Bulletin of Networking, Computing, Systems, and Software – www.bncss.org, ISSN 2186-5140
Volume 8, Number 1, pages 50–53, January 2019

GPU implementation of the fast Smith-Waterman algorithm using BPBC technique

Takahiro Nishimura*, Jacir L. Bordim†, Yasuaki Ito*, and Koji Nakano*
*Department of Information Engineering, Hiroshima University
Kagamiyama 1-4-1, Higashi-Hiroshima, 739-8527 JAPAN
†Department of Computer Science, University of Brasília
70910-900 Brasília - DF - Brazil

***Abstract*—The main purpose of this work is to propose a GPU implementation for the Smith-Waterman algorithm using Bitwise Parallel Bulk Computation (BPBC). The Smith-Waterman algorithm is based on a dynamic Programming approach that obtains the optimal local alignment between two sequences. The idea of this work is to perform the computation of the Smith-Waterman algorithm with bit-level parallelization using the the BPBC technique. We implemented the proposed BPBC technique for the Smith-Waterman algorithm on the GPU and evaluated the performance. Experimental results show that the GPU implementation runs 8.54 times faster than the multi-core CPU implementation.**

***Index Terms*—Smith-Waterman, GPU, CUDA, bitwise operations, bulk computation**

I. はじめに

バイオインフォマティクスの分野において，未知の DNA 塩基配列やタンパク質のアミノ酸配列の性質を既知の配列から推定することは，基本的かつ重要な手法である．性質を推定する方法の 1 つとして，既知の配列と未知の配列の類似度を調べる方法がある．類似度を調べたのち，もし類似度が高ければ未知の配列が既知の配列と類似した性質をもっている可能性が高いと考える手法である．この類似度を計算する手法として，主に Smith-Waterman Algorithm(SWA) [1] が用いられている．

SWA とはローカルアラインメントを正確に求めるアルゴリズムである，ここでローカルアラインメントとはアラインメントの一種であり，グローバルアラインメントとローカルアラインメントに分類される．アラインメントとは DNA やタンパク質の類似領域を特定できるように並べたものであり，配列間の類似性を示す．グローバルアラインメントとは 2 つの配列全体の対応関係を調べるもので，ほぼ同じ長さの配列間での比較において有効である．一方ローカルアラインメントとは 2 つのシーケンス間の部分的な対応関係を調べるもので，長さの異なる配列間によく用いられる．2 つの配列長をそれぞれ m, n で表すと，時間計算量は $O(mn)$ となる．SWA は類似度計算とバックトラックの 2 つの処理で構成される．ローカルアラインメントを求めるためにはバックトラック処理が必須であるが、本論文では主に類似度に焦点を当てているため，バックトラックに関しては言及しない．

GPU とはグラフィックス処理を行うハードウェアであり，画像・映像のモニタディスプレイ出力，3D グラフィックス処理などを行っている．これらの演算機能をグラフィックス処理に限定せず汎用的に活用する GPGPU と呼ばれる技術があり，これらに関する技術が盛んに行なわれている．

II. Bitwise Parallel Bulk Computation 手法

本実装で用いる Bitwise Parallel Bulk Computation (BPBC) 手法 [3] について説明する．

A. *BPBC* 手法について

BPBC 手法とは，bitwise 論理演算を用いて 32 個の入力ベクトルに対する組合せ回路を同時にシミュレートし，バルク計算を高速化する手法である．ここで bitwise 論理演算とは 32-bit ワードの各 bit に対する論理 OR, AND, XOR, NOT 演算を同時に実行する命令であり，この bitwise 論理演算を用いて大量の入力 instance を同時に計算することが可能である．BPBC 手法のアイディアとして以下の 2 つが挙げられる．

1) 各入力 instance のデータ bit をデータ word の対応する bit にそれぞれ格納
2) 32-bit bitwise 論理演算を用いて，同時に 32 個の入力ベクトルに対する組合せ回路をシミュレート

