社団法人　電子情報通信学会
THE INSTITUTE OF ELECTRONICS,
INFORMATION AND COMMUNICATION ENGINEERS

信学技報
TECHNICAL REPORT OF IEICE.

# 大脳皮質モデルBESOMのGPGPUによる並列化

中田　秀基†　井上　辰彦†,††　一杉　裕志†

† 産業技術総合研究所　〒 305–8560 茨城県つくば市梅園 1-1-1
†† 株式会社創夢　〒 151–0072 東京都渋谷区幡ヶ谷 1 － 34 － 14
E-mail: †{hide-nakada,tatuhiko-inoue,y-ichisugi}@aist.go.jp

**あらまし**　大脳皮質モデル BESOM は、最新の神経科学的な知見に基づき、大脳皮質をベイジアンネットによってモデル化したものである。BESOM における学習、認識には確率伝搬法の反復計算が必要なため計算量が大きく、並列化による実行速度向上が必須である。一般に機械学習機の並列化手法としてはモデル間並列とモデル内並列が考えられるが、本稿ではモデル内並列化を GPGPU で実現した技法について報告する。データ構造の見直しによる GPGPU によるモデル内並列化と、ミニバッチ学習化による並列化を併用することで約 50 倍の速度向上を得た。
**キーワード**　BESOM, GPGPU, ミニバッチ, 機械学習

# GPGPU parallelization of a cerebral cortex model BESOM

Hidemoto NAKADA†, Tatsuhiko INOUE†,††, and Yuuji ICHISUGI†

† National Institute of Advanced Industrial Science and Technology　Umezono 1–1–1, Tsukuba, Ibaraki, 305–8560 Japan
†† Soum Corporation　1 Chome － 34 － 14, Hatagaya, Shibuya, 151–0072 Tokyo
E-mail: †{hide-nakada,tatuhiko-inoue,y-ichisugi}@aist.go.jp

**Abstract**　We have been proposing a computational model of the cerebral cortex called BESOM, that models the cerebral cortex as Bayesian network based on recent findings in the neuroscience area. BESOM requires a compute intensive process called belief propagation for recognition and learning. Hence, parallelization is mandatory for practical application of BESOM. In general, there are two ways to parallelize machine learning system; i.e. inter–model parallelization and intra-model parallelization. We already implemented and reported the former. This paper addresses the latter; intra-model parallelization of BESOM. Using GPGPU and mini-batch method, we could achieve around 50 times speed-up compared with single thread CPU implementation.
**Key words**　BESOM, Master-Worker, parameter server

## 1. はじめに

BESOM モデル (BidirEctional Self Organizing Maps) [1] [2] [3] は、大脳皮質をベイジアンネットでモデル化した機械学習モデルである。BESOM は、大脳皮質に存在するマクロコラムを**ノード**として表現し、これらのノードを結んだ**ベイジアンネット**によって大脳皮質をモデル化するもので、パターン認識や強化学習など、人間の脳で行われる処理を計算機上で実現することを目指している。

一般に機械学習は、大量の学習データを処理しなければならないことから、計算機に対する負荷が大きい。それに加えて BESOM は、ベイジアンネット上で高機能な推論動作を実現するために確率伝搬法と呼ばれるある種の反復計算を用いるため、一般的なニューラルネットと比較して、より負荷が大きい。したがって、BESOM を実用的に利用するには、並列化による高速化が不可欠である。

機械学習を並列化する手法には大別して、ひとつのモデルの学習を並列化する方法と、複数のモデルを並列に学習させる方法がある。後者に関しては、すでに簡単なパラメータサーバを用いた複数モデルによる並列化を実現し、報告している [4]。

本稿では、前者の単一モデル内の並列化について、GPGPU による手法を報告するデータ構造を大幅に書き換えてモデル内を並列化するとともに、ミニバッチを導入することで、約 50 倍の高速化を実現する事ができた。本稿の内容の一部は [5] で報告済みである。

本論文の構成は以下の通りである。2. で BESOM を概説する。4. で実装について述べる。5. で評価を行う。6. で関連する研究を議論する。7. では、まとめと今後の課題について述べる。

