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

2008－ARC－177 (3)
2008－HPC－114 (3)
2008／3／5

統合開発環境CUDAを用いたGPUでの配列アライメントの高速化手法

宗川裕馬[†] 伊野文彦[††] 萩原兼一[††]

本稿では，生物学分野における配列アライメント処理の高速化を目的として，CUDAを用いた手法を提案する．提案手法はSmith-Watermanアルゴリズムを対象とする．高速な共有メモリ上で大部分の計算を処理することにより，低速なビデオメモリ・マルチプロセッサ間のデータ転送量を削減する．さらに，問い合わせ配列および対象配列をベクトル化することにより，演算回数を削減する．実験では，OpenGL実装と比較した．結果，アミノ酸配列データベースに対して，OpenGL実装と比べて最大で6.4倍の速度向上を達成した．このとき，ビデオメモリ・マルチプロセッサ間のデータ転送量を1/139に削減できた．

A Fast method for Sequence Alignment Using the CUDA-equipped GPU

Yuma Munekawa,[†] Fumihiko Ino[††] and Kenichi Hagihara[††]

In this paper, we propose a CUDA-based method for accelerating biological sequence alignment. The proposed method is based on the Smith-Waterman algorithm. It reduces the amount of data transfer between multiprocessors and video memory by performing computation mainly on fast shared memory. Furthermore, we also reduce the number of operations by applying vectorization to query and database sequences. We show some experimental results comparing the proposed method with an OpenGL-based method. As a result, the speedup over the OpenGL-based method reaches a factor of 6.4 when using amino acid sequence database. We also find that the amount of data transfer is reduced to 1/139.

1. はじめに

GPU[1] (Graphics Processing Unit) とは，コンピュータシステムにおけるグラフィクス処理に特化した半導体チップである．近年，CPUを超える浮動小数点演算性能に着目し，GPUの並列アーキテクチャを汎用計算（GPGPU : General Purpose Computation on GPUs）に応用する研究が盛んである．また，nVIDIA社よりCUDA[2] (Compute Unified Device Architecture) と呼ばれるGPGPUを対象とした統合開発環境が提供されており，プログラマビリティも向上している．

一方，配列アライメントとは，2本以上の配列を入力として，それらの類似性を判定する操作である．この操作はDNAの塩基配列やたんぱく質のアミノ酸配列を対象として，生物学分野で広く用いられている．特に，2本の配列間の類似性を判定する操作をペアワイズアライメントと呼び，動的計画法による解法としてSW (Smith-Waterman) アルゴリズム[3]が存在する．

配列アライメントは，実用上は1本の問い合わせ配列とデータベース（DB）に格納した多数の対象配列を入力として，各組み合わせに対するペアワイズアライメントを繰り返す．このDBは年率1.5～2倍の速度で増大しているため，高速化の要求がある．SWアルゴリズムをGPU上で高速化する試みとしては，OpenGL[4]を用いた実装[5]が存在する．しかし，CUDAを用いた研究は我々の知る限り存在しない．

本稿では，SWアルゴリズムをCUDAを用いて高速化する手法を提案する．共有メモリや文字型のベクトルテクスチャなどのOpenGLで用いることのできない機能により，低速なビデオメモリ・マルチプロセッサ間のデータ転送量と演算回数を削減する．

以降では，2節でCUDAの概要を説明し，3節でSWアルゴリズムについて述べる．その後，4節で我々の実装を説明し，5節で実験結果を示す．最後に，6節で本稿をまとめる．

2. 統合開発環境CUDA

CUDAはGPUにおける汎用計算のためのプログ

† 大阪大学基礎工学部情報科学科
Department of Information and Computer Sciences, School of Engineering Science, Osaka University
†† 大阪大学大学院情報科学研究科コンピュータサイエンス専攻
Department of Computer Science, Graduate School of Information Science and Technology, Osaka University

