情報処理学会論文誌　Vol. 0　1–11 (??? 1959)

グラフィックプロセッサを用いた自己組織化マップアルゴリズムの実装と評価

設樂 明宏[†1]　西川 由理[†1]
吉見 真聡[†2]　天野 英晴[†1]

本稿では，NVIDIA 社製のグラフィックプロセッサを用いた自己組織化写像 (SOM) のアルゴリズムを実装と性能評価について述べる．プログラムには NVIDIA 社の提供する CUDA を用い，並列プログラミング，データフローの最適化よびプロファイリングを行った．評価にはグラフィックカードを 3 種類用い，ストリームプロセッサ数やメモリ容量と性能の関係について調査した．また問題 (マップ) サイズおよび次元数，学習係数などの SOM に関するパラメータを変化させ，実行速度への影響を調べた．その結果，GTX280 を用いた場合, マップサイズ 1372×1372, ベクトル数 128, 学習サイズ 128 個のとき、Intel Core 2 Quad 2.40GHz と比べて 150 倍程度の性能が示された．

Implementation and Evaluation of Self-Organizing Map Argorithm by using Graphic Processor

Akihiro Shitara,[†1] Yuri Nishikawa,[†1]
Masato Yoshimi[†2] and Hideharu Amano[†1]

In this paper, we introduce an implementation of algorithm for self-organizing map(SOM) using GPUs and discuss its evaluation. We used CUDA provided by NVIDIA Corporation for parallel programming, profiling, and data flow optimization so as to exploit inherent data-level parallelism of the algorithm. By using three NVIDIA's graphic cards for evaluation, we investigated the relationships among the number of processor elements, amount of memory device and performance. As the result of performance evaluation with various parameter combinations, we found that implementation on GTX280 achieved 150 times higher performance of Intel Core 2 Quad 2.40 GHz when parameters of map size, number of vectors and learning size were 1372×1372, 128 and 128, respectively.

1. はじめに

自己組織化マップ (Self-Organizing Maps: SOM，以降「基本 SOM」と呼ぶ) は，1989 年に Kohonen[1] が提案した教師なし学習ニューラルネットワークである．その用途はロボット制御や画像処理など幅広く，[2]．中でも近年は，データマイニング分野で注目を集めている．基本 SOM は，多次元で大量のデータから特徴を抽出して多角的に分類し，結果を可視化できることから，遺伝子情報の分析などの分野への応用も試みられている[3)4)]．

従来，基本 SOM のアルゴリズムを実行するにあたり，SIMD またはシストリックアレイ構成のクラスタが用いられてきた[5)6)]．またクラスタ計算機の運用コスト等の問題に対処するため，2000 年以降は高性能化した FPGA を用いた専用ハードウェアの実装例が登場し，汎用 CPU に対して数十から数百倍の高速化が可能であることが確認されている[7)8)]．しかし，これらの実装例は対応する問題 (マップ) のサイズが小さく，大規模の問題に対する実用性の改善が課題となっていた．

本研究では，基本 SOM のアルゴリズムが持つ高いデータレベル並列性を活かし，グラフィックプロセッサ上に実装を行ない，その性能を評価した．プログラミングには NVIDIA 社の提供する CUDA (Compute Unified Device Architecture) を用い，パイプラインレイテンシの隠蔽や，結合アクセス，分岐命令の削減等の最適化やスレッド数の最適化を行うとともに，実行時間のプロファイリングを行った．評価では，同系列でピーク性能の異なるグラフィックプロセッサを 3 種類用いた．その結果，ローエンドなグラフィックプロセッサを用いても，対応するデータサイズは大きく，高い実用性とコスト対性能比を実現した．ハイエンドな NVIDIA GeForce GTX280 を用いた場合，Core 2 Quad 2.40 GHz の約 150 倍の性能であることを確認した．

また，GPU での基本 SOM アルゴリズムの評価を元に，ゲノム解析用に改良されたバッチ学習 SOM (Batch-Learning SOM: BL-SOM) の実装をグラフィックプロセッサ上で行い，ゲノムデータを用いて実行時間のプロファイリングを行った．その結果，

†1 慶應義塾大学大学院 理工学研究科
Graduate School of Science and Technology, Keio University
†2 同志社大学 理工学部
Faculty of Science and Engineering, Doshisha University

2. 関 連 研 究

基本 SOM のハードウェアアクセラレータに関する研究が現在までに行われている．1987 年に Carpenter らにより基本 SOM のハードウェア実行に関する検討が行われた[5]．1992 年には Speckmann らが，並列に動作する 8 つの計算ユニットで構成される基本 SOM 用に設計された SIMD 型プロセッサ COKOS(COprocessor for KOhonen's Self-organizing map) を開発している[6]． 1996 年には Ruping らによって基本 SOM 用プロセッサ NBISOM25 が開発され，定量的評価が行われた[9]．

FPGA を用いた基本 SOM 用アクセラレータは，Porrmann ら，Tamukoh らによる実装がそれぞれ開発されている[7][8]．Porrmann らは，リング状に接続された 6 つの計算コアで基本 SOM を実行するシステムを開発した．Xilinx XCV812E-6 上での実装で，ベクトル次元数 9，マップサイズ最大 250 × 250 の基本 SOM を実行を確認している．Tamukoh らは，基本 SOM の学習過程において，入力ベクトルと重みベクトル間の距離をマンハッタン距離で定義することによって，演算の簡素化を図った.[8] これにより，実装した際の回路面積を削減することに成功し，16 ビット幅データで 128 次元，16×16 のサイズの基本 SOM を単一の Xilinx XC2V6000-C FPGA 上に実装した．この中で，1 つのマップに対して，距離と学習を行う回路ユニット，重みベクトル格納用のメモリを組み合わせた回路を多数並べることで，並列計算を可能にした．結果，マップサイズの変化から影響を受けない一定の時間で計算できることが示され， Intel Xeon 2.80 GHz CPU に対して約 350 倍の CUPS 値 (17500MCUPS) を達成した．

3. SOM : Self Organization Map

SOM は，多量のデータの特徴量を抽出する目的で利用されるニューラルネットワークの一種である．基本的な SOM アルゴリズムでは，競合層と呼ばれるニューロンが配置された空間が，順に入力される入力データを学習していく．入力ベクトル $\boldsymbol{x} = \{x_1, \cdots, x_v\}$ は，v 次元の数ベクトルである．各ニューロン i には，入力ベクトルと同じ v 次元の重みベクトル $\boldsymbol{m}_i = \{m_{i1}, \cdots, m_{iv}\}$ が設定される．また，学習の効果と範囲を示す近傍距離 N_c および，近傍関数 h，学習率係数 $\alpha(t)$ が設定される．全入力データの学習が T 回繰り返された後，競合層に学習結果が形成される．

3.1 基本 SOM の計算手順

基本 SOM のアルゴリズムは，以下の手順で実行される．

(1) 各パラメータおよび競合層の全ニューロンの重みベクトル $\boldsymbol{m_i}$ の値を乱数で初期化する．制御変数を設定する $t \leftarrow 0$

