GPGPU に適した XML インデクス生成とキーワード検索手法

武藤　教宏† 　横山　昌平† 　福田　直樹† 　石川　博†

† 静岡大学情報学部情報科学科　〒 432–8011 静岡県浜松市中区城北 3–5–1
E-mail: †cs07094@s.inf.shizuoka.ac.jp, ††{yokoyama,fukuta,ishikawa}@inf.shizuoka.ac.jp

あらまし　XML は Web との親和性に優れ，データの蓄積・共有の標準フォーマットとして広く普及している．Web 上のシステムは多数のユーザからの並列同時アクセスを効率良く処理する事が求められる．並列処理に関して，GPU を汎用計算に用いる GPGPU が注目されている．GPU による処理では，処理単位を効率的に細分化し，多数のコアを有効活用することが課題である．本論文において GPGPU に適した XML のインデクス手法を提案する．提案手法は GPU を用いて多数のクエリを並列処理する場合に処理が高速化されることを目的とする．また，CPU でクエリから検索条件を作成し検索処理に GPU を用いる．GPU を用いた検索処理が有効であることを示すために検索処理に CPU のみと GPU と CPU の両方を用いた場合と提案手法を実装した場合の比較を行う．実験により，一部データ構造において提案手法が有効であることを確認した．また，データサイズを変更した場合の実行時間について実験を行い，実行時間が線形的な増加を示すことを確認した．

キーワード　XML キーワード検索，半構造データ処理，GPGPU

Towards Better GPGPU Based Indexing for XML Data

Norihiro MUTO†, Shohei YOKOYAMA†, Naoki FUKUTA†, and Hiroshi ISHIKAWA†

† Department of Computer Science, Faculty of Informatics, Shizuoka University　Johoku 3–5–1, Naka-ku, Hamamatsu-shi, Shizuoka, 432–8011 Japan
E-mail: †cs07094@s.inf.shizuoka.ac.jp, ††{yokoyama,fukuta,ishikawa}@inf.shizuoka.ac.jp

1. はじめに

XML はデータ蓄積・共有の標準フォーマットとして普及している．例えば，Open Directory Project (ODP) は，ウェブディレクトリを XML フォーマットで公開している．Web 上において，クエリ処理を行うシステムには多数のユーザから同時にクエリが入力される可能性がある．そのため，クエリ処理を行うシステムは多数のクエリを効率的に並列処理する性能が必要である．

近年，GPU の高性能化とともに，GPU が持つ高い演算能力を汎用的な処理に利用する GPGPU が普及している．GPGPU は GPU の並列処理能力を有効に活用することが課題である．GPGPU に関する多くの研究は，GPU の並列処理能力を有効に活用した処理の高速化である．例として，高速フーリエ変換 [1], ソーティング [2]，k 近傍探索 [3] などの高速化が研究されている．GPU のコアは CPU のコアと比較して，動作周波数が低く，1 プロセッサ当たりのコア数が多いという特徴がある．また，GPU のメモリは CPU と比較してメモリバンド幅が広いという特徴がある．GPU を用いて処理の高速化を行うためには，処理を細かい並列単位に分割し多数のコアを有効に活用することが必要である．また，GPU のメモリの特性(注1)を考慮したデータ配置やデータアクセスを行うことで，広いメモリバンド幅を有効に活用することが必要である．データアクセスの効率化により多数のコアによる並列処理とメモリレイテンシの隠蔽が可能である．一方で，GPU のメモリは CPU のメモリと比較して容量が少ないという欠点がある．そのため，多くのデータを処理対象とする場合には，メモリ特性を考慮したデータ配置とともにメモリ領域の有効活用も大きな課題となる．GPU を用いた並列処理はデータの読み込みの効率化が処理性能に大きく影響する．一方で，CPU を用いた並列処理はキャッシュメモリを有効活用できるかが処理性能に大きく影響する．そのため，CPU を用いて検索処理を行う場合，並列に処理するクエリ数，検索対象の XML データのサイズによりキャッシュミスが増大し処理性能が低下する可能性がある．

(注1)：連続した番号を持つスレッドがアクセスするデータが連続した番地に格納されている場合，メモリにアクセスする回数を削減可能 (CUDA [4] ではコアレスアクセス)

GPU を用いて処理することの利点として，CPU の負荷軽減もあげられる．CPU の負荷軽減は，多数のシステムがクライアントからの問い合わせを処理するサーバ環境で有効であると考えられる．GPGPU の普及により，スーパーコンピュータの設計も従来の CPU の多重化による演算能力の向上から，CPU と GPU を併用したものに変化している [5].

