GPU を使用したビット並列アルゴリズムに基く最長共通部分列の導出

河南　克也[†]　　藤本　典幸[†]

2 つの文字列の最長共通部分列を求める LCS 計算は遺伝子の比較などの様々な応用を持つ．本論文では Crochemore らのビット並列アルゴリズムを用いて改善した Hirschberg の CPU 用 LCS アルゴリズムを，GPU を用いて高速化する方法を提案する．Crochemore らのアルゴリズムは 1 ビット毎に同時並列実行が可能なビット毎の論理演算の他に，逐次性が強い算術加算など，GPU での実装に工夫が必要な演算も含んでいる．本論文では特にそれらの演算の効率的な実装方法について論じる．その方法に基いて設計したプログラムを，2.93GHz Intel Core i3 530 CPU と GeForce 8800 GTX，GTX 285，GTX 480 GPU を用いて評価した結果，CPU 上でのビット並列アルゴリズムに対しては最大 12.77 倍，Hirschberg の CPU 用 LCS アルゴリズムに対しては最大 76.5 倍高速であった．また，Kloetzli らの GPU を用いた既存アルゴリズムに対しては 10.9 倍から 18.1 倍高速であった．

Computing the Longest Common Subsequence with Bit-Parallelism on a GPU

Katsuya Kawanami[†] and Noriyuki Fujimoto[†]

The longest common subsequence (LCS for short) for given two strings has various applications, e.g. comparison of DNAs. In this paper, we propose a GPU algorithm to accelerate Hirschberg's CPU LCS algorithm improved using Crochemore et al's bit-parallel CPU algorithm. Crochemore's algorithm includes bitwise logical operators which can be computed in embarrassingly parallel. However, it also includes some operators with less parallelism, e.g. an arithmetic sum. In this paper, we focus on how to implement these operators efficiently in parallel. Our experiments with 2.93GHz Intel Core i3 530 CPU, GeForce 8800 GTX, GTX 285, and GTX 480 GPUs show that the proposed algorithm runs maximum 12.77 times faster than the bit-parallel CPU algorithm and maximum 76.5 times faster than Hirschberg's LCS CPU algorithm. Furthermore, the proposed algorithm runs 10.9 to 18.1 times faster than Kloetzli's existing GPU algorithm.

1. は じ め に

2 つの文字列の類似度を示す指標としては，例えば編集距離など，様々なものがある．その中の 1 つに最長共通部分列[4)]（Longest Common Subsequence，以下 LCS）がある．LCS は遺伝子配列の比較や文字列のあいまい検索，スペルチェックなどに応用できる．

2 つの文字列の間の LCS の 1 つを計算するアルゴリズムが，Hirschberg によって提案されている[5)]．このアルゴリズムは再帰的であり，毎回の呼出で，LCS の長さ（Length of LCS，以下 LLCS）を計算するアルゴリズムをサブルーチンとして呼出す．Hirschberg のアルゴリズムは $O(mn)$ の時間計算量と $O(m+n)$ の空間計算量で計算される．LLCS 計算を高速化する方法としては，ビット並列計算を用いて，$O(\lceil m/w \rceil n)$（w は計算機のワードサイズ）の時間計算量と $O(m+n)$ の空間計算量で計算する方法[2)] が知られている．LLCS 計算にビット並列アルゴリズムを用いると，Hirschberg のアルゴリズムは高速化できるが，遺伝子配列等の比較では 100 万文字以上の文字列を扱うので，更なる高速化が必要である．そこで，GPU を使用してビット並列アルゴリズムを更に高速化することを考えた．ビット並列アルゴリズムではビット毎の論理演算と算術加算を行う．ビット毎の論理演算は並列性が高いが，算術加算は逐次性が強く，並列性を高める為に工夫を行った．

本論文で提案する手法に基いて，CUDA[1),3),7),9),10)] でビット並列アルゴリズムを実装し，GeForce GTX 480 で評価実験を行ったところ，提案アルゴリズムは 2.93GHz Intel Core i3 530 CPU でのビット並列アルゴリズムに対しては 2.1 倍から 12.77 倍高速に，CPU でビット並列アルゴリズムを使わない場合に対しては，

† 大阪府立大学　大学院理学系研究科　情報数理科学専攻
Department of Mathematics and Information Sciences, Graduate School of Science, Osaka Prefecture University

© 2011 Information Processing Society of Japan

37.3 倍から 76.5 倍高速に動作した．また，Kloetzli らの既存研究[8] に対して，提案アルゴリズムは同一の GPU（GeForce 8800 GTX）を用いた場合，10.9 倍から 18.1 倍高速であることがわかった．

以降の論文の構成は次の通りである．第 2 章で LCS の定義と既存アルゴリズムについて簡単に説明する．第 3 章で提案手法を示す．第 4 章で評価実験について述べる．第 5 章でまとめと今後の課題について述べる．スペースの制限のため，本論文では GPU のアーキテクチャおよびプログラミングの概要については述べない．これらに不慣れな読者は文献[1),3),7),9),10] を参照されたい．