ラミングインタフェースであり，C 言語の拡張としてコードを記述できる．

CUDA のハードウェアモデルでは，GPU は M 個のマルチプロセッサ（MP）を持ち，さらに各 MP 内に S 個の SIMD 型ストリームプロセッサ（SP）が存在する．メモリ階層については，すべての MP から読み書き可能なグローバルメモリと，すべての MP から読み出し可能で，キャッシュを持つテクスチャメモリおよび定数メモリが存在する．テクスチャメモリはアクセスが近傍に集中するときに，レジスタ読み出しに近い性能を発揮する．一方，定数メモリはアクセスが 1 点に集中するときに同様の性能を発揮し，テクスチャメモリよりも小容量である．なお，これら 3 種類のメモリデータは GPU のビデオメモリに確保されるので，書き込みおよびキャッシュが効かない読み出しについては 400〜600 クロックサイクルの遅延が生じる．各 MP には専用の共有メモリとレジスタが存在する．共有メモリは同一ブロック内のすべてのスレッドから参照でき，読み書きがレジスタ並みに高速である．ただし，1MP 当たりの容量は前述の 3 種類のメモリよりも少なく，16KB である．

CUDA では，GPU を多数のスレッドを並列実行するためのコプロセッサとみなす．スレッドを複数個集めたものをブロックと呼び，各 MP へはブロック単位で仕事が割り当てられる．同一ブロック内のスレッドは，MP 内の S 個の SP で並列実行される．CUDA ではブロック間の同期機能は存在しない．しかし，同一ブロック内のスレッドは同期でき，すべてのスレッドのメモリアクセスのタイミングを揃える場合に利用できる．メモリの読み書きが原因で，あるブロックの実行が待ち状態になった場合，MP は他のブロックの実行に移る．しかし，この複数のブロックの並行実行は，1 ブロックあたりの共有メモリ使用量およびレジスタ消費量に制限を受ける．

3. Smith-Waterman アルゴリズム

Smith-Waterman アルゴリズムはペアワイズアライメントを実行するアルゴリズムであり，2 本の配列を入力とし，それらの配列中で最も類似度の高い部分配列の組を動的計画法により求める．長さが n の配列 Q および長さ m の配列 S を，$Q = q_1 q_2 \ldots q_n$ および $S = s_1 s_2 \ldots s_m$ と定義する．Q の位置 i までの部分配列 $q_1 q_2 \ldots q_i$ および S の位置 j までの部分配列 $s_1 s_2 \ldots s_j$ 間の類似度を $H_{i,j}$ と表す（$1 \leq i \leq n$ かつ $1 \leq j \leq m$）．$H_{i,j}$ は以下の式で定まる．

		T	C	T	C	G	A	T
	0	0	0	0	0	0	0	0
G	0	0	0	0	0	2	1	0
T	0	**2**	1	2	1	1	1	3
C	0	1	**4**	3	4	3	2	2
T	0	2	3	**6**	5	4	3	4
A	0	1	2	**5**	5	4	6	5
C	0	0	3	4	**7**	6	5	5

-------：類似度の最も高い部分配列

図 1 スコア行列とトレースバック

$$H_{i,j} = max\{0, E_{i,j}, F_{i,j}, H_{i-1,j-1} + sbt(q_i, s_j)\} \quad (1)$$

ここで，

$$E_{i,j} = max\{H_{i,j-1} - \alpha, E_{i,j-1} - \beta\} \quad (2)$$

$$F_{i,j} = max\{H_{i-1,j} - \alpha, F_{i-1,j} - \beta\} \quad (3)$$

である．任意の $0 \leq k \leq n$ および $0 \leq l \leq m$ に対し，$H_{k,0} = H_{0,l} = E_{k,0} = F_{0,l} = 0$ と初期化する．$sbt(a, b)$ は文字 a を文字 b に置換するためのコストを表す．α および β は配列長を合わせるためのギャップペナルティである．$\alpha \neq \beta$ をアフィンギャップペナルティと呼び，$\alpha = \beta$ をリニアギャップペナルティと呼ぶ．ギャップペナルティがリニアギャップペナルティのとき，式 (1) は以下のように簡略化できる．

$$H_{i,j} = max\{0, H_{i,j-1} - \alpha, H_{i-1,j} - \alpha, H_{(i-1,j-1} + sbt(q_i, s_j)\} \quad (4)$$