本研究は，GPGPU に適した XML キーワード検索手法を提案する．提案手法においては 1 つのクエリに対して 1 つのブロック(注2)を生成し，複数のブロックを並列して実行するクエリ単位での並列化を行う．また，1 つのブロックを複数のスレッドで処理する検索単位での並列化を行う．さらに，データ配置を工夫し GPU が持つ広いメモリバンド幅を利用する．XPath を用いて記述されたクエリを処理する．

提案手法が多数のクエリを処理する場合に有効であることを確認するために，評価実験を行う．また，データサイズと実行時間についての関係を測定し，提案手法の有効性について評価をする．

2. 関 連 研 究

2.1 VLCA 探索

小林ら [6] は，XML 文書に対して VLCA(Valuable LCA) の探索を高速化する手法を提案している．XML 文書に出現する全てのタグ名にユニークなビット列を割り当て，ビット列同士の論理和を用いて，タグ名の重複を判定している．また，キーワード検索は XML 文書に出現するテキストノードの単語で B+木を構成して高速化をしている．小林らが VLCA 探索の高速化にビット演算や B+木を用いるのに対して，本研究は GPU を用いた処理に適したデータ構造の提案による XPath クエリ処理の高速化を目指している．

2.2 XML 検索

Arroyuelo ら [7] は，XPath クエリ処理をキーワードの検索を高速化することで効率的に行う手法を提案している．キーワード検索を高速に行うために，テキストノードの内容を並び変えて保存し高速に検索する手法を提案している．また，対象の XML ファイルが巨大であってもメモリ上で処理可能であることを目的としている．Arroyuelo らがキーワード検索の高速化による XPath クエリを行っているのに対して，本研究は検索処理を並列処理することによる高速化を目指している．

2.3 XPath クエリ処理の並列化

野村ら [8] は，スケルトン並列プログラミングを用いて，XPath クエリを並列に処理する手法を提案している．XPath クエリのパス表現について，木スケルトンを用いて処理する手法を提案している．本研究とは並列処理をする手段，パス表現を処理する手段が異なる．本研究は GPU で並列処理を行い，パス表現にビット列同士の AND 演算を用いることを提案している．

2.4 GPGPU によるデータ検索

GPGPU をデータ検索に適用する研究として，Garcia ら [3] は，k 近傍探索を GPU を用いて高速に処理する手法を提案している．GPU を用いた k 近傍探索において GPU が有する多数のスレッドで総当たり的に要素間の距離を計算している．提案手法と CPU を用いた ANN(k 近傍探索のための C++ライブラリ)，C 言語で実装した CPU による総当たり k 近傍探索の実行時間を比較し，提案手法が有利であることを確認している．本研究とは高速化する対象が異なる．Garcia らは k 近傍法の高速化を対象としているのに対して，本研究は XML に対するデータ検索の高速化を対象としている．

(注2)：CUDA における並列度の単位．複数スレッドのまとまり

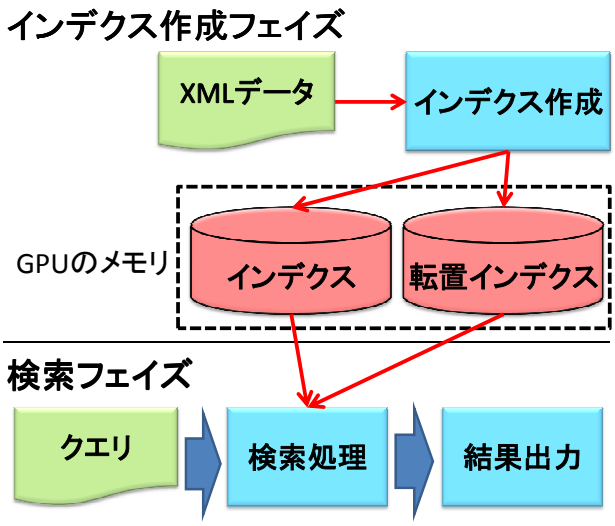

図 1　処理の手順

3. 提 案 手 法

本研究で提案する処理の手順を図 1 に示す．提案手法は，XML から検索処理のためのインデクスを作成するインデクス作成フェイズとクエリを処理する検索フェイズの 2 つのフェイズがある．インデクス作成フェイズは，XML ファイルからタグの出現位置を記録したインデクスとテキストノードの内容と出現位置を記録した転置インデクスを作成する．作成したインデクスと転置インデクスは GPU のメモリに転送する．転送したデータを検索フェイズで検索処理に用いる．検索フェイズは入力されたクエリに基づいて検索処理を行い，結果を出力する．検索フェイズでは XML ファイルを参照せず，インデクス作成フェイズで作成したインデクスと転置インデクスを用いて処理を行う．また，インデクス作成フェイズでは作成したインデクスと転置インデクスの全データを GPU のメモリに転送する．

3.1 インデクス作成フェイズ

インデクス作成フェイズは，XML データを読み込み，インデクスと転置インデクスを作成し GPU のメモリに転送する．

作成するインデクスおよび転置インデクスの例を図 2 に示す．インデクスとは，XML ファイル内の各タグの出現位置の情報を持つデータである．転置インデクスとは，XML ファイル内のテキストノードの内容と出現位置の情報を持つデータである．

インデクス作成フェイズの概要をアルゴリズム 1 に示す．インデクスは，SAX イベントの先頭から順番に XML の 1 要素を 1 ビットととして作成する．本研究は，XML ファイル内の

<Book>
<Title>Database</Title>
<Author>Alice</Author>
<Author>Bob</Author>
<Section>
<Title>Database</Title>
<Title>normalize</Title>
<Title>SQL</Title>
</Section>
</Book>

インデクス

(name,level)	
(Book,0)	11111111111111111110
(Title,1)	01100000000000000000
(Author,1)	00001101100000000000
(Section,1)	00000000001111111100
(Title,2)	00000000001101101000

