2007－HPC－111 (13)
2007／8／1

GPUグリッドによる高速な塩基配列アライメント

伊野文彦[†] 小谷裕基[†,☆] 萩原兼一[†]

本稿における GPU (Graphics Processing Unit) グリッドとは，家庭やオフィスにおける遊休資源を活用する計算グリッドの一形態である．その特長は，GPU を CPU と同様の汎用計算資源として提供する点にある．本稿では，GPU グリッドで高速に動作するアプリケーションの一例として，アミノ酸塩基配列データベースに対する配列アライメントの実装を示す．この実装は，単体 GPU 上で動作する Liu らのアルゴリズムを基にしている．マスタ・ワーカ方式を用い，GPU グリッド上でデータファイルに対する走査を並列処理し高速化する．実験では，単体 GPU 版，単体 CPU 版および CPU グリッド版と比較した．結果，8 台からなるグリッドにおいて，GPU グリッド版は CPU グリッド版に対して 5 倍以上高速であった．このときの性能は，描画部分に限れば 1 台の GPU で 47GFLOPS であり，8 台の CPU に匹敵する性能を引き出せた．

Fast Biological Sequence Alignment on the GPU Grid

Fumihiko Ino,[†] Yuki Kotani[†,☆] and Kenichi Hagihara[†]

The graphics processing unit (GPU) grid in this paper is a computational grid that utilizes idle resources in the home and office. One advantage is that it provides both GPUs and CPUs as general-purpose computational resources. This paper presents an implementation capable of accelerating alignment for biological sequences in database. Our implementation is based on Liu's algorithm that runs on a single GPU. It employs a master/worker paradigm to accelerate database scanning on GPU grids. We show some experimental results comparing the implementation with various versions running on a single GPU, a single CPU, or multiple CPUs. As a result, we find that our implementation is at least five times faster than the multiple CPU version. Furthermore, the kernel speed of our implementation reaches 47 GFLOPS on a single GPU, which is equivalent to the performance obtained on eight CPUs.

1. はじめに

GPU[1] (Graphics Processing Unit) とは，グラフィクス処理の高速化を目的としたチップである．GPU は，急速に性能を向上していて，CPU を超える演算性能を単精度で提供する．また，グラフィクスの知識を必要としない開発環境 CUDA[2] が nVIDIA 社の GPU で利用でき，プログラマビリティも向上している．

一方，グリッド技術はネットワークを通して，異なる組織に存在する複数の計算資源を 1 つの仮想的な高性能計算機に統合できる．従来のグリッドでは，GPU は画面表示のための部品に過ぎなかったが，汎用の計算資源として用いる試み[3)~5)] がある．

例えば，Folding@Home プロジェクト[3] は遊休 CPU および GPU を用い，蛋白質の折り畳みシミュレーションを高速化する．実効性能として 20 万台の CPU がおよそ 200TFLOPS を提供する一方で，900 台の GPU が 50TFLOPS を達成している．ただし，このシステムは GPU が遊休であるか否かを監視しない．したがって，遊休でない GPU へジョブが投入されることがあり，文献 4) 同様，著しい性能低下を引き起こし得る．これらに対し，我々は GPU を監視し，性能低下が生じない GPU グリッド[5] を開発してきた．

本稿では，GPU グリッド上で高速に動作できるアプリケーションを示すことを目的として，塩基配列に対するアライメント[6] の実装を示す．アライメントとは，複数の配列を入力として，配列要素間の最適な対応関係を求める処理である．生物学分野において重要な操作であり，配列の類似性判定に応用できる．

開発した実装は，単体の GPU 上で動作する Liu らのアルゴリズム[7] を基に，GPU グリッド上でアミノ酸塩基配列データベース (DB) を並列に走査する．この際，2 本の配列に対するペアワイズアライメントを

† 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
☆ 現在，JR 東海
Presently with Central Japan Railway Company