(2) 入力ベクトル $\boldsymbol{x}$ を与える

(3) $\boldsymbol{x}$ と各 $\boldsymbol{m}_i$ の距離 $||\boldsymbol{m}_i - \boldsymbol{x}||$ を評価し，式 (5) を満たす勝者ニューロン c を探索する

$$||\boldsymbol{m}_c - \boldsymbol{x}|| = \min_i ||\boldsymbol{m}_i - \boldsymbol{x}|| \tag{1}$$

(4) 勝者ニューロン c とその近傍 N_c 内にある全ニューロンの重みベクトルを，式 (2) を用いて更新し，学習させる．式 (2) の近傍関数 h_{ci} は学習率係数 $\alpha(t)$ を用いて式 (3) で定義される

$$\boldsymbol{m}_i' = \boldsymbol{m}_i + h_{ci}(\boldsymbol{x} - \boldsymbol{m}_i) \tag{2}$$

$$h_{ci} = \begin{cases} \alpha(t) & (i \leq N_c) \\ 0 & (i > N_c) \end{cases} \tag{3}$$

(5) 次の入力ベクトルの処理を (2) へ戻って繰り返す．全入力ベクトルの計算が終了した場合，$t < T$ ならば $t \leftarrow t+1$ の後に (2) へ戻って最初の入力ベクトルから繰り返す．$t = T$ ならば計算終了

$\mathbf{x}$ との一致値が最も大きい重みベクトル $\mathbf{m_c}$ を探し (式 (5))，その重みベクトルを持つニューロン c を勝者とする．ここで一致値とは，2 つのベクトルの類祖度を表すものであり，自己組織化マップの場合はユークリッド距離を用いる．即ち，入力ベクトルとの距離が最も小さい重みベクトルを持つようなニューロンが勝者となる．

3.2 BL-SOM のアルゴリズムの概要

基本 SOM を用いて大量の入力データを分類する場合，入力データの学習順序が学習結果に影響してしまう欠点がある．重みベクトルの値の更新は入力データごとに行われるため，より後に入力されたデータの方が最終的な学習結果に強く反映されてしまうためである．

大量のデータを解析する際の問題に対処するために，SOM の改良である BL-SOM が金谷，阿部らによって提案されている[4][10][3]．BL-SOM の最大の利点は，入力データの順序と学習結果に依存しない事である．特に我々のグループの対象とするゲノム解析の分野においては，ゲノムデータの特徴を大局的に分析するため BL-SOM を用いる事が有効である．

3.3 BL-SOM の計算手順

BL-SOM の計算手順を次に示す．ここでは，マップサイズを J 行 I 列とする．

(1) 重みベクトル$\boldsymbol{m_{ij}}$を式 (4) を用いて初期化する．

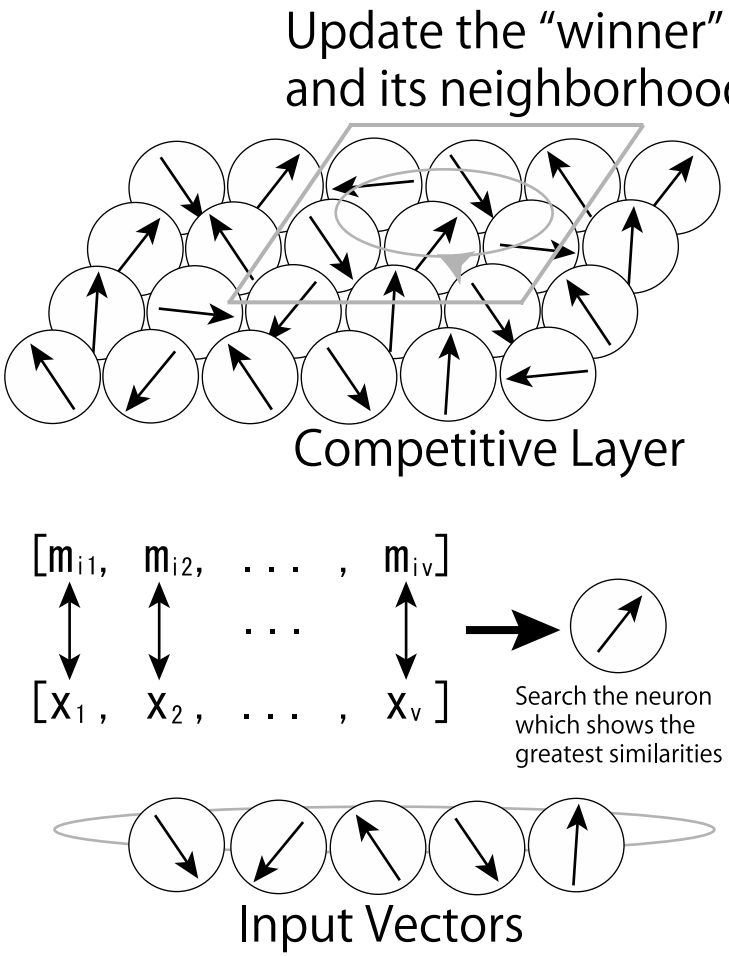


図 1　自己組織化マップ・アルゴリズムの概念図

$$\boldsymbol{m_{ij}} = \boldsymbol{x_{ave}} + \frac{5\sigma_1}{I}\left[\boldsymbol{b_1}(i - \frac{I}{2}) + \boldsymbol{b_2}(j - \frac{J}{2})\right] \tag{4}$$

ここで$\boldsymbol{x_{ave}}$は，全ての入力ベクトルを平均したベクトルで，σ_1は$\boldsymbol{x_{ave}}$の要素の標準偏差である．また，$\boldsymbol{b_1}$と$\boldsymbol{b_2}$は，入力ベクトルの主成分分析の第 1 成分と第 2 成分を表すベクトルである．

(2)　全ての重みベクトルに対して，入力ベクトル$\boldsymbol{x}$と$\boldsymbol{m_{ij}}$のユークリッド距離を計算する．

(3)　“勝者ニューロン”$\boldsymbol{m_c}$を次の式に従い探す．

$$||\boldsymbol{m_c} - \boldsymbol{x}|| = \min ||\boldsymbol{m_{ij}} - \boldsymbol{x}|| \tag{5}$$

(4)　全ての入力に対して 2 から 3 までの手順を繰り返す．

(5)　式 (6) により重みベクトルを更新する．

$$\boldsymbol{m_{ij}^{(new)}} = \boldsymbol{m_{ij}} + \alpha(t)\left(\frac{\sum\limits_{x_k \in S_{ij}} x_k}{N_{ij}} - \boldsymbol{m_{ij}}\right) \tag{6}$$