転置インデクス

Text	出現位置
“Database”	3 12
“Alice”	5
“Bob”	8
“normalize”	12
“SQL”	15

図 2 XML データからインデクスと転置インデクスを作成する例

開始タグ，テキストノード，終了タグを要素とする．開始タグから終了タグの直前までのビットを 1 にし，各タグの出現範囲を記録する．インデクスをビット列で作成することでビット演算を用いた並列処理を可能にする．また，XML ファイル内のタグ要素の名前を格納したテーブルと深さを格納したテーブルを作成する．提案手法は，2 つのタグ A と B に i 番目の要素で先祖子孫関係があるかどうかを式 (1) を用いて判定する．

$$AB_i = A_i \cap B_i \tag{1}$$

$$AB_i = \begin{cases} 1 & \text{先祖子孫関係がある} \\ 0 & \text{先祖子孫関係がない} \end{cases}$$

A_i，B_i はタグ A，B の i 番目のビットデータを示している．また，AB_i は，A_i と B_i が先祖子孫関係にあるかを判定するとともに，タグ同士の部分木を求めるのに用いる．

転置インデクスは，XML ファイル内のテキストノードの内容に基づいて作成し，テキストノードの出現位置を格納する．ノードの出現位置を管理する領域の大きさは，事前に XML ファイルを走査して決定する．

並列処理に GPU を用いるため，適切にテキストのデータを配置した転置インデクスを作成する．GPU でのキーワード検索では，転置インデクス中の各テキストデータをテキスト単位でスレッドに割り当て並列処理を行う．そのため，テキスト A の i 文字目を $text[i][A]$ に格納しキーワード検索において連続した番号を持つスレッドが連続したメモリ番地にアクセス可能なデータ配置を行う．$text$ は転置インデクスのテキストデータを格納するテーブルである．

Algorithm 1 インデクス作成処理

```
sp ← 0
node ← 0
tag_num ← 0
text_num ← 0
while Parse(element, name, value)! = EOF do
  if element = StartTag then
    search ← tagtable_search(tag_num, name, sp)
    if search = −1 then
      tag_registry(tag_num, name, sp)
      search ← tag_num + +
    end if
    stack[sp + +] ← search
  end if
  if element = EndTag then
    sp − −
  end if
  if element = Text then
    search ← texttable_search(text_num, value)
    if search = −1 then
      text_registry(text_num, value)
      search ← text_num + +
    end if
    node_registry(search, node)
  end if
  for i = 0 to sp do
    index[stack[i]][node] ← 1
  end for
  node + +
end while
```

3.2 検索フェイズ

本研究で提案する検索フェイズの詳細を図 3 に示す．解釈処理は，CPU でクエリを解釈しクエリ内のパスが記述された部分とキーワード検索の条件が記述された部分を抽出する．CPU は抽出したクエリの内容から検索条件を作成する．並行して実行するクエリが複数存在する場合，全てのクエリの解釈が終了した時点で検索条件を GPU のメモリに転送する．検索処理は GPU を用いてパス式が示す部分木の抽出処理 (以降パス式処理とする)，キーワード検索処理，集約処理を行う．並行して実行するクエリが複数存在する場合，GPU を用いた検索処理は複数のブロックによる並行処理によって行う．検索処理に必要なデータであるインデクスおよび転置インデクスは，インデクス作成フェイズで GPU のメモリに転送されている．検索処理終了後に CPU は GPU のメモリから検索結果をメインメモリに転送し検索結果の出力を行う．

3.2.1 クエリの解釈

クエリの解釈処理は，入力されたクエリから，パス式とキーワード検索条件を作成する．

パス式の作成処理はロケーションステップに記述された軸とタグ名を条件として，インデクスデータのタグ名と深さを検索する．ロケーションステップに記述された条件と一致するタグが存在する場合，パス式にタグの番号を追加する．ロケーションステップに記述された条件と一致するタグが複数存在する場合，パス式を複数作成する．ロケーションステップに記述された条件と一致するタグが存在しない場合，パス式を無効にする．タグの検索は，インデクスに保存されたタグ名とクエリのノードテストに記述されたタグ名が一致するか判定する．タグ名が一致する場合，タグの深さが適正であるかを判定する．タグの深さの判定は，クエリの軸，1 つ前のロケーションステップで追加したタグの深さ，タグ名が一致したタグの深さを用いて行う．ロケーションステップの軸が child 軸の場合，タグの深さが直前のロケーションステップで追加したタグの深さ+1 と等しければ条件に一致していると判定する．descendant 軸の場合，タグの深さが直前のロケーションステップで追加したタグの深さ+1 以上であれば条件に一致していると判定する．

キーワード検索条件の作成処理はクエリからキーワード検索に関連する部分を抽出する．抽出する内容は一致条件と検索キーワードである．一致条件として，前方一致 (starts-with)，後方一致 (ends-with)，中間一致 (contains) を処理対象とする．

作成したパス式とキーワード検索条件を検索処理のために，GPU のメモリに転送する．また，クエリの条件から複数のパス式を作成した場合，いくつのパス式を作成したかについても GPU のメモリに転送を行う．