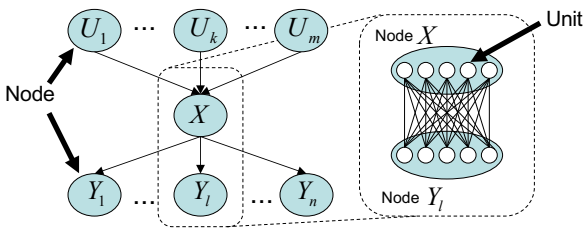


図 1　BESOM のネットワーク

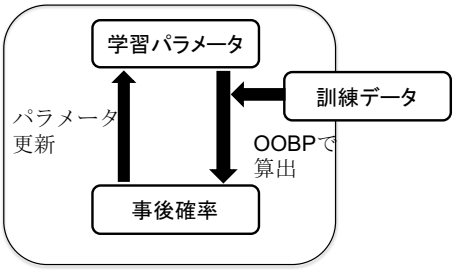


図 2　BESOM における学習の様子

## 2.　BESOM とその並列化

### 2.1　BESOM の概要

近年の計算論的神経科学における注目すべき進展として、「大脳皮質がベイジアンネットである」という仮説の登場がある。大脳皮質とベイジアンネットは、機能と構造の両面で多くの類似性を持ち、実際にさまざまな神経科学的現象がベイジアンネットを用いたモデルで再現されている。このような観点から、われわれは大脳皮質をベイジアンネットでモデル化した計算論的モデル BESOM を提案している。

ベイジアンネットを用いたディープラーニングは以下の点から有望であると思われる。

- ベイジアンネットは複数の事象の間の因果関係を確率により表現する知識表現の技術である。ベイジアンネットは信号源と観測データの間の因果関係を比較的少ないメモリで完結に表現できる場合があり、その場合は少ない計算量でさまざまな推論を行うことができる。
- 推論動作は、ネットワーク全体の情報を使って行われる。入力からのボトムアップの情報だけでなく、文脈からのトップダウンの予想の情報も用いたロバストな認識を行うことができる。したがって、フィードフォワード型ニューラルネットよりもはるかに高機能である。
- 階層的な生成モデルを素直に表現できるため、学習対象の事前知識が作りこみやすい。事前知識をネットワーク構造やパラメタ事前分布の形で作りこむことで、学習の性能を上げられる可能性がある。

BESOM は、大脳皮質に存在するマクロコラムをノードとして表現し、これらのノードを結んだベイジアンネットによって大脳皮質をモデル化する。個々のノードは複数の**ユニット**から構成される。ノードは確率変数に相当し、ユニットはその確率変数が取りうる値を示す。

図 1 に BESOM のネットワークの例を示す。楕円がノード、その中の白丸がユニットを表現している。ネットワークはこの図で示すように、一般に多階層の構造をとる。

通常のベイジアンネットでは、各ノードの条件付き確率表を表現するためには、親ノードの数 $m$ に対して $O(2^m)$ 個のパラメータが必要となるが、これは計算量の爆発をまねく。BESOM では、条件付き確率表の形をより少量のパラメータで表現できるよう制約している。各ノードのパラメータ数は $O(m)$ となる。