2. LCS

2.1 LCS の定義

文字列 C= $c_1c_2\cdots c_p$ と文字列 A= $a_1a_2\cdots a_m$ を考える．このとき，C の添字集合 $\{1,2,\cdots,p\}$ から A の添字集合 $\{1,2,\cdots,m\}$ への写像 F が次の条件 C1, C2 を満たすなら，C は A の部分列（Subsequence）であるという．

C1: $F(i)=k$ は $c_i=a_k$ の必要十分条件である．

C2: $i<j$ ならば $F(i)<F(j)$ である．

ただし空列（空の文字列）は任意の文字列の部分列とする．文字列 A の部分列であり，かつ文字列 B の部分列でもある文字列を，A と B の共通部分列（Common Subsequence）と定義する．A と B の共通部分列の中で最長のものを，A と B の LCS と定義する．例えば文字列 abcdefghij と cfilorux の LCS は cfi，文字列 abcde と baexd の LCS は ad, ae, bd, be である．

2.2 LCS の長さの計算方法

LLCS は動的計画法によって計算できる[5]．このアルゴリズムは，文字列 A の長さが m，文字列 B の長さが n のとき，$(m+1)\times(n+1)$ セルの表 L を次のルール R1〜R3 に従って埋めると，L$[m][n]$ に A と B の LLCS が格納されるというものである．

R1: $i=0$ または $j=0$ のとき，L$[i][j]$=0

R2: A$[i-1]$=B$[j-1]$ のとき，L$[i][j]$=L$[i-1][j-1]$+1

R3: その他のとき，L$[i][j]$=max(L$[i][j-1]$,L$[i-1][j]$)

表 L を埋めるのに必要な時間計算量と空間計算量はともに $O(mn)$ である．

ここで R2 と R3 に注目すると，表の i 行目（$1\leqq i\leqq m$）を求める為に必要な行は i 行目と $i-1$ 行目だけであることが分かる．この性質を利用して，表の保持する行を i 行目と $i-1$ 行目の 2 行のみにしたものが，Listing1 で示されるアルゴリズム[5] である．ここで K は 2 行 $n+1$ 列の計算用配列であり，L は 1 行 $n+1$ 列の出力用配列である．

Listing 1 Hirschberg の LLCS アルゴリズム

```
1  Input:string A of length m, B of length n
2  Output:LLCS L[j] of A and B[0...j-1]
3    for all j(0<=j<=n)
4  llcs(A,m,B,n,L){
5    for(j=0 to n) K[1][j] = 0
6    for(i=1 to m) {
7      for(j=0 to n) K[0][j] = K[1][j]
8      for(j=1 to n)
9        if(A[i-1] == B[j-1])
10         K[1][j] = K[0][j-1] + 1
11       else
12         K[1][j] = max(K[1][j-1],K[0][j])
13   }
14   for(j=0 to n) L[j] = K[1][j]
15 }
```

Listing 2 Hirschberg の LCS アルゴリズム

```
1  Input:string A of length m, B of length n
2  Output:LCS C of A and B
3  lcs(A,m,B,n,C) {
4    if(n==0) C = ""(null string)
5    else if(m==1) {
6      for(j=1 to n)
7        if(A[0]==B[j-1]) {
8          C = A[0]
9          return
10       }
11     C = ""
12   }
13   else {
14     i = m/2
15     llcs(A[0...i-1],i,B,n,L1)
16     llcs(A[m-1...i],m-i,B[n-1...0],n,L2)
17     M = max{L1[j]+L2[n-j]} (0<=j<=n)
18     k = min{L1[j]+L2[n-j] == M} (0<=j<=n)
19     lcs(A[0...i-1],i,B[0...k-1],k,C1)
20     lcs(A[i...m-1],m-i,B[k...n-1],n-k,C2)
21     C = strcat(C1,C2)
22   }
23 }
```

Listing1 の Hirschberg の LLCS アルゴリズムでは，L$[j]$ に文字列 A と文字列 B の j 文字目まで（B[0...$j-1$]）の LLCS が格納される．このようにすると時間計算量は $O(mn)$ のままで，空間計算量を $O(m+n)$ に削減できる．遺伝子配列の比較では 100 万文字以上の文字列を扱う場合が多いので，空間計算量を線形に抑えることは重要である．

2.3 Hirschberg の LCS アルゴリズム

Hirschberg の提案する LCS 計算アルゴリズム[5] を Listing2 に示す．ここで，文字列 S の部分文字列 S$[l...u]$（$l\leqq u$）を反転した文字列を S$[u...l]$ で表す．このアルゴリズムは LLCS の計算を行いながら再帰

ⓒ 2011 Information Processing Society of Japan

Listing 3　Crochemore らの LLCS を求めるビット並列アルゴリズム

```
1  Input:string A of length m, B of length n
2  Output:LLCS L[i] of A[0...i-1] and B
3    for all i(0<=i<=m)
4  llcs_bp(A,m,B,n,L){
5    for(c=0 to 255)
6      for(i=0 to m-1)
7        if(c==A[m-i-1])
8          PM[c][i] = 1
9        else
10         PM[c][i] = 0
11   for(i=0 to m-1) V[i] = 1
12   for(j=1 to n) {
13     V = (V + (V & PM[B[j]]))
14           | (V & ~(PM[B[j]]))
15   }
16   L[0] = 0
17   for(i=1 to m)
18     L[i] = L[i-1]+(1-V[i-1])
19 }
```

