修士論文

チャネル方向並列化を導入したCNNの学習のハイブリッド並列化の提案

Proposal of a hybrid parallelization scheme for training of CNN

平成30年2月1日提出

指導教員　工藤 知宏　教授

東京大学大学院

工学系研究科　電気系工学専攻

37-166424　赤沼 領大

要旨

現在、画像認識をはじめとしてCNN（畳み込みニューラルネットワーク）があらゆる分野で応用されている。しかし、パラメータや演算量が膨大なためにCNNの学習には膨大な時間がかかる。このためにCNNの学習を高速化する手法の開発が盛んにおこなわれている。演算を高速化させる方法としてデータ並列を用いた演算の並列化が一般的に用いられているが、演算器の利用率効率を高めたり、演算の並列度を上げたりするためにはミニバッチのサイズを大きくする必要がある。しかし、ミニバッチのサイズを上げすぎると学習の収束性が悪くなることが知られており、ミニバッチサイズには上限がある。この上限のためにデータ並列での並列度が制限されてしまい、今後のさらなる並列化の支障となる。

データ並列にモデル並列を導入するハイブリッド並列を行うとミニバッチサイズを維持したまま並列度を上げることが出来る。既存の層単位でのモデル並列化はロードバランシングにおいて劣っているために、各層のチャネル方向への並列化を導入した。データ並列とハイブリッド並列での実行時間をそれぞれモデル化して比較を行った。

チャネル方向への層の並列化を行うと、従来のCNNの演算よりも小さい演算量の演算が発生する。演算のサイズが小さくなることで演算性能の低下が起こりえる。この影響を評価するために小さいサイズでのCNNの演算をGPUとFPGAで行い検証した。

Abstract

Currently, CNN (Convolution Neural Network) is used in various fields including image recognition. However, because of a large number of parameters and a large amount of computation, training of CNN requires enormous time. So, various schemes for speeding up CNN training have been developed. In those schemes, parallel processing using data parallelism is generally used and in order to increase the utilization ratio of computing elements the size of mini batch must be large. However, it is known that convergence ratio of training is low if the mini batch size is too large, and therefore there is an upper limit for the usable mini batch size. This limits the degree of parallelism in data parallel schemes.

In this thesis, a hybrid parallelization scheme which combines data parallelism with model parallelism is proposed and evaluated. This scheme makes it possible to increase the degree of parallelism while maintaining mini batch size. Since load of computing elements is not well balanced in existing layer-wise parallelization schemes, channel-wise parallelization is used in the proposed scheme. The execution times of data parallel and hybrid parallel schemes were estimated and compared. In channel-wise parallelization, the size of computation at each step of execution becomes small. Impact of the size of computation to the execution efficiency is also evaluated by implementing such computation on GPU and FPGA.

目次

図目次

表目次

1.序論

1.1. 研究背景

画像認識とは画像を処理して何らかの説明を行うタスクであり、その一つに画像のクラス識別がある。これは画像認識の一分野で、画像に写っているものが事前に用意したいくつかの分類の内の何なのかを認識するタスクである。

古くは画像から SIFT [1]や HOG [2]等の経験的に作られたハンドメイド特徴量を抽出して、その特徴量をもとに分類を行っていた。しかし大規模な画像セットである ImageNet [3]を用いたコンテストで CNN を用いたチームが 2 位のチームにエラー率で 10%以上もの差をつけて優勝したことで CNN が世間からの注目を集めるようになった [4]。

CNN はニューラルネットワークの一種で、CNN 自体の技術は 1980 年代から知られていたが、当時はコンピューターの演算性能が十分でないために、MNIST [5]等の小さなデータセットの画像認識への適用例はあったが、大規模なデータセットに対して現実的な手段とは考えられていなかった。CNN が注目されて以降、CNN は画像認識以外の分野でも活用されている。R-CNN [6]では画像から候補領域を取り出してそれに対して CNN で物体認識を行うことで、写っている物の位置を検出する画像検出を行っている。また画像から自然言語による説明文生成も CNN を応用して行われている [7]。

