社団法人 情報処理学会 研究報告
IPSJ SIG Technical Report

2008－ARC－177 (24)
2008－HPC－114 (24)
2008／3／6

GPUの汎用計算環境CUDAによる主記憶上の大規模なテキストに対する高速な全文検索の検討

東 竜一[†1] 藤本典幸[†1] 萩原兼一[†1]

本稿では，NVIDIAが提供しているGPUの汎用計算環境CUDAを用いて実装した全文検索の成果について報告する．全文検索のアルゴリズムとして高速なボイヤー・ムーア法を実装し検索結果の転送を工夫することで，GPUとしてnVIDIA GeForce 8800 GTXを用いる場合，Intel Core 2 Extreme 2.93GHz CPUによる検索にくらべて3〜4倍程度の速度向上を達成した．またシフトJIS文書の検索に対しては戻り読みという操作により，シフトJISを文字コードとする日本語文書に対する検索も誤りなく高速に行う．またその成果を元にVRAMの容量を超える文書に対する主記憶上の大規模な文書に対する全文検索について検討する．

An Investigation of CUDA based Fast Full Text Search for Large-Scale Texts on Main Memory

Ryuichi Higashi,[†1] Noriyuki Fujimoto[†1]
and Kenichi Hagihara[†1]

We report an implementation of fast full text search on CUDA architecture, a general purpose computing environment for GPUs provided by NVIDIA. Our implementation concatenates all of given Japanese or English texts and then divides it into many portions to run concurrently many threads, each of which executes Boyer-Moore string search algorithm for the allocated text portion. The experimental results on nVIDIA GeForce 8800 GTX show that our implementation runs about 3 to 4 times faster than that on Intel Core 2 Extreme 2.93GHz. Also we investigate a feasibility of fast full text search for texts on main memory, which are too large to fit into VRAM, with our implementation.

1. はじめに

全文検索は，文書内の文字を1文字ずつ照合する逐次検索および事前に文書から作成する索引を用いる索引検索の2種類に分類できる[1]．逐次検索はテキストサイズに比例して時間がかかり，テキストが大規模になると検索にかなり時間がかかるようになる．また索引型には次のような欠点がある：索引の形式によるが，索引のサイズが元の文書サイズに匹敵する大きさになりうる；検索する前に索引を作成しなければならない；文書の更新が多い環境であるなら時間が経つに連れて索引に差異が生じる；索引化の方法により，単語単位でしか検索できない，文書中の正確な位置を特定できない，など検索における柔軟性が失われる．

そこで，本稿ではCUDAの性能を活かした高速な逐次型検索について報告する．英文だけでなくシフトJIS[4]により符号化された日本語文章の検索も戻り読みを行うことで正しく行う．逐次型の検索アルゴリズムとしてはボイヤー・ムーア法(以下BM法)[1),2)]を用いる．それによりGPUにnVIDIA GeForce 8800 GTXを用いる場合の検索処理の速度はIntel Core2 Extreme 2.93GHz CPUに対して4.45倍の速度を達成した．そしてGPU上での検索結果の格納方法および主記憶へ返すデータ構造の工夫により，CPU-GPU間の文書以外のデータ転送時間を含めたGPUによる全文検索の応答時間をCPUの検索時間に対して1/3.79に短縮した．ただし，これは予めVRAMに検索対象の全文書データが載っている場合である．

更に本稿では，文書転送および検索処理を非同期にして転送時間を隠蔽することで，VRAMに載らない主記憶上の大規模な文書をGPUで高速に検索できる可能性について検討する．主記憶をデータベースの領域とする主記憶データベース検索は磁気ディスクへアクセスしないので高速である[3]．

以降，2章で提案手法，3章で評価実験について述

†1 大阪大学 大学院情報科学研究科
Graduate School of Information Science and Technology, Osaka University

```
1:  主記憶上に領域を確保
2:  while (読み込むべき文書が残っている)
3:      文書を読込
4:  全文書を連結しデータを VRAM に転送
5:  while (検索を続ける)
6:      検索キーワードを受取
7:      BM 法用に 2 つのジャンプテーブルを生成
8:      VRAM 上にメモリ確保し初期化・データ転送
9:      検索用のカーネル関数を呼出
10:     結果を主記憶に転送
11:     結果を表示
```

