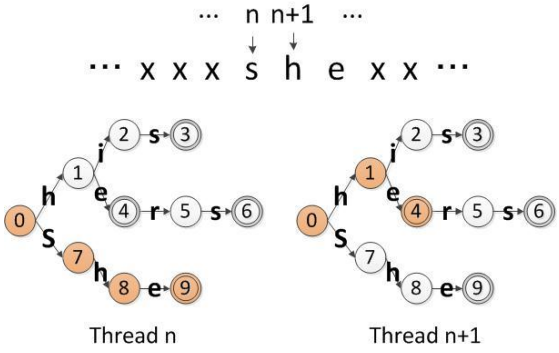

図 4　PFAC オートマトンの振る舞い

4. マルチユーザーサポート

SoR では，抽出されたコネクション毎の L7 情報に対し，異なるパターンセットを用いた探索処理を行うため，ユーザー数が増加してもスループットを維持できるスケーラビリティを保つことは重要な問題である．そこで本実装では，pthread (9)を利用したホストにおけるマルチスレッド環境を構築し，単一ホストスレッドが単一ユーザーのパターンセットからオートマトンを構築し，対応するテキストデータに対し，独立して文字列探索処理を管理する．ホストスレッドは，ホストでの PFAC オートマトンの構築や GPU へのテキストデータやオートマトン，探索結果の転送，GPU カーネルの実行命令の発行を行い，デバイスからライトバックされた探索結果データをファイル出力した時点で処理を終了する．

5. 実装・評価

5.1 実装

5.1.1 ホストメモリアロケーション

ホスト - デバイス間メモリ転送は Direct Memory Access (DMA)で行う．テキストデータのホストメモリは cudaHost Alloc() API に，フラグ ”cudaHostAllocWriteCombined” を組み合わせて設定する．このフラグによって確保されたホストメモリ領域は CPU の L1，L2 キャッシュから解放され，他のアプリケーションが利用可能になる．さらに，PCIe を通しての転送がスヌープされないため，転送速度の向上が期待できる．CPU キャッシュラインから切り離されるため，ホストからテキストデータの Read の速度は低下するが，テキストデータの処理は PCIe を通した，ホストからデバイスへのメモリ転送と，デバイスでの探索処理のみでアクセスされるため処理速度低下を引き起こす可能性はない．

5.1.2 デバイスメモリアロケーション

ホストが各ユーザーのパターンセットから構築したオートマトンは，デバイスメモリに転送された後，テクスチャメモリにバインドされる．テクスチャメモリは空間的局所性に対するキャッシュが有効なため，入力文字に対応する遷移先の取得速度の高速化が可能であると考えられる．

また，ルートノードの情報を取得するため，スレッドが頻繁にテクスチャメモリのルートノード領域にアクセスすることはパフォーマンスのボトルネックとなることが考えられる．そこで，オートマトンに定義される状態遷移の内，ルートノードを起点とする遷移，すなわち初期遷移を高速なアクセスが可能なシェアードメモリに格納することで，テクスチャメモリへの初期アクセスを回避し，探索処理を高速化する．シェアードメモリの容量は数十 KB と小容量であるので，初期状態における ASCII コード 256 個に対する遷移先の整数型状態番号のみを保持することで，各スレッドが最初の入力文字に対する状態を取得するのにかかる初期オーバーヘッドを軽減する．

5.1.3 実験環境

本実験に使用した環境について述べる．文字列探索を行うパターンセットとして，オープンソースの侵入検知システムである Snort ver.2.9.0.5 (10) の web-activex.rules, web-client.rules, web-iis.rules, web-misrules パターンファイルから，content 文で指定される正規表現パターンを抽出し，利用した．1 パターン当たりの長さは 32 バイトとし，1 つ辺り 200 個のパターンを持つパターンファイルに対して探索処理の実験を行った．また，テキストデータとして，西研究室のネットワークトラヒックでキャプチャした TCP パケットデータから L7 情報を抽出して利用した．抽出には，既出の L7 解析ソフトウェアである SLIM を用いた．