的に LCS を求める．LCS の計算全体に必要な時間計算量は $O(mn)$，空間計算量は $O(m+n)$ である．

2.4　ビット並列計算による LLCS の導出法

LLCS を高速に導出するアルゴリズムとしては，ビット並列性を利用するものがある．Listing3 に示すアルゴリズムは，Crochemore らの提案するビット並列計算を用いた LLCS の計算法[2] である．V は m ビットのビット列を記憶する変数である．&はビット毎の AND を，| はビット毎の OR を，~はビット毎の NOT を，+ は算術加算を表す演算子である．ただし + はビット列 B の B[0] を最下位とみなす．

Crochemore らのアルゴリズムでは，最初にパターンマッチベクトル（以下 PMV）を作成する．文字 c に対する文字列 S の PMV P とは，$\mathrm{S}[i]=c$ となるビット $\mathrm{P}[i]$ のみ 1 であり，それ以外のビットは全て 0 である m ビットベクトルのことである．Listing3 では 5〜10 行目で各文字 c（1 バイト文字を想定しているので $0 \leqq c \leqq 255$）に対する文字列 A の PMV を作成し，それを反転したビット列を，$256 \times m$ ビットの 2 次元ビット配列 PM の $\mathrm{PM}[c]$ に格納している．

このアルゴリズムでは動的計画法の表の縦 1 列を数値配列としてではなく，1 つ上のセルとの差を表すビットベクトルとして格納し，ビット演算を繰り返すことで動的計画法の表を右方向へと計算していくのと等価な処理をしている．

なお，このアルゴリズムが出力する「LLCS を計算する表の最後の 1 列」はビットベクトルの形だが，1 であるビットを数え上げれば $O(m)$ の時間計算量で数値配列に変換できる（Listing3 の 16〜18 行目）．

Crochemore らのアルゴリズムの時間計算量は $O(\lceil m/w \rceil n)$，空間計算量は $O(m+n)$ である．ここで w は計算機のワードサイズである．

3. 提案手法

3.1　GPU 化の対象アルゴリズム

Listing2 の LCS アルゴリズムの中でもっとも多くの計算時間を必要とする箇所は，LLCS を求める関数 llcs() である．本論文では，Listing3 のビット並列アルゴリズムを用いて LLCS 計算を高速化した Listing2 の LCS アルゴリズム（すなわち，Listing2 中の llcs() 呼出に対して，次に述べる変更および改善を施したものを，Listing3 の llcs_bp() に変更したもの）を，GPU を用いて更に高速化する手法を提案する．なお，GPU は 64bit モードでも，整数レジスタのサイズは 32bit であるので，ワードサイズ w は 32 である．

Listing2 の 16 行目の llcs() では，A と B を反転した文字列の部分文字列に対して LLCS を求めている．しかし毎回の呼出で文字列を反転しているとオーバーヘッドが大きくなるので，入力文字列を後ろから見る LLCS 計算関数として，新たに関数 llcs'() を作る．llcs'() は文字列 A,B を見る順序が異なる以外は llcs() と変わらないので，llcs() とほぼ同様にして作ることができる．llcs'() に対応するビット並列アルゴリズム llcs_bp'() も作る．

Listing1 の Hirschberg の LLCS アルゴリズムが出力するのは，LLCS を計算する表の第 m 行である．しかし Listing3 に示した Crochemore らのアルゴリズムでは表の縦 1 列をビットベクトルで表現し，表を第 0 列から第 n 列まで計算する．つまり，表の第 n 列が出力されてしまう．そこで，行毎に計算している元の Listing1 の LLCS アルゴリズムを，列毎に計算するように変更する．それと同時に Listing2 の LCS アルゴリズムも列毎の計算に対応した形に変更する．

また，提案手法では 32 文字分の情報をビット列として 1 つの unsigned int 型変数に埋め込むので，文字列長は 32 の倍数でなければならない．その為，文字列長が 32 の倍数でない場合には，パディングを行って 32 の倍数にする必要がある．パディングに使えるのは A と B の中に現れない適当な文字，つまり制御文字や 2 バイト文字の 1 バイト目などである．

3.2　提案手法の概要

GPU を使って計算するのは Listing2 の 15〜16 行目の llcs() だけで，その他の部分は CPU が実行する．

ⓒ 2011 Information Processing Society of Japan

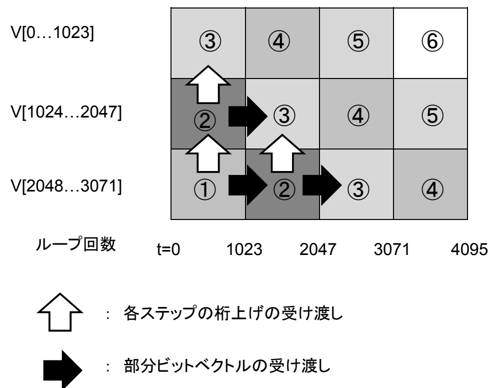

図 1　m=3072,n=4096 の場合のブロック分割

これは，Listing2 の 19〜20 行目に lcs() の再帰呼出の部分があるが，GPU では，モデルによって，再帰呼出が全く，あるいは深くはできない為である．