ソフトウェアとしての BESOM は、Java で記述されており、ハイパーパラメータをインタラクティブに操作するための GUI を備えている。

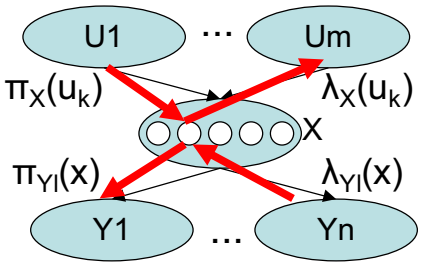


図 3　OOBP の概要

### 2.2　BESOM の学習と OOBP

BESOM での訓練データを用いた教師あり学習は次のように行う (図 2)。

（1）　訓練データ（入力値と教師信号）を、ネットワークの対応するノードに与える。

（2）　この条件下で、その他のノード（隠れ層のノード）の取る事後確率分布を決定する。

（3）　事後確率分布を観測データの分布とみなして、ネットワークのパラメータ（重み）を更新する。

また、学習結果のパラメータを用いた識別は次のように行う。

（1）　入力値のみをネットワークの対応するノード（入力層のノード）に与える。

（2）　この条件下で、その他のノード（隠れ層のノード）の取る事後確率分布を決定する。

（3）　出力層のノードの事後確率分布を識別結果として解釈する。

いずれの場合でも 2 の過程は全く同じであり、もっとも計算量の多い過程である。この過程を以下では認識とよぶ。BESOM における認識には、OOBP [3] と呼ばれるアルゴリズムを用いる。OOBP はループ有り確率伝搬法（Loopy Belief Propagation）[6] の一種で、反復計算によってノードの事後確率分布を計算するものである。図 3 に反復時に伝搬される情報の様子を示す。

ここでは詳細は省くが、ノードの状態を直接の親子関係の相互関係のみを用いて更新する作業を数回繰り返すことで、近似的に事後確率を算出する。繰り返し回数は現在 10 回に設定している。

OOBP における状態更新の式を 1 に示す。

$$
\begin{aligned}
\lambda_{Y_l}^{t+1}(x) &= \beta_2 \sum_{y_l} \lambda^t(y_l)(\pi^t(y_l) - \kappa_X^t(y_l) + w(y_l, x)) \\
\lambda^{t+1}(x) &= \prod_{l=1}^{n} \lambda_{Y_l}^{t+1}(x) \\
\pi_{Y_l}^{t+1}(x) &= \beta_1 \rho^{t+1}(x) / \lambda_{Y_l}^{t+1}(x) \\
\kappa_{U_k}^{t+1}(x) &= \sum_{u_k} w(x, u_k) \pi_X^t(u_k) \\
\pi^{t+1}(x) &= \sum_{k=1}^{m} \kappa_{U_k}^{t+1}(x) \\
\rho^{t+1}(x) &= \lambda^{t+1}(x) \pi^{t+1}(x) \\
BEL^{t+1}(x) &= \alpha \rho^{t+1}(x) \qquad (1)
\end{aligned}
$$

ユニットごとに、タイムステップ $t$ の情報を用いて $t+1$ での状態を算出する計算である。本質的に、ユニット単位での並列化可能な操作であることに注意して欲しい。

### 2.3 BESOM の並列化

BESOM の学習の並列化に関しては、大別して 2 つの方針が考えられる。ひとつは単一のモデル内での学習を並列化する方法である。上述のように、BESOM の学習過程のコアとなるアルゴリズム OOBP は個々のユニットに対して独立に実行することができるため、本質的にはユニット数分の並列度が存在し、比較的容易に並列化を行うことができる。

もう一つの方法は、複数のモデルを用いて独立、並列に学習を進める方法である。学習中に定期的にモデル上の学習パラメータを収集し、調停した結果を再配布してモデル間の同期を取ることで、効率的に学習をすすめることができる。もちろんこの 2 つは排他的ではなく、双方を同時に実現することも可能である。前述のとおり、本稿では単一モデルの並列化をあつかう。

### 2.4 オンライン学習とミニバッチ学習

学習過程の一部を並列化する手法の一つとしてミニバッチ学習化が挙げられる。本稿で基盤として用いた BESOM 実装ではオンライン学習を行う。すなわち、個々の訓練データに対して個別に認識を行い、その結果に基づいてモデルパラメータを更新することで学習する (図 4 左)。これに対して、少量のデータ集合（ミニバッチ）に対して認識を行い、その結果を用いてモデルパラメータを更新して学習するのがミニバッチ学習である (図 4 右)。

オンライン学習では、ある学習データの認識は、直前の学習データの認識結果に基づいて更新したモデルパラメータで行わなければならないため、学習データの認識を同時に複数行うことができない。これに対してミニバッチ学習では、バッチ内の個々のデータ認識過程には、相互に依存関係がないため、並行して行うことができる。