実験を行ったハード・ソフトウェア環境として，ホスト環境を表 2 に，デバイス環境を表 3 に示す．CPU は Intel Xeon E5630 2.67GHz，ホストメモリを 32GB，GPU は NVIDIA tesla C2070 を使用した．OS は Cent OS 6.2 を，CUDA Tool Kit はバージョン 4.2 を使用した．また，実験結果としてホストからデバイスへのテキストデータ転送時間，カーネルの実行時間，デバイスからホストへの探索結果転送時間とテキストデータサイズ，ホストスレッドの並列度を各パラメータとして取得した．

表 2　ホスト環境

CPU	Intel Xeon E5640 2.67GHz
OS	Cent OS 6.2
チップセット	Supermicro X8DTG-QF
メインメモリ	DDR3 32GB
CUDA Tool Kit Version	4.2
CUDA Driver Version	275.33

表 3　デバイス環境

NVIDIA tesla C2070	コア動作クロック	1.15 [GHz]
	CUDA コア数	448
	デバイスメモリ容量	5376 [MB]
	メモリバンド幅	384 [bit]
	メモリ動作クロック	1.49[GHz]

5.2 評価

5.2.1 実験 1

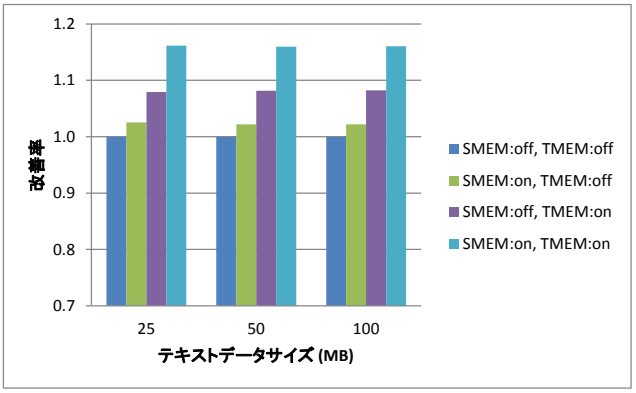

図 5　1 並列度における
シェアードメモリ，テクスチャメモリによる改善率

本実験では，被探索テキストデータ一つ，パターンセット 1 つの場合における，シェアードメモリとテクスチャメモリによる GPU のカーネル実行時間の評価を行った．図 5 は，横軸に処理を行うテキストデータの容量を，縦軸に，シェアードメモリとテクスチャメモリの両方を使用しない場合の実行時間を 1 とした場合の各実装における高速化の割合を示す．実験 1 において，グローバルメモリに加え，両メモリを使用する場合のカーネルの実行時間は最大で

16.1%高速化された．また，両メモリを使用しない場合に対する，シェアードメモリのみ，またはテクスチャメモリのみの実装を行った場合の改善率の和よりも，両メモリを同時に使用した場合の方が約 5.7%高い改善率が得られた．

5.2.1 実験 2

本実験では，実験 1 で有効性が確認されたシェアードメモリおよびテクスチャメモリを実装した場合において，マルチユーザー環境の複数コネクションにおける探索処理を想定し，ホストスレッドを 1 から 8 の並列度で探索処理を行う場合のプログラムの実行時間の評価を行った．図 6 は，横軸をテキストデータサイズとし，各並列度における実行時間の比率を示す．ここでの並列度は 1 度に探索を行うユーザー数を表す．複数のユーザーに対する探索処理を並列度 1 で逐次的に行う場合，オートマトンの構築や GPU における探索処理を含む一連の処理を，ユーザーの数だけ繰り返し行うため，実行時間はユーザー数に比例して大きくなる．一方，本実験において，図 6 は，並列度 8 と並列度 1 を比較し，テキストデータサイズが 25MB の時に約 4.6 倍，100MB の時に約 5.7 倍程度の実行時間しか掛かっていない事が読み取れる．