しかし CNN による画像認識ではパラメータの数が多く学習に多くの時間がかかるという欠点がある。CPU で 100 万枚の画像の学習を行うと簡単なニューラルネットワークで 2 日以上かかる [8]。

CNN の学習を高速化させる方法の 1 つとして演算の並列化がある。通常プログラムは 1 つの演算器上でしか実行されないが、複数の演算器を使ってプログラムを実行することで演算を高速化することが出来る。CNN の演算の並列化の方法としては大きく分けてデータ並列とモデル並列がある。データ並列は通常の処理にパラメータの更新値の集約のステージを加えるだけで実行可能でよく行われている手法だが、この方法では並列度を上げるためにはミニバッチのサイズを大きくしなければならず、これは学習効率の悪化を招く [9]。そのために使用できるミニバッチサイズには上限があり、これがデータ並列での並列度を制限する。これは今後のさらなる並列度の向上の障害となる。

1.2. 研究目的

上記の様にデータ並列にはミニバッチのサイズの限界に由来する並列度の制限がある。これを解決するために本論文ではデータ並列にモデル並列を導入したハイブリッド並列を提案する。モデル並列とは 1 つの CNN の処理を複数のノードで分割する手法である。この手法ではミニバッチのサイズ以上のノード数を使って CNN の処理を行うことが出来る。しかし既存のモデル並列の手法 [10]では CNN の層単位での並列化のみを行っているので、並列度に限界があり、また演算量がバランスするようにノードに演算を分散させることは

難しい。

そこで本論文では CNN の 1 つの層をチャネル方向に分割複数の層に分割する方式を提案する。これによって既存方式よりも演算の並列度が上がり、また適切に演算をノードに割り当てることで、演算ノードごとの演算負荷のバランスを取ることが出来る。

チャンネル方向の CNN の並列化を行うと従来の CNN の演算よりも小さい演算量の処理が発生する。このような処理の演算性能を GPU と FPGA 上で検証し、本提案方式に適したプロセッサについて検討した。

1.3. 本論文の構成

本論文は全 7 章から構成される。まず、第 1 章では本論文の研究背景と目的について述べた。第 2 章では DNN と CNN に関する既存の技術や研究について述べる。第 3 章では既存手法よりも高い並列度を達成するハイブリッド並列手法について説明する。第 4 章ではデータ並列とハイブリッド並列の各々での演算時間をモデル化し比較・検討を行った。

ハイブリッド並列では既存の並列計算機とは異なるアーキテクチャーが適している。その同期型アーキテクチャーを第 5 章で提案する。

第 6 章では GPU と FPGA を用いて提案手法で発生する演算の評価実験を行い、第 7 章ではそれについて考察した。最後に第 8 章では本論文をまとめ、今後の展望について述べる。

2.関連研究・手法

2.1. ニューラルネットワーク

人工ニューラルネットワークは、高度な情報処理を目的として脳の特性を計算機上でモデル化した物である。その中の順伝播型ニューラルネットワークは入力層、隠れ層、出力層を持ち、入力層に入ってきた入力データが、隠れ層に伝搬され、それが出力層へと 1 方向にのみ伝搬して出力データを得る。本論文ではこのような順伝播型ニューラルネットワークのことをニューラルネットワークと呼ぶ。

ニューラルネットワークは図 1（右）の様にいくつかのノードからなる入力層、隠れ層、出力層から構成されて、入力層と隠れ層、隠れ層と出力層それぞれエッジでつながっている。各エッジには重みがつけられていて、入力層の各ノードの値と対応するエッジの重みの積の総和にバイアスと呼ばれる値を足したものが隠れ層に総入力として渡されて、総入力を活性化関数と呼ばれる非線形な関数に通したものが隠れ層のノードの出力になる。隠れ層と出力層の間でも同じように値が受け渡されていく。ニューラルネットワーク中の 1 つのノードに注目した図が図 1（左）になる。注目しているノード（灰色）には 2 つのノード i_0 と i_1 がつながっていて対応する重みはそれぞれ W0、W1 で、バイアスは b_0 である。この時、活性化関数を f と置くとこの注目しているノードの出力は以下のようになる。

$$\mathbf{y} = \mathbf{f}(\boldsymbol{w_0 i_0} + \boldsymbol{w_1 x_1} + \boldsymbol{b_0}) \tag{1}$$

