Vol.2017-HPC-162 No.25
2017/12/19

大規模ニューラルネットワークモデルの Out-of-Core 学習の性能評価

今井晴基[†1]　レドゥック トゥン[†1]　根岸康[†1]　関山太朗[†1]　河内谷清久仁[†1]

概要：深層学習のニューラルネットワークモデルは年々大規模化し，今後 GPU メモリー上に載せることは困難になることが想定される．それら大規模なモデルを用いた学習を実現する手法として，各層の計算時に必要となるデータのみを GPU に配置し，その他の層のデータを CPU に退避させる Out-of-Core 学習の手法が存在する．Out-of-Core 学習は CPU-GPU 間のデータ通信によるオーバーヘッドを発生させるが，全ての層のデータが GPU 上に載らないモデルの学習も可能とする．我々はこの Out-of-Core 学習の機能を Chainer version 3 に移植・実装し，複数 GPU への対応を行った．本稿では，GoogLeNet や ResNet など既存モデルを拡張した大規模モデルと ImageNet を拡大したデータセットを用いて Out-of-Core 学習の実行時間の評価を行なった．Out-of-Core 学習により 2 倍から 6 倍のバッチサイズが実行可能となった．またその実行オーバーヘッドは最大 80%程度であった．

キーワード：深層学習，大規模ニューラルネットワーク，Out-of-Core 学習，GPU，Chainer

Performance evaluation of Out-of-Core training for large-scale neural network models

HARUKI IMAI[†1]　TUNG LE DUC[†1]　YASUSHI NEGISHI[†1]
TARO SEKIYAMA[†1]　KIYOKUNI KAWACHIYA[†1]

Abstract: Neural network models of deep learning have been expanded and it can be difficult to fit them in GPU memory. To realize training of such large model, there is an Out-of-Core training which locates only necessary data for each layer on GPU and swap out data of other layers to CPU. The Out-of-Core training incurs additional overhead of the CPU-GPU communication, but we can realize training of the very large network model. We ported and implemented the Out-of-Core training to Chainer version 3 and also implemented multi-GPU Out-of-Core training. In this paper, we evaluated execution time of the Out-of-Core training by using enlarged existing models such as GoogLeNet and ResNet and enlarged imageNet dataset. We confirmed we can run 2 to 6 times larger batch size by using the Out-of-Core training with up to about 80% overhead.

Keywords: Deep Learning, Large-scale neural network, Out-of-Core training, GPU, Chainer

1. はじめに

深層学習は画像認識，音声認識など様々な分野において既存手法を圧倒する性能を示し続けている．その性能実現には，より深く大規模なニューラルネットワークが必要不可欠となってきており，大規模画像認識の競技大会である ILSVRC[1]では，2012 年に Hinton らの 8 層の AlexNet[2]を起点として，2014 年の VGG[3]は 19 層，2014 年 GoogLeNet[4] 22 層，2015 年 ResNet[5]152 層と年々大規模なネットワークを利用したチームが優勝し続けている．また，医療画像処理など実応用においても，3 次元データにおいて 3 次元の畳み込み処理を行うことが期待されているなど[6]，今後も大規模化の流れは継続するものと考えられる．

ネットワークの大規模化に伴う演算量の増加などから，深層学習は GPU を用いて実行されることがほとんどである．GPU の演算能力は，新しいデバイスが出るたびに飛躍的に向上しているが，メモリー容量は，演算能力と比較して向上していない．最新の NVIDIA® Tesla® V100 においても 16GB と据え置かれている．GPU のメモリーは HBM など高速であるが高価なメモリーが搭載されていることなどから，今後もその飛躍的な増加は困難と考えられる．そのため，大規模ネットワークの実行には GPU メモリー容量の制約がより大きな課題となると考えられる．

その回避のために医療画像処理ではアプリケーションに特化したメモリー使用量の削減手法が開発されている[6]．また，より汎用的に利用可能なメモリー圧縮手法なども開発されている[7]．しかし，メモリー削減手法だけではさらなる大規模化に対処することに限界がある．伝統的に HPC のアプリケーションが，それを回避するために演算とメモリーを分割して分散環境でアプリケーションを実行してきたように，深層学習でもネットワークを分割するアプローチが開発されてきている[8][9]．一般的にニューラルネットワークは密に結合している部分も多いため，分散化した時の通信オーバーヘッドはより大きくなる可能性があるため

†1 日本アイ・ビー・エム株式会社 東京基礎研究所
IBM Research - Tokyo

今後様々な工夫が必要となる．NVIDIA 社の CUDA ライブラリが提供する機能 Unified Memory を利用すれば，アプリケーションが GPU の実メモリー以上のメモリーを使用することが可能になる．Unified Memory によりプログラムをほぼ変更することなくより大規模なネットワークの実行が可能になるが，実行時間が数分の一程度に低下してしまうため，現状では実用的な選択肢とならない．