## 3. GPGPU による並列化

### 3.1 GPGPU の概要

GPGPU（General Purpose Graphic Processing Unit）は、本来グラフィック描画のために設計された高並列な演算ユニットを一般的な計算に転用するもので、近年高性能計算において広く用いられている。とくに NVIDIA 社の、CUDA と呼ばれるプログラミング環境を用いるものが事実上の標準となっている。

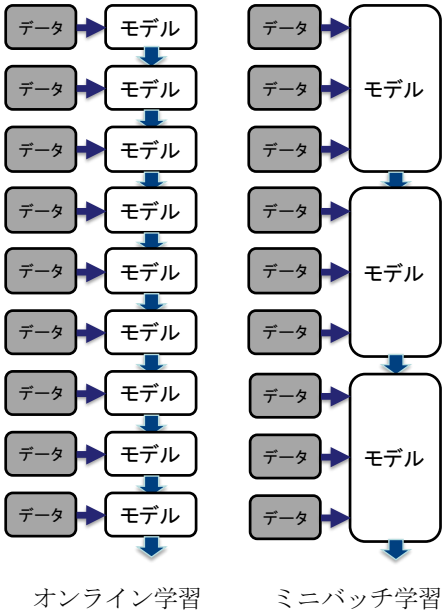


図 4　オンライン学習とミニバッチ学習

GPGPU は高性能計算一般に有効であるが、ニューラルネットワークのいわゆるディープラーニングは、精度が単精度で充分であること、計算が定型的な行列演算に帰着することなどから、非常に GPGPU と相性がよく、特に多用されている。既存のディープラーニングフレームワークのほぼ全てにおいて GPGPU がサポートされている。現在のディープラーニングブームの一端を GPGPU が支えていると言っても過言ではない。

### 3.2 GPGPU の構成

GPGPU は、通常の CPU と比較すると圧倒的に多数の演算コアを持つ。ただし、各コアのクロックは比較的低い。例えばわれわれが評価に用いた NVIDIA GeForce GTX980 は、2048 基の CUDA コアを持つが、クロックは 1.1GHz 程度である。

NVIDIA GPGPU はカーネル関数と呼ばれる一つの関数をすべてのコアで同時に実行する。一方、通常のマルチコア CPU の各コアにおいては実行するコードに何も制約がない。さらに、NVIDIA GPGPU では、計算コア群は 32 コアを単位とした warp と呼ばれるグループにまとめられている。この warp 内の 32 コアはプログラム・カウンタを共有して実行を行う。プログラムに条件分岐があり、コアによって真偽判断が別れる場合、warp は真の場合のコードと偽の場合のコードを直列化して実行する。したがって、個々のコアがそれぞれ異なる挙動を示すコードを実行すると、性能が大きく低下する。したがって、GPGPU を十分に活用するためには、各コアが均質な動作を行うようにプログラムを記述する必要がある。

現在の GPGPU は、CPU とは別のメモリ空間を持つしたがって一般に、GPGPU を用いて計算するには、下記のステッ

```
// Load the ptx file.
CUmodule module = new CUmodule();
cuModuleLoad(module, "JCudaVectorAddKernel.ptx");

// Obtain a function pointer to the kernel function.
CUfunction function = new CUfunction();
cuModuleGetFunction(function, module, "add");

// Call the kernel function.
cuLaunchKernel(function,
    gridSizeX,  1, 1,      // Grid dimension
    blockSizeX, 1, 1,      // Block dimension
    0, null,               // Shared memory size and stream
    kernelParameters, null // Kernel- and extra parameters
);
```

図 5　JCUDA の利用法

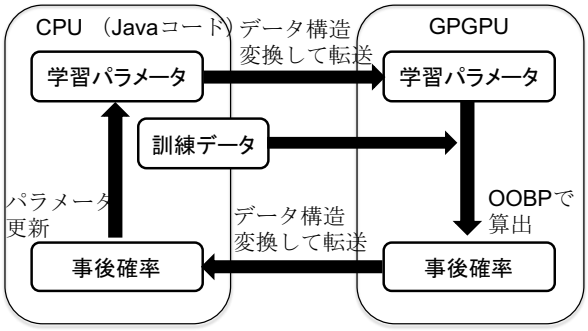


図 6　GPGPU を用いた学習

プを踏む必要がある。

- CPU 上のデータを GPGPU に転送
- GPGPU 上でデータを処理
- 処理結果を CPU に転送

また、メモリ空間が異なるということは、GPGPU に転送する際にポインタを特別に処理しなければならないことを意味する。ポインタによるリンク構造のようなデータ構造を GPGPU に転送する際にポインタを変換して、GPGPU 上でもリンク構造が保たれるようにしなければならない。このコストは大きい。