図 1　CUDA での検索における CPU の疑似コード

べ，4 章でまとめる．CUDA プログラミングの詳細については文献[5),6)] を参照されたい．

2. 提案手法

提案手法は，検索対象の全文書を連結して 1 つの大きな文書にしたものをブロック分割し，1 つのブロック内の検索を 1 つのスレッドブロックに行わせることにより全文検索を並列化する．各スレッドは BM 法を逐次実行する．**図 1** に，CUDA で検索するときの CPU の擬似コードを載せる．

2.1 CPU における前処理

1 つに結合した複数文書の境界を超えたマッチを誤って出さずに検索を行えるように，検索が失敗する 1 バイトの値を両端も含めた各文書間に埋め込んでおく．この操作は図 1 の 2，3 行目において行う．このバイト値としては，ASCII コードを基本とする文字コードであれば NUL(0x00) や SOH(0x01) などの制御文字を表すバイト値を使える．また単純に検索すると，一致している長さと検索キーワードの長さを逐一比べる必要がある．BM 法では検索キーワードの末尾から照合を行い，先頭まで一致すればマッチとなる．そこで検索キーワードの先頭バイトの前に，使用しない 1 バイトの値を番兵として埋め込んでおく．ただし文書の境界を示すバイトとは異なるものを選ぶ．

BM 法の第 1 テーブルは，分岐なしで個々のバイト値からシフト量を取れるようにする．その方法としては，個々のバイト値を第 1 テーブルのインデックスとする要素に対応するシフト量を入れる．このため，第 1 テーブルの要素数は (検索キーワードを構成するバイト値の種類数)+1 ではなく 256 とする．この要素サイズはシフト量の最大値を 255 までとして 1 バイトにする．

2.2 CUDA カーネル関数の検索処理

CUDA カーネル関数における検索方法，ブロック分割の方法および高速に検索するために行った共有メモリの使用について説明する．

検索対象の全文書を連結して 1 つの大きな文書にしたものをブロック分割し，1 つのブロック内の検索を 1 つのスレッドに行わせることにより全文検索を並列化する．各スレッドは BM 法を逐次実行する．

2.2.1 ブロック分割の方法

予備実験の結果，1 スレッドブロックは 256 スレッド，1 スレッド当たり文書 200 バイトを検索することにした．このサイズは検索キーワードの先頭バイトを走査する範囲のことで，実際には検索キーワードのバイト長が l であれば $(199+l)$ バイトの文書を走査する．したがって，1 スレッドブロック当たり 200 バイト $\times 256 = 51,200$ バイトの文書を検索する．よって，1 つの検索キーワードにつきスレッドブロック数は，$\lceil$ 文書サイズ $/51,200 \rceil$ となる．

2.2.2 共有メモリの使用

共有メモリとは同じブロック内のスレッド間で共有できるメモリで 16 バンクから成り，バンク衝突がなければアクセス遅延がない．対して，GPU の各プロセッサは VRAM へアクセスする場合，単純な演算の 100 倍から 150 倍程度の遅延が発生する．そこで，多重にアクセスする要素は共有メモリで保持する．検索キーワードおよび検索キーワードから生成した 2 つのジャンプテーブルへのアクセス回数は，平均的に 1 スレッド当たりの検索量に比例して大きくなる．そのため，検索キーワードのバイト列および 2 つのテーブルを共有メモリに入れる．文書からのアクセスは 1 つの検索キーワードによる検索で高々1 回であるので VRAM から共有メモリへ移す意味はない．

2.3 検索結果の格納方法

CUDA では GPU の計算中に動的にメモリを確保できない．しかし検索前に正確にマッチ数を見積もることは不可能である．そのため，検索結果の格納領域は検索する前にどのような結果も記録できるようなサイズを割り当てておく必要がある．以下，この条件を満たす単純な実装および 2 つの改良版の合計 3 つの格納方法について説明する．

2.3.1 1 バイト 1 結果方式

1 バイト 1 結果方式は，検索テキストを格納している 1 バイトの配列と同じ配列を作成し，マッチ位置に対応する要素を 1，それ以外を 0 にする．マッチ位置とは，マッチ部分の先頭バイト値のことを指す．このように 1 つの結果を 1 か 0 で表す方法をフラグ式と呼ぶことにする．**図 2** に例を示す．