図 1 は 32 入力の 32 個の全加算器を bitwise 論理演算を用いることで BPBC 手法を適用する例である．図 1 において X, A, B は 32-bit word であり，図中の回路 1 つ 1 つが全加算器を表している．それぞれの全加算器への 3 つの入力 bit は 32-bit word X, A, B の対応する bit にそれぞれ格納される．入力を格納した 3 つの word に対して全加算器をシミュレートする bitwise 論理演算を実行することで出力が得られ，32-bit word C, S の対応する bit にそれぞれ格納される．このように，BPBC 手法を適用することで，32 個のインスタンスが同時に実行できることがわかる．

B. *bit* 転置

BPBC 手法を用いる場合，入力を bit 転置形式に変換する必要がある．bit 転置形式とは，32 個の入力を bit 毎にビットパックして 32-bit word に格納した形式のことである．図 2 には 32 個の 2-bit の入力に対する bit 転置の例を示す．

bit 転置形式に変換する手法として，bit 毎に交換していく手法は自明であるが，非常に計算コストがかかる．しかし，[5] の 2.7 節で紹介されているビット行列の転置手法を用いることで，非常に効率的に bit 転置形式に変換することが出来る．基本的な考え方は，スワップ操作を正方形のブロックに対して複数回実行していくことである．

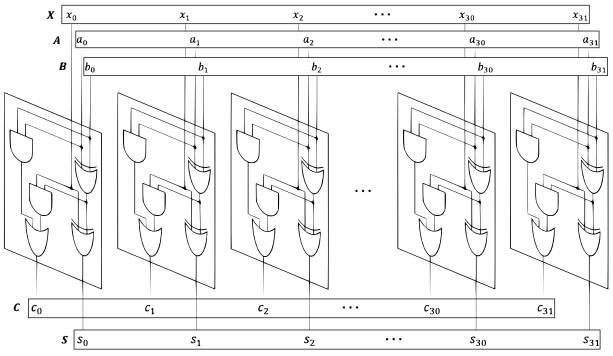

Fig. 1. 全加算器に対する BPBC 手法の適用

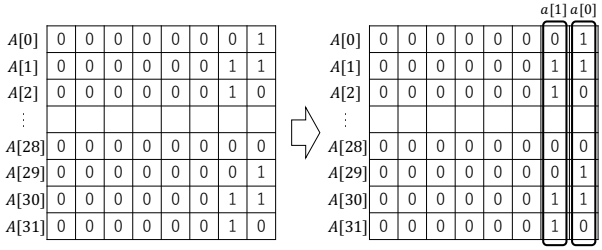

Fig. 2. 2-bit の入力に対する bit 転置の例

本研究ではこの手法を 32 個の 32-bit word によって構成される 32×32 の行列に対して適用し，bit 転置を試みた．対象を DNA シーケンスとしており，DNA は 4 文字のみ用いるため，入力は 2-bit word となる．また bit 転置形式の word に対して bit 転置と逆の操作を順番に行うことで，元の形式に戻すことが可能である．

スワップ操作を用いた bit 転置操作を GPU を用いて高速化していたが，GPU を用いた bit 転置処理のさらなる高速化を試みた．GPU には ballot 関数と呼ばれる関数が存在する．ballot 関数とは，同一 warp 内の各 thread の論理演算結果を 32-bit word の対応する bit 位置に格納し，1 つの 32-bit word として返すことが出来る関数である．ballot 関数をうまく用いることで bit 転置に必要な命令数を削減し，bit 転置をさらに高速化することが可能である．本研究では入力を 2bit としているため，32 個の入力文字列に対して 1warp を使用し，ballot 関数命令を 2 回実行するだけで bit 転置と同様の結果を得ることが出来る．図 3 に本手法を用いた例を示している．2-bit の入力 $A[0], A[1], \ldots, A[31]$ に対して 2 回 ballot 関数命令を実行することで bit 転置形式に変換できていることがわかる．

また表 I には SWAP と ballot 関数のそれぞれを用いた場合の bit 転置の実行時間比較を行っている．実験には NVIDIA Tesla V100 を用いた．表 I に示す通り，bit 転置手法を SWAP を用いるものから ballot 関数を用いるものに変更することで最大 2.11 倍の高速化となった．

