Vol.2017-HPC-160 No.38
2017/7/28

ooc_cuDNN: GPU計算機のメモリ階層を利用した大規模深層学習ライブラリの開発

伊藤　祐貴[1,a)]　松宮　遼[1]　遠藤　敏夫[1]

概要：畳み込みニューラルネットワーク (CNN) による深層学習は計算量が非常に大きく，その計算は GPU を用いることで汎用 CPU を用いた場合よりも高速に行われることが示されている．しかし，GPU メモリ容量が小さいために，ネットワークや画像サイズが大きい CNN を GPU で高速に計算するプログラムを実装することは困難である．本論文では CPU メモリの容量を活用して GPU メモリ容量を超える CNN の計算を可能とする out of core cuDNN(ooc_cuDNN) ライブラリの設計と実装を述べる．ooc_cuDNN は，深層学習ライブラリの cuDNN を拡張したものであり，性能モデルに基づいた計算の分割により大規模問題を高速に計算することを可能とする．ooc_cuDNN を利用することで，60 GB 以上のメモリを必要とする CNN の計算が，メモリ容量 16 GB の GPU 一台で可能となり，その際のオーバーヘッドは 13 %であった．

1. はじめに

近年，画像認識 [1] や音声認識 [2] などの分野にニューラルネットを適用することにより，高い精度が示されている．ニューラルネットワークとは，人間の脳の学習メカニズムをモデルにしたアルゴリズムである. その中でも，主に畳み込み演算を使用するものは畳み込みニューラルネットワーク (CNN) と呼ばれる．CNN は画像認識 [3] と画像処理 (超解像化 [4] や画像生成 [5] など) において用いられることが多い．

CNN アルゴリズムの計算は計算量が多いが，GPU を用いることで高速に行うことができる．GPU で CNN の計算を行う手法としては，NVIDIA が開発した深層学習ライブラリ cuDNN[6] を使用することが挙げられる．しかし，cuDNN を使用して計算できる CNN の大きさは GPU のメモリ容量によって制限されてしまう．例として，バッチサイズを 1024 とした場合の VGG16[7] では 60 GB 以上のメモリを必要とする．これに対して NVIDIA の Tesla P100 のメモリ容量は 16 GB しかない．そのため，Tesla P100 一台ではこの CNN の計算を cuDNN を用いて行うことは困難である．

GPU メモリ容量を超えるデータを処理するためには，データの一部をより広大な CPU メモリに退避させればよい．しかし，CPU・GPU 間の通信性能が GPU メモリ性能より低いことを考慮する必要がある．たとえば，Tesla P100 GPU を PCI Express 3.0(Gen3) で接続した計算機では，GPU メモリバンド幅 732 GB/sec に対して CPU・GPU 間通信バンド幅は片方向 8GB/sec に限定される．これに起因する通信オーバヘッドをいかに隠蔽するかが課題となる．また，通信をアプリケーションレベルで記述すると，通信の制御をするためにソースコードも複雑になる．

本論文では cuDNN を拡張したライブラリである out of core cuDNN(ooc_cuDNN) の設計と実装を述べる．ooc_cuDNN は CPU メモリを使用し，かつ計算を分割することにより GPU メモリ容量を越える CNN の計算を可能とする．ooc_cuDNN では高速化のために，性能モデルから適切な分割サイズを求め，パイプライン処理をすることで通信によるオーバーヘッドを小さくする．また, 計算をより効率的に行うために，複数の計算をまとめて行うライブラリ関数も新たに用意する．

ooc_cuDNN を利用することで，60 GB 以上のメモリを必要とする CNN の計算がメモリ容量 16 GB の GPU 一台で可能となった．また，その際のオーバーヘッドは 13 %であった．

本研究の貢献を以下にまとめる．

- GPU メモリ容量を超える CNN の問題を計算する手法の提案
- CPU・GPU 間通信を考慮した CNN の計算の性能モデルの構築
- 性能モデルに基づいた計算の分割とパイプラインによる高速化
- 計算を複合して行うことによる高速化

[1] 東京工業大学
[a)] itou.y.aj@m.titech.ac.jp

© 2017 Information Processing Society of Japan

2. 背景

2.1 CNN

2.1.1 CNN の構造

CNN は複数の層から構成され，各層はさらに複数の特徴マップから構成されている．ここで，特徴マップとは前の層の出力を入力として受け取り，計算を行うことで出力されるデータである．本論文ではある層に含まれる特徴マップの数をその層のチャンネル数と呼ぶこととする．

層の種類は計算の内容によって畳み込み層やプーリング層, 全結合層などに分類される．また, 畳み込み層や全結合層では計算のために重みフィルタなどのパラメータを必要とする．そして, CNN の学習ではそれらのパラメータが最適化対象となる．

2.1.2 順伝播と逆伝播

CNN の学習は誤差逆伝播法と呼ばれるアルゴリズムを用いて行われる．誤差逆伝播法とは CNN の出力と正解データの誤差を用いてパラメータの更新を行う学習法である．そのため, 誤差逆伝播法では以下の 3 つの処理を行う．

(1) 学習のためのサンプルデータをネットワークの入力として, 各層の特徴マップを計算する [順伝播]
(2) ネットワークの出力と正解データの誤差を用いて, 各層の特徴マップや重みフィルタなどに対する勾配を求める [逆伝播]
(3) 計算した勾配を用いて, パラメータを更新する

ここで, 各層の特徴マップは前の層の特徴マップから計算されるため, 順伝播の計算は入力層から出力層へと 1 層ずつ行われる．一方, 各層における勾配は後の層の勾配から計算されるため, 逆伝播の計算は出力層から入力層へと順に行われる．

また, 一般的に勾配の計算では順伝播で計算した特徴マップも入力として用いる．よって, 順伝播で一度計算した特徴マップのデータは逆伝播で使用されるまで保持される必要がある．

2.1.3 畳み込み層の計算

各層における計算の内容は層の種類によって異なる．

以下では CNN において代表的な層である畳み込み層について述べる．畳込み層の計算はさらに「重みフィルタを用いた画像の畳み込み計算」,「バイアスの加算」,「活性化」の 3 つの計算から成り立つ．

層 X を重みフィルタ W で層 Y に畳み込む計算について説明する．畳み込み計算のパラメータを表 1 に示す．X と Y はそれぞれ c_X，c_Y 枚の特徴マップを持つ．図 1 のように層 X と層 Y の特徴マップは全結合されており，結合している特徴マップの組み合わせでそれぞれ畳み込み計算を行う．特徴マップの組み合わせごとの計算では, X の特徴マップの一部分とフィルタの各点についての内積が Y の特徴マップの 1 点の値となる．この処理を Y の特徴マップのすべての点について行う．

表 1 層 X を重みフィルタ W で層 Y に畳み込む計算のパラメータ

パラメータ	意味
c_X	X のチャンネル数
h_X	X の特徴マップの高さ
w_X	X の特徴マップの幅
c_Y	Y のチャンネル数
h_Y	Y の特徴マップの高さ
w_Y	Y の特徴マップの幅
r_W	W のフィルタの高さ
s_W	W のフィルタの幅
n	バッチサイズ
d	データ 1 要素のサイズ [Byte]