このように入力データから順伝播をして出力データを得る過程をニューラルネットの推論と呼ぶ。また推論で目的の処理が出来るように重みとバイアスを更新する演算を学習と呼ぶ。

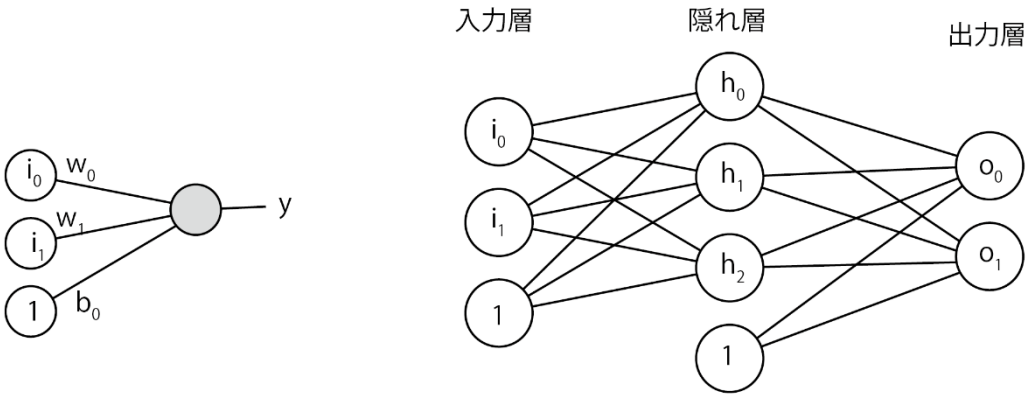

図 1　ニューラルネットワークの模式図

ニューラルネットワークは機械学習の一手法ではある。本論文ではその中でも教師付き学習で画像のクラス分類を行うニューラルネットワークモデルに注目する。画像のクラス分類とは画像に写っている物が何なのかを識別するタスクである。例えばネコの画像が写っている画像を入力した時に「ネコ」というラベルを返すというものである。従来は SIFT

[1]や HOG [2]のような経験的な特徴量を画像から抽出してそれを用いて画像の認識をしていた。機械学習ではこの特徴量をアルゴリズムに従って自動的に抽出する。教師付き学習では大量の画像と、それに何が写っているのかを示す正解ラベルからなるデータセットを用意してそれを用いて画像の認識モデルを構成する。適切に学習が行えればニューラルネットワークは未知の画像に対しても正しいラベルを推論出来るようになる。