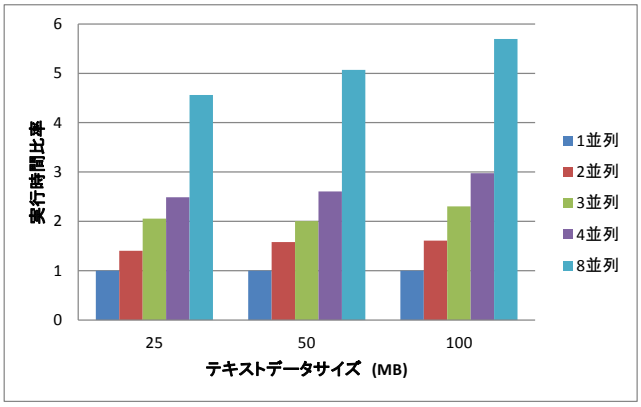

図 6　各並列度における実行時間比率

並列度 1 による逐次的実行の場合，1 つのホストスレッドが処理を 8 ユーザー分逐次的に実行されるため，1 ユーザー分の処理の 8 倍の時間がかかる．それに対して，高並列度におけるマルチホストスレッドの処理は CPU のマルチコアを利用して並列処理を行うため，それぞれの処理を並列に実行することが出来る．本実験環境では 8 コア CPU を使用しているため，並列度 8 は CPU の並列処理性能を十分利用できる実装であったと考えらえる．また，本実験で使用したデバイスは，メモリ転送エンジンとカーネル実行エンジンが独立していることから，ホストスレッドの並列度を高めることにより，ホスト - デバイス間メモリ転送処理とカーネル実行を非同期的に動作させることが出来た事が高速化の原因だと考えられる．

5.2.2 実験 3

本実験では，実験 2 と同様にホストスレッドの並列度を 1 から 8 に変化させた場合の，テキストデータのデバイス転送とデバイスにおけるカーネル実行時間の動作遅延を評価する．ホストがデバイスにメモリの転送命令やカーネル実行命令を発行してから，実際にデバイスでその処理が開始されるまでの遅延には，デバイスへのリクエストキューの輻輳状況が影響すると考えられる．本実験では 2 つの時刻取得 API を利用し，動作遅延を計算している．1 つは gettimeofday()関数で，これにより取得した CPU 時間で，ホストが命令を発行してからデバイスでの処理が終了するまでの時間を取得する．もう 1 つは，CUDA API である cudaEventRecord()関数で，デバイスが実際に処理を開始して制御をホストに返すまでの GPU 時間である．この 2 つの時間の差を取る事により，処理のリクエストの発行から処理開始までの遅延を計算し，並列度との関係を評価した．ホストが処理のリクエストを発行した時刻をt_{c_start}，デバイスが実際に GPU 上で処理を開始した時刻をt_{d_start}，デバイスが処理を終了しホストと同期をとった時刻をt_{stop}とする．図 7 はホストスレッドの並列度と，ホストスレッドがデバイス処理の終了を待つ総実行時間スレッド平均，

$$t_{whole} = t_{c\_start} - t_{stop}$$

に占める実行遅延時間平均，

$$t_{delay} = \frac{t_{c\_start} - t_{d\_start}}{t_{stop} - t_{c\_start}}$$

の割合を示す．並列度が 1 から 3 の範囲では，遅延の割合はほぼ 0%に等しく極めて小さいが，最大並列度 8 においては処理時間の約 17.4%が遅延となっている．並列度 4 以上ではデバイス処理に輻輳による大幅な待機時間がが発生していると考えられる．