Listing3 の LLCS アルゴリズムは m ビットのビットベクトル V に対して，ビット毎の論理演算（&, |, ˜）と，算術加算（+）の演算を行う．ビット毎の論理演算の並列化は容易である．しかし算術加算には桁上げが存在し，最悪の場合には最下位から最上位まで桁上げが伝播するので，逐次性が強い．この為，算術加算の計算の並列性を高めるには工夫が必要である．

このビットベクトル V を部分ベクトルに分割して並列処理することを考える．GPU 上ではビットベクトルは unsigned int 型変数の配列で表現する．実験に使用した GPU では unsigned int 型は 32 ビットである．CUDA では同一ワープ内の 32 スレッドの実行は命令レベルで同期している（SIMD 実行）ので，1 ブロックのスレッド数は 32 とする．これにより，ブロック内のスレッド間の同期コストがゼロになる．また，1 スレッドが 1 要素（32 ビット）を処理することにより，1 ブロックが 1024 ビットを処理するようにする．

なお, 一度のカーネル関数呼出でループ n 回分全てを計算するのではなく，一度の呼出で計算するのはループ 1024 回とする．この 1024 ループを 1 ステップとする（第 j ステップは $1024\times j$ ループ目から $1024\times(j+1)-1$ ループ目である．また $0 \leqq j \leqq \lceil n/1024 \rceil -1$ である）．ブロックとステップのサイズを揃えるのは，こうすると 1 スレッドにつき 1 変数だけの読み書きで，桁上げ情報の読み書きができる為である．

図 1 は $m = 3072, n = 4096$ の場合の，ブロック・ステップ分割の例である．この図の中の四角形 1 つは，1 ブロックの 1 ステップ分を表す．以下，これを計算ブロックと呼ぶ．図 1 で同じ番号が振られた計算ブロックは同時並列に計算が可能である．ただし各計算ブロックは，その左と下の計算ブロックが終わっていなければ計算を開始できない．その為，最初のカーネル関数呼出時に計算できる計算ブロックは，最も左下にある V[2048 $\cdots$ 3072] を担当するブロックの第 0 ステップのみである．2 回目のカーネル関数呼出時には V[2048 $\cdots$ 3072] の第 1 ステップと V[1024 $\cdots$ 2047] の第 0 ステップが計算できる．各カーネル呼出の時点で計算できる計算ブロックは全て計算するものとすると，図 1 の番号の順になり，$\lceil m/1024 \rceil + \lceil n/1024 \rceil - 1$ 回のカーネル呼出で LLCS が計算できる．

図 1 の黒矢印は，部分ベクトル V[$i \cdots i$+1023] を担当するブロックが j ステップ目の処理を終了した時点での V[$i \cdots i$+1023] の値を，同ブロックの j+1 ステップ目に渡すことを表す．また，白矢印は，部分ベクトル V[$i \cdots i$+1023] を担当するブロックが j ステップ目に記録した，各ループ回数での上位ブロックへの桁上げを，次回のカーネル関数呼出で V[$i - 1024 \cdots i - 1$] を担当するブロックの j ステップ目に渡すことを表す．黒と白の矢印で示したセル間の値の受け渡しは，カーネル関数内のループの外側で行う．特に白矢印で示した各ループ回数での桁上げは，ループの実行中は shared メモリに記録し，ループ終了後に shared メモリから global メモリに書き込む．読み出す時もループ開始前に global メモリから shared メモリに読み出し，ループ実行中は shared メモリから読み出す．

3.3 カーネル関数の引数と機能

1 ステップ分のループを回すカーネル関数 llcs_kernel() と，呼出元である llcs_gpu() の機能について説明する．これらの擬似コードは Listing4 に示した．この llcs_gpu() が，Listing3 の llcs_bp() を GPU 上で実装したものである．また，llcs_bp'() を GPU 上で実装した関数 llcs_kernel'() と llcs_gpu'() も作成したが，こちらはほぼ同様なので説明は省略する．

まず llcs_kernel() から説明する．入力の m, n は文字列 A, B の長さを，dstr2 は global メモリ上にコピーした文字列 B を表す．g_V はビットベクトル V を格納する global メモリ上の配列である．g_PM は，各文字 c に対する文字列 A の PMV を格納する二次元配列である．g_PM[c] が，文字 c に対する文字列 A の PMV にあたる．car は桁上げ情報を格納する配列である．使用する時には二次元配列とみなして，ダブルバッファリングを行う．num はカーネル関数の実行回数を表し，llcs_kernel() 内でこのブロックが現在何ステップ目を計算するのかを求めるのに使う．11〜15