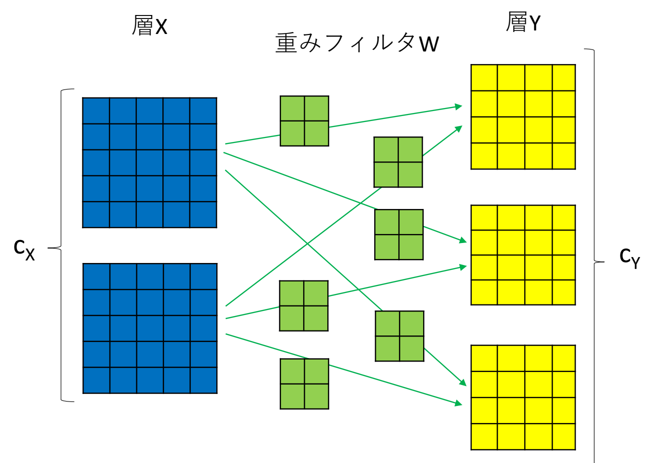

図 1 層 X と層 Y の結合 (バッチサイズ $n = 1$)

バイアスの加算では層の各点にバイアスと呼ばれる値を加える．また, 活性化では層の各点に対して活性化関数を適用する．活性化関数としてはシグモイド関数などの非線形関数が用いられる．

2.1.4 バッチ処理

画像 1 枚を入力として CNN の計算を GPU で行う場合，小規模な CNN では並列度が低いため，GPU の演算性能を活かせないことがある．そのため，複数の入力画像をまとめて CNN への入力とするバッチ処理を行う．ひとまとめにした画像の数をバッチサイズと呼び，以下では n とする．

2.1.5 CNN のメモリ使用量

各層は [バッチサイズ] × [チャンネル数] × [特徴マップの高さ] × [特徴マップの幅] の 4 次元で構成される．データ 1 要素のサイズを d Byte とすると，層 X のために必要なメモリ量は $(n \times c_X \times h_X \times w_X) \times d$ Byte となる．つまり，層あたりのメモリ使用量はバッチサイズ，チャンネル数，特徴マップのサイズに比例して増加する．また, 特徴マップだけでなく, 重みフィルタなどのパラメータのためのメモリも確保する必要がある．

CNN では逆伝播で使用するために層ごとに特徴マップを保持する必要がある．つまり, CNN 全体のメモリ使用量

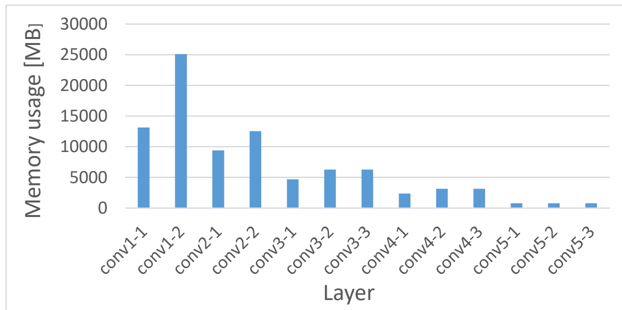

図 2 VGG16 の各畳み込み層の計算に必要なメモリ量

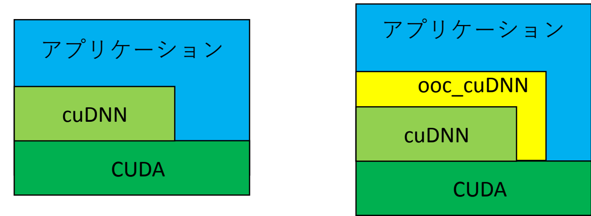

図 3 cuDNN を用いる場合のソフトウェアスタック (左) と ooc_cuDNN を用いる場合のソフトウェアスタック (右)

は層数によっても変化する．そのため，大規模な CNN の計算を行うためには多くのメモリが必要となる．

例として，バッチサイズ $n = 1024$ とした場合の VGG16 ネットワーク [7] の各畳み込み層の順伝播計算に必要なメモリ量を図 2 に示す．一層の計算におけるメモリ使用量で見ても，conv1-2 という層では最大で 24 GB 以上のメモリを必要としている．一方，Tesla P100 のメモリ容量は 16 GB しかない．これは一つの層の計算だけでも GPU メモリが足りなくなる場合があることを表している．

2.2 cuDNN

cuDNN は NVIDIA が開発した，深層学習を GPU で行うためのライブラリである．Caffe[8] や TensorFlow[9]，Chainer[10] など多くの機械学習フレームワークにおいても，GPU での計算の高速化のために cuDNN が使用されている．なお，本論文ではアプリケーションは C++で記述されており，cuDNN および CUDA ライブラリを直接呼ぶことを想定する．このとき，アプリケーションのソフトウェアスタックは図 3 の左側のようになる．

cuDNN には CNN の計算のための各種ライブラリ関数が用意されている．例として，画像の畳み込みを行う `cudnnConvolutionForward()` 関数，バイアスの加算を行う `cudnnAddTensor()` 関数，層の活性化を行う `cudnnActivationForward()` 関数がある．

一方，cuDNN の各関数は入力・出力ともに GPU 上のデータしか扱うことができない．そのため，cuDNN を使用するには各層の特徴マップや重みフィルタなどのデータをすべて GPU メモリ上に配置する必要がある．

3. GPU メモリ容量を超える深層学習

GPU メモリ容量を超える CNN の計算をするために，本論文では cuDNN を拡張した *ooc_cuDNN* (*out of core cuDNN*) を提案する．バッチサイズ，チャンネル数，特徴マップのサイズが大きいために一層の計算で GPU メモリ容量を超えてしまう場合でも，ooc_cuDNN では GPU での計算を可能とする．また，原則的には ooc_cuDNN のライブラリ関数は cuDNN のライブラリ関数と互換性がある．

ooc_cuDNN を用いたアプリケーションのソフトウェアスタックは図 3 の右側のようになる． また，3.1 節で示すように ooc_cuDNN ではメモリ管理のために一部の CUDA 関数も拡張する．

ooc_cuDNN の基本的な設計方針は以下のようになる．

(1) GPU メモリ上に配置することができないデータは CPU メモリ上に配置する
(2) 計算は GPU で cuDNN の関数を使用して行う
(3) 層のバッチサイズ，チャンネル数，特徴マップのサイズを分割して一部分ずつ計算を行う
(4) CPU メモリ上にあるデータが計算に必要となる場合，CPU・GPU 間で通信を行う

また，通信によるオーバーヘッドを小さくするため，計算と通信をパイプライン処理する．

3.1 メモリ管理関数

ooc_cuDNN ではメモリ確保のために `ooc_cudaMalloc()` 関数を提供する．`ooc_cudaMalloc()` 関数ではデータを GPU メモリ上で確保しようとするときにメモリ容量が足りない場合，代わりに `cudaMallocHost()` 関数で CPU メモリを確保する．このとき，GPU メモリの容量に空きがあったとしても，一部だけ GPU メモリで確保するということはせずにすべて CPU メモリで確保する．