3.2.2 パス式処理

パス式処理の例を図 4 に示す．パス式処理は，クエリから作成したパス式に対応する XML の部分木を抽出する．複数のスレッドを生成しビット列を分割して並列に処理する．各スレッドは $LENGTH/THREAD$ 個の要素を処理する．$LENGTH$ はインデクスデータの長さであり，$THREAD$ は生成するスレッド数である．スレッド T_{id} は $loop * THREAD + T_{id}$ 番目の要素について処理を行う．$loop$ は 0 から $LENGTH/THREAD$ まで変化する変数である．T_{id} はスレッド識別子であり，生成するスレッドの数が 1024 の場合，0 から 1023 の値である．T_{id} を差分値とすることで，連続したメモリ番地へのアクセスを可能にし，インデクスデータの読み出しをコアレスアクセスにより行う．パス式処理の結果は GPU のメモリのパス式処理の結果を格納する変数に保存する．

3.2.3 キーワード検索処理

キーワード検索処理のため，クエリから作成したキーワード検索条件に対するポインタを作成する．ポインタとはキーワード検索条件に一致したテキストノードの出現位置を 1 としたビット列であり，インデクスと同じビット長を持つ．複数のスレッドを生成しテキストを並列に検索する．各スレッドは $PATTERN/THREAD$ 個のテキストについて検索処理とポインタ作成処理を行う．$PATTERN$ は転置インデクスに格納されたテキストの種類数である．スレッド T_{id} は $loop * THREAD + T_{id}$ 番目の要素について処理を行う．$loop$ は 0 から $PATTERN/THREAD$ まで変化する変数である．T_{id} はスレッド識別子であり，生成するスレッドの数が 1024 の場合，0 から 1023 の値である．T_{id} を差分値とすることで，連続したメモリ番地へのアクセスを可能にし，インデクスデータの読み出しをコアレスアクセスにより行う．キーワード検索の結果，キーワード検索条件に一致するテキストの出現位置を転置インデクスから取得する．取得した出現位置に対応するビットを 1 に変更することでポインタを作成する．ビットを 1 に変更する処理は，並列して動作する他のスレッドが同一のメモリ番地のデータを更新する可能性がある．そのため，atomic 命

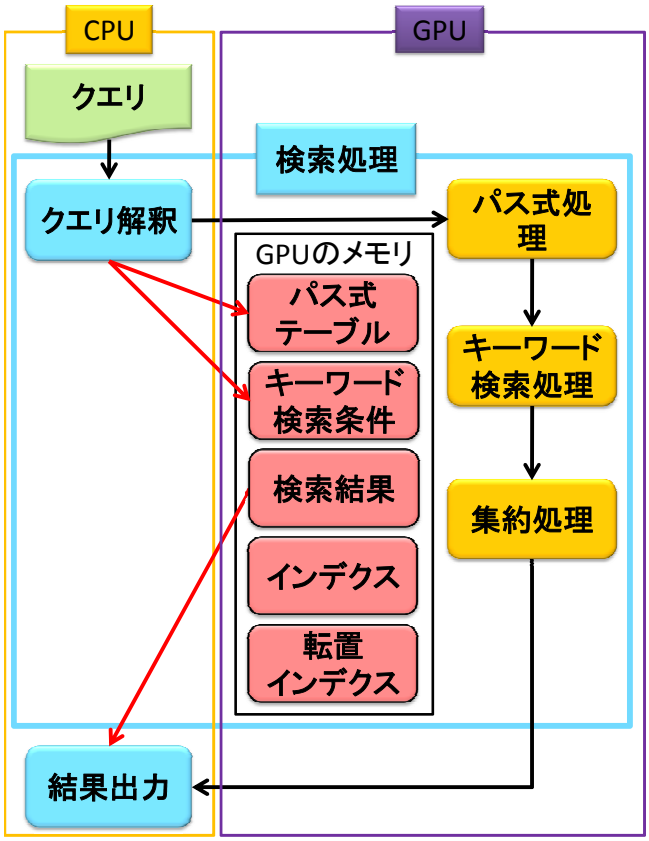

図 3　検索フェイズの詳細

例 : /Book//Title

(Book,0)	1111 1111 1111 1111 1111 10	
	AND	
(Title,1)	0110 0000 0000 0000 0000 00	
/Book/Title	0110 0000 0000 0000 0000 00	

(Book,0)	1111 1111 1111 1111 1111 10	
	AND	
(Title,2)	0000 0000 0001 1011 0110 00	
/Book/*/Title	0000 0000 0001 1011 0110 00	

図 4　パス式処理の例

例 : /Book//Title[text()=“Database”]

/Book/Title	0110	0000	0000	0000	0000	00
				OR		
/Book/*/Title	0000	0000	0001	1011	0110	00
/Book//Title	0110	0000	0001	1011	0110	00

/Book//Title	0110	0000	0001	1011	0110	00
				AND		
[text()=“Database”]	0010	0000	0001	0000	0000	00
Result	0010	0000	0001	0000	0000	00

図 5　集約処理の例

表 1　DBLP から生成したデータの概要

ファイルサイズ (MB)	要素数	テキストノード数
4.93	273,344	136,672
9.88	556,598	278,299
24.72	1,396,052	698,026
49.49	2,810,560	1,405,280
98.51	5,471,472	2,735,736
197.01	10,944,690	5,472,345