もう一つのアプローチは，本稿で評価を行う Out-of-Core 学習である．Out-of-Core 学習とは全てのネットワークのデータを GPU メモリーに確保し続けるのではなく，演算対象のレイヤーのデータのみを GPU に載せ，他のレイヤーのデータは CPU 側のメモリーに退避させる手法である[10]. GPU はホスト CPU にアクセラレータとして搭載されているが，演算量は GPU の方が圧倒的であるため，深層学習では CPU はあまり利用されておらず，CPU のメモリーも十分に活用されていないことが多い．このアプローチの課題は CPU-GPU 間の通信が非常に多くなるため，そこがボトルネックになってしまう点である．NVLink など高速なリンクも採用されることもあるが，GPU のメモリーバンド幅と比較して低速である．

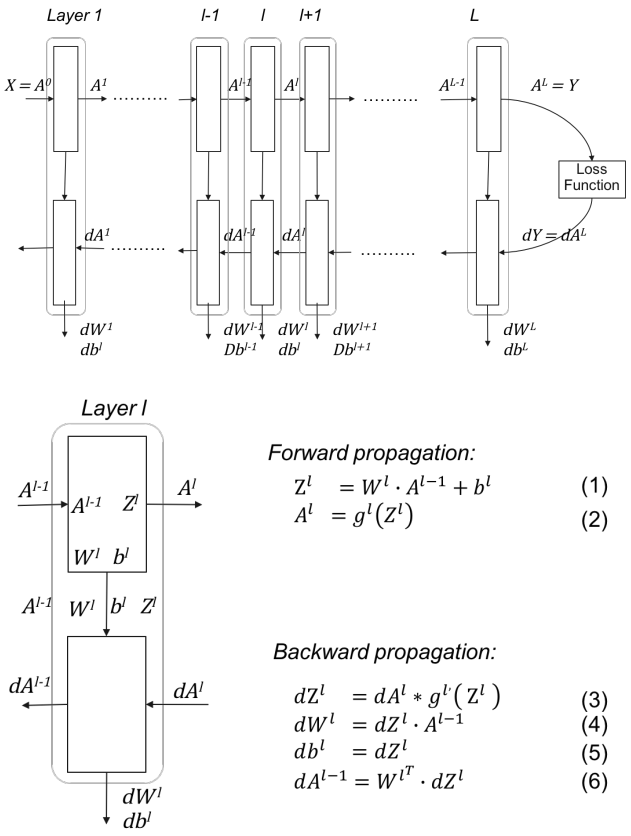
本稿では，その Out-of-Core 学習の実行時の CPU-GPU 間の通信の影響の評価を中心に性能評価を行い，その高速化の可能性や今後の課題を検討する．以下，第 2 章では，Out-of-Core 学習の動作を解説し，第 3 章では，今回評価に用いる Chainer における実装について述べ，第 4 章では計測条件と計測結果の評価を行い，第 5 章でまとめとする．

2. Out-of-Core 学習

本章では本稿で評価を行う Out-of-Core 学習について述べる．図 1 に L 層のニューラルネットワークの誤差逆伝搬法における処理とデータの流れを示す．上段に順伝搬処理，下段に逆伝搬処理が書かれている．

順伝搬処理の l 層においては，l-1 層の出力 $A[l-1]$を入力とし，重み $W[l]$とバイアス $b[l]$から(1)式を用いて，出力 $Z[l]$を求める．Z[l]に(2)式のように活性化関数 g を適用し l+1 層の入力とする．1 から L 層まで同様の処理を行い，Loss 関数の出力 $dY(=dA[L])$を逆伝搬処理の入力とし，逆伝搬処理を行う．逆伝搬の l 層においては，l+1 層の出力 $dA[l]$, l 層の順伝搬で入力 $A[l-1]$, 計算結果の $Z[l]$，重み $W[l]$, バイアス $b[l]$を用いて，(3)式から(6)式を用いて，l 層の逆伝搬における出力 $dA[l-1]$，重み更新の勾配 $dW[l]$, バイアス更新の勾配 $db[l]$を求める．この処理を各層で順次行うことで全ての層の dW, db を求め，これらを用いて，重み W, バイアス b を更新する．この順伝搬処理，逆伝搬処理を 1 回のパラメータ更新としてこれを継続して行うことで Loss 関数の出力を小さくしていく．Out-of-Core 学習でない通常の場合(以下，In-Core 学習とする)，逆伝搬での入力 $A[l-1], Z[l], W[l], b[l]$は順伝搬で計算された後，逆伝搬で利用されるまで GPU メモリー上に保持される．畳み込みニューラルネットワークの場合，最初の層の方がそのデータサイズが大きい傾向があり，それらはその最初の層の逆伝搬処理が完了するまで保持され続けるため，多くの GPU メモリーが消費され続けることになる．