この手法は，以降で示す 2 手法と比較して結果の記録にかかる計算量は少ない．しかし検索対象の文書と

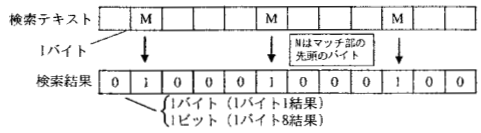

図 2　1 バイトに 1 個および 8 個の結果を入れる格納方法

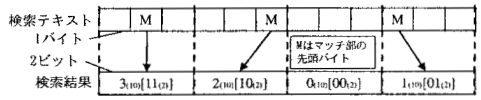

図 3　$n = 3$ におけるオフセット方式の格納方法

検索結果の保持に必要な記憶容量が同じであり，検索結果を CPU に転送する時間が大きく無視できない．

2.3.2　1 バイト 8 結果方式

2.3.1 節の 1 バイト 1 結果方式は 1，0 の 2 通りの状態を記憶するのに 1 バイトを使う．そこで 1 バイトではなく 1 ビットに 1 つの結果を記録する．これにより，検索結果の保持に必要な記憶容量が 1/8 になる．図 2 に例を示す．この方式を 1 バイト 8 結果方式と表記する．

2.3.3　オフセット方式

オフセット方式は，検索キーワードを構成するバイト値の種類数を利用して結果の数自体を減らす．ただし，フラグ式とは異なり 1 つの結果は 1 か 0 の 1 ビットではなくなる．'x', 'y', 'z' を任意の 1 バイトの値として，ある検索キーワードが "xyyzz" という 5 バイトの構成であるとする．この時，バイト値の種類数は 'x', 'y', 'z' の 3 個である．ある検索キーワードを構成するバイト値の種類が n であり文書中のある位置でマッチしているとする．そのとき，その次にマッチするのは少なくとも n バイトは先である．証明は付録に示す．つまり任意の連続する n バイトにおいてマッチは高々1 回となる．よって n バイト毎に 1 結果を記録するようにすればよく，記録する要素数は $1/n$ に抑えられる．ただし，上記の 2 つの方法とは異なり，10 によるマッチの有無ではなく n バイト中のマッチした位置を残す必要がある．そこで各区間の先頭からマッチ位置へのオフセットを格納することになる．ただし，マッチしていない場合も同時に表す必要があり，マッチがなかった場合は 0，マッチがあった場合は（マッチがあった n バイト区間の先頭からのオフセット数）+1 を記憶する．つまり，0〜n の $(n+1)$ 通りの数を記憶する必要がある．この値を記憶するのに必要なビットサイズ s は $s = \lceil \log_2(n+1) \rceil$ である．ここで n は検索キーワードを構成するバイト値の種類数であるので最大で，s は $2^8 = 256$ 種類となるが，対象としている文字コードでは 256 種類のバイト値を全て使うことはありえないので，事実上 $n < 256$ となる．$\log(n+1) \leq 8$ であり s は 8 以下となるので，1 つの結果は 1 バイトで表せる．更に容量を削減するために 1 バイト中に複数個の結果を埋め込むことを考える．埋め込める数 m は $m = \lfloor 8/s \rfloor$ である．**図 3** に，$n = 3$ における例を示す．これより，検索文書 n バイトにつき検索結果の領域は $1/m$ バイトとなるので，検索文書 1 バイト当たりの記憶容量 t は，$t = 1/(mn) = 1/(\lfloor 8/s \rfloor n) = 1/\left(\lfloor 8/\lceil \log_2(n+1) \rceil \rfloor n\right)$ バイトとなる．**表 1** に，n による m および t の値を示す．また 2.3.2 節の 1 バイト 8 結果方式では検索文書 1 バイト当たり 1/8 バイトの結果領域を使うので，その比 $z(= 8t)$ を示す．この値が小さければ小さい程，1 バイト 8 結果方式より記憶効率がよい．

表 1　バイト値の種類数を利用した格納方法による結果サイズ

n	m	t	z
1	8	1/8	1
2	4	1/8	1
3	4	1/12	2/3
4	2	1/8	1
$4 \leq n \leq 15$	2	1/2n	4/n
15	2	1/30	4/15
16	1	1/16	1/2
$n \geq 16$	1	1/n	8/n

n: 検索キーワードを構成するバイト値の種類数
m: 1 バイト当たりの結果格納数
t: 文書 1 バイト当たりの結果サイズ (B)
z: 1 バイト 8 結果方式に対する結果サイズの割合