ここで集合 S_{ij} は，式 (7) と式 (8) を満たす重みベクトル$\boldsymbol{m_{i'j'}}$を勝者ニューロンとする入力ベクトルの集合である．

$$i - \beta(t) \leq i' \leq i + \beta(t) \tag{7}$$

$$j - \beta(t) \leq j' \leq j + \beta(t) \tag{8}$$

$\alpha(t)$ と $\beta(t)$ はそれぞれ，学習の t 回目の繰り返し過程における学習係数と近傍距離である．それぞれ次の式で定義される．

$$\alpha(t) = max\{0.01, \alpha(0)(1 - \frac{t}{T})\} \tag{9}$$

$$\beta(t) = max\{1, \beta(0) - t\} \tag{10}$$

(6)　2 から 5 までの手順を時間 t が T に達するまで繰り返す．

BL-SOM は式 (6) を用いて入力ベクトルの平均をとるため，入力順序が結果に影響を及ぼさない．

4. GPU アーキテクチャ

本研究では，SOM のプラットホームとなる GPU として，Geforce GTX280, 9800GTX+, および 9600GT を用いる．GTX280 は GT200 シリーズ，9800GTX+と 9600GT は G80 シリーズに大別され，特に前者は倍精度浮動小数点演算をサポートしていることが特徴である．その点を除き，両シリーズはハードウェア構成および制御方式が類似しているため，ここでは GTX 280 シリーズを例にとって解説する．

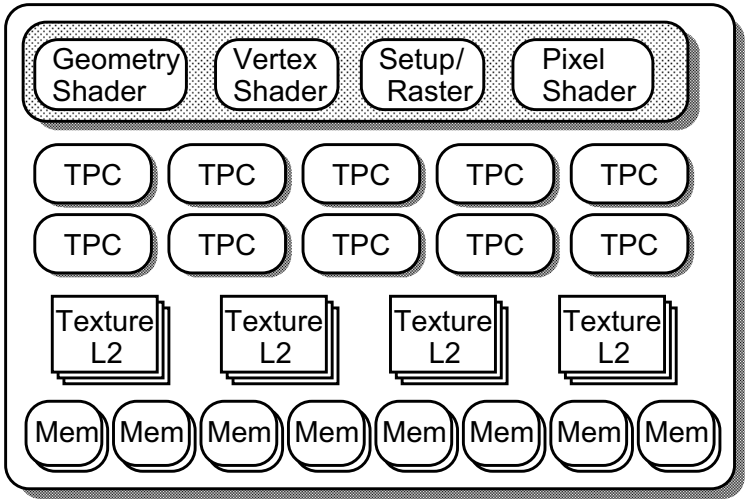


図 2　GT200 シリーズの構造

4.1　グラフィックプロセッサアーキテクチャ

4.1.1　Texture Processor Cluster (TPC)

図 2 に，NVIDIA 製グラフィックカードに搭載されている GT200 シリーズデバイスのアーキテクチャの概観を示す．また以下に主な構成を述べる．

- Texture Processor Cluster (TPC)
- スレッドスケジューラ
- デバイスメモリ
- L2 キャッシュ

図 2 における Texture Processor Cluster (TPC) は，複数のコアを持つ演算処理部である．2009 年 1 月現在，GT200 シリーズには GTX280 と GTX260 プロセッサがあり，前者は TPC を 10 個，後者は 8 個持つ．各 TPC 内部には，Texture Filtering Unit(図中の TF) および複数個の Streaming Multiprocessor(SM) を持つ．各 SM の構成を以下に述べる．

- Streaming Processor(以下，SP)(8 個): 単精度型浮動小数演算パイプライン
- Special Function Unit(SFU)(2 個): 超越関数計算用演算パイプライン
- DP(1 個): 倍精度型浮動小数演算パイプライン
- 共有メモリ: 8 個の SP で共有されるオンチップメモリ

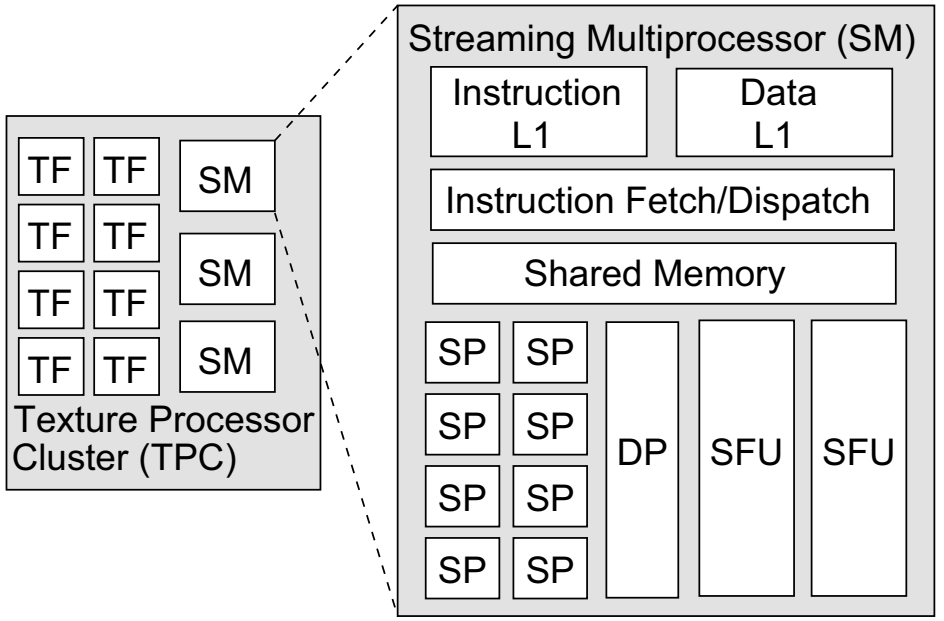


図 3　Texture Processor Cluster(TPC) の構造

- 命令およびデータ用 L1 キャッシュ
- プログラムカウンタ

4.1.2　SIMT アーキテクチャ

NVIDIA 社製の GPU の制御方式は，Flynn による 2 種類の分類にあてはめることができる．GPU アーキテクチャ全体の制御方式は，各 TPC が独自のデータを用いて独立したスレッドを実行できるため，MIMD 型 (Multiple Instruction，Multiple Data) として捉えることができる．一方，各 SM は後述する SIMT(Single Instruction，Multiple Thread) と呼ばれる方式にて制御される．

SM 内部の 8 個の SP では，プログラムカウンタが共有され，1 サイクル内に同一の命令を実行する．これは 1 サイクル単位で見れば，SIMD(Single Instruction, Multiple Data) 制御方式のように見える．しかしここで，SM が命令を発行するクロックレートは TPC の 1/3 程度である点に注意する必要がある．TPC が SM に対して 1 サイクル毎に命令を発行すると，SM 内の演算パイプライン稼働率が低下する．そこで演算効率を向上させるため，

表 1　GPU の諸元

GPU name 0	GeForce GTX280	GeForce 9800GTX+	GeForce 9600GT
マルチコア数	30	16	8
マルチコア数/TPC	3	2	2
コアクロック (GHz)	1.27	1.84	1.60
レジスタ/コア	16KB	8KB	8KB
共有メモリ/コア	16KB	16KB	16KB
メモリクロック (GHz)	1.1	1.1	0.9
メモリバス幅 (bit)	512	512	256
メモリプロセッサ間			
バンド幅 (GB/sec)	142	70.4	57.6
メモリ容量	1GB	512MB	512MB