一方，Out-of-Core 学習においては，逆伝搬処理で利用する順伝搬処理の入力，演算結果を GPU メモリー上に保持し続けるのではなく，各層の順伝搬処理後，これを一旦 CPU に退避させる．そして，逆伝搬でその層の演算を行う前に，CPU から GPU に戻し，各層の処理を行う．深いネットワークの場合, 大量のデータを CPU メモリーに退避させるため，CPU メモリーを大量に消費することになるが，CPU メモリーは GPU メモリーと比較すると安価で容量も大きいため，多くの場合問題とならない．課題は，CPU メモリーにデータを退避させる通信時間であり，それが大きくなりすぎると In-Core 学習に対して大幅な処理時間の増加を招く．



$$Z^l = W^l \cdot A^{l-1} + b^l \quad (1)$$
$$A^l = g^l(Z^l) \quad (2)$$
$$dZ^l = dA^l * g^{l\prime}(Z^l) \quad (3)$$
$$dW^l = dZ^l \cdot A^{l-1} \quad (4)$$
$$db^l = dZ^l \quad (5)$$
$$dA^{l-1} = W^{l^T} \cdot dZ^l \quad (6)$$

図 1 Out-of-Core 学習の処理とデータの流れ

3. Chainer における Out-of-Core 学習の実装

本稿では，2 章の Out-of-Core 学習の評価を Chainer を用いて評価を行った．Chainer は，Preferred Networks 社が開発している深層学習用のフレームワークであり，Python の行列演算用ライブラリ Numpy 互換のインターフェースを持つ CUDA 実装の行列演算ライブラリである Cupy を利用し

GPU 上で動作させることができる．また，Chainer versin2 においては，Out-of-Core 学習をサポートする派生レポジトリー[14][15]を利用することで実現可能である．今回我々は，最新の正式リリースである Chainer version3 に，Version 2 の Out-of-Core 学習のコードを移植し，それに対して，複数 GPU 対応や性能向上のための修正を行った．

3.1 Chainer version2 における Out-of-Core の実装

Chainer における順伝搬，逆伝搬の計算過程を説明するために，図 2 に計算グラフの構造を簡易的に示す．分岐はないケースである．

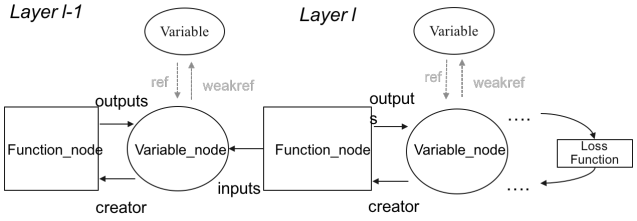


図 2 Chainer 計算グラフ構造

順伝搬においては，ユーザーが作成したネットワークのモデルファイルを元に，各レイヤー(Function)ごとに Function_node オブジェクト，Variable_node オブジェクトを順次作成しながら，計算グラフを作成していく．

逆伝搬処理においては，順伝播で作成した Function_node オブジェクトと Variable オブジェクトを逆順にたどる．Chainer version2 の場合，入力データを含む Variable_node を inputs 変数で参照することができ，Variable_node はそれを作成した creator_node 変数で参照できる．逆伝播処理は Variable_node 中の backward 関数を creator_node がなくなるまで順次呼び出すことで実行する．

Out-of-Core 学習の実行時のオプションとしては，async, fine_granurality がある．async は CPU-GPU 間のデータ退避復帰の通信を同期して行うか非同期に行うかの設定である．非同期に行う場合，通信時間が隠されることから Out-of-Core 学習のオーバーヘッドが小さくなるため，高速に実行できるが，逆伝搬処理の時，次のレイヤーのデータをプリフェッチしておくことになるため，そのレイヤーの処理のためのデータに加え次のレイヤーのデータも GPU メモリーに配置しておくことになるため，メモリー使用量が増加する．

またモデルファイルを修正することなく，自動的に CPU-GPU 間の通信を行い Out-of-Core 学習を実現可能である．fine_granurality オプションは，これを True にするとレイヤーごとに細粒度で CPU-GPU 間の通信を行う設定である．

3.2 Chainer version3 への移植のための変更点

本節では，Chainer version2 の Out-of-Core 機能を Chainer version3 に移植し，複数 GPU 対応するために行った変更点について説明する．Version3 においても上記の基本構造は同様である．

3.2.1 逆伝播処理方法の変更への対応

Version2 ではすべての Variable_node が逆伝搬処理の関数 backward()を持っていたため，どの Variable_node からも逆伝搬処理を開始することができたが，Version3 では Variable オブジェクトの開放を積極的に行うために[16]，一部の Variable_node のみが bakward 関数を持つ事となり，逆伝播処理の実装方法が変更された．このため Out-of-Core 学習の実装に際しては，この変更への対応を行う必要があった．