なお，本研究では $\alpha = \beta = 1$ のリニアギャップペナルティを用い，置換については a および b が一致するとき $sbt(a, b) = 2$，そうでないとき $sbt(a, b) = -1$ とした．

式 (4) に基づき，すべての i および j について $H_{i,j}$ を計算すれば，**図 1** のようにスコア行列 H を得る．求める部分配列の組は，0 から H 内の最大値 $H_{i,j}$ を導出するまでの過程を逆向きに辿る（トレースバックする）ことで定まる．

本研究の対象問題は，1 本の問合せ配列を入力として，DB 内の D 本すべての対象配列とのペアワイズアライメントを実行することである．したがって，1 本の問合せ配列 Q に対し，DB を走査し，対象配列 S を変えながら最大の類似度を与える部分配列の組を探索することになる．ゆえに，本研究ではスコア行列の最大値のみ算出することとする．この場合，トレースバックは DB 内で最大（あるいは上位 10 件など）の $H_{i,j}$ を返す対象配列 S に対してのみ行えばよい．トレースバックはスコア行列 H の生成に比べて実行回数が十分に少なく，CPU で実行できる[5]．

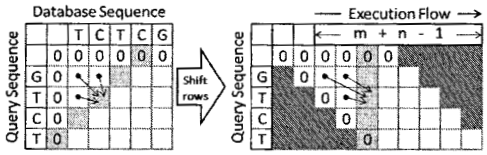

図 2　スコア行列のデータ依存およびその変換

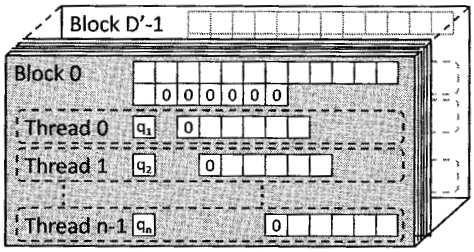

図 3　提案手法のブロック分割およびスレッド分割

4. 提案手法

説明を容易にするために，スカラ版の基礎的な実装を述べたのちに，提案手法の核となる共有メモリの使用およびデータのベクトル化について述べる．

4.1 処理の分割

図 2 に，スコア行列計算におけるデータ依存関係を示す．式 (4) より，要素 $H_{i,j}$ の計算は，隣接する左の要素 $H_{i,j-1}$，上の要素 $H_{i-1,j}$，および左上の要素 $H_{i-1,j-1}$ に依存する．したがって，対角線上に存在する要素は独立であり，これらを並列処理できる．ただし，t 番目の対角線の計算には，$t-1$ 番目および $t-2$ 番目の対角線が必要である．提案手法では，対角線上の要素が同一列に並ぶようスコア行列を変換し (図 2)，左の列から順に計算する．対象配列長および問い合わせ配列長を各々 m および n とすれば，計算すべき対角線の数は $m+n-1$ 本である．

図 3 に，提案手法におけるブロック分割およびスレッド分割を示す．DB 内の D 本の対象配列のうち，D' 本ごとを GPU に転送しスコア行列の生成を繰り返す．このとき，提案手法はスコア行列を D' 個ずつ生成する．スコア行列間にはデータ依存がないため，提案手法では 1 ブロックが 1 本の対象配列に対するスコア行列を計算する．一方，スコア行列内に着目すると，前述の通り同一対角線上の要素は独立に計算できる．したがって，ブロック内のスレッドは，スコア行列の 1 行を担当する．このとき，1 ブロック当たりのスレッド数は n 個である．

なお，計算すべき対角線の数 $m+n-1$ は，m に依存してブロック（対象配列）ごとに異なる．したがって，ブロックごとの m を実行開始時に読み出す必要がある．ブロック内のすべてのスレッドが同じ値を読み出すため，提案手法では，この情報を定数メモリに格納する．ただし，定数メモリの容量に制限があるため，D' は 8192 としている．