令(注3)を用いて同期制御を行う．キーワード検索の結果は GPU のメモリのキーワード検索処理の結果を格納する変数に保存しておく．

3.2.4　集 約 処 理

集約処理の例を図 5 に示す．集約処理は，クエリに基づいて複数作成されたパス式について，パス式処理の結果を集約を OR 演算を用いて集約する．キーワード検索結果の反映は，集約したパス式とキーワード検索処理の結果について，AND 演算を用いて行う．パス式処理と同様にスレッド T_{id} は $loop * THREAD + T_{id}$ 番目の要素について処理を行う．集約処理の結果は GPU のメモリの検索処理の結果を格納する変数に保存しておく．

3.2.5　結 果 出 力

GPU のメモリに保存された検索処理の結果をメインメモリに転送し，検索の結果とする．検索の結果はクエリの条件に一致するノードの出現位置を 1 としたビット列であり，インデクスと同じビット長を持つ．

4. 実　　験

GPU を用いたクエリの並列処理の有効性を実験より示す．実験はクエリ数を増加させた場合の CPU との比較実験とデータサイズと実行時間の関係を比較する実験を行う．クエリは XPath で記述されたクエリを用いる．クエリ処理の結果として検索条件に合致するテキストノードの出現位置を出力する．実行時間を計測する処理の範囲は検索条件をメインメモリから GPU のメモリに転送する処理の開始時点から検索結果を GPU のメモリからメインメモリに転送する処理が終了した時点まで (以降検索処理とする) とする．その検索処理を 100 回繰り返す．その処理を 10 回行い平均実行時間をその手法の実験結果とする．実験のデータセットは異なる 2 種類のデータセットを用いる．1 つ目のデータセットは XML 文書生成器 [9] を用いて生成した DBLP [10] に近い構造を持つ XML ファイル (以下 DBLP とする) である．生成された XML ファイルは，同一の内容を持つテキストノードが多数存在するため，転置インデクスのメンバ領域を大きく設定している．生成したファイルのデータサイズ，要素数，テキストノード数を表 1 に示す．2 つ目のデータセットとしては XMark ベンチマーク [11] のために提供されている XML 文書生成器である xmlgen を用いて生成した XML ファイル (以下 xmlgen とする) である．生成された XML ファイルは，テキストノードの内容の多様性が高いため，転置インデクスのメンバ領域を小さく設定している．生成したファイルのデータサイズ，要素数，テキストノード数を表 2 に示す．実験に利用するデータセットは，avg-depth が DBLP は 2.90，xmlgen は 5.55 と異なる特徴をもつデータである．また，データサイズに対するテキストノードの数も異なる．キーワード検索の対象となるテキストノードは，同一の内容を持つテキストノードが多数存在する DBLP のほうが数が少なく，テキストノードの内容の多様性が高い xmlgen のほうが多い．

表 2　xmlgen から生成したデータの概要

ファイルサイズ (MB)	要素数	テキストノード数
5.6	151,649	68,116
11.32	305,082	137,217
28.35	758,082	340,922
56.29	1,512,877	679,966
113.06	3,027,035	1,360,720
226.79	6,062,204	2,724,555

4.1　実行時間比較

4.1.1　目的および条件

表 3 に実験に使用した装置およびコンパイラを示す．GPU を用いた検索処理は前述したインデクスと転置インデクスを用いて行う．GPU と Opteron を用いた検索処理，Opteron を用いた検索処理，Core i7 を用いた検索処理は，転置インデクスをテキストノードの親要素のタグごとに作成しキーワード検索に用いる．これは，メモリ領域が限定された GPU と比較して CPU のメインメモリが豊富にあることを考慮した実装であり，キーワード検索における計算量は，処理に GPU を用いた場合のほうが大きくなっている．パス式処理および集約処理を GPU を用いた検索処理，GPU と Opteron を用いた検索処理では，GPU を用いて行う．一方で，Opteron を用いた検索処理，Core i7 を用いた検索処理では CPU を用いて行う．GPU での処理の並列化が有効であることを確認するために，高い演算能力を持つ Core i7 を比較対象とする．また，提案手法が GPGPU に適していることを確認するために，GPU を用いた検索処理でデータの読み込みの効率化 (コアレスアクセス) を行わない場合 (結果の凡例では non coalesced access とする) を比

(注3)：CUDA で利用可能な同期制御命令で更新しているメモリ番地に対する他のスレッドのアクセスを禁止する

表 3　実験に用いた装置

	GPU	GPU+Opteron	Opteron	Core i7
CPU	AMD Opteron 1222 dual-core			Intel Core i7 975 quad-core
GPU	NVIDIA GTX 460 7SM 768MB			
Memory	8GB			12GB
OS	Windows 7			Fedora 12
コンパイラ (C++)	C/C++ Optimizing Compiler Version 15.00.21022.08 for x64			GNU コンパイラ (v2.4)
コンパイラ (GPU)	CUDA toolkit 3.2			