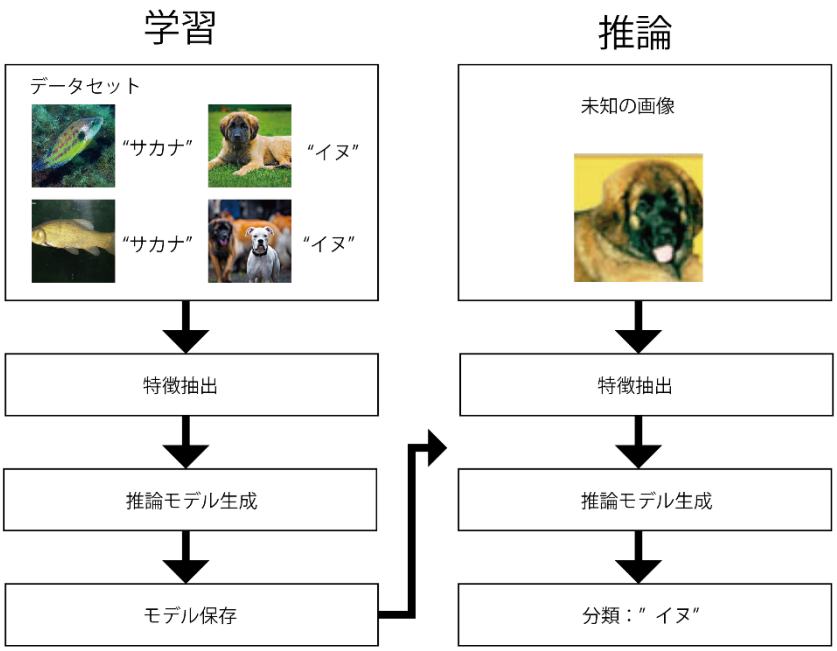

図 2　機械学習の概要

2.2. DNN

DNN（ディープニューラルネット）とはニューラルネットワークのうち、複数の隠れ層を持つものを指す。またこれを用いた機械学習を深層学習（ディープラーニング）と呼ぶ。深層学習のアルゴリズム自体は古くから知られていたが、演算能力の不足やデータセットの不足により注目を向けられなくなった。しかし ILSVRC という大規模画像データセットを用いた画像認識コンテストで 2012 年にトロント大学の Hinton らのチームが DNN を用いてエラー率で 10%もの改善を行い優勝した [4]ことで DNN への注目があつまった。図 3 の様にそれまで年数%の改善にとどまっていたものが、この時では 10%もの改善を行いブレークスルーとなった。以来画像認識以外でも既存の手法を凌駕する性能の機械学習が可能なことが報告されている [6] [7]。

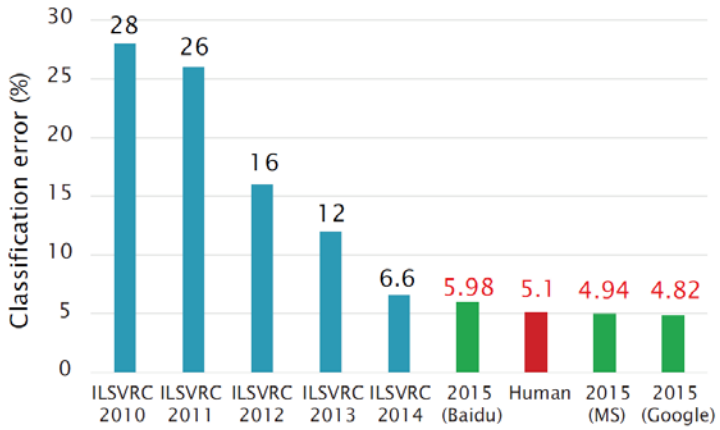

図 3　ILSVRC のエラー率の推移 [54]より
ILSVRC 2012 以降で CNN が使用されている。

DNN では何層の隠れ層を持ち、それぞれの隠れ層の中のノード数やノードの接続の方式がどうなるのが最適なのかは問題によって異なる。このような DNN のノードの構成をネットワークモデルと呼ぶ。

2.2.1. DNN の学習

ニューラルネットが目的のタスクをこなすように、重みとバイアス（以下この 2 つを総称してパラメータと呼ぶ）を更新する処理を DNN の学習と呼ぶ。学習には様々な方法があるが、広く使われている勾配降下法を本論文では用いる。

DNN のパラメータの初期値は通常乱数で与えられるので、推論した結果は正しい値にならない。ニューラルネットワークでクラス識別を行う場合、ニューラルネットからは各クラスの該当確立が出力される。この出力と正解ラベルのデータから誤差関数を用いて Loss(E)を計算する。Loss はニューラルネットの推論の結果と正解ラベルとの誤差を表す。勾配降下法ではパラメータ(w)を以下の式の様に順次更新することでニューラルネットが目的のとするクラス分類が行えるように学習を進める。

$$\boldsymbol{w^{t+1} = w^{t} - \varepsilon \frac{\partial E}{\partial w}} \tag{2}$$