表 1　スレッドごとの処理の概要

入力：問い合わせ配列テクスチャ $QueryTex$，
　　対象配列テクスチャ $SubjectTex$
出力：スコア行列の最大値

```
1.  idx1 := threadID + 1;
2.  idx2 := threadID;
3.  que := QueryTex[idx1];
    // m + n − 1 本の対角線を計算
4.  for t := 0 to m + n − 2
5.    idx3 := t − threadID;
      // 計算の有無を判定
6.    if (idx3 < 0||idx3 > m) ; // 計算なし
7.    else // 計算あり
8.      sub := SubjectTex[idx3][blockID];
        // sbt の計算
9.      if (que == sub)sbt := 2;
10.     else sbt := −1;
        end if
        // 対角線の計算
11.     H := max(max(max(0, Dia2[idx2] + sbt),
          Dia1[idx2] − 1), Dia1[idx1] − 1);
        // 最大値の更新
12.     MAX := max(MAX, H);
        // 次のループのための対角線の格納
13.     Dia2[idx1] := Dia1[idx1];
14.     Dia1[idx1] := H;
      end if
      // スレッドの同期
15.   __syncthreads();
    end for
    // 最大値をグローバルメモリに格納
16. result[blockID * width + threadID] := MAX;
```

4.2 共有メモリ上でのスコア行列の計算

式 (4) より，スコア行列内の要素を計算するために，その隣接要素を 3 つ読み出す必要がある．したがって，グローバルメモリよりも高速な共有メモリにスコア行列を格納すれば，実装を高速化できる可能性がある．また，第 i 行を担当するスレッドは第 $i-1$ 行と第 i 行の要素を読み出す必要があるため，スレッド間で計算結果を共有できれば，読み出し部分の効率を向上できる．なお，共有メモリの容量は小さいが，スコア行列全体を保持する必要がないため，問題にならない．具体的には，$t-1$ 番目および $t-2$ 番目の対角線を保持すれば，t 番目の対角線を更新できる．

表 1 に，スレッドごとの処理を疑似コードで示す．このコードでは，ブロックおよびスレッドの識別子を各々 $blockID$ $(0 \leq blockID < D')$ および threadID

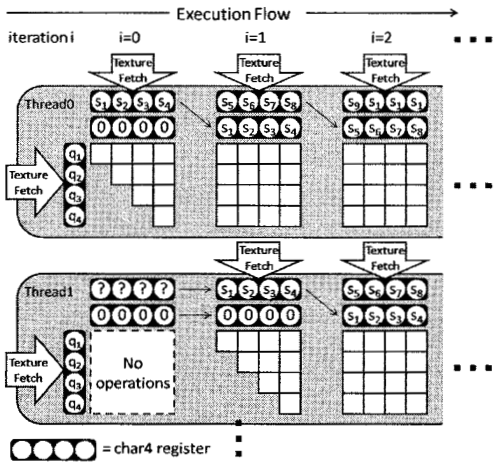

図 4　ベクトル化を施したのちの計算の様子

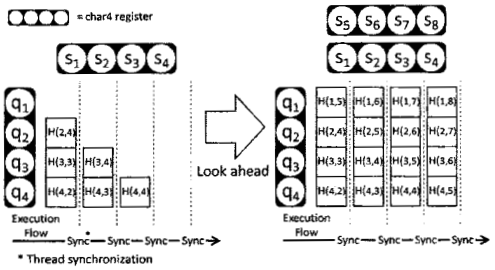

図 5　対象配列の先読みによる並列性の向上

$(0 \leq threadID < n)$ としている．また，問い合わせ配列を 1 次元テクスチャに格納し，D' 本の対象配列を 2 次元テクスチャに格納している．4 行目の t ループは，計算対象となる $m+n-1$ 本の対角線を走査している．ただし，すべてのスレッドがスコア行列を必ず計算するわけではない．図 2 の塗りつぶし部分の計算は不要であるため，6〜7 行目で計算の有無を判定している．その後，11 行目で対角線の計算を行う．ここで，Dia1[] および Dia2[] は共有メモリ上に確保した配列であり，それぞれ $t-1$ 番目および $t-2$ 番目の対角線を保持している．15 行目では，ブロック内のスレッドを同期し，対角線ごとにスコアの計算が進むように制御している．最後に，すべての対角線を計算し終えたのちに，スレッドごとに計算した最大値をグローバルメモリへ格納する（16 行目）．