### 3.3　JCUDA

GPGPU のカーネルは、C,C++から呼び出すのが一般的な利用法であるが、C,C++以外の言語から呼び出すライブラリも、さまざまな言語に対して提供されている。JCUDA [7] は、Java から直接 CUDA で記述したコードを呼び出すことを可能にするライブラリである。Cublas や Cudnn などの、CUDA で記述されたライブラリを直接サポートしている。

Java から CUDA カーネルを利用するには、CUDA ソースコードをコンパイルすると生成される中間コードである ptx ファイルをロードして関数ポインタを取り出して、それに対して呼び出しを行う。図 5 に、簡単な利用イメージを示す（[7] より引用）。

## 4.　GPGPU による並列化の実装

OOBP の GPGPU 実装について述べる。BESOM 実装本体のコードは Java によって記述されており、ユーザインターフェースやデータの入出力など、さまざまな機能を提供している。これらすべてを CUDA ベースに移行することには意味が無い。今回の実装では OOBP の部分のみを対象として GPGPU 化を行った。Java による BESOM 実装本体から GPGPU コードの呼び出しは JCUDA を用いた。

### 4.1　GPGPU 上でのデータ構造

Java による OOBP のコードでは、各ノードをオブジェクトで表現し、個々のオブジェクトの中にパラメータを配列で保持している。このようなデータ構造は GPGPU には適さない。

われわれは、ノードの持つパラメータをそれぞれ 2 次元の配列に展開した。これによってデータへのアクセスが均質化される。

GPGPU を用いた学習の様子を図 6 に示す。まず、学習パラメータと訓練データを GPGPU に転送する。この際にパラメータを GPGPU での処理に適した配列に展開する。GPGPU では OOBP を行い事後確率を算出する。算出した事後確率をデータ構造変換したうえで CPU 側に回収する。この事後確率情報を用いて、CPU で学習パラメータを更新する。

### 4.2　ミニバッチ学習化

OOBP の単純な GPGPU 化では限定的な速度向上しか得られなかったことから、ミニバッチ学習化を行った。ミニバッチ学習化のメリットは 2 点ある。

- CPU - GPGPU 間のパラメータ転送頻度を下げることができる。パラメータの更新を CPU でおこなうため、パラメータを更新するたびにパラメータを GPGPU に転送しなおさなければならない。ミニバッチ学習化を行うと、パラメータの更新頻度が低下するので、転送頻度が小さくなり、実行時間削減に繋がる。
- 並列実行の機会が増える。GPGPU は大量のスレッドを用いることで、メモリのレイテンシを隠蔽する。GPGPU の性能を十分に引き出すには実際のコア数よりもさらに大きなスレッド数が必要である。ミニバッチ学習化をすることで、一度に処理するデータの量が増大するので、並列実行の機会が増大する。

ミニバッチ学習化した OOBP では、バッチサイズ分の OOBP 処理を同時に行う。イメージとしては、行列とベクトルの演算を行列と行列の演算に変更することに相当し、大きな性能向上が期待できる。

## 5.　評　　価

### 5.1　評 価 環 境

本研究での評価には、NVIDIA GeForce GTX 980 を 1 基持つ PC を使用した。CPU には Intel(R) Xeon(R) CPU W5590（3.33GHz, 4 コア） x 2 ソケット、メモリは 48GByte 搭載している。カーネルは Linux 3.13.0 である。Java 処理系には Oracle JDK 1.8.0_45 を用いた。CUDA のバージョンは 7.5 である。

評価実験には MNIST 手書き文字認識 [8] を用いた。これは 0 から 9 までの手書きの数字を 28x28 の白黒 8bit の画像としたラベル付きデータで、60000 件の訓練データと 10000 件のテストデータから構成される。図 7 に画像の例を示す。