3.2.2 マルチ GPU 対応

深層学習のデータ並列は多くの深層学習のフレームワークで実装されており，Chainer においても In-Core 学習で実行可能である．データ並列実行は各 GPU は，1GPU での実行と同じ処理を行うが，重み・バイアスの更新を複数 GPU の結果を用いて更新するものであり，実質的にバッチサイズを増加させたことに相当する．複数 GPU の結果を集約する部分は Reduction 演算であり，ここに比較的大きな通信が発生する．Out-of-Core 学習の Version 2 の Out-of-Core 実装では，1GPU で実行のみが実装されていた．このため，Version3 への移植に際して複数 GPU でのデータ並列で実行するための拡張を行った．各 GPU はそれぞれ CPU へデータ退避を行うため Out-of-Core 学習においても各 GPU の処理は 1GPU と変わらない．ただし，NUMA 構成になっている時は，CPU と GPU の Affinity が正しく設定されていないと CPU-GPU 間の通信が効率的に行われない．そのため，NVIDIA Management Library(NVML)の Python インターフェースである pynvml/py3nvml を利用し，CPU-GPU affinity を設定することとした．具体的な API としては nvmlDeviceSetCpuAffinity()を利用して各プロセスに各 GPU との Affinity を設定している．

4. 計測

4.1 計測対象ネットワーク

本稿では 3 章で述べた Chainer version3 を用いて Out-of-Core 学習の性能評価を行った．評価に利用するニューラルネットワークは GoogLeNet, ResNet50, ResNet152 を用いた．訓練画像として ImageNet[11]の画像を用いる．確率的勾配降下法においては計算効率向上のため訓練画像をいくつかのバッチに分けて，そのバッチごとに重みの更新を行う．本計測では，まずこれらのネットワークのバッチサイズを増加させることで In-Core 学習では実行できないバッチサイズを利用して Out-of-Core 学習を行う．バッチサイズを増加させても，様々なパラメータ調整を行うことで，学習は高速に進む報告もある[12]が，一般に大きすぎるバッチサイズの利用は推奨されていない[13]．そこで，それらのネットワークを簡易的な方法で拡大し，これらも評価に用いることにした．既存のネットワークの入力画像サイズは 224x224 となっているが，これを 2240x2240 と 100 倍に拡

大し，ImageNet の画像も 100 倍に拡大して用いる．GoogleNet，ResNet 等多くの Convolution Network が，前半が Convolution 層，後半が全結合層から構成されている．この内, Convolution 層だけを入力サイズに合わせて修正し，全結合層のサイズは変更しない． Convolution 層においては，入力サイズに比例して使用メモリー量と計算量が増加する．一方，全結合層においては，入力サイズの二乗に比例して，使用メモリー量と計算量が増加するので，入力サイズに合わせて全結合層を拡大することは現実的ではない．Convolution 層と全結合層を結合する層 (例えば，max pooling 層) において, 整合性を取るようにパラメータを調整する．GoogLeNet における変更例を以下に示す．

```
def __call__(self, x, t):
    h = F.relu(self.conv1(x))
    h = F.local_response_normalization(
        F.max_pooling_2d(h,  3, stride=2), n=5)

    h = F.relu(self.conv2_reduce(h))
    h = F.relu(self.conv2(h))

    h = F.max_pooling_2d(
        F.local_response_normalization(h,   n=5), 3, stride=2)

    h = self.inc3a(h)
    h = self.inc3b(h)
    h = F.max_pooling_2d(h,  3, stride=2)
    h = self.inc4a(h)

    l = F.average_pooling_2d(h,  5 * (GoogLeNet.insize  // 224),
                stride=3 * (GoogLeNet.insize  // 224))
    l = F.relu(self.loss1_conv(l))
    l = F.relu(self.loss1_fc1(l))                  プーリング層全結合層：
    l = self.loss1_fc2(l)                          プーリング層で調整。この範囲は拡大しない
    loss1 = F.softmax_cross_entropy(l, t)

    h = self.inc4b(h)
    h = self.inc4c(h)
    h = self.inc4d(h)

    l = F.average_pooling_2d(h,  5 * (GoogLeNet.insize  // 224),
                stride=3 * (GoogLeNet.insize  // 224))
    l = F.relu(self.loss2_conv(l))
    l = F.relu(self.loss2_fc1(l))                  プーリング層全結合層：
    l = self.loss2_fc2(l)                          プーリング層で調整。この範囲は拡大しない
    loss2 = F.softmax_cross_entropy(l, t)

    h = self.inc4e(h)
    h = F.max_pooling_2d(h,  3, stride=2)
    h = self.inc5a(h)
    h = self.inc5b(h)

    h = F.average_pooling_2d(h,  7 * (GoogLeNet.insize  // 224),
                stride=1 * (GoogLeNet.insize  // 224))
    h = self.loss3_fc(F.dropout(h, 0.4))
    loss3 = F.softmax_cross_entropy(h, t)          プーリング層全結合層：
                                                   プーリング層で調整。この範囲は拡大しない

    loss = 0.3 * (loss1 + loss2) + loss3
    accuracy = F.accuracy(h, t)
```

図 3 モデル修正例（GoogLeNet）

4.2 計測環境

計測マシンには IBM® Power Systems™ S822LC for High Performance Computing (通称” Minsky”, 以下 Minsky)と NVIDIA® DGX-1(以下,” DGX-1”)を用いた．Minsky は CPU に POWER8 プロセッサー(10 コア, 3.54GHz)を 2 基搭載し，CPU メモリー512GB であり，2 ソケットの NUMA 構成となっている．GPU は NVIDIA® Tesla® P100(GPU メモリー16GB)を 4 基搭載している. CPU-GPU 間は NVLink1.0(片方向 40GB/sec のバンド幅)で接続されている. OS は RHEL7.3, CUDA9, cuDNN7 を利用した．