一般に，SIMD アーキテクチャでは分岐命令が性能を低下させる．この疑似コードでは，6 行目に分岐命令が存在するが，この命令では大半のスレッドが計算があるものとして 8 行目以降に分岐する．したがって，分岐命令に起因するオーバヘッドは小さいと考える．

4.3　ベクトル化

データ転送量および演算回数の削減を目的として，文字型データのベクトル化を行う．具体的には，問い合わせ配列および対象配列を char4 型のベクトルテクスチャに格納し，ベクトルごとに読み出す．

図 4 に，ベクトル化を施したのちの計算の様子を示す．スカラ版では各スレッドがスコア行列の 1 行を担当するのに対し，ベクトル版では一連の 4 行をまとめて担当する．これにより，ベクトル版では 1 ブロックあたりのスレッド数は $\lceil n/4 \rceil$ 個となる．さらに，対象配列もベクトル単位で読み出すため，各スレッドはループ 1 回の実行でスコア行列の 4×4 要素を計算する．したがって，一度に 4 本の対角線を計算することになり，ループの繰り返し回数は $m+n-1$ 回から $\lceil (m+n-1)/4 \rceil$ 回に減少する．

なお，対象配列の読み出し先として，スレッドごとに char4 型のレジスタを 2 つ用いる（**図 5**）．2 倍のレジスタを用いることにより，対象配列を先読みでき，データ転送量を削減できる．もしレジスタをベクトル 1 個分しか用意しなければ，図 5 に示すように，4 本の対角線上において計算できる要素が下三角形行列の部分に限定されてしまい，出力先が計算の進行とともに減少してしまう．この場合，並列性も低下してしまう．これらを防ぐためには，対角線上の要素が欠けないように計算すればよく，これは対象配列に対してベクトル 1 個分を先読みすれば実現できる．具体的には，片方のレジスタを先読み専用とし，次のループに移る前にもう一方へコピーしておけば，各スレッドは常に 8 個の要素を保持でき，4×4 要素を計算できる．

最後に，ベクトル化によるデータ転送量の削減効果について述べる．スカラ版では，スコア行列の 1 要素を計算するために，対象配列の要素を 1 つ読み出す必要がある．一方，ベクトル版では，スコア行列の 4×4 要素を計算するために，平均で対象配列の 4 要素を読み出せばよい．したがって，スカラ版と比較してテクスチャの転送量を 1/4 に削減できる．また，1 ブロック当たりのスレッド数が約 1/4 となるため，表 1 における 6 行目の分岐命令や 15 行目の同期命令も 1/4 の実行で済む．

なお，各スレッドが初めて 4×4 要素の計算を実行するとき，下三角形行列部分の 6 要素は計算する必要がない．しかし，この制御は煩雑となるため，先読み用のレジスタを 0 で初期化しておくことにより，不要な部分の計算結果が 0 になるようにしている．

表 2　実験環境

CPU	Intel Xeon E5440 2.83GHz
メインメモリ	8GB
GPU	nVIDIA GeForce 8800 GTX
ビデオメモリ	768MB
OS	Windows XP Professional x64 Edition SP2
ビデオドライバ	169.09

表 3　データベースの概要

データベース名	UniProtKB/Swiss-Prot protein knowledgebase
エントリ数 D（個）	250,143
アミノ酸塩基数（個）	90,588,910
1 エントリあたりの長さ m の平均	362
ファイルサイズ（MB）	121

表 4　各実装の実行時間の比較（単位：秒）

Query Length n	SSEARCH	OpenGL		CUDA	
	T	T'	T	T'	T
128	84	6.0	17.9	1.5	3.4
256	159	11.1	28.1	2.8	4.7
384	253	16.9	40.2	4.8	6.7
512	318	22.9	52.2	6.3	8.2
640	428	30.7	66.3	9.5	11.5
768	507	39.4	81.4	11.3	13.3
896	598	49.2	97.5	17.4	19.4
1024	677	59.4	113.9	19.7	21.7