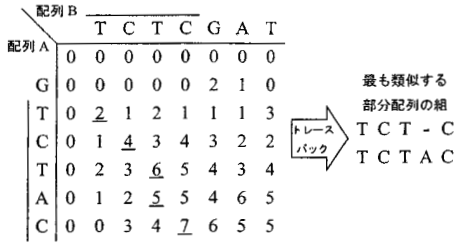

図 1 Smith-Waterman アルゴリズムの実行例

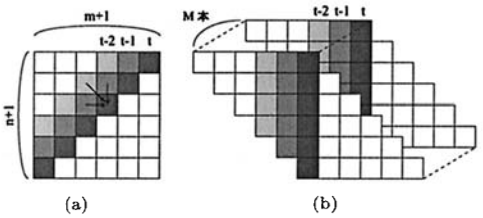

図 2 スコア行列生成時のデータ依存関係および GPU に適したスコア行列の変形

繰り返す．アライメントには，最も類似性の高い部分配列の組を返す Smith-Waterman（SW）アルゴリズム[6] を用いた．

以降では，2 節で SW アルゴリズムについて述べ，3 節で Liu らのアルゴリズムを示す．その後，4 節で GPU グリッドの概要および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. Smith-Waterman アルゴリズム

2 本の配列 $A = a_1a_2 \ldots a_n$ および $B = b_1b_2 \ldots b_m$ が与えられたとする．アライメントとは，配列要素の挿入および削除を繰り返し，A を B に変換することを指す．変換に要する挿入および削除の回数をコストとみなし，このコストに基づいて配列間の類似度を表す．特に，最も類似する部分配列の組を抜き出すことを局所アライメントと言う．

SW アルゴリズム[6] は，動的計画法に基づいて局所アライメントの厳密解を返す．a_i および b_j で終端する部分配列 $a_1a_2 \ldots a_i$ および $b_1b_2 \ldots b_j$ 間の最大類似度を $H_{i,j}$ とする（$1 \leq i \leq n$ かつ $1 \leq j \leq m$）．このとき，$H_{i,j}$ は以下のように定まる．

$$H_{i,j} = max\{H_{i-1,j-1} + s(a_i, b_j), E_{i,j}, F_{i,j}, 0\} \quad (1)$$

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．また，$s(a,b)$ は文字 a を文字 b へ置換するためのコストを表し，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．さらに，任意の $0 \leq k \leq n$ および $0 \leq l \leq m$ に対し，$H_{k,0} = H_{0,l} = E_{k,0} = F_{0,l} = 0$ である．なお，本研究でのギャップペナルティは $\alpha = \beta = 1$ とし，置換については a および b が一致するとき $s(a,b) = 2$，そうでないとき $s(a,b) = -1$ とした．

式 (1) 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** に示すスコア行列 H を得る．局所アライメントの解（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．

なお，本研究の対象問題は，N 本の問合せ配列を入力として，問合せ配列ごとに DB 内の M 本すべての対象配列との局所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 A に対し，DB を走査し対象配列 B 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．この場合，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 B に対してのみトレースバックすればよい．ゆえに，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[7]．

3. 単体 GPU によるアライメント

本節では，Liu らのアルゴリズム[7] について述べる．

図 2(a) に，スコア行列生成時におけるデータ依存関係を示す．式 (1) 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．しかし，t 番目の対角線は，$t-1$ 番目および $t-2$ 番目の対角線を必要とする．したがって，平面に対して直線上のデータにしか並列性がなく，効率が悪い．

そこで，対象問題が M 本の配列とのアライメントであることに着目し，このアルゴリズムは M 個の局所アライメントを同時に処理する．つまり，図 2(b) に示すように，置行き方向に DB 内の対象配列を並べ，M 個のスコア行列の対角線を並列計算する．この場合，t 番目の対角線をすべて保持するために $(n+1) \times M$ 画素のテクスチャを用意すればよい．さらに，$t-1$ 番目と $t-2$ 番目の対角線も同じサイズのテクスチャに保持し，これらの入出力を順に切り替えれば，テクスチャは 3 枚で済む．このように，t をインクリメントしながら描画を $n+m-1$ 回ほど繰り返せば，1 本の問合せ配列に対するすべてのスコア行列を計算できる．

さらに，テクスチャは画素ごとに RGBA チャネル

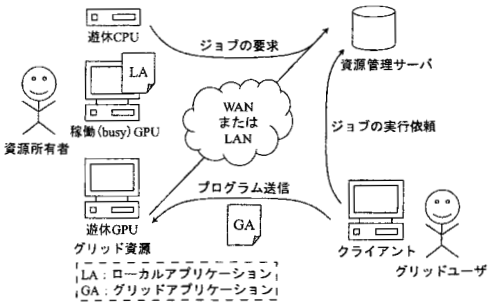

図 3 GPU グリッドの構成

を持つことから，ベクトル演算を用いて 4 種類の値を 1 度に書き込める．具体的には，$H_{i,j}$，$E_{i,j}$，$F_{i,j}$ および $H_{i,j}$ の最大値を各チャネルへ書き込む．