III. Smith-Waterman Algorithm

Smith-Waterman Algorithm(SWA) [1] について説明する．シーケンスアラインメントとはバイオインフォマティクスの分野において生物の進化の過程の解析や DNA や RNA，タンパク質の構造情報の分析のために広く用いられている．

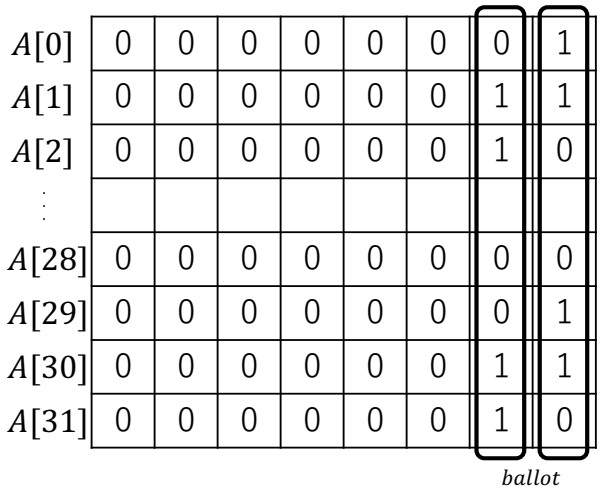

Fig. 3. ballot 関数を用いた bit 転置

SWA は動的計画法を用いてローカルアラインメントを計算する．Edit Distance [4] などで求められる，文字列全体をマッチさせるグローバルアラインメントと異なり，ローカルアラインメントは一致度合いが非常に高い部分文字列を特定したものである．よって，ローカルアラインメントはバイオ系のアプリケーションに好んで用いられる．

SWA ではギャップペナルティ W_k を用いて計算が行われる．ギャップペナルティには線形ギャップペナルティ，アフィンギャップペナルティとそれぞれ呼ばれる 2 種類が存在する．線形ギャップペナルティは $W_k = kd$ で計算され，k はギャップ長，d はギャップコストを表す．線形ギャップペナルティはギャップ長に比例した値になる．対してアフィンギャップペナルティは $W_k = u(k-1)$ で計算され，k はギャップ長，u はギャップ伸長ペナルティ，v はギャップ開始ペナルティを表す．通常 $v > u$ といった関係にある．例えばギャップ長が 5 の場合，線形ギャップペナルティを用いると $W_k = 5d$，アフィンギャップペナルティを用いると $W_k = v + 4u$ となる．SWA においてギャップペナルティに線形ギャップペナルティを用いた場合，ギャップとして欠失する文字数に比例したペナルティが課されるが，これは生物の進化の観点では不適切なスコアとされることが多い．ギャップを挿入することは配列が切断されることを意味するが，その間に挿入される配列の長さは配列の切断と比較すると大して重要な意味を持たないといわれているからである．ギャップペナルティにアフィンギャップペナルティを用いることで，より生物の進化の過程に即した計算を行うことが出来る．そこで本研究ではアフィンギャップペナルティを用いた SWA を実装した．

A. アフィンギャップペナルティを用いた *SWA*

SWA について詳しく述べていく．入力として文字列長 m の入力文字列 $X = x_1 x_2 \cdots x_m$ と文字列長 n のパターン文字列 $Y = y_1 y_2 \cdots y_m$ が与えられたとき，SWA は 2 つのシーケンスの要素の全てのペアの類似度を計算する．類似度の計算のため，SWA はサイズ $(m+1) \times (n+1)$ のスコア行列 d を使用する．スコア行列中の要素 $d[0][j] (0 \leq j \leq n)$ と $d[i][0] (0 \leq i \leq m)$ は 0 で初期化される．スコア行列 d 中の