利用したネットワークの構造を図 8 に、その際のパラメータを表 1 に示す。ネットワークは、入力層、出力層に加えて 2 つの隠れ層を持つ。入力層に画像データを与え、出力層に教師ラ

表 1　評価に用いたネットワークのパラメータ

| | 入力層 | 隠れ層 1 | 隠れ層 2 | 出力層 |
|---|---|---|---|---|
| ノード数 | 784 (28 * 28) | 81 (9 * 9) | 9 (3 * 3 ) | 1 |
| 1 ノードあたりのユニット数 | 2 | 20 | 100 | 11 |
| 総ユニット数 | 1568 (784 * 2) | 1620 (81 * 20) | 900 (9 * 100) | 11 |
| 接続する親ノード数 | - | 16 (4 * 4) | 9 (3 * 3) | 9 |
| パラメータ数 | - | 51840 (81 * 20 * 2 * 16) | 162000 (9 * 100 * 20 * 9) | 9900 (1 * 11 * 100 * 9) |

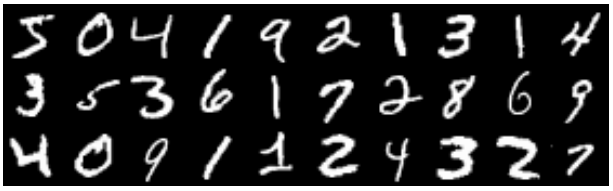

図 7　MNIST 手書き文字の例

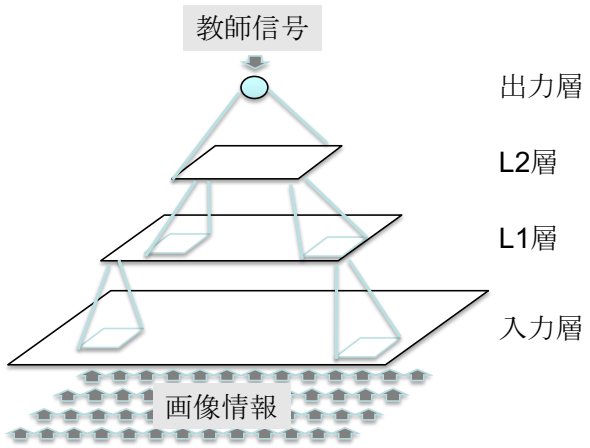


図 8　MNIST 認識に用いるネットワーク

ベルを与える。各層は 2 次元平面に配置される。各層のノードは、上位層のノードのうち、2 次元的に局所配置された一部のノードとのみ接続される。隠れ層 1 は入力層の 4x4 の領域に、隠れ層 2 は隠れ層 1 の 3x3 の領域に接続される。出力層は隠れ層 2 のすべてのノードに接続される。

### 5.2　評価結果

OOBP の実行時間と、学習を含めた実行時間を計測した。1280 枚の画像を学習し、そのうち OOBP にかかった時間のみを積算し、OOBP の実行時間とした。ミニバッチを用いる場合にはバッチサイズを 128 とした。計測はそれぞれ 10 回行い、最小値を値とした。

#### 5.2.1　OOBP の実行時間

図 9 に OOBP の実行時間を示す。ベースラインとして、CPU で 1 スレッドのみ用いた場合を計測した (**CPU 1 Thread**)。実行には 30 秒以上かかっている。**CPU 8 Thread** は、ノード単位の並列化を CPU を用いて実行した結果である。8 スレッド用いているにもかかわらず、3 倍程度の高速化しか得られていない。これは、負荷に大きな不均衡があるためである。各ノードの OOBP の計算時間は、そのノードと親ノードとの間のパラメータ数に依存する。この数は層によって大きく異なるため、負荷が均一にならないためである。