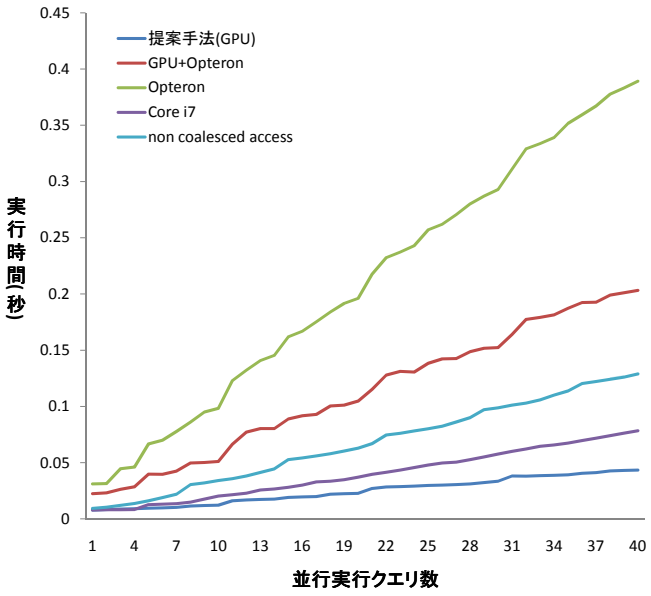

図 6　クエリ数を変化させて DBLP を検索した時間の変化

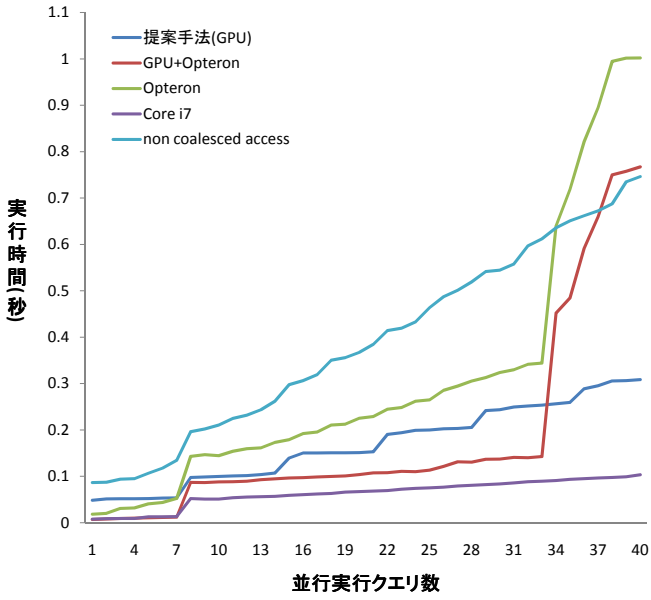

図 7　クエリ数を変化させて xmlgen を検索した時間の変化

較対象として検索時間を比較する．GPU を用いた検索処理では 1 クエリに対して 1 ブロック，1 ブロックに対して 1024 スレッドを生成する．GPU と Opteron を用いた検索処理のキーワード検索と Opteron を用いた検索処理では_beginthread を用いて 1 クエリに対して 1 スレッドを生成する．GPU と Opteron を用いた検索処理のパス式処理および集約処理は 1 クエリに対して 1 ブロック，1 ブロックに対して 1024 スレッドを生成する．Core i7 を用いた検索処理は pthread を用いて 1 クエリに対して 1 スレッドを生成する．

並行して実行するクエリの数を 1〜40 まで変化させ，40 回実行時間を計測する．実験に利用するデータは DBLP を用いる場合表 1 に示したデータサイズが 98.51MB の XML ファイル，xmlgen を用いる場合表 2 に示したデータサイズが 113.06MB の XML ファイルである．

4.1.2　DBLP を用いた場合の結果

DBLP を対象にクエリ数を変化させて検索を行った結果を図 6 に示す．並行して処理するクエリ数に関わらず，GPU を用いた検索処理の実行時間が最も短くなっている．また，並行して処理するクエリ数の増加に対して，提案手法の実行時間の増加率が最も小さくなっている．GPU と Opteron を用いた検索処理と Opteron を用いた検索処理の結果より，GPU と CPU による協調処理が有効であることがわかった．また，提案手法と GPU を用いた検索処理でコアレスアクセスを行わない場合の結果より，提案手法は GPU の特性を有効に活用できている．

4.1.3　xmlgen を用いた場合の結果

xmlgen を対象にクエリ数を変化させて検索を行った結果を図 7 に示す．並行して処理するクエリ数に関わらず，Core i7 を用いた検索処理の実行時間が最も短くなっている．一方で，並行して処理するクエリ数の増加に対して，提案手法の実行時間の増加率が最も小さくなっている．GPU と Opteron を用いた検索処理と Opteron を用いた検索処理の結果の比較より，GPU と CPU による協調処理が有効であることがわかった．また，提案手法と GPU を用いた検索処理でコアレスアクセスを行わない場合の結果より，提案手法は GPU の特性を有効に活用できている．提案手法の結果は並行実行クエリ数が 7 の倍数で大きく増加する傾向がみられた．これは今回の実装で 1 クエリを 1SM(注4)で処理していること，実験に用いた GPU が 7SM であることより，7 の単位で処理が直列化されているためと考えられる．そのため，提案手法はクエリ数の増加に対してクエリの内容に関わらず実行時間が規則的に増加する，並列処理に適した手法であると考えられる．一方で，提案手法以外の結果について実行時間が並行実行クエリ数が 7 の時点で大きく増加しているのは，入力したクエリから作成した検索条件が多くの