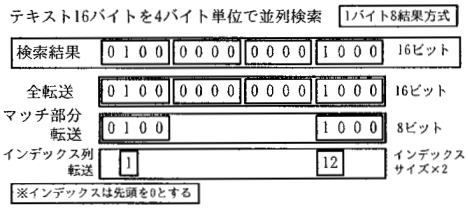

図 4　3 種類の検索結果の転送方法の例

2.4　検索結果の転送

検索結果の単純な転送方法としては，検索して得た結果を全て転送する全転送手法がある．その全転送手法におけるデータ転送量を削減し高速化する．そのために，以下の 2 通りの方針がある．(1) マッチが検出できたスレッドが生成した結果のみを転送する手法 (マッチ部分転送)．(2) マッチ位置を示すインデックスのみを転送する手法 (インデックス列転送)．**図 4** に，3 手法の実例を示す．本節では，(1) および (2) を説明する．

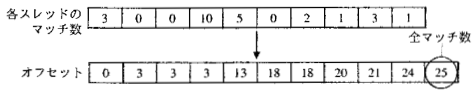

図 5 Scan によるオフセット列の生成

2.4.1 マッチ部分転送

マッチを検出したスレッドが生成した結果のみをCPU に転送する．ただし結果領域は連続ではないので，連続してマッチしているスレッドの結果列を 1 回の転送で取得するようにする．そのため，各スレッド毎に検索時にマッチの有無を記録する．

2.4.2 インデックス列転送

インデックス列転送は，マッチした位置のアドレス (インデックス) の列を生成し転送する．このアドレスは検索テキストの先頭を 0 とする．ただし，GPU においてインデックス列を並列に作成するには，各スレッドごとにインデックス値を書き込む場所へのオフセットが必要になる．そのため，検索処理中に各スレッドにそのマッチ数を記録させる．このスレッド毎のマッチ数の列に対して，CUDPP（CUDA Data Parallel Primitives Library）[7] による Scan（Parallel Prefix Sum）[8] を用いて，オフセット列を生成する．図 5 に例を示す．文書の先頭からのインデックスを 32 ビットで記憶するとすると，1 つの結果の記録容量は 1 バイト 8 結果方式の 32 倍となる．したがって，転送量の観点ではマッチ数が検索文書の全バイト数の 32 分の 1 以下であれば本手法の転送時間は 1 バイト 8 結果方式より短い．また，マッチ数が 0 であることが分かれば，インデックス列を生成し転送する意味は全くないのでその時点で検索を終了する．

2.5 シフト JIS テキストに対する戻り読み

シフト JIS は半角文字を 1 バイト，全角文字を 2 バイトで符号化する．その符号化方式により，半角文字および全角文字の 1 バイト目の領域は完全に分離している．そして全角文字の 2 バイト目の領域は，半角文字および全角文字の 1 バイト目の領域と重なる部分がある．よって，半角文字か全角文字の 1 バイト目で始まる検索キーワードのバイト列が全角文字の 2 バイト目から始まるバイト列と一致する可能性がある．これは当然誤ったマッチであるのでマッチとしてはならない．これを防ぐにはテキストの先頭から 1 バイトずつ判別していき，全角文字の 2 バイト目であれば読み飛ばす必要がある．しかし BM 法はシフトすることによりテキストを読み飛ばしてしまう．そこで，戻り読みという操作により一致した部分の先頭のバイト値が全角文字の 2 バイト目でないことが分かればマッチとする．この戻り読みの詳細な操作は付録で示す．

表 2 評価実験に用いたマシンの仕様

項目	内容
CPU	Intel Core 2 Extreme X6800 (2.93GHz)
GPU	nVIDIA GeForce 8800 GTX
メモリ	PC2-5300 DDR2 SDRAM 1GB×2