また，データが CPU メモリ・GPU メモリのどちらに配置されるかはアプリケーションの実行時に決まる．したがって，ooc_cuDNN では CPU・GPU 通信関数として `ooc_cudaMemcpy()` 関数を提供する．この関数では通信方向の指定を `cudaMemcpyDefault` として `cudaMemcpy()` 関数を実行する．

3.2 計算関数

ooc_cuDNN の計算関数は入力・出力データがそれぞれ CPU メモリ・GPU メモリのどちらにあっても対応することができる．計算関数を実行するとき，データが CPU メモリ上にある場合は計算時にデータの分割を行う．

計算時の層の分割ではバッチサイズ，チャンネル数，特

© 2017 Information Processing Society of Japan

表 2 畳み込み計算時の分割サイズのパラメータ

パラメータ	意味
d_n	一分割あたりのバッチサイズ
d_c_Y	一分割あたりの Y のチャンネル数
d_c_X	一分割あたりの X のチャンネル数
d_h_Y	一分割あたりの Y の特徴マップの高さ
d_h_X	一分割あたりの X の特徴マップの高さ (d_h_Y と r_W から決まる)

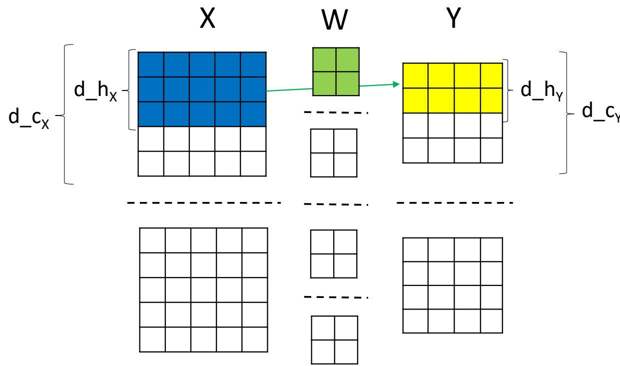

図 4 畳み込み計算の分割 ($d_n = 1$ の場合)

徴マップのサイズそれぞれについて分割する．現在の設計ではメモリアクセスの局所性を考慮して，特徴マップのサイズの分割は高さについてのみ行うこととする．重みフィルタの分割では出力チャンネル数，入力チャンネル数についてのみ分割する．

計算関数の入出力データが CPU メモリ上にある場合，CPU・GPU 間通信が必要となる．また，その場合 GPU メモリ上に一時的にデータを置くためのバッファを確保する必要がある．入出力データが元々 GPU メモリ上にある場合は，それらのデータを直接扱うことができるので通信やバッファは不要である． また，CPU・GPU 間の通信によるオーバヘッドを小さくするために, 計算と通信をパイプライン処理する．パイプライン化は CUDA の stream 機能を用いて行う．

以下では, ooc_cuDNN における畳み込みの順伝播計算について述べる．

層 X と重みフィルタ W を入力として，層 Y を出力する畳み込み計算を行う．計算の分割は n，c_Y，c_X，h_Y について行う．計算時の分割サイズのパラメータを表 2 に示す．また, 計算の分割の例について図 4 に示す．図 4 のように X，W，Y はそれぞれ分割され，X および W の一部から Y の一部が計算されることになる．

この畳み込み計算の実装について述べる．d_X，d_W，d_Y をそれぞれ X，W，Y の計算用のバッファとして GPU メモリ上に配置する．また, n，c_Y，c_X，h_Y について分割するため，計算はそれぞれに対応した 4 重ループで行う．各ループのイテレーション回数はそれぞれ $\lceil \frac{n}{d_n} \rceil$，$\lceil \frac{c_Y}{d_c_Y} \rceil$，$\lceil \frac{c_X}{d_c_X} \rceil$，$\lceil \frac{h_Y}{d_h_Y} \rceil$ である．ループ内では以下の 3 つの処理 (HtoD，Convolution，DtoH) を行い，Y の一部分を計算していく．

(1) HtoD ：X が CPU メモリ上にある場合，計算に対応する X の一部分を d_X へとコピーする．W についても同様の処理を行う．

(2) Convolution ： GPU 上で d_X，d_W を入力として，d_Y を出力する畳み込み計算を `cudnnConvolutionForward()` 関数を用いて行う

(3) DtoH : Y が CPU メモリ上にある場合，d_Y から Y の計算に対応した部分へとコピーする

X が GPU メモリ上にある場合は計算に対応する X の一部分を d_X として扱う．W，Y についても同様である．

ここで, 通信回数を考慮してループの順番は外側から n，c_Y，h_Y，c_X についてのループとした．そして, HtoD と Convolution は c_X についてのループ, DtoH は h_Y についてのループで行う．

通信のオーバーヘッドを隠すために，パイプライン処理により HtoD，Convolution，DtoH をオーバーラップして行う．このパイプライン処理のタイムラインの例を図 5 に示す．h_Y についてのループの 1 回分を 1 ステップとして，3 つの stream でパイプライン処理する．

現在, ooc_cuDNN では VGG ネットワークの順伝播・逆伝播に必要な CNN 関数が実装済である． 畳み込み層の逆伝播関数, およびバイアスの加算や活性化・プーリングなどの順伝播・逆伝播関数についても上記と同様の手法で設計・実装した．

4. 性能モデルによる最適化

ooc_cuDNN ではバッチサイズ，チャンネル数，特徴マップのサイズについてデータを分割して計算を行う．その際の性能は各分割サイズに大きく影響を受ける．また，分割サイズは計算時に GPU メモリ容量を越えないようなものでなければならない．より適切な分割サイズを決めるため，関数ごとに性能モデルを構築し，そのモデルに基づいて最適化を行った．

4.1 節では通信の性能モデル，4.2 節では cuDNN の性能モデルについて説明する．それらの結果を用いて，4.3 節では ooc_cuDNN の性能モデルについて述べる．また，4.4 節ではその性能モデルを用いた最適化について述べる．

4.1 通信の性能モデル

CPU・GPU 間の通信の性能については通信するデータサイズの線形モデルで表し，以下のような式とする．

$$t_{HtoD} = \alpha_{HtoD} \times (\text{通信するデータサイズ}) + \beta_{HtoD}$$

ここで係数 α_{HtoD} は 1 Byte あたりのメモリ転送時間，β_{HtoD} は通信のレイテンシを表す．これらの係数はハードウェアによって決まる．ooc_cuDNN では α_{HtoD} と β_{HtoD}