各 SM は図 4 に示すように，同一の命令を複数サイクルに渡って実行する必要がある．例えば，あるサイクルにて SP0 がスレッド 0 を，SP1 がスレッド 1 を実行する場合，次のサイクルではそれぞれスレッド 8，スレッド 9 を実行する．この同一の命令による複数スレッドの実行を，SIMD に代わって SIMT と呼ぶ．GTX200 シリーズでは，一命令が 8 個の SP により 4 サイクル，すなわち 32 スレッド実行したとき，最も高い演算効率が得られる．この 32 スレッドの単位は warp と呼ばれる．

なお，CUDA の利点として，スレッド数を明示することなくプログラムを記述できることが挙げられる．よってプログラマは，スレッド毎の例外処理 (分岐後の振舞いなど) で異なる点があれば，それを記述するのみでよい．この特徴は，SIMD 処理の並列度を意識する Intel SSE 命令を用いた SIMD プログラミングとは大きく異なる．

4.2　高速化のテクニック

4.2.1　パイプラインレイテンシの隠蔽

Vasily らは，$a = a * b + c$ や $a = a * b + c$ のような命令レベル並列性を引き出すことの困難なレジスタ-レジスタ間演算を，ループアンローリングした状態で反復実行し，パイプラインレイテンシを見積っている．その結果，レイテンシは 24 サイクルであることがわかり，これを隠蔽するために，SM あたり最小 6 つの warp が実行されている必要があることを示した[11]．ただし，1 つのブロック 1 つの SM 内で複数のブロックが実行されることもあるため，複数のブロックをとおしてこの値を満たされればよい．この値は本研究における実装にも適用されている．

4.2.2　結合アクセス

前節にて述べたように，SM 内部の各 SP は SIMT 方式であるため，メモリアクセス命令も同一に行われる．このとき，アクセス対象のメモリ番地が連続するケースが多いため，個々ではなくまとめてアクセスすることにより高速化を図るデータ転送方式を，結合アクセスと呼ぶ．

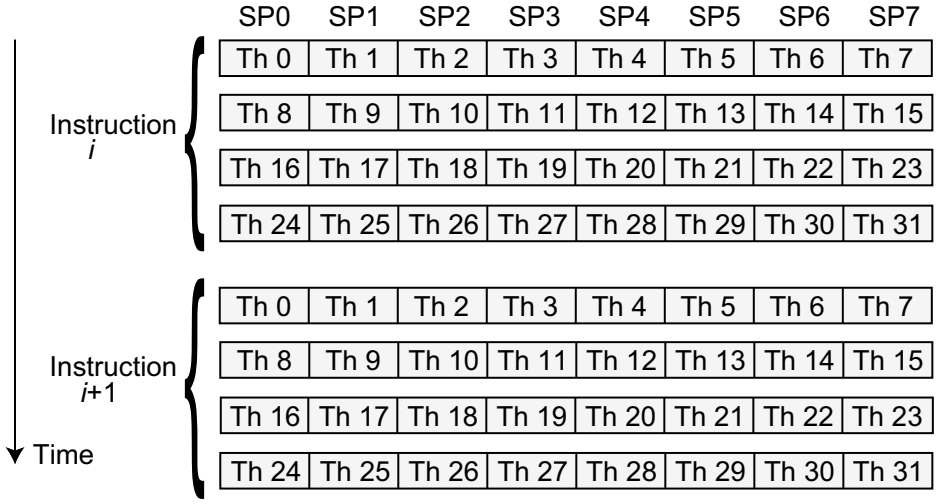


図 4　warp によるスレッドの実行

G80/GT200 シリーズでは，アライメントされた連続する 64 バイトのデバイスメモリ領域に対し，16 スレッド単位で協調しながらアクセスする場合が最も高効率であり，同一のデータ量を単一に転送する場合と比べて 10 倍程度のメモリアクセスの高速化が可能である[12]．

4.2.3　分岐の削減

1 つの warp に含まれる全てのスレッドが，同一のパスを実行する様子を図 4 に示す．ここで，実行中の命令が条件分岐命令であり，warp 内のスレッドによって次に実行するパスが異なる場合を考える．1warp は SIMT 実行されるため，異なる命令は同時に処理することができない．まず一方のパスをとるスレッドのみを含んだ warp を実行し，次のサイクルにて片方のスレッドを含む warp を実行することで両者の分岐先の命令実行時間が生じる．条件分岐の少ないプログラムを記述することは，GPU 高性能を達成する上で重要である．

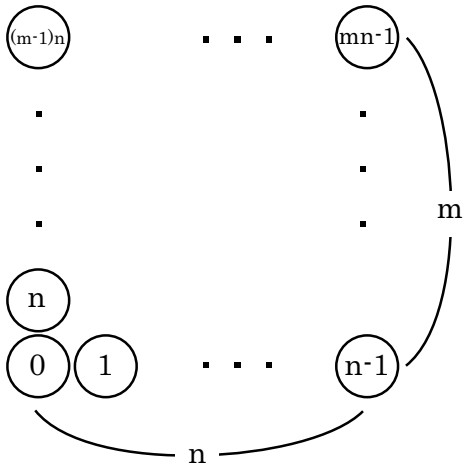


図 5　競合層ニューロンの配置

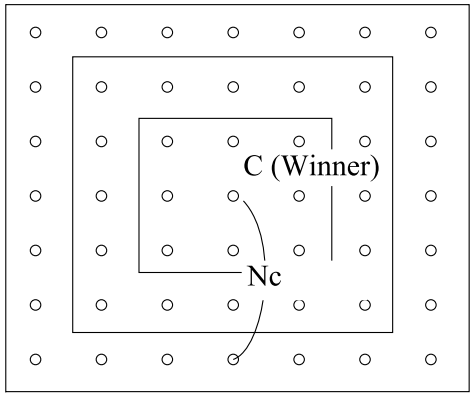


図 6　学習範囲の定義

5. 実　　装

5.1 SOM のデータ表現

競合層の重みベクトル，入力ベクトルは全て単精度浮動小数点型の 1 次元配列として表す．また，競合層ニューロンは，図 5 のように $n \times n$ の正方形状に配置し，0 から順に番号をつける． 全ての重みベクトルのデータは，結合アクセスを可能にするため，同一の属性のデータの構造体を並べた Structure of Array(以下 SoA) のデータとして表す．

配列の大きさは，パイプラインレイテンシを隠蔽できる数に，スレッド数は結合アクセスを行いやすい数に設定する．プロセッサの SP 演算ユニットのパイプラインレイテンシを隠蔽するのに必要なブロックあたりのスレッド数である 256 スレッドに設定し，隠蔽する．結合アクセスは，16 個のスレッド (64KB) 単位で行うことができるので 256 スレッド数という値はレイテンシを最小化し，最適な結合アクセスを行うための条件を満たす．ニューロンを SoA のデータとして表現するために，1 つの属性の配列の長さを 256 個 (768 バイト) 単位で増やしながら十分な長さの配列長を決定する．さらに同じ大きさのメモリ領域を確保し，その先頭から次の属性の重みベクトルの要素を順に格納する．また，勝者ニューロンの近傍距離 N_c を図 6 のように定義し学習させる．