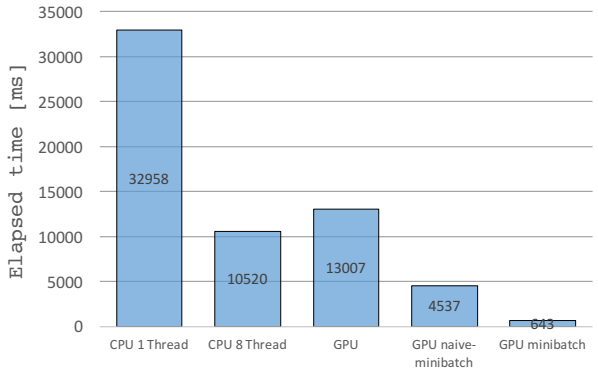


図 9　OOBP 実行時間

**GPU** は、ミニバッチを用いずに GPGPU で並列化した結果である。OOBP を行うたびに学習パラメータを GPGPU に転送している。GPU 並列化しているにもかかわらず、十分な速度向上が得られていない。この理由は、学習パラメータの転送が負荷になっていることと、並列度がそもそも不足しているためであると考えられる。

**GPU naive-minibatch** は、上述の学習パラメータ転送コストを排除するため、バッチサイズ (128) 回に一度だけパラメータを転送するようにしたものである。**GPU naive-minibatch** と **GPU** の差分は、このパラメータ転送時間であると考えて良い。一度あたりの転送コストは小さいが、およそ 1200 回の転送に、約 8.5 秒掛かることがわかる。

**GPU minibatch** は、OOBP の計算をミニバッチ的に行った結果である。すなわち、128 枚の画像を同時に認識している。重み配列の転送も 128 枚につき一度だけとなっており、パラメータ転送時間に関しては **GPU naive-minibatch** と同じであると考えて良い。**GPU naive-minibatch** と **GPU minibatch** の差分は、演算のミニバッチ化による並列度増大の効果である。これによって約 7 倍の高速化が得られている。ミニバッチでは、1280 枚の画像を 643ms で認識できている。したがって 1 枚あたりの認識時間は 0.5ms である。これを **CPU 1 Thread** と比較すると、高速化は約 51 倍となる。

#### 5.2.2　学習の実行時間

図 10 に学習を含めた全体での実行時間を示す。CPU で 1 スレッドのみの場合（**CPU 1 Thread**）と、GPGPU によるミニバッチ学習の場合で計測した。

GPGPU によるミニバッチでは、パラメータの更新を CPU で行う場合（**GPU minibatch, CPU learning**）と、パラメータの更新も含めて GPGPU で行う場合（**GPU minibatch, whole**）の二通りを測定した。**CPU 1 Thread** では、約 45

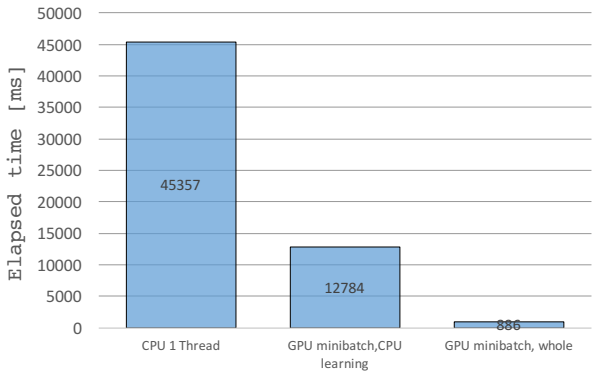


図 10　学習も含めた実行時間

秒となっており、1280 回のパラメータの更新に 22 秒が費やされていることがわかる。**GPU minibatch, CPU learning** は、12.8 秒となっている。OOBP 部分は 643ms であるから、12 秒程度がパラメータの更新に費やされていることがわかる。**GPU minibatch, whole** では、パラメータの更新も含めた学習が 886ms で終了している。これは計算がほぼ GPGPU 内で完結するため、CPU との通信が、ミニバッチごとの学習データの送信と、最初と最後のパラメータ送受信のみとなるためである。

**CPU 1 Thread** と **GPU minibatch, whole** を比較すると、約 45 秒から、約 0.9 秒へと 51 倍高速化している。

## 6. 関 連 研 究

GPGPU による並列化やミニバッチを用いた並列化はニューラルネットのディープラーニングにおいては標準的な実装技法であり、既存の殆どのライブラリ、フレームワークが実装している [9] [10] [11] [12] [13] [14]。