4. GPU グリッドによるアライメント

図 3 に，GPU グリッドの構成を示す．GPU グリッドは，GPU を明示的に汎用の計算資源として扱う点を除き，既存のデスクトップグリッドと同じ構成である．主に 3 つの構成要素からなる．

- グリッド資源．グリッド資源はインターネットに接続しているデスクトップ PC である．通常は，資源所有者がローカルアプリケーションを実行するが，遊休状態のときにグリッドアプリケーションを実行する．GPU の有無に関わらず，任意の PC はグリッド資源になり得る．なお，遊休状態の判定は，いくつかのセンサを備えるスクリーンセーバ[5] に基づく．スクリーンセーバが起動するまでの待ち時間は 5 分である．
- 資源管理サーバ．資源管理サーバは登録済のグリッド資源を監視および選択する．また，グリッドユーザからのジョブの実行要求を受け付ける．要求に対しては，資源とのマッチメイキング[8] を行い，適切な資源にジョブを割り当てる．
- クライアント．クライアントはグリッドアプリケーションを実行したいグリッドユーザのためのフロントエンドである．また，グリッド資源にもなり得る．

図 4 に，GPU グリッドにおけるアライメントの流れを示す．現在の実装はマスタ・ワーカ方式に基づいて，アライメントを並列処理する．その際，1 台の計算資源に 1 本の問合せ配列 A を割り当て，DB の走査を実行する．この走査により，問合せ配列 A および M 本の対象配列に対する局所アライメントが完了する．したがって，N 本の問合せ配列に対して N 個のタスクが存在し，その各々を繰り返し計算資源に割

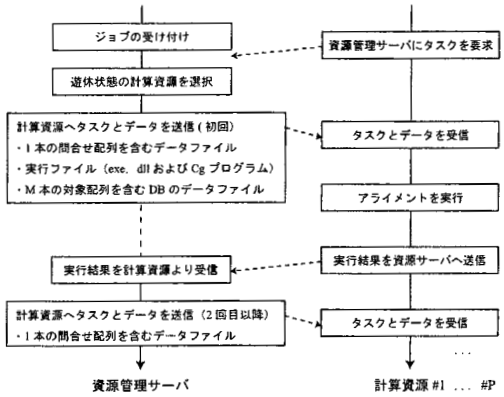

図 4 GPU グリッドでのアライメントの流れ

り当てて並列実行することになる．また，任意のタスクを独立に処理できるため，計算資源間でデータをやりとりする必要はない．このように，このアプリケーションはパラメータスイープ型であり，GPU グリッドに限らず CPU グリッドにも適している．

各計算資源に転送するデータは，実行に必要なプログラム一式および入力ファイル一式である．前者は，CPU 上で動作する実行ファイル，各種ライブラリおよび GPU 上で動作するプログラムである．後者は，M 本の対象配列を含むデータファイルおよび 1 本の問合せ配列を含むデータファイルである．ただし，これらの一部は一度転送すれば，再送の必要はない．したがって，2 回目以降のタスク実行には，問合せ配列を含むデータファイルのみを転送すればよく，転送量を抑えることができる．

実装には，C++言語，OpenGL ライブラリ[9] および Cg 言語[10] を用いた．また，スコア行列を格納するためのバッファには FBO[11]（Frame Buffer Object）を用いた．なお，前世代の GPU では，扱える 2 次元テクスチャの大きさが 4096×4096 画素以下である．したがって，本実装は 1 本の問合せ配列および 4096 本の対象配列に対するアライメントを一回の描画で処理する．M が 4096 本を超える場合，4096 本ごとのアライメントを繰り返す．なお，最新の G80 世代では 4096×4096 画素を超えるテクスチャも扱える．

5. 評価実験

開発した実装を評価するために，GPU グリッドを用いてアライメントを実行し，その性能を検証する．

5.1 実験の手順

表 1 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 M はおよそ 24 万

表 2　実験環境

計算資源	R1	R2	R3	R4	R5	R6	R7	R8
CPU	Pentium 4	Xeon	Pentium 4	Pentium 4	Xeon	Core2 Duo	Pentium D	Core Duo
周波数 (GHz)	3.4	2.8	2.8	2.8	3.8	2.4	3.0	2.0
メインメモリ (GB)	2	4	2	1	4	2	2	2
GPU	GeForce 8800 GTX				GeForce 8800 GTS	GeForce 7950 GX2	GeForce 7900 GTX	GeForce Go 7900 GTX
VRAM 容量 (MB)	768				640	512	512	512
コアクロック (MHz)	575				500	500	650	500
メモリクロック (MHz)	1800				1600	1200	1600	1200
VRAM バンド幅 (GB/s)	86.4				64.0	76.8	51.2	38.4
フィルレート (Gpixel/s)	36.8				24	24	15.6	12
OS	Windows XP							
ネットワーク	100 Mb/s Ethernet							