BL-SOM の場合，基本 SOM とは異なり，重みベクトルを更新するために入力ベクトルの平均を求める必要がある．すなわち，重みベクトルは全ての入力がそれぞれの勝者ニューロンに関連付けられるまで更新されない．したがって式 (6) において，入力ベクトルの合計 $\sum_{\mathbf{x_k} \in S_{ij}} x_k$ と，関連付けられた入力ベクトルの数 N_{ij} を保存するための 2 つのメモリ領域がさらに必要である．ここで前者を Sum Buffer (SB)，後者を Learn Count Buffer (LCB) と呼ぶことにする．SB は，GPU 上での基本 SOM の実装するのに必要なメモリ領域よりもさらに m(行) $\times$ n(列) $\times$ v (次元数) の float 型領域が必要で，LB は m(行) $\times$ n(列) の整数型領域が必要である．

5.2 基本 SOM のカーネル

基本 SOM のアルゴリズムに従い，下の 3 つのカーネルを実装した．

5.2.1 calc_euclid_kernel()

calc_kernel() は，入力ベクトルと全ての重みベクトルとのユークリッド距離を求めるカーネルである．以下に，1 つの競合層ニューロンに割り当てられたスレッドの動作を示す．

(1)　重みベクトルの 1 次元分の要素をデバイスメモリから読み出し，レジスタに格納する．ここでは，スレッドが必ず結合アクセスによって読み出される．

(2)　重みベクトルの要素と入力ベクトルの各要素の差の自乗和を求める．

(3)　1 と 2 を次元数の回数だけ繰り返す．

結合アクセスの増加や，分岐による実行する warp 数の増加を避けるなどの理由から，SoA 内の重みベクトルの要素として実際には利用されていないデータについても自乗和を求める．また，計算量を減らすために，自乗和の値を用いて大小を比較する．

5.2.2 get_winner_kernel()

get_winner_kernel() は，calc_euclid_kernel() で求めたユークリッド距離の配列の中から最小値をもつインデックスを探し出すカーネルである．最小値は，図 7 のように，二分木のデータ構造を利用してリダクション演算を行う．GPU は，1 つのブロックに最大 512 個のスレッドを生成することができるので，1024 個ずつにデータを分け，繰り返し演算を行う．
以下は，1 つのスレッドの動作である．

(1)　ブロック内のユークリッド距離のデータが 1024 個埋まっていなければ，GPU で表せる整数型の最大の値を代入する．ここで，アライメントされて連続した 64 バイトのデータを読み出すのであれば，結合アクセスによって高速にアクセスが可能である。

(2)　配列前半の 512 データと後半 512 データについてを 1 つずつ大小関係を比較し，前半 512 個の領域小さい方の値を書き込み，後半 512 個にはそのニューロンの番号を

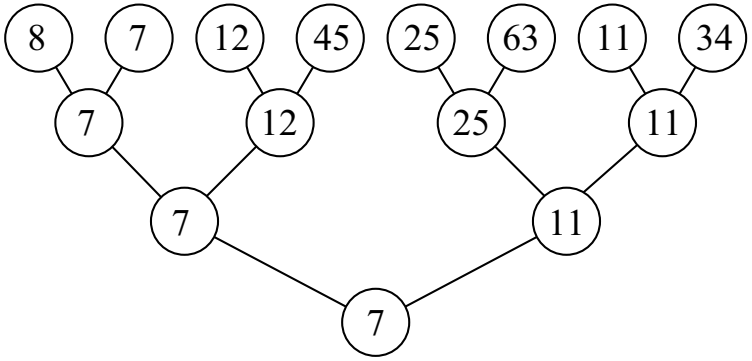


図 7　リダクション演算

書き込む．そして 1024 個から 512 個へのリダクションを行う．

(3)　以降，512→ ... → 4→2→1 個と，リダクションを行いながら小さい値を選び，ブロック内での最小値を決定する．

5.3　learn_kernel()

learn_kernel() は，勝者ニューロンとその周辺のニューロンの重みベクトルの値を更新し，学習させるカーネルである．calc_euclid_kernel() と同様に，1 スレッドに 1 ニューロンの更新を割り当てる．しかし，全ての重みベクトルが更新の対象となるは限らない．そこで，マップ全体を 8×16 のニューロンから成る区間に分割し，区間ごとにスレッドの起動とメモリデバイスメモリへのアクセスを行う．以下は，1 つの競合層ニューロンに割り当てられたスレッドの動作である．

(1)　スレッド ID，勝者ニューロンの位置を元に，割り当てられたスレッドのニューロンが，学習させる範囲の中でどの位置にあるのかを求める．

(2)　割り当てられたスレッドのニューロンがマップサイズ全体でどの位置にあるのかを求める．

(3)　割り当てられたスレッドのニューロンの位置がマップ全体からはみ出していなければ，次元の低い方から順に重みベクトルの要素を更新する．

5.4　BL-SOM のカーネル

BL-SOM のアルゴリズムに基づき，カーネルを実装した．BL-SOM の計算手順の内，(2) および (3) については基本 SOM と同じであるため，ここでは (5) および (5) の実装について述べる．

5.5　sum_up_kernel()

sum_up_kernel() は，勝者ニューロンとその近傍に対して関連づけられた入力ベクトルの合計 S_{ij} を求める．calc_euclid_kernel() と同様に，1 つのスレッドは一つのニューロンの S_{ij} の計算を担当し，分割された区間ごとにスレッドの起動と SoA の形で表現されたデータに対するメモリアクセスを行う．各スレッドは次のとおり計算する．

(1)　ベクトルの 1 つの次元の数について，勝者ニューロンの近傍と重なるニューロン区間の SB と LCB のデータを SM の共有メモリに転送する．

(2)　それぞれのスレッドは，スレッドが担当するニューロンが勝者ニューロンの近傍の範囲内にあり，値の更新の対象となるかをスレッドのインデックスより決定する．

(3)　スレッドが担当したニューロンが値の更新の対象となった場合，スレッドは SB に入力ベクトルの要素の値を足し，LCB の値を 1 つインクリメントする．

(4)　以上の計算をベクトルの次元数だけ繰り返す．

5.6　bl_learn_kernel()

bl_learn_kernel() は，式 (6) に従い重みベクトルの値を更新する．はじめにデバイスメモリから重みベクトルと SB，LCB の 1 つの要素をそれぞれ SM の共有メモリに転送し，式 (6) に従って更新する．最後に更新された重みベクトルの値をデバイスメモリに書き戻す．この操作をベクトルの次元数だけ繰り返す．

6. 性 能 評 価

設計した基本 SOM および BL-SOM の CUDA プログラムを用いて，マップサイズ，ベクトル次元数，学習させる範囲の大きさと，カーネルの実行速度の関係について評価を行った．本稿では，表 2 の実行環境で評価を行った．