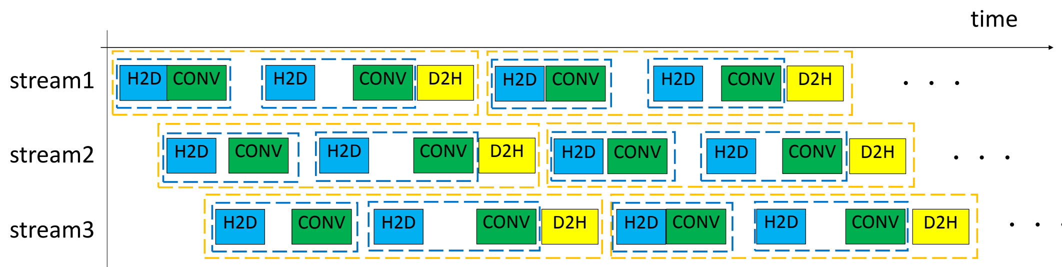

図 5　畳み込み計算のパイプライン処理のタイムライン (青の点線で囲まれた部分が c_X ループ，オレンジの点線で囲まれた部分が h_Y ループの 1 イテレーションに相当する)

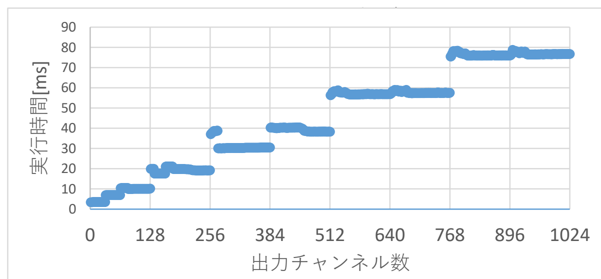

図 6　cuDNN を用いた畳み込み計算の実行時間

は予備実験から推定したものを用いる．

同様に GPU から CPU への通信の性能モデルは以下の式になる．

$$t_{DtoH} = \alpha_{DtoH} \times (\text{通信するデータサイズ}) + \beta_{DtoH}$$

4.2　cuDNN の性能モデル

cuDNN の性能モデルは計算関数ごとに構築した．各計算の性能モデルは計算量についての線形モデルで表す．

以下では, cuDNN を用いた畳み込み計算の性能モデルについて述べる．畳み込みの計算量は $O(n \times c_Y \times h_Y \times w_Y \times c_X \times r_W \times s_W)$ である．

cuDNN を用いた畳み込み計算の性能を解析するために予備実験を行った．予備実験では Tesla P100 で cudnnConvolutionForward() 関数の実行時間を c_Y を変化させて測定した． 結果を図 6 に示す．

この予備実験の結果，cuDNN を用いた畳み込み計算の実行時間は出力チャンネル数 c_Y に対して線形ではないことがわかった．図 6 では実行時間が大きく増加する c_Y は多くの場合 32 の倍数であった．簡単のため，および c_Y の値が小さいときの誤差を小さくするために, 現在の設計では c_Y が 32 増加するごとに実行時間が増加するものとして扱う．よって，畳み込み計算の性能モデルは以下のようになる．

$$t_{conv} = \alpha_{conv} \times \{n \times \left(32 \times \left\lceil \frac{c_Y}{32} \right\rceil\right) \times h_Y \times w_Y \times c_X \times r_W \times s_W\} + \beta_{conv}$$

α_{conv} と β_{conv} は通信と同様に予備実験から推定したものを用いる．

cuDNN における畳み込み計算以外の関数の性能モデルについても上記と同様の手法で構築した．

4.3　ooc_cuDNN の性能モデル

ooc_cuDNN の各性能モデルは 3.2 節での設計・実装に基づいて構築した．

以下では, ooc_cuDNN を用いた畳み込みの順伝播計算の性能モデルについて述べる．まず，d_X，d_W，d_Y のデータサイズ $memory_{d_X}$，$memory_{d_W}$，$memory_{d_Y}$ はそれぞれ以下の式で計算される．

$$memory_{d_X} = (d_n \times d_c_X \times d_h_X \times w_X) \times d \quad (1)$$

$$memory_{d_W} = (d_c_Y \times d_c_X \times r_W \times s_W) \times d \quad (2)$$

$$memory_{d_Y} = (d_n \times d_c_Y \times d_h_Y \times w_Y) \times d \quad (3)$$

このとき, 1 回の HtoD にかかる時間 t_{HtoD}, および DtoH にかかる時間 t_{DtoH} はそれぞれ以下の式で表される．

$$t_{HtoD} = (\alpha_{HtoD} \times memory_{d_X} + \beta_{HtoD}) \times p_X + (\alpha_{HtoD} \times memory_{d_W} + \beta_{HtoD}) \times p_W$$

$$t_{DtoH} = (\alpha_{DtoH} \times memory_{d_Y} + \beta_{DtoH}) \times p_Y$$

ここで，p_X は X が CPU メモリと GPU メモリのどちらにあるかを表すパラメータである．p_X の値は X が CPU メモリ上にあれば 1，GPU メモリ上にあれば 0 である．p_W，p_Y についても同様である．

また，1 回の Convolution の実行時間 t_{conv} は 4.2 節より以下の式で表される．

$$t_{conv} = \alpha_{conv} \times \{d_n \times \left(32 \times \left\lceil \frac{d_c_Y}{32} \right\rceil\right) \times d_h_Y \times w_Y \times d_c_X \times r_W \times s_W\} + \beta_{conv}$$

層全体の畳み込み計算の実行時間 T_{conv} は, パイプライン処理を行っているため t_{HtoD}，t_{conv}，t_{DtoH} の大小関係によって決まる．畳み込み計算のパイプラインの 1 ステップでは HtoD と Convolution を $\left\lceil \frac{c_X}{d_c_X} \right\rceil$ 回ずつと DtoH を 1 回行うため，$\left\lceil \frac{c_X}{d_c_X} \right\rceil t_{HtoD}$，$\left\lceil \frac{c_X}{d_c_X} \right\rceil t_{conv}$，$t_{DtoH}$ の大小関係で場合分けをする．また, 計算全体ではこのパイプライ

ン処理を $\left(\left\lceil\frac{n}{d_n}\right\rceil\left\lceil\frac{c_Y}{d_c_Y}\right\rceil\left\lceil\frac{h_Y}{d_h_Y}\right\rceil\right)$ ステップ行う．よって，T_{conv} は以下の式で表される．

$$\begin{aligned}T_{conv} &= t_{HtoD} + t_{conv} + t_{DtoH}\\ &+ \left(\left\lceil\frac{c_X}{d_c_X}\right\rceil - 1\right) max(t_{HtoD}, t_{conv})\\ &+ \left(\left\lceil\frac{n}{d_n}\right\rceil\left\lceil\frac{c_Y}{d_c_Y}\right\rceil\left\lceil\frac{h_Y}{d_h_Y}\right\rceil - 1\right)\\ &\times max\left(\left\lceil\frac{c_X}{d_c_X}\right\rceil t_{HtoD}, \left\lceil\frac{c_X}{d_c_X}\right\rceil t_{conv}, t_{DtoH}\right)\end{aligned}$$

ooc_cuDNN における畳み込み計算以外の関数の性能モデルについても上記と同様の手法で構築した．

4.4 性能モデルによる分割サイズの最適化

以上の性能モデルを用いて各分割サイズの最適化を行う．この最適化では GPU メモリの空き容量による制約の下で，性能モデルの値がより小さくなる分割サイズを探索する．