ⓒ 2011 Information Processing Society of Japan

Listing 4 llcs_kernel() と llcs_gpu() の擬似コード

```
1  __global__ void llcs_kernel(
2    int m, n, char *dstr2,
3    unsigned int *g_V, *g_PM, *car,
4    int num)
5  {
6    index = 全体でのスレッド番号;
7    count = このブロックのステップ数;
8    cursor = 1024 * count;
9    V = g_V[index];
10   global メモリ上のcar から桁上げ情報を読み出す;
11   for (j=0 to 1023) {
12     if(cursor+j >= n) return;
13     PM = g_PM[dstr2[cursor+j]][index];
14     V = (V & (~PM)) | (V + (V & PM));
15   }
16   g_V[index] = V;
17   global メモリ上のcar に桁上げ情報を書き込む;
18 }
19
20 void llcs_gpu(
21   char *A, *B, int m, n,
22   char *dstr1, *dstr2, int *output,
23   unsigned int *g_V, *car, *g_PM)
24 {
25   dstr1 = パディングしたA のコピー;
26   dstr2 = B のコピー;
27   num_x = (m+1023)/1024;
28   num_y = (n+1023)/1024;
29   for (i=0 to ((m+31)/32)-1)
30     g_V[i] = 0xFFFFFFFF;
31   PMV の作成;
32   for(i=1 to num_x+num_y-1)
33     llcs_kernel() in Parallel on GPU
34       (gridDim=num_x, blockDim=32)
35   for(i=0 to m) {
36     g_V のi ビット目までの中で0 であるビットを数える;
37     output[i] に数えた結果を書き込む;
38   }
39 }
```

行目の for ループが，1 ステップ分（1024 ループ分）だけのループを回す処理にあたる．図 1 で白または黒矢印で示したセル間の値の受け渡しは，ループの外の 9, 10, 16, 17 行目で行っている．

次に llcs_gpu() を説明する．ここでは 32〜33 行目の for ループで，カーネル関数 llcs_kernel() を $num_x + num_y - 1$ 回呼んでいる．前処理として 25 行目で文字列 A のパディングを行い，27〜28 行目でブロック数 num_x とステップ数 num_y を計算．29〜30 行目でビットベクトル V を全て 1 で初期化している．後処理として 35〜38 行目では，ビットベクトルを数値配列に変換して，出力配列 output に書き込んでいる．

3.4 算術加算の並列化

3.2 節の最初で述べた通り，+ には桁上げがある為，逐次性が強い．この + の並列性を高める為に，Sklansky が提案する全加算器の並列化法，条件付合計加算 (Conditional-Sum Addition)[6] を応用した．なお，条件付合計加算については文献[11] が詳しいので，そちらを参照されたい．

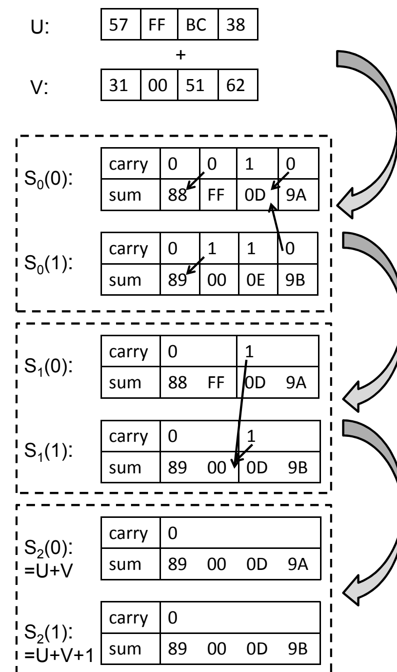

図 2 n ビット全加算器の並列化（$n = 32$ の場合）

Sklansky の条件付合計加算は，桁上げが 0 と 1 の 2 通りしか無いことを利用している．n ビットの加算を 1 ビット毎で並列に行うとき，各半加算器はあらかじめ下位からの桁上げがある場合と無い場合の両方について，和と上位ビットへの桁上げを計算しておく．その後に，桁上げの伝播を並列に行う．提案手法ではこの条件付合計加算を，1 ビット毎の加算器ではなく，32 ビット毎の加算器に応用して利用する．

図 2 の例を用いて，n ビット全加算器の並列化法を説明する．なお，実装に用いたのは 4 バイト単位 ×32 要素の 1024 ビット全加算器であるが，この例は 1 バイト単位 ×4 要素の 32 ビット全加算器である．図 2 では 2 つの部分ベクトル U, V と下位からの桁上げ carry の，和と上位への桁上げを求めている．U+V+carry を求める為にまず 1 バイト毎の和と上位バイトへの桁上げを並列に求める．$S_0(0)$ は下位バイトからの桁上げが無かった場合の和と桁上げであり，$S_0(1)$ は桁上げがあった場合の和と桁上げである．次に $S_0(0)$, $S_0(1)$ のそれぞれを 2 バイト毎にまとめることを考える．その為に，$S_0(1)$ の 3 バイト目に $S_0(0)$ の 3 バイト目をコピーする．こうして求めた 2 バイト毎の和が，$S_1(0)$, $S_1(1)$ である．同様にして $S_1(0)$, $S_1(1)$ から 4 バイト毎の和を求めると，$S_2(0)$,$S_2(1)$ となる．$S_2(0)$ が U+V（下位 4 バイトからの桁上げが無かった場合