6.1　パラメータ

本研究報告で設計したプログラムは，以下の 3 つの項目をパラメータ化している．

- 競合層ニューロンマップの 1 辺の個数: n
- ベクトル次元数: v
- 勝者ニューロンの近傍距離 N_c

これらはいずれも，プログラム実行時の引数として与えられ， 3 つの値のみで確保するメモリの領域や各カーネルのスレッド数やブロック数といった変数が決まる．

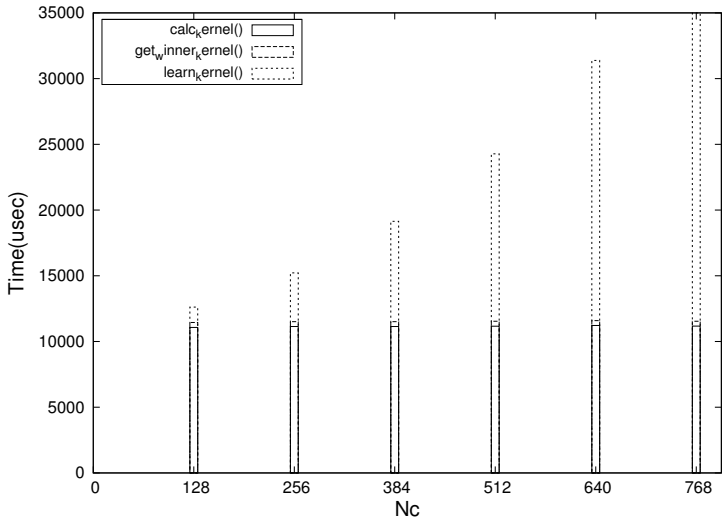


図 8　学習サイズ Nc と基本 SOM の実行時間の関係

6.2　基本 SOM の性能評価

6.2.1　近傍距離と基本 SOM 実行時間の関係

ベクトル次元数を 128，マップ数を 1372×1372 とし，近傍距離 N_c を変化させたときの GeForce 280GTX での実行時間を図 8 に示す.

6.2.2　マップサイズ，ベクトル次元数と基本 SOM の実行時間の関係

基本 SOM のマップサイズ，ベクトル次元数を変化させた場合の実行時間を図 9 に示す. GPU は GeForce GTX280 を使用し，近傍距離 N_c を 128 とした． 競合層ニューロン数の増加に対して，calc_euclid_kernel()，learn_kernel() が線形に増加している事が確認できる.

get_winner_kernel() は，リダクションにより最小値を求めるので，ニューロン数の増加に対して，$O(n \log n)$ で増加するが， 128 次元ベクトルでの実行において，GeForce GTX280 上で実行可能な最大ニューロン数が 1400×1400 程度なので getWinner_kernel() は最大でも 3 回のみの実行となるため全体での割合は少ない.

ベクトル次元数の増加に対しても calc_euclid_kernel()，learn_kernel() が線形に増加する. これは，ユークリッド距離を求める際の積和演算や，重みベクトルの要素の更新が，ベクトルの次元数の回数で行われるからである.

表 2　実行環境

CPU	Intel Core 2 Quad Q6600 2.40GHz
Memory	4GB
OS	Linux 2.6.23 (Fedora 8)
CPU コンパイラ	gcc-4.1.2
デバイス実行環境	CUDA 2.0
GPU コンパイラ	nvcc 2.0
GPU 測定環境	CUDA Visual Profiler 1.0

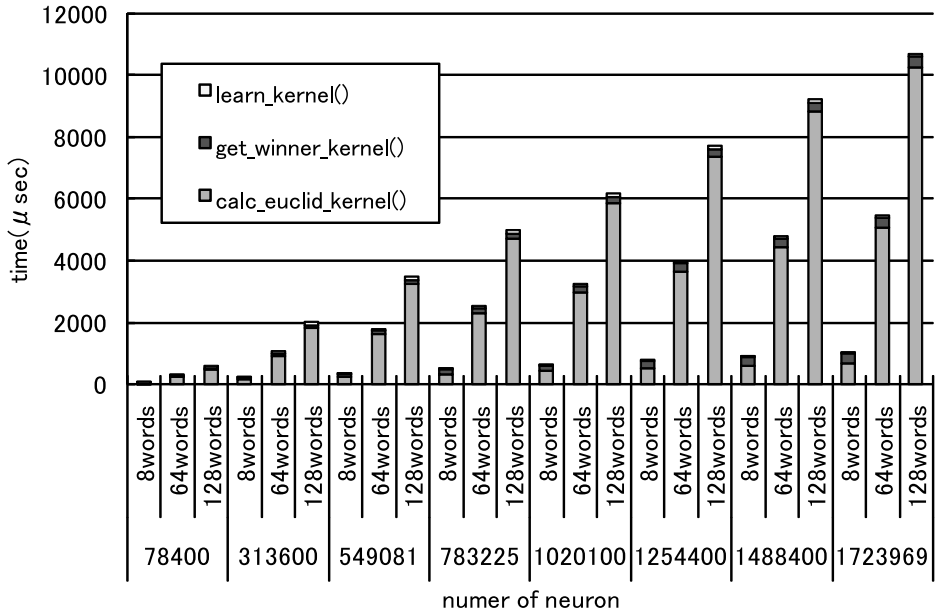


図 9　マップサイズ，ベクトル次元数の変化と基本 SOM の実行時間の関係

6.2.3　CPU との実行速度比較

表 1 の 3 枚のグラフィックカードを用いて，CPU との実行時間の比較を行った．ベクトル次元数を 128，近傍距離 N_c を 128 とし，マップサイズを変化させたとき，CPU と GPU の実行時間比の変化を図 13 に示す．値は (CPU の実行時間 / GPU の実行時間) として示す.