TABLE I
SWAP と BALLOT のそれぞれを用いた BIT 転置の速度比較

$Num\ of\ Input =$	1024	2048	4096	8192	16384	32768	65536
SWAP[ms]	0.20	0.36	0.67	1.24	2.49	4.79	9.76
ballot[ms]	0.18	0.20	0.40	0.75	1.39	2.75	4.62
SWAP/ballot	1.11	1.80	1.67	1.65	1.79	1.74	2.11

各要素 $d[i][j](0 < i \leq m, 0 < j \leq n)$ は以下の式を繰り返し適用することで計算される．

$$d[i][j] = \max \begin{cases} 0 \\ p[i-1][j] \\ q[i][j-1] \\ d[i-1][j-1] + w(x_i, y_j) \end{cases}$$

$d[i][j]$ は部分文字列 X と Y の類似度を示している．またここで $w(x_i, y_j)$ は $match/mismatch$ コストを示しており，$p[i][j], q[i][j]$ は位置 (i, j) がそれぞれパターン文字列方向と入力文字列方向に必ずギャップを挿入した場合のスコアを示している．p, q 中の要素 $p[0][j](0 \leq j \leq n)$ と $p[i][0](0 \leq i \leq m)$,$q[0][j](0 \leq j \leq n)$ と $q[i][0](0 \leq i \leq m)$ は 0 で初期化される．$w(x_i, y_j), p[i][j], q[i][j]$ はそれぞれ以下の式で定義される．

$$w(x_i, y_j) = \begin{cases} c_1 & \text{if } x_i = y_j \\ -c_2 & \text{if } x_i \neq y_j \end{cases}$$

$$p[i][j] = \max \begin{cases} d[i-1][j] - \alpha \\ p[i-1][j] - \beta \end{cases}$$

$$q[i][j] = \max \begin{cases} d[i][j-1] - \alpha \\ q[i][j-1] - \beta \end{cases}$$

$p[i][j], q[i][j]$ の更新式は，それまで続いていたギャップを伸長するか，新たに位置 (i, j) からギャップを展開するか決定することを意味しており，2 式の内スコアが小さいほうで決定される．更新式において，α はギャップ展開ペナルティを，β はギャップ伸長ペナルティをそれぞれ意味する．こうして $p[i][j], q[i][j]$ には文字列の先頭からギャップを挿入し続けた場合のスコアを記録しておき，このスコアを $d[i][j]$ と比較することでスコア行列を計算していく．

スコア行列は非負の値のみとなっており，シーケンス間の一部の領域が異なる場合でも，うまくローカルアラインメントを抽出できるようになっている．上記の式を再帰的に用いることで，スコア行列 d の全ての値は以下の逐次アルゴリズムによって計算される．

[Sequential algorithm for the SWA]
for $j \leftarrow 0$ to n do
  $d[0][j] \leftarrow 0, p[0][j] \leftarrow 0, q[0][j] \leftarrow 0$
for $i \leftarrow 0$ to m do
  $d[i][0] \leftarrow 0, p[i][0] \leftarrow 0, q[i][0] \leftarrow 0$
for $j \leftarrow 1$ to n do
  for $i \leftarrow 1$ to m do
    $p[i][j] \leftarrow \max(d[i][j-1] - \alpha, p[i][j-1] - \beta$
    $q[i][j] \leftarrow \max(d[i-1][j] - \alpha, q[i-1][j] - \beta$
    $d[i][j] \leftarrow \max(0, p[i][j], q[i][j]$
                $d[i-1][j-1] + w(x_i, y_j))$
Function $w(x_i, y_j)$
  if $(x_i \neq y_i)$ return c_2 else return c_1

SWA ではスコア行列が完成した後，スコア行列中の最大の類似度を持つセルからトレースバックを行い，最適ローカルアラインメントと呼ばれるもっとも類似した部分文字列を抽出する．本研究では SWA のスコア行列を計算する部分に研究対象を限定し，複数のインスタンスに対してスコア行列計算を実行することで，閾値以上の類似度を持つインスタンスを抽出できるような実装を行っている．このようなふるいにかける計算を BPBC 手法を用いて計算し，抽出後のインスタンスは CPU を用いてきめ細かくトレースバックまで計算することを想定している．