上式のεは学習係数と呼ばれパラメータの更新量を決める値で、その値は学習に大きな影響を与える [11]。学習係数は小さすぎると、更新をその分多く行う必要があり、一方学習係数を大きくしすぎると学習が発散する恐れがある。そのために学習係数は Adam [12]等のアルゴリズムを用いて決定する。

確率的勾配降下法によって DNN を学習することは可能だが$\frac{\partial E}{\partial w}$を計算することには困難

が生じる。出力層近くの w でなら計算は単純であるが、出力層が遠い w については複雑で大きな計算量が必要である。これを解決するためには微分の連鎖則を用いたバックプロパゲーション（誤差逆伝播法） [13]を用いる。

図 4 のように 2 層のネットワークで誤差関数に 2 乗誤差を選択した時を例にして説明する。ニューラルネットの出力を(z_0,z_1)、正解ラベルを(t_0,t_1)としたときに 2 乗誤差は

$$\mathrm{E} = \frac{1}{2}\sum(z_i - t_i)^2$$

と表せる。この時出力層の重みの微分は以下のように表せる。

$$\frac{\partial \boldsymbol{E}}{\partial \boldsymbol{v}} = (\mathbf{z} - \mathbf{t})\intercal \frac{\partial \boldsymbol{z}}{\partial \boldsymbol{v}} = (\mathbf{z} - \mathbf{t})\intercal \mathbf{y} \tag{3}$$

また隠れ層の重みの微分は

$$\frac{\partial \boldsymbol{E}}{\partial \boldsymbol{w}} = \frac{\partial \boldsymbol{E}}{\partial \boldsymbol{y}}\frac{\partial \boldsymbol{y}}{\partial \boldsymbol{w}} = (\mathbf{z} - \mathbf{t})\boldsymbol{v}^{\boldsymbol{t}}\intercal \boldsymbol{x} \tag{4}$$

となる。

つまりパラメータを求めるには推論側の特徴マップと Loss から逆向きに計算される誤差の微分の伝播（一般に勾配と呼ばれる）の 2 つが必要になる。

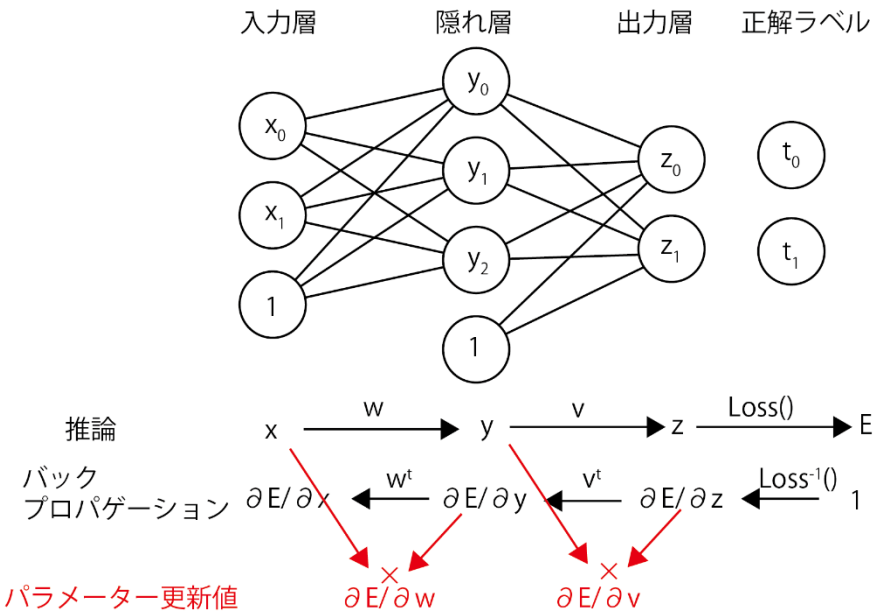

図 4　バックプロパゲーションの概要

ここまでをまとめると DNN の学習は 3 つのステージからなる。最初に Loss を計算するために推論を行い、その後バックプロパゲーションを用いて微分値を計算する。そして最後にパラメータを更新する。