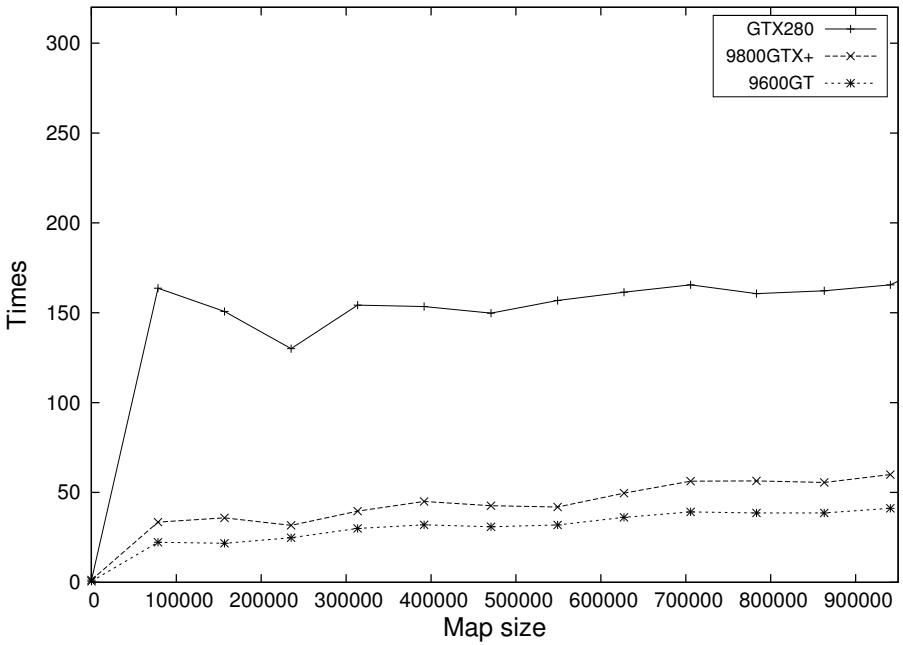


図 10　CPU と GPU の基本 SOM の実行時間比

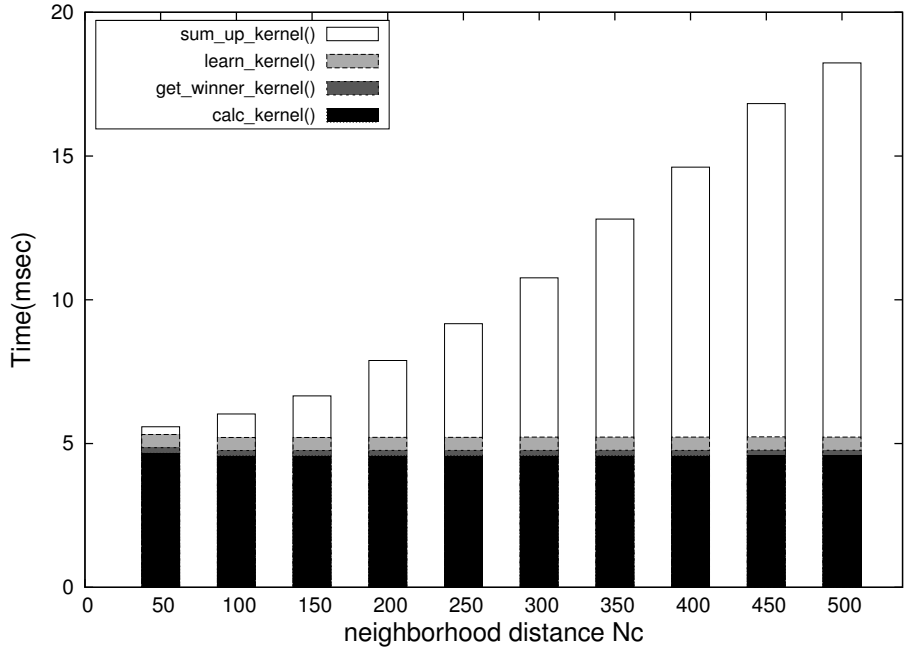


図 11　近傍距離 Nc と BL-SOM の実行時間の関係

6.3　BL-SOM の性能評価

GPU 上で BL-SOM を用いたゲノム解析を行うために，1 枚の NVIDIA GeForce GTX280 GPU 上で BL-SOM の性能評価を行った．

ホストマシンは基本 SOM と同じく表 2 のとおりである．本章ではマップサイズや近傍距離と性能の関係を示す．

6.3.1　近傍距離と BL-SOM の実行時間の関係

図 11 は，136 次元，960×960 の BL-SOM のマップにおいて，1 つの入力ベクトルを 1 回学習させたときの近傍距離と実行時間の関係を示している．sum_up_kernel() の実行時間は n_c^2 のオーダで増加することが分かる．しかし，この増加は緩やかになる．これは，勝者ニューロンの近傍距離の範囲がマップ全体よりも大きくなったためである．

6.3.2　マップサイズ，ベクトル次元数と BL-SOM 実行時間の関係

図 (9) は，GeForce GTX280 において 1 つの入力データを 1 回学習するときのマップサイズ，次元数と実行時間の関係を表す．近傍距離は 70 である．calc_euclid_kernel() と getWinner_kernel() については基本 SOM と同様である．また，sum_up_kernel() と bl_learn_kernel() の実行時間は，データの次元数とマップ内のニューロンの総数に対して線形に増加する．

ベクトルデータの次元数が 128 で GTX280 のメモリに格納できる最大のマップサイズが 960×960 のとき，getWinner_kernel() はせいぜい 3 回しか実行されないため，getWinner_kernel() の計算時間は実行時間の大部分を占めることはない．

6.3.3　CPU での実行時間との比較

実際のゲノムデータを用いて CPU での BL-SOM の実行時間と GPU での実行時間を比較した．ゲノムデータとして，ミトコンドリアの特徴を表す 136 次元ベクトル 8259 件を用いた．データの次元数，近傍距離，学習の繰り返し数はそれぞれ 136, 70, 1 としてミトコンドリアの学習データを学習させた時の CPU と GPU の実行時間の比を図 13 に示す．値