畳み込みでは d_n，d_c_Y，d_c_X，d_h_Y の 4 つのパラメータが最適化対象である．そのため，全探索した場合の計算量が大きくなる．そこで，以下のようなヒューリスティクスを導入することで計算量を削減する．

(1) 各パラメータについて分割しない状態を初期状態とする ($d_n = n$，$d_c_Y = c_Y$，$d_c_X = c_X$，$d_h_Y = h_Y$)
(2) まだ，選ばれていないパラメータから一つを選ぶ (すべてのパラメータが選ばれていたならば終了)
(3) 選んだパラメータの分割サイズについて探索する．その中で性能モデルの値が最小となった分割サイズをそのパラメータの分割サイズとする
(4) (2) へ戻る

このように探索した場合，全探索と比較して探索範囲が小さくなってしまう．そのため，より適切なパラメータを求めるには探索するパラメータの順番が重要となる．ooc_cuDNN では分割した場合に発生するオーバヘッドを考慮して，パラメータを d_n，d_h_Y，d_c_X，d_c_Y の順に決めることとした．

他の計算でも同様のヒューリスティクスを用いて分割サイズを決定する．

5. CNN 計算の複合

これまでに述べた設計と最適化により，ooc_cuDNN は効率的な大規模計算を畳み込みのように計算量が大きい計算では実現できる．しかしながら，6.3 節で後述するように，ooc_cuDNN でバイアスの加算・活性化・プーリングのような計算量が小さい計算をそれぞれ単体で実行した場合，cuDNN と比較して性能が大きく低下してしまう．これは計算量が小さい場合，通信を隠蔽するのに十分な計算時間を確保できないためである．

通信によるオーバーヘッドを小さくするためには大きな計算量が必要となる．一方，多くの CNN アプリケーションでは計算量が大きい畳み込み計算とバイアスの加算や活性化，およびプーリングは続けて行われる．そこで，ooc_cuDNN ではそれらの計算をまとめて行う複合関数を新たに用意する．

以下では畳み込み・バイアスの加算・活性化の複合関数を例にして説明する． この 3 つの計算を別々に行う場合，図 7 の上側のように各計算で CPU・GPU 間で通信を行う必要がある． これに対して複合関数を用いる場合では，図 7 の下側のように一度の通信に対して 3 つの計算をまとめて行う．そのため，畳み込み計算と同じ通信回数で 3 つの計算を行うことができる．そして，この複合関数では畳み込み計算によって通信のオーバーヘッドが隠された上で，バイアスの加算・活性化を行うことができると期待される．

既存のアプリケーションでこの複合関数を使用する場合，ソースコードの書き換えは図 8 のようになる．この書き換えは多くのアプリケーションにおいては局所的な書き換えで済むと期待される．

6. 実験と評価

ooc_cuDNN を用いて実験を行い，性能を評価した．実験環境を表 3 に示す．この環境では GPU メモリ容量は 16 GB である．ooc_cuDNN によってこの容量を超える問題サイズの CNN の計算が可能となることを実証した．

ooc_cuDNN を使用するときは，特徴マップはすべて

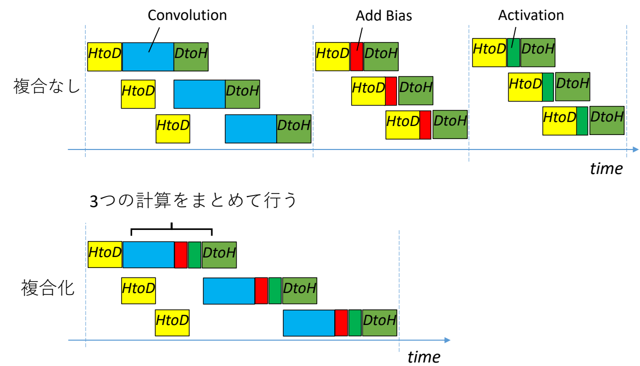

図 7 畳み込み・バイアスの加算・活性化を別々に行う場合のタイムライン (上) と複合して行う場合のタイムライン (下)

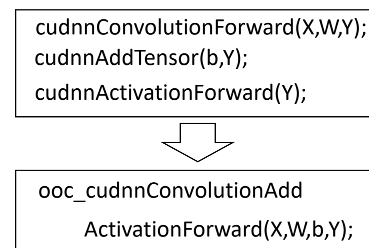

図 8 畳み込み・バイアスの加算・活性化を cuDNN で行う場合のソースコード (上) と ooc_cuDNN で複合関数を用いて行う場合のソースコード (下)

表 3 実験環境

GPU	Tesla P100 (Pascal 世代)
GPU メモリ容量	16 GB
CPU	Intel Xeon E5-2680v4 (Broadwell 世代)
CPU メモリ容量	128 GB
PCI-Express	gen3 x16
OS	CentOS Linux 7.2
CUDA	CUDA 8.0
cuDNN	cuDNN 5.0

表 4 分割サイズ最適化についての実験結果 [GFLOPS]

問題サイズ	(1)	(2)	(3)
提案最適化手法	2574	6787	2681
局所探索法	2522	5142	2383
実際の最適パラメータ使用	2654	7024	2721

CPU メモリ上，重みフィルタとバイアスは GPU メモリ上に配置した．これは重みフィルタとバイアスは特徴マップと比較して小さく，実際の CNN の計算の際に GPU メモリに十分収まると考えられるからである．また，cuDNN との比較も行った．

6.1 分割サイズ最適化についての評価

ooc_cuDNN による畳み込み計算におけるパラメータの最適化について評価した．実験では 4.4 節で提案した最適化手法でパラメータを選んだときの性能と，局所探索法でパラメータを選んだときの性能を比較した．また，すべてのパラメータの組み合わせでそれぞれ実行して最も高かった性能とも比較した．

局所探索法では各パラメータを分割しない状態を初期状態として実験を行った．また，現在の解に対してパラメータのうちいずれかを 1 増加 (または減少) させたものを近傍解とした．

この実験は以下の 3 通りの問題サイズについて行った．フィルタサイズはすべて 3×3 とした．

(1) $n = 1, c_X = c_Y = 64$, 画像サイズ $= 8192 \times 4096$

(2) $n = 1, c_X = c_Y = 256$, 画像サイズ $= 1024 \times 1024$

(3) $n = 512, c_X = c_Y = 64$, 画像サイズ $= 256 \times 256$

結果を表 4 に示す．提案最適化手法でパラメータを選んだ場合でも，実際の最適パラメータで実行した場合と比較して性能低下は 3.4 %以下であった．

また, 表 4 によれば局所探索法を用いた場合と比較して, 提案最適化手法を用いた場合のほうが性能が高い．これは局所探索法では近傍解を用いて各分割サイズを更新するので, 局所解に陥りやすいためである．一方，提案手法ではパラメータごとに全探索を行うので局所解に陥りにくい．