3. 評価実験

CUDA による検索速度を評価するために CPU 版と比較する．CUDA 版は提案手法に基づくアルゴリズムを使用する．CPU 版では検索結果を 1 バイト 1 結果方式で書き込むが，結果を格納する領域を確保するための時間は小さくはないが無視して以降の実行時間の測定結果には含めない．これは CPU 版の評価を有利にするので，提案手法の評価方法の公正さには問題ない．CPU 版のコンパイラは Intel C++ Compiler 10 で推奨最適化オプションを設定している．評価項目は，検索時間および応答時間 (検索キーワードを受け取ってから検索結果を表示する直前までの時間) とする．検索に用いる文書は，戻り読みの必要がない英文文書および戻り読みの必要があるシフト JIS の日本語文書とする．各容量は 100KB の文書を繰り返し読み込むことで実現している．英文検索に対しては戻り読みを行わない．なお，本稿では 1KB=1000B，1MB=1000KB とする．表 2 に評価実験に用いたマシンの仕様を示す．

図 6 に，1 バイト 8 結果方式における英文文書の容量ごとの応答時間を底が 10 の対数グラフで示す．使用している検索キーワードは 5 バイトで 100KB 当たり 278 マッチする．容量が 1MB までならカーネル関数の呼び出しオーバーヘッドなどのために CPU の方が速い．100MB 以上では CUDA の方が約 3 倍速い．

図 7 に，200MB の日本語検索文書に 3 つの検索キーワードを与えた結果をそのマッチ数ごとに示す．戻り読みは 1 マッチにつき 1 回行うのでマッチ数がそのまま戻り読みの回数となる．CPU 版は戻り読み回数が多い 944000 マッチのとき，戻り読みをしない場合に対して戻り読みをする場合は約 2.24%検索時間が増えている．しかし CUDA では戻り読みによるオーバヘッドは確認できないほど小さい．これは個々マッチ要素に対するの戻り読みはスレッドごとに逐次で行うが，全体としては並列に行うためであると考えられる．

表 3 に，200MB の英文検索における時間の内訳を示す．結果処理時間とは検索が終了してから結果を転送し終わるまでの時間である．インデックス列を生成するもののみ，その生成をするために時間がかかるの

でこの時間を加えている．左から 3 つは検索結果を変更せず全て転送する方式で結果転送時間はマッチ数に依存しない．したがって，マッチ数としては多いもの 1 種類だけを示す．右の 6 つの項目はその結果の処理方法によりマッチ数に依存して変わる．そこで，多・中・少のマッチ数に対する載せている．この多・中・少のマッチ数はそれぞれ 556000，2000，278 である．この 3 種類のうちマッチ数が多・中のものは，マッチ要素が各マッチ位置が文書の広範に分布しているが，少のものは局所的に存在する．CPU はマッチ数が多で 197.0ms かかる．またテキストの転送時間は 200MB で 131.5m である．

検索時間について注目する．オフセット方式以外は高々6%程度の差であり，CPU の検索時間の 4 分の 1 程度となっている．オフセット方式に時間がかかっている理由は，検索時におけるオフセット列の生成において各スレッドにおけるレジスタの使用数が多く，並列に実行できるスレッド数が減るためである．

次に結果の処理にかかった時間について注目する．1 バイト 8 結果方式の転送時間は 1 バイト 1 結果方式の約 12.7% であり容量比に合わせて短縮できている．更にオフセット方式の結果転送時間は 1 バイト 8 結果の約 78%となっている．この理由は，検索キーワードを構成するバイト値の種類数が 5 個であるので，図 1 より検索結果のサイズが 4/5 になっているからである．部分転送を行う方法は，マッチ数としては少である場合は中である場合の約 13.9%である．しかし少の転送時間は中の約 8.75%となっている．これは，少はマッチが局所的であるので，複数のスレッドが生成した結果を一度の転送命令で取得するからである．多は転送回数が多いために非常に時間がかかっている．インデックス列の方法は，オフセットの導出は検索時間の 2%以内で完了している．インデックス生成および結果転送の時間はマッチ数に依存し，マッチ数が多いときはそれぞれ検索時間の 5%程度の時間がかかっている．それらを合計して，結果の処理時間は検索時間の約 12.3%となる．マッチ数が多いもので次に速いのはオフセット方式であるが，その約 32.2%まで減らせているので他の方法にくらべて十分高速である．その結果，テキストの転送を含めない応答時間 (応答時間 T_1) は，CPU の検索時間の約 26.2%まで削減している．しかし基本のアルゴリズムが BM 法であるので，検索キーワードが長いほど検索時間が短くなり，マッチ数も減る傾向にある．そのような状況ではオフセット方式および部分転送の方が高速である可能性がある．