ⓒ 2011 Information Processing Society of Japan

Listing 5　算術加算関数 plus() のコード

```
 1  #define tx threadIdx.x
 2  __device__ void plus(unsigned int *v1,
 3    unsigned int *v2, unsigned int *a0,
 4    unsigned int *a1, int j,
 5    unsigned int *u_c, unsigned int *l_c,
 6    int index, int lenv1, unsigned int bit)
 7  {
 8    __shared__ bool s_c0[32],s_c1[32];
 9    unsigned int maximum;
10
11    s_c0[tx] = false; s_c1[tx] = true;
12    if(index >= lenv1/32) return;
13    maximum = d_max(*v1, *v2);
14    *a0 = *v1+*v2; *a1 = *s_a0+1;
15    if(*a1 >= maximum)
16      s_c1[tx]=false;
17    if(*a0 < maximum) {
18      s_c0[tx]=true; s_c1[tx]=true;
19    }
20    if (tx & 1 == 0)
21      if (s_c0[tx+1]) {
22        *a0 = *a1; s_c0[x] = s_c1[tx];
23      } else if (!s_c1[tx+1]) {
24        *a1 = *a0; s_c1[tx] = s_c0[tx];
25      }
26    if (tx & 2 == 0)
27      if (s_c0[(tx/2+1)*2]) {
28        *a0 = *a1; s_c0[tx] = s_c1[tx];
29      } else if (!s_c1[(tx/2+1)*2]) {
30        *a1 = *a0; s_c1[tx] = s_c0[tx];
31      }
32    if (tx & 4 == 0)
33      if (s_c0[(tx/4+1)*4]) {
34        *a0 = *a1; s_c0[tx] = s_c1[tx];
35      } else if (!s_c1[(tx/4+1)*4]) {
36        *a1 = *a0; s_c1[tx] = s_c0[tx];
37      }
38    if (tx & 8 == 0)
39      if (s_c0[(tx/8+1)*8]) {
40        *a0 = *a1; s_c0[tx] = s_c1[tx];
41      } else if (!s_c1[(tx/8+1)*8]) {
42        *a1 = *a0; s_c1[tx] = s_c0[tx];
43      }
44    if (tx <= 15)
45      if (s_c0[16]) {
46        *a0 = *a1; s_c0[tx] = s_c1[tx];
47      } else if (!s_c1[16]) {
48        *a1 = *a0; s_c1[tx] = s_c0[tx];
49      }
50    if(tx > 0) return;
51    if(l_c[j/32] & bit)
52      if(s_c1[tx]) u_c[j/32] |= bit;
53    } else {
54      if(s_c0[tx]) u_c[j/32] |= bit;
55  }
```

の和と桁上げ）となり，$S_2(1)$ が U+V+1（下位 4 バイトからの桁上げがあった場合の和と桁上げ）となる．

この方法の最大の利点は，$S_t(0)$ と $S_t(1)$ から $S_{t+1}(0)$ と $S_{t+1}(1)$ を求める処理が 2^t バイト毎で同時並列に行えることである．U, V の持つ要素数が l の場合，最終形の U+V+carry を求めるのに必要なこの処理の回数は，$log_2 l$ 回である．

この考えに基いて算術加算を実装したコードを，Listing5 に示す．unsigned int 配列 v1 と v2 が，加算する 2 本のビットベクトル（入力）である．a0 は下位からの桁上げが無い場合の 1024 ビット毎の和を，a1 は桁上げがある場合の 1024 ビット毎の和を代入する，出力用の配列である．u_c は上位ブロックに渡す桁上げを，l_c は下位ブロックから渡された桁上げを格納する配列である．lenv1 は文字列 A の長さであり，Listing5 内ではビットベクトルの長さを超えたところのスレッドの処理を止める為に使用している．j は現在のカーネル関数呼出でのループ回数であり，桁上げを l_c のどこから読み込むか，u_c のどこに書き込むかを求めるのに使用する．j と bit は下位ブロックからの桁上げを l_c のどこから読み込むか，また，上位ブロックへの桁上げを u_c のどこに書き込むかを求めるのに使用する．

入力ビットベクトルは，この plus() により加算された結果，$\lceil m/1024 \rceil$ 個の 1024 ビット毎の和（下位からの桁上げがある場合と無い場合との 2 通り）になる．最後にこれらの部分ベクトルは，前回のカーネル関数実行時に下位のブロックが求めた桁上げに従って，下位からの桁上げがある場合か無い場合のどちらであるかが決定される．このブロックが現在のステップで求めた「上位ブロックへの桁上げ」は，次のカーネル関数呼出時に 1 つ上のブロックが plus() を実行する時に必要なので，配列 u_c に保持している．ただし u_c はカーネル関数内のローカル変数なので，カーネル関数の終了前に global メモリ上に保存している．

3.5　その他の工夫

global メモリ上配列への memcpy は，llcs_kernel() の呼出元のホスト関数 llcs_gpu() や，Listing2 の再帰呼出に入る前に行っている．ただし global メモリ上配列の malloc は全て再帰呼出に入る前に行うものとし，呼出元のホスト関数 llcs_gpu() 内では行わない．

デバイス上で文字列にパディングを行う時には，ホストから元の文字列をコピーしてこなければならない．しかしホスト・デバイス間での転送はデバイス同士やホスト同士の転送に比べて低速であるので，ホスト・デバイス間での転送回数は極力減らしたい．そこでホスト・デバイス間での文字列の転送は 1 回のみとし，その 1 回でデバイス上に元の文字列のコピーを作成する．llcs_gpu() や llcs_gpu'() で文字列を使用する場合には，デバイス上に作成した元の文字列のコピーをデバイス上の作業用メモリ領域にコピーし，それに対してパディングを行う．