表 1　データベースの情報

データベース名	UniProtKB/Swiss-Prot protein knowledgebase
リリース	51.0
エントリ数 M (個)	241,242
アミノ酸塩基数 (個)	88,541,632
1 エントリあたりの長さ m の平均	367
ファイルサイズ (MB)	118

件であり，1 エントリ当たりの長さ m は平均で 367 である．エントリ全体のファイルサイズは 118MB である．一方，問合せ配列はアミノ酸塩基配列とし，その数 N および長さ n は各々64 個および 367 である．これら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，CPU 上で動作する SSEARCH[12] を用いた．この実装は，我々の実装と同じ SW アルゴリズムに基づいている．なお，SSEARCH を基に拡張命令 SSE2 を用いて高速化した報告[13] がある．この SSE2 版は SSEARCH に対して概ね 3 倍ほど高速である．

表 2 に，実験に用いた計算機の仕様を示す．最新の GPU を搭載した PC を 5 台（資源 R1〜R5）用い，残り 3 台（資源 R6〜R8）は前世代の GPU を搭載している．これらは 100Mb/s の LAN 環境で接続されている．なお，資源 R8 はノート PC である．これらの PC を用い，GPU グリッド版，CPU グリッド版，単体 GPU 版および単体 CPU 版の性能を比較する．

5.2　専有環境での評価

まず，ユーザが計算資源を専有できる環境を想定して各実装を評価する．この環境は，休日や夜間あるいはクラスタなどのように，資源所有者およびユーザ間における資源競合が発生しない状況に対応する．

図 5 に，実装ごとの実行時間を示す．ここで，実行時間とは，問合せ配列や DB エントリの転送からすべての走査が完了し計算結果を回収するまでの時間を指す．したがって，118MB のファイル転送や VRAM・主記憶間の転送に要する時間を含んでいる．また，単体 GPU 版および単体 CPU 版の結果は，最短の実行時間を返した資源 R1 および R6 のものである．

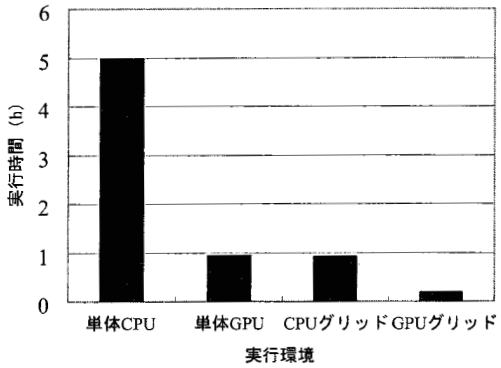

図 5　アライメントに要する実行時間

8 台からなる GPU グリッドは，単体 CPU が約 5 時間（17,920 秒）を要するアライメントを 1/26 の約 11 分（673 秒）に短縮できている．一方，単体 GPU は約 57 分（3392 秒）で処理を終えていて，この性能は 8 台の CPU グリッドにおける約 56 分（3345 秒）とほぼ同等である．さらに，速度向上の観点では，GPU 版および CPU 版のいずれも 8 台の投入に対して 5 倍程度の速度向上を達成している．

表 3 に，走査 1 回当たりの実行時間の内訳を示す．資源 R8 のノート PC でさえ，CPU 版で最速の資源 R6 の 280 秒よりも短い 88 秒で 1 回の走査を終えている．また，GPU 版の計算時間のうち GPU が描画に要した時間は，資源 R1〜R4 において全体の 30%程度（18 秒）である．したがって，残りの 70%は CPU 側の処理に要していて，この部分の短縮が課題である．この問題は，描画領域の大きさに対してテクスチャの