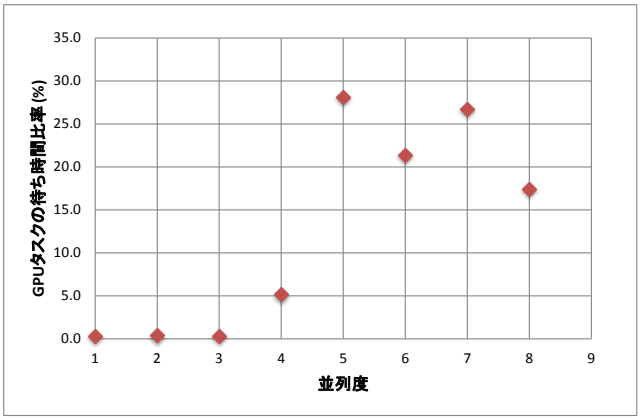

図 7　並列度と実行遅延時間割合の関係

また，図 8 に各並列度における，テキストデータのホスト - デバイス間転送速度を示す．転送速度は，全ホストスレッドにおける総転送データ量を、転送実行時間で割ることで導出した．並列度 1 の場合，約 5.7GBps の転送速度が出た．PCIe の理論上の転送速度は．Gen2 のリンク幅×16

に接続した場合，6.4GBps であり，並列度 1 の時，最大で 89%の転送速度が出ている．一方で，並列度を上昇させると，転送速度は低下し，並列度 4 の時の転送速度は 2.2GBps となる．この原因として，複数のホストスレッドがデバイスにメモリ転送をリクエストすると，そのメモリ転送はデバイスメモリの複数のアドレスに対して平行に実行されることにより，各転送ジョブの切り替えに伴うコストが発生し，転送速度が低下する事が考えられる．また，並列度 4 以上の並列度にした場合，転送速度の低下が認められないのは，4 リクエストが GPU デバイスドライバが許容する最大平行転送リクエスト数の最大値となっている可能性が考えられる．図 7 において，並列度 4 以上で実行遅延の割合の急激な増加は，ホスト - デバイス間通信の際の PCIe の転送速度の低下と，同時に 4 つを超える転送リクエストが発行された場合，超過リクエスは待機することで，ホストで処理を待機するメモリ転送リクエストが増加したことにより引き起こされたと考えられる．

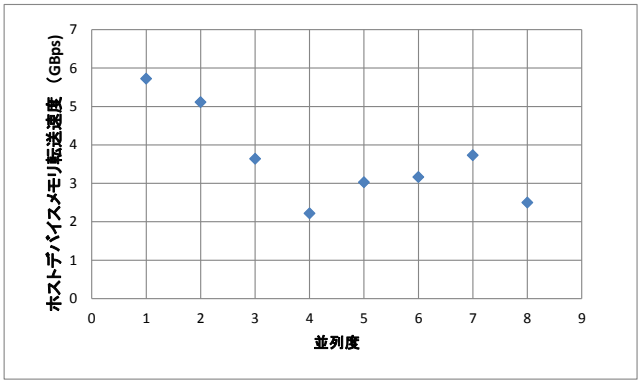

図 8　並列度とホストデバイスメモリ転送速度の関係

さらに，ホストスレッドの並列度とカーネルの実行速度の関係を図 9 に示す．図 8 と同様に，全ホストスレッドにおける総データ処理量をカーネル処理時間で割る事で導出した．並列度が 4，および 5 の時に顕著な実行速度の低下がみられる．デバイスのカーネルの並列実行処理は最大で 16 カーネル行える．一方で，デバイスメモリの帯域幅は 144GBps で非常に高いデータ供給力を持つが，デバイスメモリへのリード・ライト負荷の高いカーネルが複数処理を行う場合は，この処理性能を超過する可能性もある．また，デバイス内でのコンテキストスイッチがボトルネックになることも考えられる．ここで，並列度 6 以上の場合，カーネル実行速度が並列度 1 の場合とほぼ同程度であるのは，図 7，8 に示す，ホスト - デバイス間メモリ転送速度の待機時間の増加と転送速度の低下により，各ホストスレッドがカーネル実行に追いつかなくなり，デバイス内でのカーネルの輻輳が解消された事が原因だと考えられる．