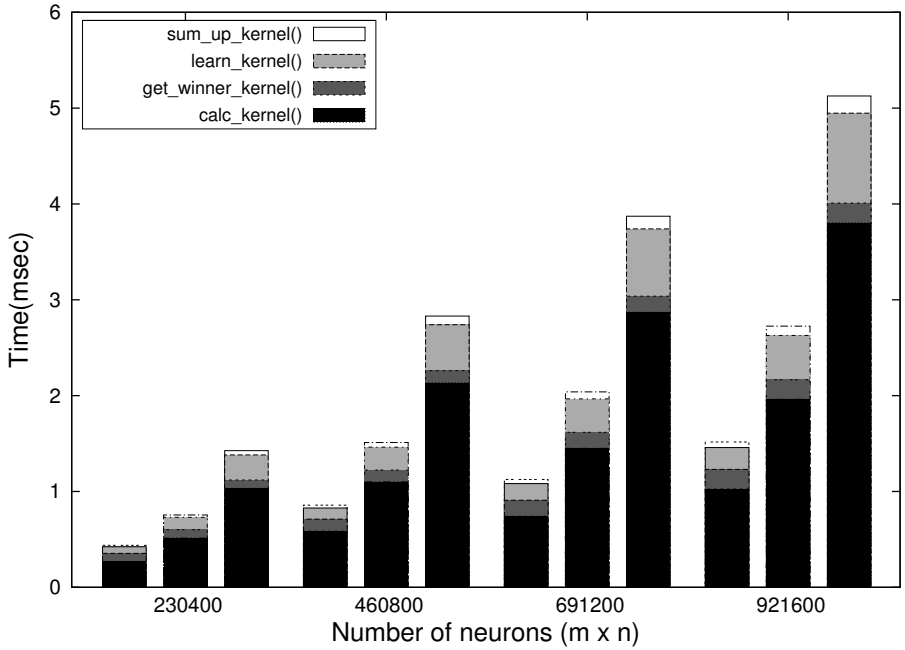


図 12　マップサイズ，ベクトル次元数の変化と BL-SOM の実行時間の関係

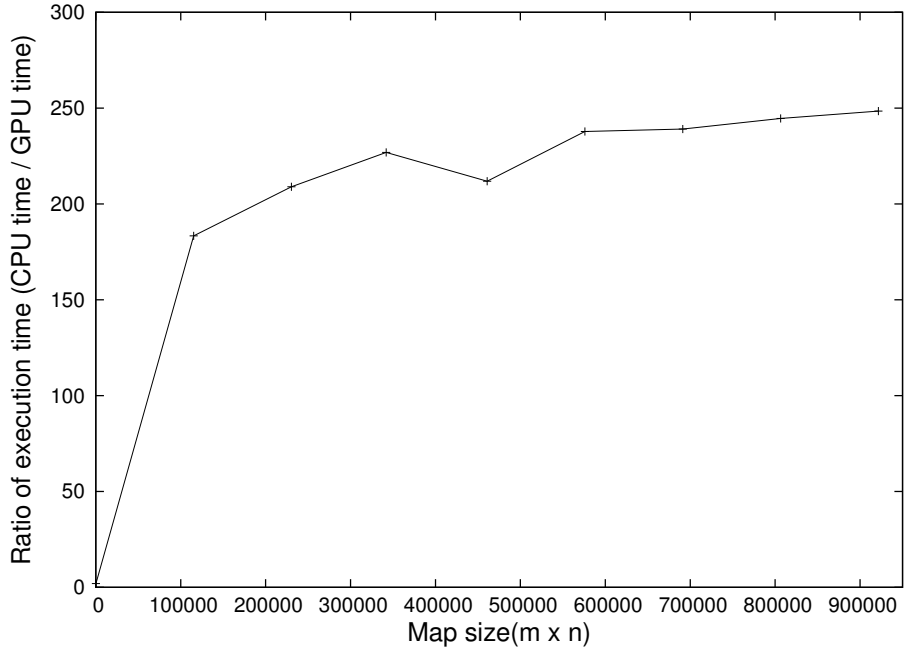


図 13　Execution time ratio of CPU to GPUs

は (CPU の実行時間/GPU の実行時間) として表す．結果，GPU での実装は CPU の 250 倍程度であった．

0

7. ま と め

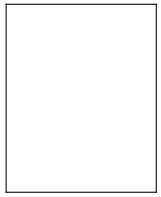
本稿では，GPU を用いた SOM のプログラムの実装を行い，評価した．実装には，パイプラインレイテンシの隠蔽，結合アクセス，分岐の削減の 3 つの最適化を行った．マップサイズやベクトル次元数，近傍距離を変化させて 3 種類の GPU 上で実行したところ，128 次元，280×280 のマップサイズ，近傍距離 128 の条件下において，GeForce GTX280 が Intel Core 2 Quad 2.40 GHz と比べ約 150 倍高速であることを確認した．さらに，128 次元のベクトルのマップで 68GCUPS の性能を示し，Virtex XC2V6000 FPGA を用いた実行時間と比較したところ，3.89 倍の性能が得られることがわかった．本実装は，実質的にマップサイズの制約を解消し，問題の形状変動に対する高い柔軟性と，高速性を併せ持つことが確認された．

参 考 文 献

1) Kohonen, T.: *Self-organization and associative memory*, Springer-Verlag New York, Inc. New York, NY, USA (1989).
2) 伊藤則夫，　白木渡，　安田登：“地盤物性値の空間分布推定問題への自己組織化特徴マップの応用”，土木学会論文集，Vol.VI-47, No.651, pp.145–156 (2000).
3) Abe, T., Kanaya, S., Kinouchi, M., Ichiba, Y., Kozuki, T., and Ikemura, T.: Informatics for unveiling hidden genome signatures, *Genome Research*, Vol.13, pp. 693–702 (2003).
4) Abe, T., Sugawara, H., Kanaya, S. and Ikemura, T.: Sequences from Almost All Prokaryotic, Eukaryotic, and Viral Genomes Available Could be Classified According to Genomes on a Large-Scale Self-Organizing Map Constructed with the Earth

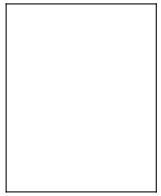
Simulator, *Journal of the Earth Simulator*, Vol.6, pp.17–23 (2006).
5) Carpenter, G. and Grossberg, S.: A massively parallel architecture for a self-organizing neural pattern recognition machine, *Computer Vision, Graphics, and Image Processing*, Vol.37, No.1, pp.54–115 (1987).
6) Speckmann, H., Thole, P., Rosentiel, W., Aleksander, I. and Taylor, J.: Hardware Implementations of Kohonen's Self-Organizing Feature Map, *Artificial Neural Networks, 2*, Vol.2, pp.1451–1454 (1992).
7) Porrmann, M., Franzmeier, M., Kalte, H., Witkowski, U. and Ruckert, U.: A Reconfigurable SOM Hardware Accelerator, *Proceedings of the 10th European Symposium on Artificial Neural Networks, ESANN*, pp.337–342 (2002).
8) Tamukoh, H., Aso, T., Horio, K. and Yamakawa, T.: Self-organizing map hardware accelerator system and its application to realtime image enlargement, *Neural Networks, 2004. Proceedings. 2004 IEEE International Joint Conference on*, Vol.4 (2004).
9) Ruping, S. and Ruckert, U.: A scalable processor array for self-organizing feature maps, *Microelectronics for Neural Networks, 1996., Proceedings of Fifth International Conference on*, pp.285–291 (1996).
10) Kanaya, S., Kinouchi, M., Abe, T., Kudo, Y., Yamada, Y., Nishi, T., Mori, H. and Ikemura, T.: Analysis of codon usage diversity of bacterial genes with a self-organizing map: characterization of horizontally transferred genes with emphasis on E. coli O157 genome, *Gene, Vol. 276*, pp.89–90 (2001).
11) Volkov, V. and Demmel, J.W.: Benchmarking GPUs to Tune Dense Linear Algebla, *SC'08* (2008).
12) NVIDIA: NVIDIA CUDA Compute Unified Device Architecture (2008).

(平成？年？月？日受付)
(平成？年？月？日採録)



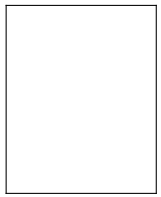
設樂　明宏

平成 21 年 慶應義塾大学理工学部情報工学科卒業．現在，同大大学院理工学研究科開放環境科学専攻修士課程在籍中．



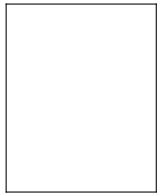
西川　由理（学生会員）

平成 18 年 慶應義塾大学理工学部情報工学科卒業．平成 20 年 同大大学院理工学研究科開放環境科学専攻修士課程修了．現在，同大大学院理工学研究科開放環境科学専攻博士課程在籍中．平成 20 年度より日本学術振興会特別研究員 DC1．ハイパフォーマンスコンピューティングとインターコネクトに関する研究に従事．



吉見真聡（正会員）

平成 16 年 慶應義塾大学理工学部情報工学科卒業．平成 21 年 同大大学院理工学研究科開放環境科学専攻博士課程修了．現在，



天野　英晴（正会員）

昭和 56 年 慶應義塾大学工学部電気工学科卒業．昭和 61 年 同大大学院工学研究科電気工学専攻博士課程了．工博．現在，慶應義塾大学理工学部情報工学科教授．計算機アーキテクチャの研究に従事．