表 3　走査 1 回当たりの実行時間の内訳（左欄が GPU 版で右欄が CPU 版）

項目		R1		R2		R3		R4		R5		R6		R7		R8	
転送時間（秒）	初回	21		16		15		16		15		15		15		15	
	以降	0.02		0.01		0.01		0.01		0.01		0.01		0.01		0.01	
計算時間（秒）	全体	53	385	59	458	58	462	59	463	61	338	86	280	80	428	88	339
	GPU	18	—	18	—	18	—	18	—	25	—	62	—	48	—	60	—
実効性能*（GFLOPS）		47	2.2	47	1.9	47	1.9	47	1.9	35	2.5	14	3.1	19	2.0	15	2.5
実効バンド幅*（GB/s）		87	4.2	88	3.5	87	3.5	87	3.5	65	4.7	26	5.7	36	3.7	27	4.7
走査回数		10	8	8	7	8	6	9	7	8	9	7	11	7	7	7	9

*：CPU 側の処理を含めないカーネル部分から算出

切り替え回数が多すぎることに起因している．

次に，転送時間を調べると，初回の走査のみ 15〜21 秒もの時間を 118MB のファイル転送に要している．しかし，それ以降は数 KB の転送で済み，転送時間は 20 ミリ秒程度である．一般に，アライメント対象となる配列の組み合わせは多いため，初回に転送した M 本の対象配列は，以降のアライメントで再利用できる可能性が高い．なお，今回のネットワーク環境における転送性能は実効で 45〜63Mb/s であり，この性能は広域分散環境では期待できない．仮に，転送性能が 3Mb/s に低下すれば，転送時間に 5 分強を要する．この転送時間に見合う問合せ配列の数 N（もしくは長さ n）が必要であり，これらを調節してタスクを構成すべきである．

全体で 64 回の走査に対して 8 台を用いることから，各資源は 8 回の走査を担当すれば，偏りはない．しかし，表 3 において各資源が走査した回数に着目すると，6〜11 回の範囲で CPU 版および GPU 版ともに偏りが生じている．この理由は，すべての資源に初回のデータを送り終える前に，初回の走査を完了する資源があるためである．また，資源そのものが異種であることも理由の 1 つである．例えば，前世代の資源 R6〜R8 は R1〜R5 よりも約 1.5 倍ほど長い計算時間を要している．一般に，GPU の性能向上は著しく，世代間の性能差が CPU よりも大きいことは GPU グリッドで注意すべき特徴である．

次に，GPU の実効性能および実効バンド幅を調べる．ここで，これらの値は GPU の描画時間から算出している．したがって，CPU 側の処理を含めた全体の計算時間に基づく性能ではないことに注意されたい．表 3 より，資源 R1〜R4 は 47GFLOPS を達成している．一方，これらが持つ GPU の理論性能は 518GFLOPS であるため，描画部分に限定しても効率は 10%に達していない．この理由は，アライメントの性能ボトルネックが VRAM にあるためである．実際に，実効バンド幅を表 2 の理論バンド幅と比較すると，資源 R1〜R5 は VRAM のバンド幅をほぼ使い切っている．したがって，実行効率を向上するためには，転送に対する計算の比率を高める必要がある．なお，DB に対する走査はパラメータスイープ型のアプリケーションであることから，上記の結果は，最新の GPU を 22 個用いれば 1TFLOPS を達成できる可能性があることを示している．

表 3 は，CPU 性能が GPU 版の計算時間を決めることも示している．例えば，資源 R1 を R6 と比較すると，R1 は R6 よりもカーネル部分の実効性能が 3.4 倍ほど高いが，全体の計算時間ではこの倍率が 1.6 倍に低下している．この理由は，資源 R6 の CPU 性能が R1 よりも高いためである．例えば，CPU 版の性能比が 385 対 280 であるのに対し，GPU 版における CPU 側の計算時間は 35 対 24 であり，これらの比率はほぼ一致する．したがって，GPU アプリケーションと言えども，GPU グリッドでは CPU 性能の高い資源を選択することも大切である．

5.3　共有環境での評価

資源所有者およびグリッドユーザが計算資源を共有する場合，すなわち本来の環境での性能について検証する．計算資源として R1，R5，R6 および R7 の 4 台を用い，5 日間にわたって実験した．なお，資源所有者は大学院生であり，GPU に関する研究を遂行している．また，各資源は 1 日当たり約 8 時間ほどオンライン状態になり，夜間や休日などのオフライン期間は観測対象から除外した．各資源における走査は資源が遊休状態になってから 1 秒以内に始まる．

日中 8 時間あたりの走査回数は，平均で 202 回であった．その内訳は，資源 R1 が 62 回，R5 が 30 回，R6 が 71 回および R7 が 39 回である．これらの処理は，単体 CPU（資源 R6）で約 16 時間弱を要することから，専有できる PC を 2 台ほど省けたことになる．