以下の実験では，提案最適化手法によって選択したパラメータを用いた．

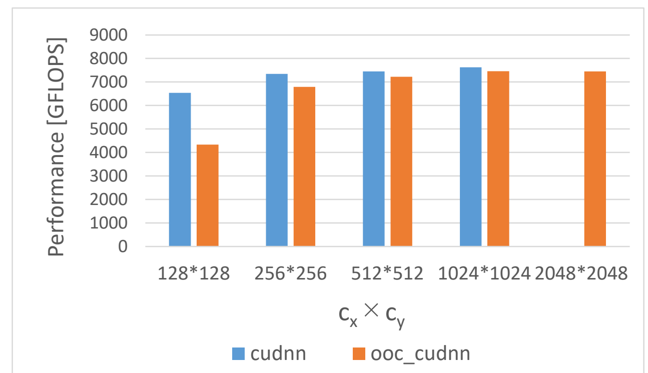

図 9 チャンネル数を変化させたときの畳み込みの性能

6.2 性能の評価

層 X を重みフィルタ W で層 Y に畳み込む計算を cuDNN，ooc_cuDNN でそれぞれ行った．実験では $n = 1$, $h_Y = w_Y = 1024$, $r_W = s_W = 3$ で入出力チャンネル数 c_X, c_Y を変化させたときの性能を測定した． また，この実験では c_X, c_Y が 2048 以上のとき, 必要なメモリ量が GPU メモリ容量を超えるため cuDNN では実行することができなかった．

結果を図 9 に示す．cuDNN を用いたときの性能は最大で 7629 GFLOPS であった．これに対して，ooc_cuDNN を使用した場合の性能は 4332 ~ 7455 GFLOPS であった．特に GPU 容量を超えてしまうチャンネル数に対しては，cuDNN を使用した場合の最大性能と比較して 97 %の性能となった．

図 9 ではチャンネル数が小さいほど, cuDNN と ooc_cuDNN の性能差が大きい． ここで, チャンネル数による ooc_cuDNN の性能への影響について考える．畳み込みの計算量は c_X, c_Y の両方に比例して増大する．これに対して, HtoD 通信における層 X に関する通信量は c_X には比例するが, c_Y には依存しない．そのため, c_Y が大きい場合では HtoD 通信にかかる時間より計算時間のほうが長くなり, HtoD 通信を隠蔽することができる．一方, c_Y が小さい場合では HtoD 通信にかかる時間が支配的になり性能低下につながる．同様に, DtoH 通信を隠蔽できるかは c_X の値に依存する．

6.3 計算の複合についての評価

5 章で提案した複合関数の評価を以下のように行った．

畳み込み・バイアスの加算・活性化の計算を cuDNN, ooc_cuDNN でそれぞれ行った．ooc_cuDNN では複合関数を使用した場合と，3 つの計算を分けて行った場合のそれぞれで実行時間を測定した．

結果を図 10 に示す．ooc_cuDNN では計算を複合しない場合, 実行時間のうちバイアスの加算と活性化が占める割合が cuDNN と比較して増加した．これはバイアスの加算・活性化のように計算量が小さい計算では, 通信の大部分が

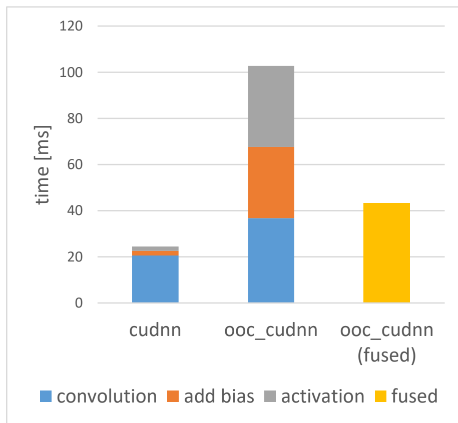

図 10　畳み込み・バイアスの加算・活性化の実行時間の内訳 ($n = 1$, $c_X = c_Y = 64$, $h_Y = w_Y = 1024$, $r_W = s_W = 3$)

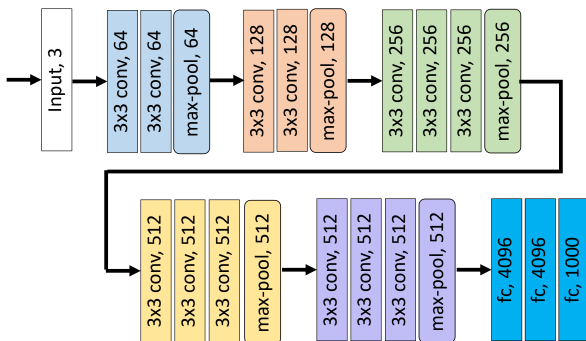

図 11　VGG16 のネットワーク構造

計算によって隠蔽されないためである．一方, 3 つの計算を複合することにより，複合しない場合と比較して実行時間は 42 %となった．

このように計算量が小さい計算でも，計算量が大きい計算とまとめて行うことで, 通信によるオーバーヘッドを小さくして効率的に行うことができる．

6.4　CNN アプリケーションへの適用

VGG16 ネットワーク [7] の順伝播・逆伝播の計算を cuDNN と ooc_cuDNN をそれぞれ使用して行った．VGG16 のネットワーク構造を図 11 に示す． VGG16 は画像認識に用いられる CNN であり, 主に 13 層の畳み込み層と 3 層の全結合層から構成される．また, 各畳み込み層におけるフィルタサイズはすべて 3×3 である．実験には cuDNN と ooc_cuDNN をそれぞれ用いて C++で我々が実装したプログラムを使用した．

この実験では ooc_cuDNN で計算を行うとき, GPU メモリに収まる特徴マップのデータは GPU メモリ上に配置した．また, 逆伝播時のメモリ使用量を削減するために, 勾配データ用のメモリは層ごとに確保するのではなく, メモリプールを用意してその中のメモリを使いまわすこととした．

この実験ではバッチサイズを 256 以上とした場合，計算に必要なメモリ量が GPU メモリ容量を超えてしまうため cuDNN では計算を行うことができなかった． また, バッチサイズ $n = 1024$ とした場合, 計算全体では 60 GB 以上のメモリを必要とし, さらには図 2 に示したように 1 つの層の計算だけでも GPU メモリが不足した．