(注4)：GPU のプロセッサに内蔵されたコアの集合の単位．1 ブロックは 1SM で実行される

テキストデータをキーワード検索の対象としていたことが影響していると考えられる．また，キーワード検索に Opteron を用いている GPU と Opteron を用いた検索処理と Opteron を用いた検索処理の場合，並行して実行するクエリ数が 34 を超えた時点から実行時間が非常に大きく増加している．これは，CPU のキャッシュメモリ不足の可能性が考えられ，キャッシュミスの増大によって処理性能が悪化したことが原因となっていると考えられる．

4.1.4 結果および考察

提案手法が並行して処理するクエリ数の増加に対して，実行時間の増加率の面で有利であることがわかった．また，GPU と Opteron を用いた検索処理と Opteron を用いた検索処理の結果より，GPU を用いて処理の一部を行うことの有効性がわかった．提案手法と GPU を用いた検索処理でコアレスアクセスを行わない場合の結果より，提案手法が GPU の特性を有効に活用できる手法であるとわかった．実行時間の面では，DBLP を用いた場合提案手法が最も高速に，xmlgen を用いた場合 Core i7 を用いた検索処理が最も高速に処理をしている．実験に用いた 2 つのデータのノード数には大きな差が存在しないため，ビット長に処理時間が依存するパス式処理や集約処理にかかる実行時間の差は大きくないと考えられる．一方で，2 つのデータはテキストノードの内容に偏りに差があり，キーワード検索にかかる時間に影響していると考えられる．xmlgen については，テキストノードの内容の多様性が高いため，キーワード検索にかかる時間が大きくなっていると考えられる．また，転置インデクスの作成方法がキーワード検索に CPU を用いる場合と GPU を用いる場合で異なり，キーワード検索に GPU を用いる場合のほうが検索対象となるテキストの数が多く，計算量が大きくなったと考えられる．

xmlgen を用いた実験の Opteron を用いてキーワード検索を行った場合に大きく性能が悪化している．これは，CPU のキャッシュメモリが不足しメモリアクセスが増加したことが原因であると考えられる．並列に処理するデータの増加によるキャッシュメモリの不足は CPU が持つ課題であるため，並列して実行するクエリ数の増加や処理対象 XML データの増加によって Core i7 を用いた場合でも同様に処理が悪化する可能性は考えられる．

4.2 データサイズ変更実験

4.2.1 目的および条件

表 3 に示した GPU の列の装置およびコンパイラを実験に用いる．データサイズと実行時間の関係は提案手法についてのみ実験する．GPU を用いた検索処理は，前述したインデクスと転置インデクスを用いて行う．検索処理は 1 クエリに対して 1 ブロック，1 ブロックに対して 1024 スレッドを生成する．提案手法がスケーラブルであることを示すためにデータサイズのみを変更した実験を行う．