2.2.1. 学習の方式と過学習

前述したような DNN の学習を何回も繰り返すことで目的とする DNN を得る。本節ではその全体像を説明する。

DNN の学習を行う場合は、その目的に合わせてデータセットを用意する。データセットとは画像とその画像に対応する正解ラベルの集合である。DNN の学習の前にデータセットをトレーニングセットとテストセットに分ける。まず、トレーニングセットのすべての画像を用いて DNN の学習を行う。この後に、学習の結果を確認するためにテストセットを用いて推論のみを行い認識率を計測する。これが学習の 1 つの単位であり epoch と呼ぶ。1 epoch の学習だけではパラメータの更新が十分ではないので、またトレーニングセットのすべての画像を用いて学習を繰り返す。このように epoch を繰り返すことで学習された DNN を得る。

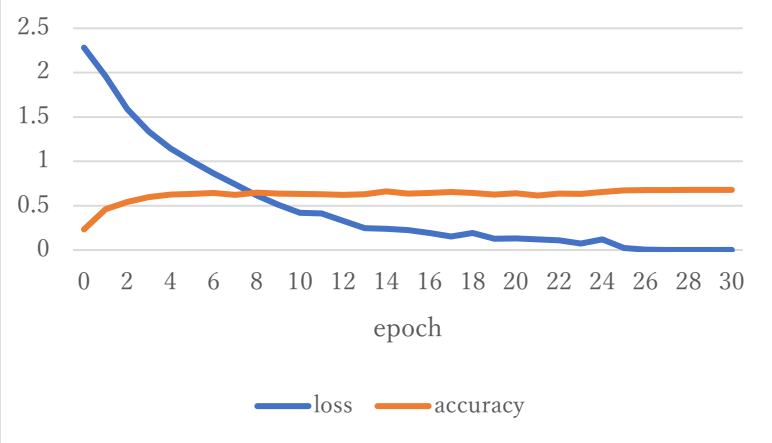

図 5　DNN の学習の例　loss の値は徐々に減少して、accuracy は徐々に増加していく

しかし、epoch を過剰に繰り返すと、DNN がトレーニングセットのデータに過学習（オーバーフィッティング）してしまって、未知のデータに対応する性能（汎化性能）を失うことがありうる。これを防止するために学習に用いないデータであるテストセットを用いて accuracy を計り、accuracy が低下していないかを確認することで過学習を確かめる。過学習が確認された場合早期終了をする。

1epoch の中の学習の方法として大きく分けて以下の 3 種類の方法がある。

1. オンライン学習

   この方式は確率的勾配降下法とも呼ばれる。1 つの画像推論を行う度に、バックプロパゲーションとパラメータの更新を行う。最も基本的な学習方法であるが、最も時間がかかる方法であるために大規模な CNN の学習では使われることは稀である。

2. バッチ処理
   データセット中の対象とするすべての画像に対してフォワード処理を行い、その後でバックワード側の処理をまとめて行い、最後にパラメータの更新をする。演算をまとめて行うために演算のサイズが大きくなり、演算器の並列度を十分に使うことで、演算器の使用率が良く、オンライン学習よりも高速に学習が行える。
3. ミニバッチ処理
   上記 2 つの学習方式の間に位置する方式がミニバッチ学習である。データセットをいくつかのミニバッチに分ける。そのミニバッチの中でバッチ処理を行い、それをミニバッチの数だけ繰り返す。ミニバッチの数を適切にとれば、バッチ処理による演算器の使用効率を維持しながら学習の収束率が良くなる。