本研究の特徴はニューラルネットにおいて標準的な実装手法を、ベイジアンネットを基盤とする `BESOM` に導入したという点にある。ニューラルネットの演算は多くの場合、行列演算に帰着するため、CUDA で記述された行列演算ライブラリを用いることで比較的容易に GPGPU 化することができる。これに対して `BESOM` の OOBP は、行列演算に帰着することはできないため、GPGPU 実装は容易ではない。

## 7. お わ り に

本稿では、大脳皮質モデル `BESOM` の GPGPU による単一モデル内並列化について述べた。データ構造を大幅に書き換えてモデル内を並列化するとともに、ミニバッチ学習を導入することで、約 50 倍の大幅な高速化を実現する事ができた。

今後の課題としては以下が挙げられる

- 学習率の調整

本研究では認識過程の大幅な高速化を実現できたが、識別精度の向上には繋げられなかった。これは、ミニバッチ化に伴って必要な学習率の調整ができていないためであると考えられる。今後対応する予定である。

- マルチ GPGPU 対応

GPGPU は単一の PC ノードに対して複数台接続する事が可能であり、4 基搭載可能なサーバも入手可能となっている。これらの計算資源を有効に活用するために、複数 GPGPU を用いた高速化を実現する必要がある。

- 分散化との併用

本稿ではモデル内を並列化したが、これをモデル間並列化と併用することで更に高速化することが可能である。今後実装を進めていく。

## 謝　　辞

この成果は、国立研究開発法人新エネルギー・産業技術総合開発機構（ＮＥＤＯ）の委託業務の結果得られたものです。

## 文　　献

[1] 一杉裕志，“大脳皮質のアルゴリズム besom ver.1.0,” AIST09 J00006，産業技術総合研究所，2009.
[2] 一杉裕志，“大脳皮質のアルゴリズム besom ver.2.0,” AIST11 J00009，産業技術総合研究所，2011.
[3] Y. Ichisugi and N. Takahashi, “An efficient recognition algorithm for restricted bayesian networks,” Proc. of 2015 International Joint Conference on Neural Networks (IJCNN 2015)., 2015.
[4] 黎　明曦，谷村勇輔，一杉裕志，中田秀基，“マスタ・ワーカ型パラメータサーバの実装と besom への適用,” 信学技報, vol. 115, no. 174, CPSY2015-33，pp.179–184，2015.
[5] 中田秀基，黎　明曦，井上辰彦，一杉裕志，“大脳皮質モデル besom のクラスタ分散化と gpgpu 並列化,” 第 18 回情報論的学習理論ワークショップ（IBIS2015）ポスター，2015.
[6] K.P. Murphy, Y. Weiss, and M.I. Jordan, “Loopy belief propagation for approximate inference: An empirical study,” Proceedings of the Fifteenth Conference on Uncertainty in Artificial Intelligence, pp.467–475, UAI’99, Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1999. http://dl.acm.org/citation.cfm?id=2073796.2073849
[7] “jcuda.org: `http://www.jcuda.org`”. Accessed: 2015-11-04.
[8] “The mnist database of handwritten digits `http://yann.lecun.com/exdb/mnist/`”. Accessed: 2015-06-20.
[9] Y. Jia, E. Shelhamer, J. Donahue, S. Karayev, J. Long, R.B. Girshick, S. Guadarrama, and T. Darrell, “Caffe: Convolutional architecture for fast feature embedding,” CoRR, vol.abs/1408.5093, 2014. http://arxiv.org/abs/1408.5093
[10] “github cuda-convnet2: `https://github.com/akrizhevsky/cuda-convnet2`”. Accessed: 2015-11-04.
[11] “Tensorflow : `https://tensorflow.org/`”. Accessed: 2015-11-04.
[12] “chainer.org: `http://chainer.org`”. Accessed: 2015-11-04.
[13] “torch : `http://torch.ch`”. Accessed: 2015-11-04.
[14] “Pylearn2 : `http://deeplearning.net/software/pylearn2/`”. Accessed: 2015-11-04.