5. 評価実験

開発した実装の性能を評価するために，対象配列 DB にアミノ酸配列 DB を用いて配列アライメントを**表 2** にまとめる計算機上で実行した．

5.1　実験の手順

表 3 に，実験に用いたアミノ酸塩基配列 DB の情報を示す．この DB のエントリ数 D はおよそ 25 万件であり，1 エントリ当たりの長さ m の平均は 362 である．DB のファイルサイズは 121MB である．一方，問合せ配列はアミノ酸塩基配列とし，長さ 128 から 1024 の 8 本の配列を用意した．問い合わせ配列 1 本のファイルサイズは数 KB 程度である．

比較のための実装として，SW アルゴリズムの OpenGL 実装および CPU 実装 SSEARCH[6] を用いた．OpenGL 実装は，スコア行列をテクスチャ上で計算する点で提案手法と異なる．CPU 実装は，発見的手法を用いて高速化を施している．これにより実行時間は半減するが，スコア行列の要素のうち，数%に誤りが発生する．

5.2　実行時間の比較

表 4 に，問い合わせ配列長 n を 128 から 1024 まで変えて計測した各実装の実行時間 T を示す．T は問い合わせ配列と対象配列 DB のファイル処理から，データの GPU への転送，配列アライメントの実行，計算結果のリードバックまでを含んだ総実行時間である．T' は総実行時間のうち，GPU 上で行う配列アライメントの実行に要する GPU 実行時間である．

提案手法は，OpenGL 実装と比べて T' で 2.8〜4.0 倍，T で 5.0〜6.4 倍の速度向上を達成している．前者の速度向上は，提案手法が OpenGL 実装のテクスチャよりも高速な，共有メモリを用いてスコア行列を計算するためである．また，T ではさらに高い速度向上を得ていることから，CPU 側の実行時間も短縮できている．提案手法では API の呼び出しやテクスチャの切り替えによるオーバーヘッドがないことがその理由である．

一方，提案手法は SSEARCH に比べて T で 24.7〜38.8 倍高速である．GPU を使用する場合，CPU・GPU 間のデータ転送が性能のボトルネックとなるケースがある．しかし提案手法では転送時間が 400 ミリ秒程度と計算時間 T' に対して短いので，総実行時間 T での高速化に繋がっている．

5.3　CUDA 実装の性能

提案手法におけるベクトル化の効果を検証する．その後，開発した実装が GPU の性能をどれだけ引き出しているかについて調べる．ベクトル化の有無に関して 2 種類の実装を用意した．これらと OpenGL 実装を用い，ビデオメモリ・マルチプロセッサ間のデータ転送量 A およびその実効バンド幅 B を計測した（**表 5**）．

CUDA 実装のデータ転送量 A は，スカラ版においても OpenGL 実装の約 1/35 に削減できている．これは OpenGL 実装でのテクスチャの読み書きが CUDA 実装では共有メモリの読み書きへと置き換わったためである．ベクトル版では，データ転送量 A がさらに 1/4 に減少し，OpenGL 実装と比べると 1/139 に削減できている．

一方，バンド幅 B は OpenGL 実装が最大で 69.7GB/s を記録したのに対して，スカラ版の CUDA 実装は最大で 5.5GB/s にとどまる．しかし，GPU 実行時間はスカラ版が OpenGL 実装の 3 倍高速である．このことから，データ転送量 A を約 1/35 に削減したことで，性能ボトルネックがデータ転送から計算部分へと移ったと考えられる．同様に，ベクトル版のバンド幅 B は最大で 2.0GB/s であり，スカラ版よりも小さいが，GPU 実行時間はベクトル版のほうが短い．