SWA のスコア行列計算には依存関係が存在し，一つのセルを計算するのためには，左，左上，上の 3 つのセルを参照する必要がある．しかし，図 4 に示すように，斜めに並んだセルには依存関係が存在しないため，並列に計算することが可能である．

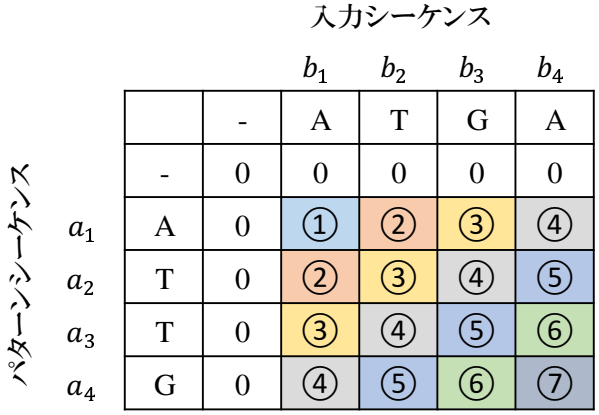

Fig. 4. 並列計算可能なスコア行列中のセル

並列計算の考え方は，全ての行を左から右に並列に計算していくことである．$d[i][j]$ は $d[i-1][j]$ の計算が終わった後に計算される．こうすることで，スコア行列の左上から右下に向かって並列計算が可能になる．

IV. SWA への BPBC 手法の適用

BPBC 手法を用いた SWA について説明する．本実装は DNA シーケンスを対象としており，DNA は A, C, G, T の 4 種類の文字から構成されるため，入力は 2-bit で表現される．BPBC 手法を用いた SWA のアルゴリズムを以下に示す．

[SWA using BPBC technique]
for $i \leftarrow 1$ to m do
  for $j \leftarrow 1$ to n do
    $p[i][j] \leftarrow \max(d[i][j-1] - \alpha, p[i][j-1] - \beta$

TABLE II
SWA の実行時間比較

入力シーケンス長	CPU[ms]	既存 GPU[ms]	提案 GPU[ms]	CPU / 提案 GPU	既存 GPU / 提案 GPU
1024	128.76	24.44	9.91	8.20	2.47
2048	245.29	41.96	20.49	8.54	2.05
4096	480.02	71.80	41.41	8.52	1.73
8192	771.17	136.54	81.33	6.84	1.68
16384	1274.71	264.19	162.70	5.74	1.62
32768	2677.11	513.32	321.63	6.01	1.60
65536	4919.16	950.09	639.08	5.39	1.47

$q[i][j] \leftarrow \max(d[i-1][j]-\alpha, q[i-1][j]-\beta$
$temp \leftarrow \max(p[i][j], q[i][j])$
$d[i][j] \leftarrow d[i-1][j-1]+w(x_i, y_j)$
$d[i][j] \leftarrow \max(d[i][j], temp)$

BPBC 手法を用いた SWA は 5 ステップで実行される．上記のアルゴリズムにおいて 3 行目と 4 行目では，入力シーケンス方向またはパターンシーケンス方向に必ずギャップを挿入した場合のスコアを格納する $p[i][j]$ と $q[i][j]$ の値を決定し，格納している．5 行目は $p[i][j]$ と $q[i][j]$ の大小を比較し，大きいほうを変数 $temp$ に格納している．6 行目では $d[i][j]$ に更新セルの左斜め上の値である $d[i-1][j-1]$ に $w(x_i, y_j)$ を加算した値を格納している．ここで $w(x_i, y_j)$ とは，文字 x_i と y_j を比較し，一致していれば $+match$，不一致であれば $-mismatch$ となる関数である．そして 7 行目で $d[i][j]$ と $temp$ の大小を比較し，大きいほうを最終結果として $d[i][j]$ に格納している．