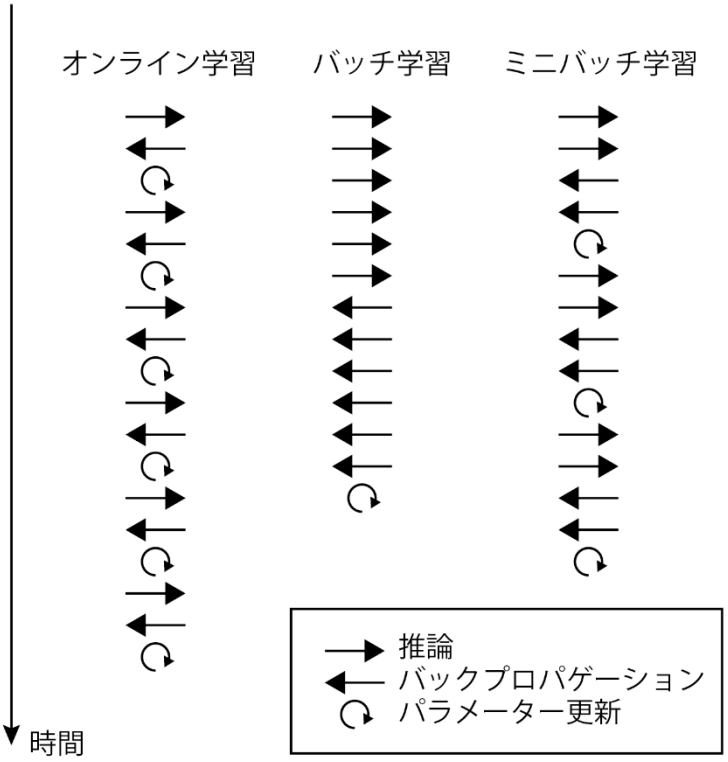

図 6　各学習方式の模式図

3 つの演算方法をまとめたものが表 1 である。計算時間という面で見ると多くの演算をまとめて処理することのできるバッチ処理が一番良いように見えるがバッチ処理やミニバッチ処理を行うとその分 1epoch でのパラメータの更新回数が少なくなる。そのためにオンライン学習と同程度の学習を 1epoch で行おうとすると学習係数をバッチでまとめた分だけ大きくする必要がある [14]。しかし学習係数を大きくとりすぎると学習の収束が悪くなり、最終的な accuracy が低下することが知られてる。一方、ミニバッチ処理やバッチ処理ではそのバッチ内での演算には依存性が無く演算を独立に行うことが出来るために並列化が可

能である。そのため演算の収束と高速性の両立により一般的にディープラーニングではミニバッチ処理が採用される。

表 1　各学習方式の比較

	オンライン学習	ミニバッチ学習	バッチ学習
演算性能	×	△	○
パラメータ更新回数	○（多い）	△	×（少ない）
並列化	×	○	○

2.3. CNN

DNN の中で、特に畳み込み層を持つものを CNN（畳み込みニューラルネットワーク）と呼ぶ。畳み込み層とはデータの局所性を表現できる層である。CNN は層が進むにつれて異なるスケールでの特徴が抽出されると言われており、画像認識に適している。このようなネットワークモデルは 1980 年に福島らによって提唱された［15］。

2.3.1.CNN を構成する層

CNN では主に以下の 4 つの種類の層が使用される。ネットワークモデルによってはこれ以外の種類の層も使用されるが本論文では以下の 5 つの層を対象とする。

○畳み込み層

畳み込み層は CNN の中心的な層で、CNN の全体の演算時間の約 90%を占める［16］。入力画像に対してカーネルフィルタで畳み込み演算を行った結果を出力する。

カーネルフィルタのサイズが K x K の場合、図のように注目画素を中心とした K x K の領域の画素の値とカーネルフィルタとの積和を出力画像の注目画素の部分の値をする。