表 5　各 GPU 実装の GPU 実行時間，データ転送量および実効バンド幅

Query Length n	GPU Time T' (s)			Data Transfer A (GB)			Bandwidth B (GB/s)		
	CUDA		OpenGL	CUDA		OpenGL	CUDA		OpenGL
	ベクトル	スカラ		ベクトル	スカラ		ベクトル	スカラ	
128	1.5	2.0	6.0	2.8	11.1	389	1.9	5.5	64.8
256	2.8	4.4	11.1	5.5	22.1	777	2.0	5.1	69.7
384	4.8	7.3	16.9	8.3	33.2	1117	1.7	4.5	66.1
512	6.3	13.4	22.9	11.0	44.3	1554	1.8	3.3	67.8

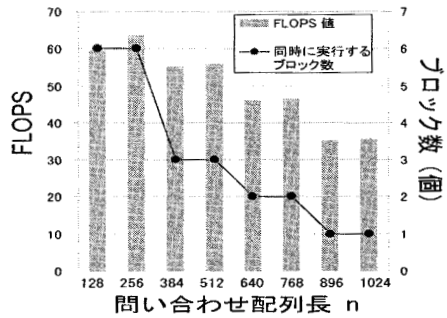

図 6　FLOPS と同時実行するブロック数の関係

これはベクトル化によりデータ転送量 A が 1/4 になったことでバンド幅 B が減少し，実行する演算数が減少したことで GPU 実行時間 T' が短くなった，と説明できる．

ベクトル版の CUDA 実装について，SW アルゴリズムの実行に必要な演算数から FLOPS 値を計算した．**図 6** に結果を示す．実効性能の最大値は 63.4GFLOPS と，GPU の MP の理論性能 345.6GFLOPS の 18%程度である．GPU の性能を出し切れていない理由は，理論性能が 1 クロックに積和の 2 演算ができることを前提にしている一方，配列アライメントでは積和演算をほとんど実行しないことと，if 文などの実行制御のための命令の実行回数が多いためである．実際に，所要クロック数換算でアルゴリズムの実行に必要な命令の 1.2 倍程度の制御命令が存在している．

図 6 に示すように，問い合わせ配列長が長くなるにつれて実効性能が階段状に減少している．これは問い合わせ配列長とともに 1 ブロック当たりの共有メモリ使用量およびレジスタ消費量が増加し、1MP で同時に処理できるブロック数が減少するためである。実際に，1MP で同時に実行できるブロック数は，FLOPS 値とともに減少している．

6. ま と め

本稿では，配列アライメントを GPU 上で高速化することを目的として，SW アルゴリズムの CUDA 実装を示した．開発した実装では，高速な共有メモリ上で大部分の計算を行うほか，問い合わせ配列および対象配列をベクトル化することにより高速化を図る．

実験の結果，配列数約 25 万のアミノ酸配列 DB の走査において，最大で OpenGL 実装の 6.4 倍高速であった．

今後の課題は，長さが 2048 を超えるような問い合わせ配列に対する配列アライメントである．

謝辞　本研究の一部は，科学研究費補助金基盤研究 (B) (2) (18300009)，若手研究 (B) (19700061) および大阪大学グローバル COE プログラム「予測医学基盤」の補助による．

参 考 文 献

1) Owens, J.D. and et al.: A Survey of General-Purpose Computation on Graphics Hardware, *Computer Graphics Forum*, Vol.26, No.1, pp. 80–113 (2007).
2) nVIDIA Corporation: CUDA Programming Guide Version 1.1 (2007). `http://developer.nvidia.com/cuda/`.
3) Smith, T.F. and Waterman, M.S.: Identification of Common Molecular Subsequences, *J. Molecular Biology*, Vol.147, pp.195–197 (1981).
4) Shreiner, D., Woo, M., Neider, J. and Davis, T.: *OpenGL Programming Guide*, Addison-Wesley, Reading, MA, fifth edition (2005).
5) Liu, W., Schmidt, B., Voss, G. and Müller-Wittig, W.: GPU-ClustalW: Using Graphics Hardware to Accelerate Multiple Sequence Alignment, *Proc. 13th Int'l Conf. High Performance Computing (HiPC'06)*, pp. 363–374 (2006).
6) Pearson, W.R.: Searching protein sequence libraries: Comparison of the sensitivity and selectivity of the Smith-Waterman and FASTA algorithms, *Genomics*, Vol.11, No.3, pp.635–650 (1991).