最後に，テーブルなどの転送に加えて文書を検索時

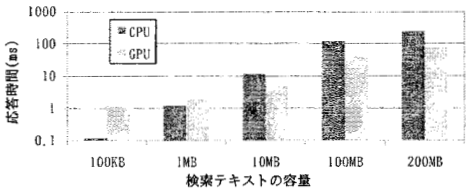

図 6　1 バイト 8 結果方式における英文検索における応答時間

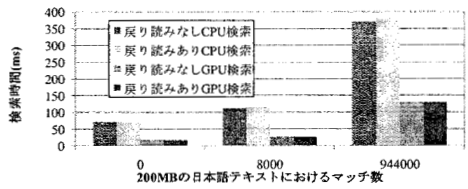

図 7　日本文検索における検索時間

に転送する場合の応答時間 (応答時間 T_2) について述べる．文書の転送時間だけで CPU 検索時間の約 2/3 もあるために応答時間 T_2 は短いものでも CPU 検索時間と大差がない．この検索は検索キーワードを 1 つ与えたときであり，複数キーワードを並列に検索しても文書の転送は最初の 1 回だけである．そのため，理論的には複数のキーワードを検索するなら文書の転送時間は相対的に小さくなる．これより，1 回の検索で複数のキーワードを検索するならば主記憶上にある文書を検索対象として高速に検索できる．更に，実験に用いた GPU の GeForce 8800 GTX のインターフェースは PCI-Express であるが，最新の GPU のインターフェースは PCI-Express 2.0 に対応している．PCI-Express 2.0 のバンド幅は PCI-Express の 2 倍の速度であり，理論的には 2 倍の速度で転送できる．また，計算および転送を並行して行うことが可能となっている．したがって，検索と主記憶からの文書の転送を並行して行えばデータの転送時間を（一部）隠蔽できる．ただし，検索したい文書を転送している間は当然検索できないので，数百 MB 以上の大規模な文書を 100MB 程度に分割して一部分を検索している間に次に検索する文書を転送するようにすればよい．仮に転送時間を完全に隠蔽できたとすると主記憶上の文書でも CPU の 4 倍程度の速度で検索できる．

4. 結　　論

GPU の処理能力を活かすために，GPU の汎用計算環境 CUDA を用いて BM 法による逐次検索の高並列マルチスレッド処理を実装した．検索は BM 法をデータ分割で並列実行する．実験の結果，CPU にく

表 3　200MB の英文検索における実行時間の内訳 (ms)

格納方法	1 バイト 1 結果	1 バイト 8 結果	オフセット	1 バイト 8 結果			1 バイト 8 結果		
転送方法	全転送			マッチ部分転送			インデックス列転送		
マッチ数	556000	556000	556000	556000	2000	278	556000	2000	278
前処理	3.167	1.390	1.154	1.453	1.604	1.940	1.858	1.860	2.120
検索	43.31	44.50	63.01	45.98	37.16	37.66	44.29	36.93	36.51
オフセット生成	-	-	-	-	-	-	0.7025	0.6594	0.6583
インデックス生成	-	-	-	-	-	-	2.656	1.020	0.6334
結果転送	168.9	21.49	16.95	1965	36.96	3.233	2.107	0.02736	0.02022
結果処理時間	168.9	21.49	16.95	1965	36.96	3.233	5.466	1.706	1.312
応答時間 T_1	215.3	67.38	81.11	2013	75.72	42.83	51.61	40.50	39.94
応答時間 T_2	346.8	198.8	212.6	2144	207.2	174.3	183.1	172.0	171.4
T_1/CPU 時間	1.09	0.342	0.41	10.2	0.384	0.217	0.262	0.206	0.203
T_2/CPU 時間	1.76	1.01	1.08	10.9	1.05	0.885	0.929	0.873	0.870

CPU の検索時間は 197.0ms　T_1 は前処理+検索+結果処理時間　T_2 は T_1+文書転送時間 (131.5ms)　多:556000　中:2000　少:278