検出した遊休状態のうち 40%ほどが 5 分未満である．平均では，12〜20 分ほど遊休状態が持続していた．一方，GPU 版はアライメントを 1〜2 分で終えている．したがって，1 本の問合せ配列を 1 つのタスクとする割り当ては，今回の環境では適切なタスクサイ

ズであると考える．一方，CPU グリッド版をそのまま使うことは得策ではない．なぜなら，大半の資源が5分以上の計算時間を要していて，遊休期間内に完了するタスクが少ないためである．したがって，対象配列を分割してタスクサイズを削減する必要がある．このことは，環境が変われば GPU グリッド版にも当てはまる．しかし，タスクサイズを削減することはオーバヘッドを相対的に増大させるため，GPU グリッドの方が効率がよい．

このように，GPU グリッドは従来のグリッドよりもアプリケーションを高速化できる可能性がある．特に，本実験で用いたようなパラメータスイープ型のアプリケーションは，独立タスクの集合で構成されていることが多いため，高速化が期待できる．

6. ま と め

本稿では，GPU グリッドが効率よく高速化できるアプリケーションを提示することを目的として，アミノ酸塩基配列に対するアライメントの実装を示した．開発した実装は，Liu らのアルゴリズム[7] を基に，GPU グリッド上でデータファイルに対する走査を並列処理し高速化する．

実験の結果，8台からなるグリッドにおいて，GPU版はCPU版に対して5倍以上高速であった．このときの性能は，描画部分に限れば1台の GPU で 47GFLOPS であり，8台の CPU に匹敵する性能を引き出せた．さらに，4台の計算資源を共有する実環境においても，単体 CPU を2台ほど専有し続ける場合の性能を引き出せた．また，GPU は CPU よりも高速であることから，より多くの処理を1つのタスクに詰め込める．このように，GPU グリッドは従来のグリッドに埋もれている計算資源を活用でき，パラメータスイープ型のアプリケーションを高速化できる可能性がある．

今後の課題は，広域分散環境における GPU グリッドの構築や CUDA を用いた場合の検証である．

謝辞 本研究の一部は，科学研究費補助金若手研究 (B) (19700061) および基盤研究 (B) (2) (18300009) の補助による．

参 考 文 献

1) Owens, J.D., Luebke, D., Govindaraju, N., Harris, M., Krüger, J., Lefohn, A.E. and Purcell, T.J.: 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 0.8.2 (2007). http://developer.nvidia.com/cuda/.

3) The Folding@Home Project: Folding@Home Distributed Computing (2007). http://folding.stanford.edu/.

4) Yamagiwa, S. and Sousa, L.: Design and Implementation of a Stream-based Distributed Computing Platform using Graphics Processing Units, *Proc. 4th Int'l Conf. Computing Frontiers (CF'07)*, pp.197–204 (2007).

5) Kotani, Y., Ino, F. and Hagihara, K.: A Resource Selection Method for Cycle Stealing in the GPU Grid, *Proc. 4th Int'l Symp. Parallel and Distributed Processing and Applications Workshops (ISPA'06 Workshops)*, pp.939–950 (2006).

6) Smith, T.F. and Waterman, M.S.: Identification of Common Molecular Subsequences, *J. Molecular Biology*, Vol.147, pp.195–197 (1981).

7) 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).

8) Raman, R., Livny, M. and Solomon, M.: Resource Management through Multilateral Matchmaking, *Proc. 9th IEEE Int'l Symp. High Performance Distributed Computing (HPDC'00)*, pp.290–291 (2000).

9) Shreiner, D., Woo, M., Neider, J. and Davis, T.: *OpenGL Programming Guide*, Addison-Wesley, Reading, MA, fifth edition (2005).

10) Mark, W.R., Glanville, R.S., Akeley, K. and Kilgard, M.J.: Cg: A system for programming graphics hardware in a C-like language, *ACM Trans. Graphics*, Vol.22, No.3, pp.896–897 (2003).

11) OpenGL Extension Registry: GL_EXT_framebuffer_object (2006). http://oss.sgi.com/projects/ogl-sample/registry/EXT/framebuffer_object.txt.

12) 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).

13) Meng, X. and Chaudhary, V.: Exploiting Multi-level Parallelism for Homology Search using General Purpose Processors, *Proc. 11th Int'l Conf. Parallel and Distributed Systems (ICPADS'05), Volume II Workshops*, pp.331–335 (2005).