NVIDIA® DGX-1™は CPU に Intel® Xeon®プロセッサー(20 コア，2.2GHz)を 2 基搭載され，CPU メモリー512GB であり, 2 ソケットの NUMA 構成となっている. GPU は NVIDIA Tesla P100(GPU メモリー16GB)を 8 基搭載している．CPU-GPU 間は PCIe(片方向 16GB/sec のバンド幅)で接続されている．OS は Ubuntu16.0.4.3LTS, CUDA8, cuDNN7 を利用した．図 4 に示すように DGX-1 の CPU-GPU 間の 1 リンクは 16GB/sec であり，PCIe スイッチを経由して接続されている．GPU0, 1, 2, 3 を利用した場合，CPU-GPU 間のバンド幅は 1GPU あたり 8GB/sec となり，GPU0, 2, 4, 6 を利用した場合は，1GPU あたり 16GB/sec となる．

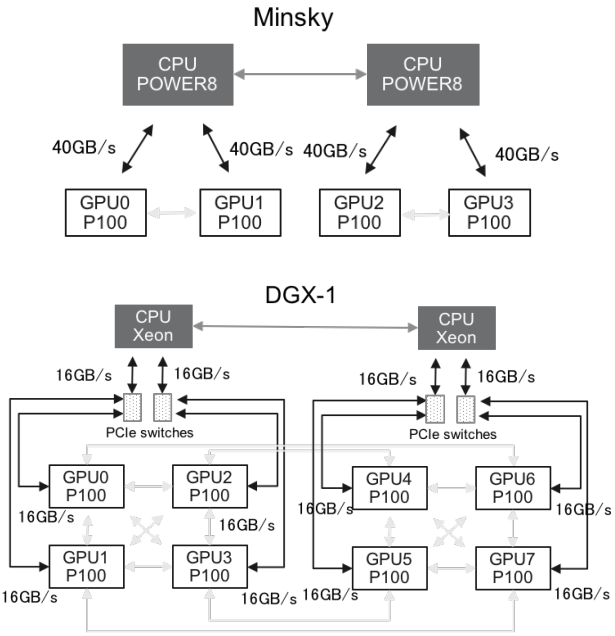


図 4 CPU-GPU 間のバンド幅（片方向）

4.3 計測結果

4.3.1 最大バッチサイズの増加

図 5 に GoogLeNet, ResNet50, ResNet152 のそれぞれについて，入力サイズ 224x224, 2240x2240 を用いた時のバッチサイズと 1 画像あたりの平均処理時間を示す． 100 バッチ分実行し, その実行時間を処理された画像数(バッチサイズ*実行バッチ数)で割って求めている．Minsky における結果である．図中には In-Core 学習(in-core), Out-of-Core 学習を CPU-GPU 間の通信を非同期に行った時(OoC)，同期して行う(OoC sync)の 3 つのケースを示している．全てのケースで CPU-GPU affinity は設定済みである．3.1 節で記述した fine_granurarity オプションは True とし，全てのレイヤーの処理に CPU-GPU 通信が発生する．

図より実行可能なバッチサイズは，In-Core 学習，OoC の非同期実行，OoC の同期実行の順に大きくなっている．表 1 に In-Core 学習に対するバッチサイズの増加率を示す. 非同期実行においては，3 章で示したように，現在演算中のレイヤーの前レイヤーのデータが GPU メモリーから解放される前に現在のレイヤーの計算が開始されていることや, 次のレイヤーのデータのプリフェッチにより，GPU に確保されるメモリー量が増加する. 一方, 同期実行においては, 現在のレイヤーにおいて利用されるデータ以外は確保されないことから，非同期実行の場合に対して，最大バッチサ