らべて 4 倍〜5 倍の速度で計算できた．またデータ構造の工夫により，GPU における文書の転送時間を含めない応答時間を GPU の検索時間の 1〜2 割の増加に抑えられた．しかし文書の転送時間を含めるとその応答時間は CPU の検索時間と大差がなくなる．よって，現状では主記憶上の文書を検索する意味がない．新しい GPU を用いることができれば，検索と転送を非同期にできる上に転送速度が上がるので，主記憶にある数百 MB 以上の文書を高速に検索できるようになると考えられる．今後の課題は，オフセット方式の検索速度を上げるためにレジスタの使用数を削減することおよび検索と転送を非同期にすることである．

謝辞　本研究は一部，日本学術振興会科学研究費補助金基盤研究 (B)18300009 および若手研究 (B)18700058 の補助による．

参 考 文 献

1) 北研二，津田和彦，獅々堀正幹，“情報検索アルゴリズム”，共立出版 (2002)
2) R. S. Boyer and J. S. Moore, “A Fast String Searching Algorithm”, Communication of the ACM, Vol.20, pp. 762–772 (1977)
3) 田中利一，“主記憶データベース技術”，東芝レビュー，Vol.57, No.4, pp.56-57 (2002)
4) 深沢 千尋，”文字コード超研究”，ラトルズ (2003)
5) NVIDIA, “CUDA Programming Guide 1.1”, http://www.nvidia.com/object /cuda_develop.html (2007)
6) NVIDIA, “Technical Brief: NVIDIA GeForce 8800 GPU Architecture Overview”, http://www.nvidia.com/page /8800_tech_briefs.html (2006)
7) GPGPU, “CUDPP: CUDA Data Parallel Primitives Library”, http://www.gpgpu.org/ developer/cudpp/ (2007)
8) S. Sengupta, M. Harris, Y. Zhang and J. Owens, “Scan Primitives for GPU Computing”, In Proceedings of Graphics Hardware 2007, pp. 97–106 (2007)

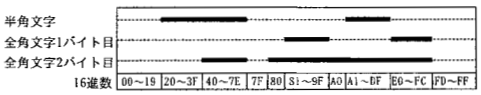

図 8　シフト JIS の符号領域

付　　録

証　　明

長さ L バイトのテキスト T から長さ $l(\leq L)$ バイトの検索キーワード W を見つける文字列照合問題を考える．W の h バイト目を W_h，W の h バイト目から k バイト目の部分テキストを $W_{h..k}$ と表す．W 中の異なるバイト値の個数が n 個であるとき，次の定理が成り立つ．

定理　T が s バイト目から W を含むとすると，任意の $i \in \{1, 2, \cdots, n-1\}$ について，T が $(s+i)$ バイト目から W を含むことはない．

証明：T が $(s+m)$ バイト目 $(m > 0)$ から W を含むとすると，任意の $j \in \{1, 2, \cdots, l-m\}$ について $W_j = W_{m+j}$ となる．$m \geq l - m$ のとき，W 中の $(l-m)$ 個のバイト値は重複しているので，W 中の異なるバイト値の数は高々 $(l - (l-m)) = m$ 個となる．$m < l - m$ のとき，W は $W_{1..m}$ を長さ l 以上になるまで繰り返し連結したものの先頭 l バイトに等しいので，W 中の異なるバイト値の数は高々 m 個となる．ゆえにいずれの場合でも $n \leq m$ となる．∎

戻り読み

シフト JIS テキストにおいて指定された位置のバイト値が 2 バイト文字の 2 バイト目かどうかは次のように判定できる．まず目的のバイト値から 1 バイトずつ文書を遡る．確実な文字の切れ目を探す．そしてその切れ目の後ろのバイトから更に後ろへ 1 バイトずつ各バイト値を分類していく．

図 8 にシフト JIS の符号領域を示す．目的のバイト値が 0x80 か 0xA0 であれば全角文字の 2 バイト目であり，0x3F 以下に含まれていれば全角文字の 2 バイト目ではないと直接決定できる．そしてバイト値がそれ以外であれば，0x80 以下，0xA0 および 0xDF のいずれかが出てくるまで文書を遡る．その範囲にあるバイト値が見つかれば，そのバイト値の次のバイト値が半角文字か全角文字の 1 バイト目となる．仮にそのバイト値が全角文字の 1 バイト目であれば次のバイトは 2 バイト目であるので読み飛ばす．これを目的のバイト値の分類が分かるまで繰り返す．