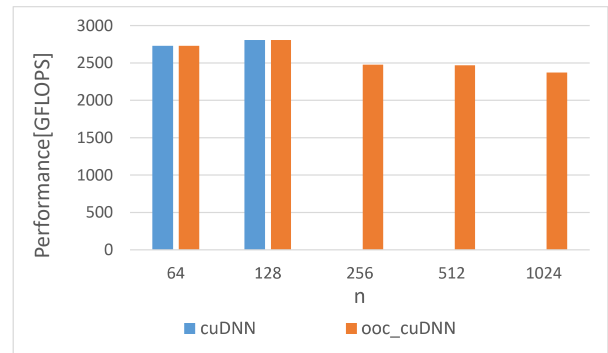

図 12　VGG16 への適用時の性能

バッチサイズ n を変化させて性能を測定した際の結果を図 12 に示す．cuDNN を使用した場合の性能は最大で 2805 GFLOPS であった．これに対して，ooc_cuDNN を使用した場合の性能は 2372 ∼ 2805 GFLOPS となった．特に GPU 容量を越える問題サイズに対して ooc_cuDNN を用いた場合は，10 ∼ 13 %のオーバーヘッドで計算を行うことができた．

ooc_cuDNN を使用した場合の性能について分析するために, VGG16 の順伝播計算における各畳み込み層の HtoD・計算・DtoH の実行時間を図 13 に示す． 図 13 では層が進むにつれて, 通信時間と比較して計算時間のほうが長くなっている．これは, VGG16 では図 11 のように層が進むにつれてチャンネル数が増加していくためである．6.2 節で述べたように, チャンネル数が大きい層では計算時間が支配的になり性能低下を抑えることができる．逆伝播計算についても同様の傾向があり, 図 12 では計算全体における性能低下を抑えることができた．

6.5　Unified Memory を使用した場合との性能比較

6.5.1　Unified Memory

GPU メモリ容量を超えるデータを扱う方法として，CUDA のメモリ管理機能の一つである Unified Memory[11] を使用することが挙げられる．Unified Memory として確保されたデータは使用時に CPU・GPU 間通信が自動で行われることにより，CPU と GPU の両方から扱うことができる．以下では CUDA8.0 および Pascal 世代アーキテクチャにおける Unified Memory の挙動について説明する．

Unified Memory はメモリをページ単位で管理しており，データ使用時の CPU・GPU 間通信は Page fault 発生時に行われる．また，Unified Memory では GPU メモリに収まりきらないデータは自動で CPU メモリに退避される．そ

© 2017 Information Processing Society of Japan

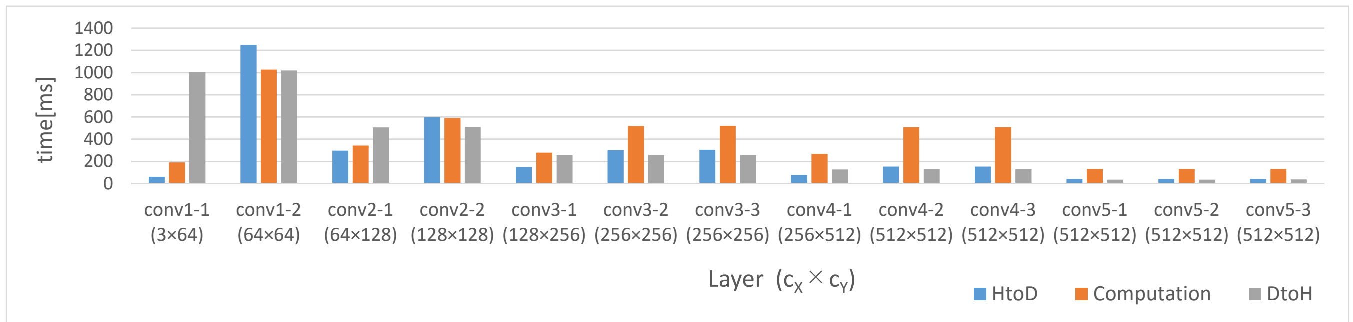

図 13　VGG16 の順伝播計算における各畳み込み層の HtoD･計算･DtoH の実行時間 ($n = 1024$)

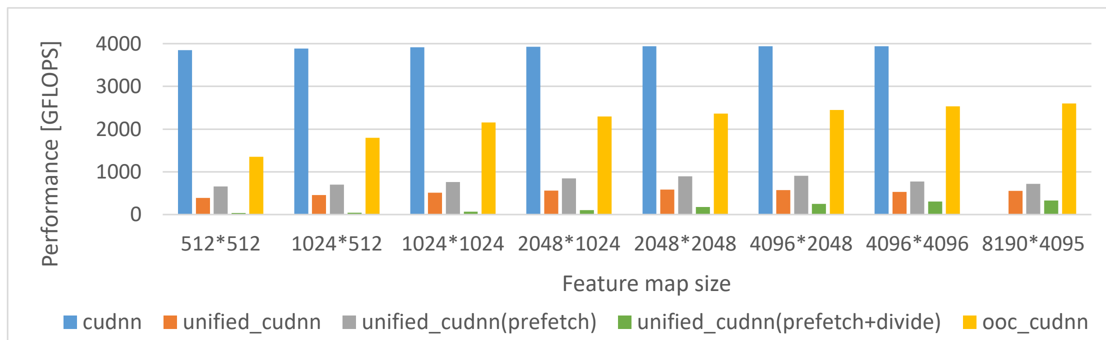

図 14　Unified Memory を使用した場合との性能比較 ($n = 1$, $c_X = c_Y = 64$, $r_W = s_W = 3$)

のため，GPU メモリ容量を超えるサイズのデータを GPU で扱うことができる．

Unified Memory には性能チューニングのため，`cudaMemPrefetchAsync()` 関数というプリフェッチ関数が用意されている．この関数を使用してデータのプリフェッチを行うことにより，Page fault 回数の削減が期待される．

6.5.2　性能比較

Unified Memory を用いて畳み込み計算を行い，性能について評価した．また，通常の cuDNN および ooc_cuDNN との性能比較も行った．

Unified Memory を用いた畳み込み計算は以下の 3 つの手法でそれぞれ行った．計算はすべて GPU 上で cuDNN を使用して行った．

- unified_cuDNN : 性能チューニングせずに計算
- unified_cuDNN(prefech) : データをプリフェッチ
- unified_cuDNN(prefetch+divide) : データをプリフェッチ，および ooc_cuDNN と同様の計算分割

ここで，unified_cuDNN(prefetch+divide) では，2 次元データについてプリフェッチする場合は連続したメモリ領域ごとに `cudaMemPrefetchAsync()` 関数を呼び出す．

実験では特徴マップのサイズを変化させたときの性能を測定した．Unified Memory を使用するときのデータの初期配置としては，特徴マップはすべて CPU メモリ上，重みフィルタは GPU メモリ上に配置した．

結果を図 14 に示す．unified_cuDNN の性能は 390 ～ 584 GFLOPS となった．Unified Memory を性能チューニングなしに使用した場合の性能低下の原因としては，以下の 2 点が挙げられる．

- 計算と通信がオーバーラップされない
- GPU 上での Page fault によるオーバーヘッドが大きい

次に unified_cuDNN(prefetch) の性能は 660 ～ 906 GFLOPS と，unified_cuDNN より高くなった．これはデータをプリフェッチすることにより，GPU 上での Page fault の発生回数が削減されたためである．

unified_cuDNN(prefetch+divide) の性能は 39 ～ 326 GFLOPS となった．これは計算の分割により `cudaMemPrefetchAsync()` 関数の非効率的な呼び出しが増加したためである．