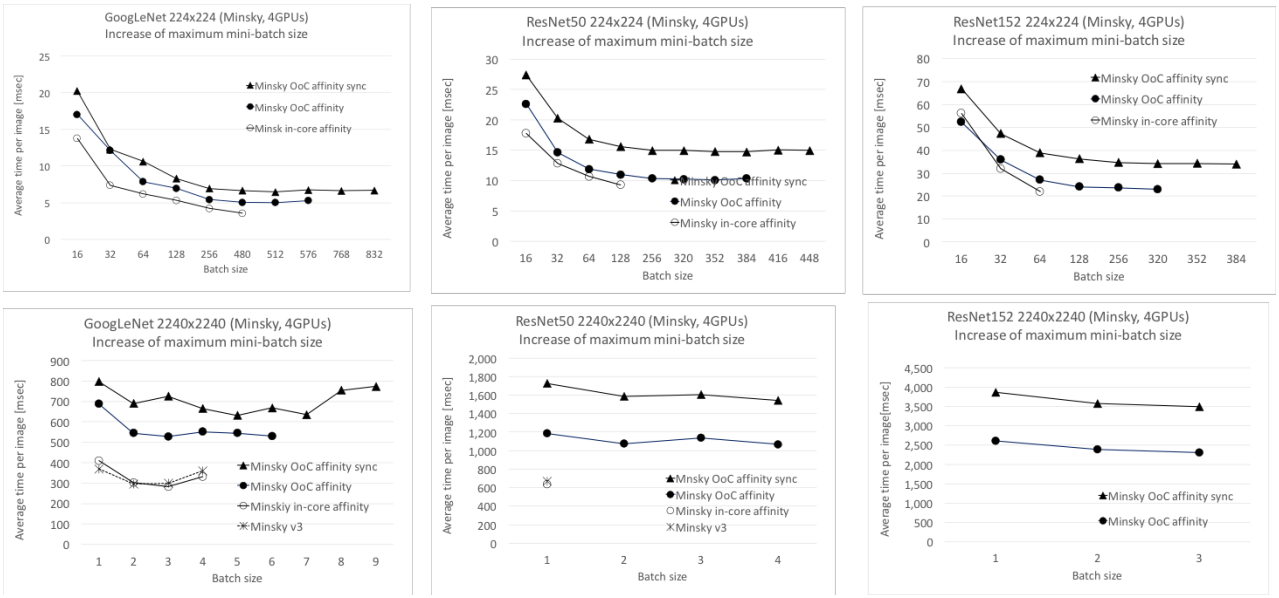


図 5 最大バッチサイズ

イズが大きくなっている．同期実行においては，最大バッチサイズは大きくなるが，大量の通信処理の待ちが発生するため，実行速度の低下が見られる．その低下はバッチサイズを十分大きくし，実行時間が安定した後で，GoogLeNet でおおよそ 20-30%，ResNet50, ResNet152 で 40-50%となっている．入力サイズ 224x224，2240x2240 で速度低下の割合は同様であった．

In-Core 学習に対する，OoC 実行のオーバーヘッドは 224x224 の GoogLeNet では 20-30%程度，ResNet50 と ResNet152 では 10-20%程度となっている．2240x2240 では GoogLeNet も ResNet50 も 70-80%程度である．2240x2240 においてオーバーヘッドが大きい理由は今後調査が必要である．

表 1 に In-Core 学習に対する実行可能な最大バッチサイズの増加率を示す。ResNet50 より ResNet152 と深いネットワークにおいてより増加率が高いことがわかる。入力サイズ 2240x2240 の ResNet152 においては In-Core ではバッチサイズ 1 でも実行できなかった．特にこのような大きなネットワークでは，Out-of-Core 学習は有益である．ResNet152 では，レイヤー数が 152 層と ResNet50 に対して 3 倍になり，In-Core 学習においてメモリー使用量は大幅に増加する．しかし，ResNet152 の OoC における最大バッチサイズは ResNet50 に対して 20-30%程度の低下に止まっている．これは，ResNet152 では，各レイヤーの構成は ResNet50 と同じで，各レイヤーでのメモリー使用量は大きくは変わらないためと考えられる．ResNet152 では退避されるメモリー量は増加し，入力サイズ 2240x2240 のバッチサイズ 3 の場合における CPU メモリー使用量は，ResNet50 は 101GB に対して ResNet152 は 208 GB と倍増していた．ResNet152 のパラメータサイズは ResNet50 の倍であるため，これは妥当な値である． このような深いネットワークにおける Out-of-Core 学習の利用は，より多く GPU メモリーを退避することができるため効果が大きい．一般に深層学習の実行では CPU リソースはあまり使われないことが多いが，Out-of-Core 学習ではそれを活用することができる．

入力 224x224 のケースでは，小さいバッチサイズでは，実行時間が遅くなっている．これは，GPU の計算量が少ないことからファイル IO や通信のオーバーヘッドの割合が大きくなっているためと考えられる．入力サイズ 2240x2240 の場合は，バッチサイズ 1 から計算量は多く，安定した結果を示している．2240x2240 の GoogLeNet と ResNet50 において，Chainer version 3 の正式版の結果(Minsky v3)を示した．実行時間は In-Core 学習とほぼ同じになっていることから，3 章で示した我々の実装による新たなオーバーヘッドはないことを示している．

表 1 In-Core 学習に対するバッチサイズの増加率

入力サイズ	ネットワーク	実行モード	増加率
224x224	GoogLeNet	OoC affinity	1.2
		OoC affinity sync	1.7
	ResNet50	OoC affinity	3.0
		OoC affinity sync	3.5
	ResNet152	OoC affinity	5.0
		OoC affinity sync	6.0
2240x2240	GoogLeNet	OoC affinity	1.5
		OoC affinity sync	2.3
	ResNet50	OoC affinity	4.0
		OoC affinity sync	4.0
	ResNet152	OoC affinity	In-Core 不可
		OoC affinity sync	In-Core 不可