V. GPU 実装

既存 GPU 実装及び BPBC 手法を用いた SWA の提案 GPU 実装について説明する．どちらの実装も SWA に用いるシーケンスデータは全てホスト PC に保存されているものとする．どちらの実装も，各インスタンス中の類似度の最大値を計算するものである．

A. 既存 GPU 実装

既存 GPU 実装 [1] ではまず，ホスト PC から入力データを転送する．その後，インスタンスの 1 行に 1-thread を割り当て，1 インスタンスを 1-block に割り当てることで並列化している．よって総起動 block 数は総インスタンス数と等しい．各 thread は入力シーケンスを char4 型や int4 型といったベクトル型を用いてデータをフェッチしており，またフェッチするデータは連続アドレスに格納されているので，コアレスドアクセスでかつアクセス回数を削減し，メモリアクセスにかかる時間を削減している．各 thread は自分の担当する行の計算が終わり次第，レジスタに格納している最大の類似度ととシェアードメモリを用いて共有されているインスタンス全体の最大の類似度の大小を比較し，大きい方をシェアードメモリに書き込む．このようにして，各インスタンスの最大の類似度を計算している．最後に計算結果をホスト PC に転送して処理は完了となる．

B. 提案 GPU 実装

提案 GPU 実装は 5 つのステップで実行される．
Step 1: 全ての入力データを wordwise 形式のままホスト PC からグローバルメモリに転送.
Step 2: グローバルメモリ上の入力データを ballot 関数を用いてビット転置形式に変換．変換後のデータはグローバルメモリに格納.
Step 3: BPBC 手法を用いて SWA を計算．計算結果をビット転置形式でグローバルメモリに格納.
Step 4: ビット転置形式で格納されている結果を通常の wordwise 形式に逆変換.
Step 5: wordwise 形式の計算結果をグローバルメモリからホスト PC に転送.
Step2 および Step4 の転置部分では 1-block あたり 1024-thread を起動して処理している．BPBC 手法を用いた SWA には 1-block あたりパターンシーケンス長分の thread を起動し，32 インスタンスを担当させている．

VI. 性能評価

評価実験では，GPU には NIVDIA Tesla V100 (5120 cores, 1.455GHz) を，CPU には Intel Xeon CPU E7-8870 v4 (20 cores, 3.000GHz)4 機をそれぞれ使用した．パターンシーケンス長は 128，インスタンス数は 32768 で固定とした．各シーケンスは CPU 上で乱数を用いて生成した．CPU 実装は 160thread を用いた OpenMP 実装となっている．CPU 実装，既存 GPU 実装，提案 GPU 実装を比較した実験結果を表 II に示す．表 II より，OpenMP を用いた CPU 実装と比較して最大 8.54 倍，既存 GPU 実装と比較して最大 2.47 倍の高速化を達成した．

VII. まとめ

本研究では Bitwise Parallel Bulk Computation 手法を用いた SWA を GPU 上に実装し，大量のインスタンスに対して高速化を行った．実験の結果，OpenMP を用いた CPU 実装と比較して最大 8.54 倍，既存 GPU 実装と比較して最大 2.47 倍の高速化を達成した．

REFERENCES

[1] Munekawa, Yuma, I. N. O. Fumihiko, and Kenichi Hagihara. "Accelerating Smith-Waterman algorithm for biological database search on CUDA-compatible GPUs." IEICE transactions on information and systems 93.6, pp.1479-1488, 2010.
[2] NVIDIA,"CUDA C Programming Guide Version 9.1.8", 2018
[3] Toru Fujita, Koji Nakano, Yasuaki Ito, "Bitwise Parallel Bulk Computation on the GPU, with Application to the CKY Parsing for Context-Free Grammars", Proc. of International Parallel and Distributed Processing Symposium Workshops, pp.589-598, May, 2016.
[4] Chacón, Alejandro, et al. "Thread-cooperative, bit-parallel computation of levenshtein distance on GPU." Proceedings of the 28th ACM international conference on Supercomputing. ACM, 2014.
[5] H. S.Warren, "Hacker's Delight (2nd Edition)." Addison-Wesley, 2012.