並行して実行するクエリの数は 5, 15, 25 の 3 種類とする．

4.2.2 DBLP を用いた場合の結果

DBLP を対象にデータサイズを変更させて並列に検索処理を行った結果を図 8 に示す．グラフ中のマーカーで示された部分

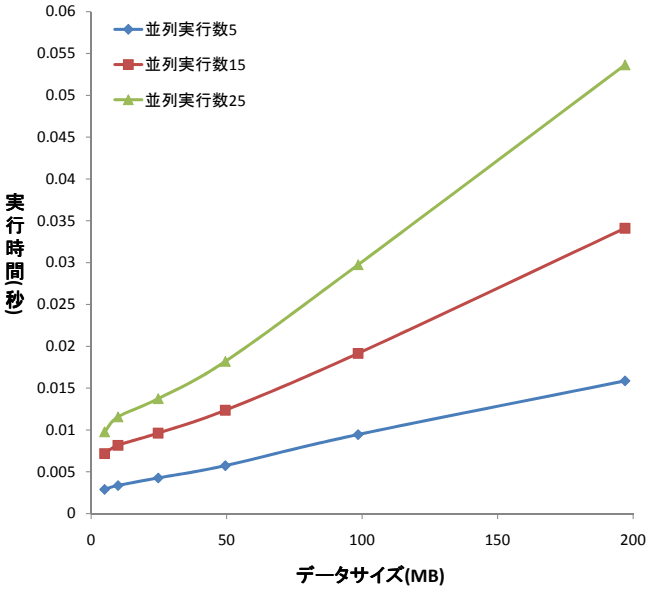

図 8　データサイズを変化させて DBLP を検索した時間の変化

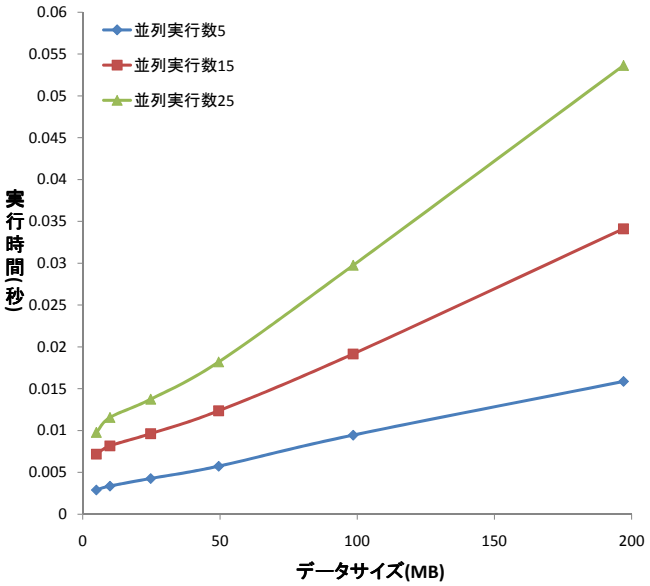

図 9　データサイズを変化させて xmlgen を検索した時間の変化

が実際に計測された結果である．3 種類のクエリ数の結果すべてで，データサイズの増加に対して実行時間が線形的な増加を示した．したがって，GPU を使って検索処理を行った場合，クエリ数に関わらずデータサイズの影響は線形的であると考えられる．

4.2.3 xmlgen を用いた場合の結果

xmlgen を対象にデータサイズを変更させて並列に検索処理を行った結果を図 9 に示す．グラフ中のマーカーで示された部分が実際に計測された結果である．3 種類のクエリ数の結果すべてで，データサイズの増加に対して実行時間が線形的な増加を示した．したがって，GPU を使って検索処理を行った場合，クエリ数に関わらずデータサイズの影響は線形的であると考えられる．

4.2.4 考　　察

提案手法はクエリ数に関わらずデータサイズの増加に対して，

線形的に実行時間が増加することがわかった．並列処理は並列単位の設定方法により処理時間が大きく増加する可能性があるが，提案手法はデータサイズの影響が線形的でありスケーラブルであると考えられる．

5. おわりに

本研究では，GPGPU に適した XML キーワード検索手法を提案した．実験より，GPU を用いた検索処理が多数のクエリを並列に処理する場合に有効となる場合があることを示した．また，提案手法が GPU のメモリ特性を有効に活用する GPU に適した手法であることを示した．

今後の課題として，CPU と GPU の性能や対象 XML の構造によってクエリの処理方法を変更するアルゴリズムについても検討をする．処理対象のファイルの大きさは転置インデクスの必要メモリ量と密接に関係するためキーワード検索に CPU を用いるなどの CPU と GPU による協調処理が必要であると考えられる．協調処理では，CPU と GPU にどのように処理を割り振るかが重要であり，適切なパラメータ設定について検討を行う．

また，限られた GPU のメモリをどのように有効活用するかについても検討をする．GPU のストリーム処理の活用により，検索に必要なデータが GPU のメモリ領域に収まらない場合にデータを転送しながら検索処理を行う手法についても検討する．

実験評価として，既存の XMLDB，XPath 処理系等との比較実験を行い，提案した手法が有効であることを確認したい．

謝辞 本研究の一部は科学研究費補助金（課題番号 22650012）の助成による．

文　献

[1] Kenneth Moreland, and Edward Angel, “The FFT on a GPU”, In Proceedings of the ACM SIGGRAPH/EUROGRAPHICS conference on Graphics hardware, Eurographics Association, pp.112-119, 2003.

[2] Nadathur Satish, Changkyu Kim, Jatin Chhugani, Anthony D. Nguyen, Victor W. Lee, Daehyun Kim, and Pradeep Dubey, “FAST Sort on CPUs and GPUs: A Case for Bandwidth Oblivious SIMD Sort” ACM, SIGMOD, pp.351-362, 2010.

[3] Vincent Garcia, and Frank Nielsen, “Searching High-Dimensional Neighbours:CPU-Based Tailored Data-Structures Versus GPU-Based Brute-Force Method”, MIRAGE 2009, LNCS 5496, pp.425-436.

[4] NVIDIA CUDA. http://developer.nvidia.com/object/cuda.html

[5] Satoshi Matsuoka, “The TSUBAME Cluster Experience a year later, and onto petascale TSUBAME 2.0” Lecture Notes In Computer Science, 2007. 4757: pp.8-9.

[6] 小林径, 横田治夫, “タグ名をビットマップ化した索引による効率的な Valuable LCA 探索手法”, DEIM2010, pp.C8-3.

[7] Diego Arroyuelo, Francisco Claude, Sebastian Maneth, Veli Mäkinen, Gonzalo Navarro, Kim Nguyen, Jouni Sirén, and Niko Välimäki, “Fast in-memory XPath search using compressed indexes”, In ICDE 2010.

[8] 野村 芳明, 江本 健斗, 松崎 公紀, 胡 振江, 武市 正人, “木スケルトンによる XPath クエリの並列化とその評価”, コンピュータソフトウェア 24, pp.3_51-3_62, 日本ソフトウェア科学会, 2007.

[9] 原崎 真奈美，横山 昌平，福田 直樹，石川 博, “実データとユーザ定義クエリセットに基づくスケーラブルな XML ベンチマーク文書生成手法”, DEIM Forum 2010, pp.C8-5.

[10] The DBLP Computer Science Bibliography, http://www.informatik.uni-trier.de/~ley/db/

[11] Albrecht Schmidt, Florian Waas, Martin Kersten, and Michael J. Carey, “XMark: A Benchmark for XML Data Management”, Proc.of the 28th VLDB conference 2002, pp.974-985.