また，Unified Memory を使用した場合，出力層 Y についても HtoD 通信が行われていた．そのような不必要な通信によるオーバヘッドも性能低下の一因である．

最後に，ooc_cuDNN の性能は 1350 ～ 2599 GFLOPS となった．特に GPU メモリ容量を越える問題サイズに対して ooc_cuDNN を用いた場合，unified_cuDNN(prefetch) と比較して 3.6 倍の性能で計算を行うことができた．

7.　関連研究

本研究と同様に GPU メモリ容量を越える深層学習をするための手法として，Rhu らは virtualized Deep Neural

© 2017 Information Processing Society of Japan

Network(vDNN) を提案している [12]．vDNN は CPU メモリを使うことで GPU メモリ容量による問題を解決しようとしている点では本研究と同じである．一方，vDNN では層単位でメモリを管理することを考えており，ある層の計算と別の層のデータの通信をオーバーラップすることで通信によるオーバーヘッドを減らしている．しかし，この手法は一つの層の計算に必要なデータがすべて GPU メモリ上に収まることを前提としていて，そうでない場合には対応できない．

Cui らは GPU を用いた深層学習の分散処理において，GPU メモリの総容量を越える問題サイズの計算を行う GeePS を提案している [13]．GeePS では各ノードにおいて GPU メモリ容量が足りない場合，そのノードの CPU メモリを使用してデータをスワップすることで対処している．計算と通信のオーバーラップも行っているが，データの分割サイズの最適化はしていない点で本研究とは異なる．

Dean らは CNN の計算を複数のノードで分散処理するモデル並列について述べている [14]．モデル並列ではノードの数だけ使用できるメモリ量が増えるため大規模な CNN の計算が可能となる．しかし，本研究で行っているようなメモリ階層の利用は行っていない．

8. まとめと今後の課題

本論文では CPU メモリを使い，かつ CNN の各層を分割することで大規模な CNN の計算を GPU で行うことを可能とする ooc_cuDNN ライブラリの設計と実装について述べた．ooc_cuDNN では性能モデルから適切な分割サイズを求め，パイプライン処理を行うことで通信によるオーバーヘッドを小さくする．また，さらなる高速化のためにでは複数の計算をまとめて行う複合関数も提供する．

ooc_cuDNN を使用することで，GPU 容量を超える CNN でも 10 ∼ 13 %の性能低下で計算できることを示した．また，Unified Memory を使用した場合と比較して最大で 3.6 倍の性能で計算できることを示した．

今後の課題としては，各関数の設計・実装だけでなく，CNN 全体を考慮したデータの初期配置の決定や計算の複合などの最適化を行っていきたい．そして，既存の機械学習フレームワークに対する ooc_cuDNN の適用も行っていきたい．また，今回の設計では 4.2 節の性能モデルにおいて，簡単のため c_Y が 32 増加するごとに実行時間が増加するものとした．しかし，今後は cuDNN の関数の詳細な性能解析を行い，その結果を性能モデルに反映したい．

謝辞 本研究は JST-CREST の研究課題「ポストスケール時代のメモリ階層の深化に対応するソフトウェア技術」の支援を受けております．

参考文献

[1] Le, Q. V., Ngiam, J., Coates, A., Lahiri, A., Prochnow, B. and Ng, A. Y.: On optimization methods for deep learning, *International Conference on Machine Learning (ICML)*, pp. 265–272 (2011).

[2] Frank Seide, Gang Li, D. Y.: Conversational Speech Transcription Using Context-Dependent Deep Neural Networks, *Annual Conference of the International Speech Communication Association (Interspeech)*, pp. 437–440 (2011).

[3] Sutskever, A. K. I. and Hinton, G.: ImageNet Classification with Deep Convolutional Neural Networks, Advances in neural information processing systems (NIPS)，pp. 1097–1105 (2012).

[4] Dong, C., Loy, C. C., He, K. and Tang, X.: Image Super-Resolution Using Deep Convolutional Networks, *IEEE transactions on pattern analysis and machine intelligence (TPAMI)*, Vol. 38, No. 2, pp. 295–307 (2016).

[5] Radford, A., Metz, L. and Chintala, S.: Unsupervised representation learning with deep convolutional generative adversarial networks, *International Conference on Learning Representations (ICLR)* (2016).

[6] Chetlur, S., Woolley, C., Vandermersch, P., Cohen, J., Tran, J., Catanzaro, B. and Shelhamer, E.: cuDNN: Efficient Primitives for Deep Learning, `https://arxiv.org/abs/1410.0759` (2014).

[7] Simonyan, K. and Zisserman, A.: Very Deep Convolutional Networks for Large-Scale Image Recognition, *International Conference on Learning Representations (ICLR)* (2015).

[8] Jia, Y., Shelhamer, E., Donahue, J., Karayev, S., Long, J., Girshick, R., Guadarrama, S. and Darrell, T.: Caffe: Convolutional Architecture for Fast Feature Embedding, *ACM international conference on Multimedia (ACMMM)*, pp. 675–678 (2014).

[9] Abadi, M., Barham, P., Chen, J., Chen, Z., Davis, A., Dean, J., Devin, M., Ghemawat, S., Irving, G., Isard, M., Kudlur, M., Levenberg, J., Monga, R., Moore, S., Murray, D. G., Steiner, B., Tucker, P., Vasudevan, V., Warden, P., Wicke, M., Yu, Y. and Zheng, X.: TensorFlow: A System for Large-Scale Machine Learning, *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pp. 265–283 (2016).

[10] Tokui, S., Oono, K., Hido, S. and Clayton, J.: Chainer: a Next-Generation Open Source Framework for Deep Learning, *Machine Learning Systems Workshop in conjuction with NIPS* (2015).

[11] 成瀬　彰：Pascal 世代で進化する Unified Memory，*GPU Technology Conference Japan (GTC Japan)*, `https://www.gputechconf.jp/assets/files/1012.pdf` (2016).

[12] Rhu, M., Gimelshein, N., Clemons, J., Zulfiqar, A. and Keckler, S. W.: vDNN: Virtualized Deep Neural Networks for Scalable, Memory-Efficient Neural Network Design, *IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pp. 1–13 (2016).

[13] Cui, H., Zhang, H., Ganger, G. R., Gibbons, P. B. and Xing, E. P.: GeePS: Scalable deep learning on distributed GPUs with a GPU-specialized parameter server, *European Conference on Computer Systems (EuroSys)* (2016).

[14] Dean, J.，Corrado, G. S.，Monga, R.，Chen, K.，Devin, M.，Le, Q. V.，Mao, M. Z.，Marc' AurelioRanzato，Senior, A.，Tucker, P.，Yang, K.，Ng, A. Y.：Large Scale Distributed Deep Networks, *Advances in neural information processing systems (NIPS)*, pp. 1223–1231 (2012).

© 2017 Information Processing Society of Japan