4.3.2 CPU-GPU 間バンド幅の効果

図 6 にそれぞれのネットワークの入力サイズ 2240x2240, バッチサイズ 3 を利用した時の CPU-GPU 間のバンド幅の影響を示す．CPU-GPU 間のバンド幅を変えた時と，CPU-GPU affinity の有無の比較である. 片方向バンド幅は Minsky において 40GB/sec, DGX-1 で GPU0, 1, 2, 3 を利用した時 8GB/sec, GPU 0，2，4，6 を利用した時 16GB/sec である．

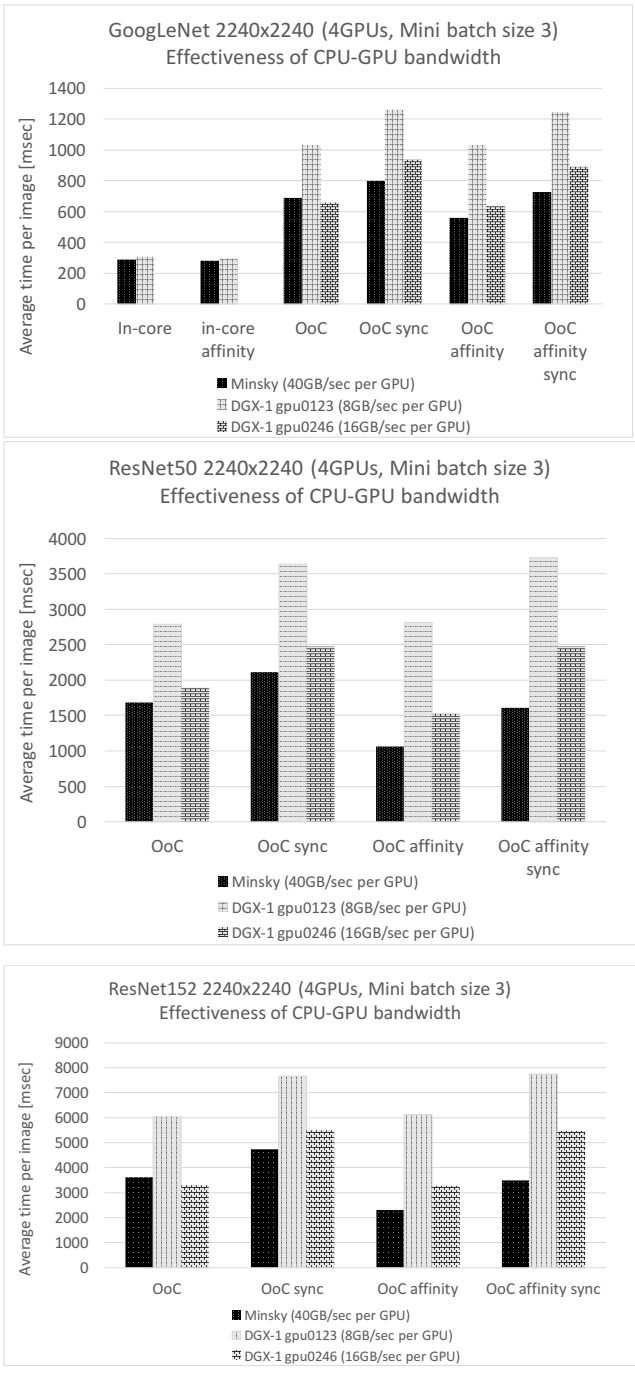


図 6　CPU-GPU 間バンド幅の性能への影響

In-Core 学習の時，Minsky，DGX-1 において，Minsky の方が多少速いが，ほぼ実行時間に違いは見られない．バッチサイズ 3 では GoogLeNet のみ In-Core 学習可能であるが，ResNet50 のバッチサイズ 1 においても同様の傾向を示していた．In-Core 学習においては，CPU-GPU 間の通信は入力データや Reduction 通信のみであり，通信の割合が小さいため大きな差は観測されなかったと考えられる. In-Core 学習においては，affinity の設定の有無でも性能向上は見られなかった．これも同様に CPU-GPU 間の通信が少ないためと考えられる．

Out-of-Core 学習における affinity 設定の有無での比較を行うと，Minsky では affinity を付けた場合の速度向上は GoogLeNet では 19 %，ResNet では 30%程度と大きいが，DGX-1 では速度向上は数%とほぼ速度向上は見られなかった．これは DGX-1 の NUMA ノード間が PCIe の速度に対して十分高速なため影響が現れない，もしくは，明示的にプログラム中で設定しなくても OS が正しく設定している可能性が考えられる．

表 2 に affinity 設定後の Out-of-Core 学習において, DGX-1 gpu0123 に対する Minsky の性能向上の加速率を示す．GoogLeNet においては 1.7 倍以上，ResNet においては 2.2 倍以上と大きな性能向上を示している．Out-of-Core 学習において CPU-GPU バンド幅の重要であることを確認した．

表 2　CPU-GPU NVLink による性能向上率

ネットワーク	実行モード	加速率
GoogLeNet	OoC affinity	1.84
	OoC affinity sync	1.71
ResNet50	OoC affinity	2.64
	OoC affinity sync	2.32
ResNet152	OoC affinity	2.66
	OoC affinity sync	2.22