対象の文字列が一定値よりも短い場合には，PMV の作成やデバイス上の値のコピー等に必要な時間が，CPU 上で動的計画法を使って LLCS を計算する時間

© 2011 Information Processing Society of Japan

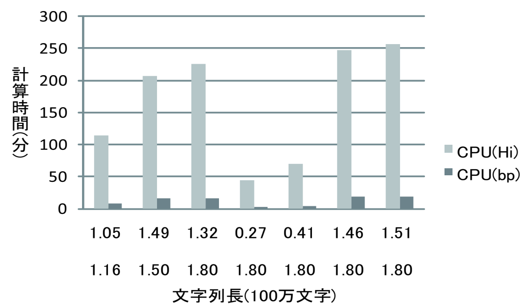

図 3 CPU(Hirschberg) と CPU(BitParallel) の比較

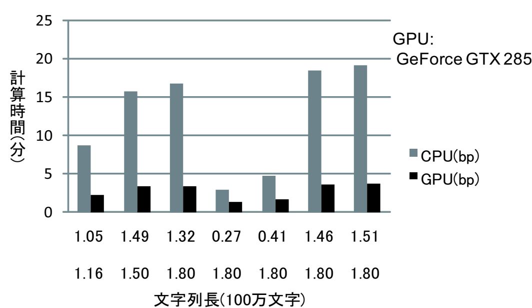

図 4 CPU(BitParallel) と GPU(BitParallel) の比較

よりも大きくなる．この場合はビット並列アルゴリズムを使った方が低速になってしまう．そこで，llcs_gpu() を呼び出す前に文字列の長さをチェックし，文字列 A の長さ m が 4096 未満ならば CPU 上で動的計画法を使って LLCS を計算している．

4. 評価実験

本章では提案アルゴリズムと Kloetzli らの既存 GPU アルゴリズム，および，Hirschberg と Crochemore らの既存 CPU アルゴリズムとの比較を行う．CPU は 2.93GHz Intel Core i3 530 の 1 コアを用い，OS は Windows 7 Professional 64bit，開発環境は CUDA 3.1 および Visual Studio 2008 Professional を用いた．コンパイルオプションは，Release モードのデフォルト値を用い，SSE 命令は用いていない．

4.1 既存 CPU アルゴリズムとの比較

CPU での動的計画法と CPU でのビット並列アルゴリズムとを比較した結果のグラフを図 3 に示す．また，CPU ビット並列アルゴリズムと GeForce GTX 285 を使用した場合の GPU ビット並列アルゴリズムとを比較した結果のグラフを図 4 に示す．これらのグラフの縦軸には計算時間（分単位）を，横軸には入力文字列 A, B の長さ（100 万文字単位）を示している．対象の文字列としては遺伝子配列を想定しているので，27 万文字から 180 万文字の長さの文字列を用いて実験した．また，表 1 には CPU(Hi) に対する，CPU(bp) と GeForce 8800 GTX，GeForce GTX 285，GeForce GTX 480 での GPU(bp) の速度向上率を示した．

図 3 と表 1 より，CPU(bp) は CPU(Hi) に対して，7 通りのいずれの場合にも，13 倍から 15 倍高速に動作した．図 4 より，GPU(bp) は CPU(bp) に対して，最短の 180 万文字と 27 万文字の場合でも 2.1 倍，最長の 180 万文字と 151 万文字の場合には 5.1 倍高速に動作した．

CPU(Hi) と比較すると，提案アルゴリズムは GeForce 8800 GTX を使用した場合には 26.5 倍から 43.1 倍，GeForce GTX 285 を使用した場合には 32.7 倍から 69.0 倍，GeForce GTX 480 を使用した場合には 37.3 倍から 76.5 倍高速であった．

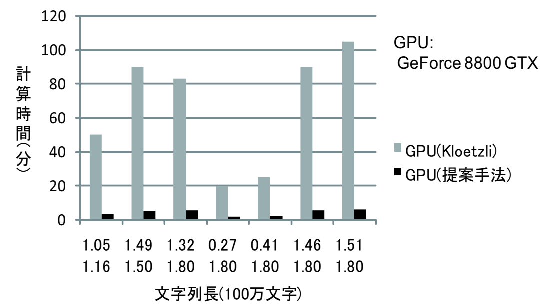

図 5 Kloetzli らの GPU アルゴリズムとの比較

4.2 既存 GPU アルゴリズムとの比較

Kloetzli らのアルゴリズム[8] との計算速度の比較を行った．Kloetzli らは CPU は AMD Athlon 64 を，GPU は GeForce 8800 GTX を使用していたので，同一の GPU で比較する為に本実験でも GeForce 8800 GTX を使用した．CPU は本実験で用いたものの方が高性能であるので同一の環境ではないが，提案アルゴリズムは CPU への依存度が低いので全く異なる結果にはならないと予想される．