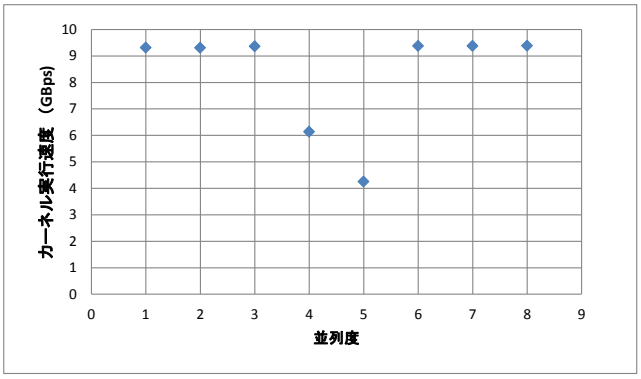

図 9　並列度とカーネルの実行速度の関係

6. 結論・今後の課題

CCN を実現する手段として、SoR における GPU を利用した文字列探索システムの提案，予備評価を行った．GPU のシェアードメモリ，テクスチャメモリを並列度 1 において利用した実装を行い，従来の GPU における PFAC 法による実装よりも，最大で 16.1%の高速化を達成した．また，ホストスレッドの並列度を大きくした際，並列度 1 から 8 に対してスケーラブルな実行が可能である事を確認した．さらに，高並列度におけるデバイスの輻輳による遅延とホスト - デバイス間通信速度，および，デバイスにおけるカーネル処理速度を評価した．

今後の課題として，複数ユーザーの異なるパターンセットを単一のテキストデータに対して探索処理を実行する際の効率的な実行方法の提案，また，探索処理効率を低下させないリアルタイムなパターンセット更新手法の提案を行いたい．

謝辞　この研究は，文部科学省　科学技術戦略推進費　気候変動に対応した新たな社会の創出に向けた社会システムの改革プログラム　「グリーン社会 ICT ライフインフラ」，独立行政法人情報通信研究機構　高度通信・放送研究開発委託研究「新世代ネットワークを支えるネットワーク仮想化基盤技術の研究開発」および，文部科学省科学技術研究費補助金基盤研究 C「安心・安全な情報提供を可能とするインターネット基盤の構築に関する研究」（22500069）の一環としてなされた．

参考文献

1)　Jacobson, V.; Smetters, D. K.; Thornton, J. D.; Plass, M. F.; Briggs, N.; Braynard, R. Networking named content. Proceedings of the 5th ACM International Conference on Emerging Networking Experiments and Technologies (CoNEXT 2009); 2009 December 1-4; Rome, Italy. NY: ACM; 2009; 1-12.
2)　Sarang Dharmapurikar; Praveen Krishnamurthy; Sproull, T.S.; Lockwood, J.W.; , "Deep packet inspection using parallel bloom filters," *Micro, IEEE* , vol.24, no.1, pp. 52- 61, Jan.-Feb. 2004
doi: 10.1109/MM.2004.1268997
URL: http://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=12689

97&isnumber=28393
3) コンテキストスイッチを利用したルータにおける TCP ストリーム再構築のメモリ削減手法［2010-ARC-190 (H22. 8. 3)］ (計算機アーキテクチャ研究会)
4) NVIDIA, http://developer.download.nvidia.com/. NVIDIA CUDA C Programming Guide, version 4.0 edition.
5) Jiangfeng Peng; Hu Chen; Shaohuai Shi; , "The GPU-based String Matching System in Advanced AC Algorithm," *Computer and Information Technology (CIT), 2010 IEEE 10th International Conference on* , vol., no., pp.1158-1163, June 29 2010-July 1 2010
doi: 10.1109/CIT.2010.210
URL: http://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=5577901&isnumber=5577816
6) NVIDIA Developer Zone URL: http://developer.nvidia.com/category/zone/cuda-zone
7) Alfred V. Aho and Margaret J. Corasick. Efficient string matching: an aid to bibliographic search. Commun. ACM, Vol. 18, pp. 333–340, June 1975.
8) Donald E. Knuth, Jr. James H. Morris, and Vaughan R. Pratt. Fast pattern matching
in strings. SIAM Journal on Computing, Vol. 6, No. 2, pp. 323–350, 1977.
9) "Programming With POSIX Threads". D. Butenhof. Addison Wesley
www.awl.com/cseng/titles/0-201-63392-2
10) The Snort Project. SNORT Users Manual, 2.9.2 edition, December 2011.