5. おわりに

本稿では, Chainer version3 に Out-of-Core 学習を実装し，それを用いて，Out-of-Core 学習の性能の性能評価を行なった．Out-of-Core 学習を用いることで，2 倍から 6 倍のバッチサイズを用いた学習を実現することを確認した．ResNet152 では通常では実行できないケースについても実行可能となることを確認した. また, CPU-GPU 間に NVLink の高速な通信を持つ Minsky において，最大 2.66 倍の性能向上を確認し，Out-of-Core 学習における CPU-GPU 間のバンド幅の重要性を確認した．

現状の実装においては, まだ Out-of-Core 学習のオーバーヘッドが大きいという課題がある．また，様々な実行モードが存在し，それらにはメモリー使用量と実行時間のトレードオフがある．これらについてもネットワークやデータに基づく最適な実行モードの設定を行うことでより効果的な実行ができると考えられる．本稿では，既存のネットワークを拡大するということで大規模なネットワークを作成したが，より実用的なデータとネットワークを利用した評価も必要である．今後はこれらの課題について検討を進め

る予定である．

参考文献

[1] ImageNet Large Scale Visual Recognition Competition: http://www.image-net.org/challenges/LSVRC/
[2] Alex Krizhevsky, Ilya Sutskever, and Geo rey E. Hinton. 2012. ImageNet Classification with Deep Convolutional Neural Networks. In International Conference on Neural Information Processing Systems. 1097–1105.
[3] Karen Simonyan and Andrew Zisserman. 2014. Very Deep Convolutional Networks for Large-Scale Image Recognition. arXiv preprint arXiv:1409.1556 (2014).
[4] Christian Szegedy, Wei Liu, Yangqing Jia, Pierre Sermanet, Scott E. Reed, Dragomir Anguelov, Dumitru Erhan, Vincent Vanhoucke, and Andrew Rabi- novich. 2015. Going deeper with convolutions. In IEEE Conference on Computer Vision and Pattern Recognition. 1–9.
[5] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. 2015. Deep Residual Learning for Image Recognition. CoRR abs/1512.03385 (2015). http://arxiv.org/ abs/1512.03385
[6] Jose Dolz, Christian Desrosiers, Ismail Ben Ayed, 3D fully convolutional networks for subcortical segmentation in MRI: A large-scale study, In NeuroImage, 2017, , ISSN 1053-8119, https://doi.org/10.1016/j.neuroimage.2017.04.039.
[7] K. Shirahata, Y. Tomita and A. Ike, "Memory reduction method for deep neural network training," *2016 IEEE 26th International Workshop on Machine Learning for Signal Processing (MLSP)*, Vietri sul Mare, 2016, pp. 1-6. doi: 10.1109/MLSP.2016.7738869
[8] Jeffrey Dean, Greg S. Corrado, Rajat Monga, Kai Chen, Matthieu Devin, Quoc V. Le, Mark Z. Mao, Marc'Aurelio Ranzato, Andrew Senior, Paul Tucker, Ke Yang, and Andrew Y. Ng. 2012. Large scale distributed deep networks. In *Proceedings of the 25th International Conference on Neural Information Processing Systems - Volume 1* (NIPS'12), F. Pereira, C. J. C. Burges, L. Bottou, and K. Q. Weinberger (Eds.), Vol. 1. Curran Associates Inc., USA, 1223-1231.
[9] Alex Krizhevsky. One weird trick for parallelizing convolutional neural networks. CoRR abs/1404.5997 (2014). http://arxiv.org/abs/1404.5997
[10] Rhu, Minsoo & Gimelshein, Natalia & Clemons, Jason & Zulfiqar, Arslan & W. Keckler, Stephen. (2016). vDNN: Virtualized deep neural networks for scalable, memory-efficient neural network design. 1-13. 10.1109/MICRO.2016.7783721.
[11] ImageNet: http://www.image-net.org/
[12] Priya Goyal, Piotr Dollár, Ross Girshick, Pieter Noordhuis, Lukasz Wesolowski, Aapo Kyrola, Andrew Tulloch, Yangqing Jia, Kaiming He. Accurate, Large Minibatch SGD: Training ImageNet in 1 Hour. CoRR abs/1706.02677. 2017. http://arxiv.org/abs/1706.02677
[13] Hinton G.E. (2012) A Practical Guide to Training Restricted Boltzmann Machines. In: Montavon G., Orr G.B., Müller KR. (eds) Neural Networks: Tricks of the Trade. Lecture Notes in Computer Science, vol 7700. Springer, Berlin, Heidelberg
[14] Chainer version 2 Out-of-Core 学習用派生レポジトリー: https://github.com/anaruse/chainer/tree/OOC_chainer_v202
[15] Cupy version 1 Out-of-Core 学習用派生レポジトリ: https://github.com/anaruse/cupy/tree/OOC_cupy_v102
[16] Chainer issues: Aggressive buffer release #2368: https://github.com/chainer/chainer/issues/2368