その結果を図 5 のグラフに示した．図 3 と図 4 と同様に，グラフの縦軸には計算時間（分単位）を，横軸には入力文字列 A, B の長さを示している．

GPU(Kloetzli) は Kloetzli らのアルゴリズムの計算時間であり，GPU(提案手法) は提案アルゴリズムの計算時間である．

Kloetzli らの論文には正確な数値が記載されていなかったのでグラフから読み取った値との比較になるが，

ⓒ 2011 Information Processing Society of Japan

表 1 Hirschberg の CPU アルゴリズムに対する，提案手法の速度向上率

m	n	CPU(Hi)	CPU(bp)	GPU(bp)	GPU(bp)	GPU(bp)
(100 万文字)				(8800)	(285)	(480)
1.16	1.05	1.00	13.21	35.97	51.77	60.87
1.50	1.49	1.00	13.14	41.84	62.25	70.61
1.80	1.32	1.00	13.52	42.41	67.13	74.90
1.80	0.27	1.00	15.05	26.56	32.75	37.34
1.80	0.41	1.00	14.59	30.40	42.15	47.11
1.80	1.46	1.00	13.45	42.84	68.46	76.04
1.80	1.51	1.00	13.40	43.10	69.07	76.50

CPU:2.93GHz Intel Core i3 530, Hi:Hirshberg, bp:bit-parallel,
8800:GeForce 8800 GTX, 285:GeForce GTX 285, 480:GeForce GTX 480

表 2 CPU でのビット並列アルゴリズムと提案手法の計算時間比較

入力長 (MB)	CPU(bp)(秒)	GPU(bp)(秒)	速度向上率
2	1855.39	327.82	5.66
4	7424.83	940.72	7.89
8	29772.05	2992.01	9.95
16	131574.88	10306.66	12.77

CPU:2.93GHz Intel Core i3 530, GPU:GeForce GTX 285

最短の 180 万文字と 27 万文字の場合でも 12.0 倍，最長の 180 万と 151 万の場合には 17.6 倍高速に動作した．なお，最大の倍率が得られたのは 150 万文字と 149 万文字の場合であり，その場合には 18.1 倍高速であった．逆に倍率が最小だったのは 180 万文字と 41 万文字の場合であるが，その場合でも 10.9 倍高速であった．

4.3 CPU でのビット並列アルゴリズムと提案手法の計算時間比較

CPU ビット並列アルゴリズムと提案アルゴリズムとを比較した結果を表 2 に示す．ここでは 2 つの入力文字列の長さは等しいものとして，2MB，4MB，8MB，16MB の 4 通りについて実験を行った．最も右の列には，提案アルゴリズムの CPU ビット並列アルゴリズムに対する速度向上率を示した．表 2 から，2MB の場合には 5.66 倍であった速度向上率が，4MB では 7.89 倍，8MB では 9.95 倍と，文字列長が長くなるにつれて上昇していることが分かる．最長の 16MB の場合の速度向上率は 12.77 倍であった．

5. まとめ

本論文では最長共通部分列を求めるビット並列アルゴリズムを GPU 上で実装する方法について論じ，その方法に基いて作成したプログラムの評価を行った．その結果，CPU 上でのビット並列アルゴリズムに対しては 2.1 倍から 12.77 倍の，Kloetzli らの GPU 上のアルゴリズムに対しては 10.9 倍から 18.1 倍の高速化を達成した．今後の課題としては最新の Fermi アーキテクチャへの最適化や，同様の手法に基いて他のビット並列アルゴリズムを GPU 上で実装することなどが挙げられる．

参考文献

1) 青木尊之, 額田彰: はじめての CUDA プログラミング, 工学社 (2009)
2) M. Crochemore, C. S. Iliopoulos, Y. J. Pinzon, and J. F. Reid: A Fast and Practical Bit-Vector Algorithm for the Longest Common Subsequence Problem, Inf. Process. Lett, 80(6), 279–285 (2001)
3) M. Garland and D. B. Kirk: Understanding Throughput-Oriented Architectures, Comm. ACM, 53(11), 58–66 (2010)
4) D. Gusfield: Algorithms on Strings, Trees and Sequences: Computer Science and Computational Biology, Cambridge University Press (1997)
5) D. S. Hirschberg: A Linear Space Algorithm for Computing Maximal Common Subsequences, Comm. ACM, 18(6), 341–343 (1975)
6) J. Sklansky:Conditional-Sum AdditionLogic, IRE Trans. on Electronic Computers, EC-9, 226–231 (1960)
7) D. B. Kirk and W. W. Hwu: Programming Massively Parallel Processors: A Hands-on Approach, Morgan Kaufmann (2010)
8) J. Kloetzli, B. Strege, J. Decker, and M. Olano: Parallel Longest Common Subsequence Using Graphics Hardware, Proc. Eurographics Symposium on Parallel Graphics and Visualization (EGPGV) (2008)
9) E. Lindholm, J. Nickolls, S. Oberman, and J. Montrym: NVIDIA Tesla: A Unified Graphics and Computing Architecture, IEEE Micro, 28(2), 39–55 (2008)
10) J. Sanders and E. Kandrot: CUDA by Example: An Introduction to General-Purpose GPU Programming, Addison-Wesley Professional (2010)
11) M. Vai: VLSI Design, CRC Press (2000)

ⓒ 2011 Information Processing Society of Japan