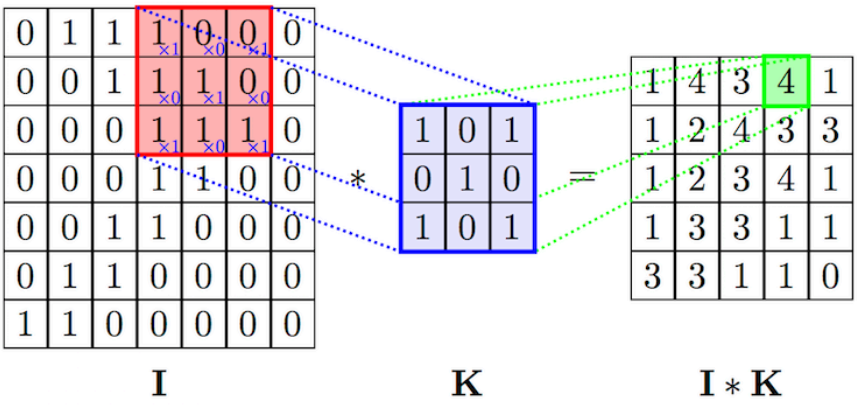

図 7　畳み込み演算［53］より

図 7 のような 1 チャネル入力、1 チャネル出力の畳み込み演算を疑似コードで表したの

が Code 1 である。このように 1 チャネル入力、1 チャネル出力の畳み込み演算自体は 4 重ループで表すことが出来る。

```
for(row=0;row<F;row++){
  for(col=0;col<F;col++){
    for(ky=0;ky<K;ky++){
      for(kx=0;kx<K;kx++){
        out[row][col]+=w[ky][kx]*in[row+ky][col+kx]
}}}}}}
```

Code 1　1 入力チャネル、1 出力チャネルの畳み込み演算

実際の CNN で使用する畳み込み演算は多チャネル入力、多チャネル出力の畳み込み演算である。入力する特徴マップのチャネル数が M、出力のチャネル数が N の時に畳み込み層はカーネルフィルタを M x N 枚持つ。一枚の画像(P)とカーネルフィルタ(K)の畳み込み演算をK⊗Pと表すとき。この畳み込み層の推論の演算は

$$\boldsymbol{Output_j} = \sum_{\boldsymbol{i=o}}^{\boldsymbol{N-1}} \boldsymbol{K_{j,i}} \otimes \boldsymbol{Input_i} \ (\mathbf{0 \le j < M}) \tag{5}$$

と表される。これを図示すると図 8 のようになる。

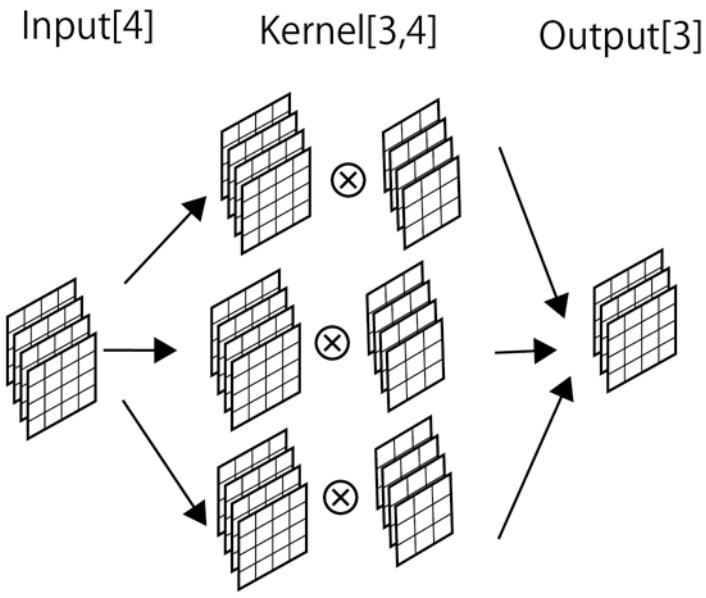

図 8　複数チャンネルの畳み込み

バックプロパゲーションは出力側から来た勾配(OutGrad)から入力側への勾配(InGrad)

の伝播を計算する過程である。この演算は Rot180()をカーネルフィルタを 180 度回転させる演算と置くと

$$\boldsymbol{InGrad_i = \sum_{j=o}^{M-1} Rot108(K_{j,i}) \otimes OutGrad_j \ (0 \le i < N)} \tag{6}$$

と表される。このように推論演算もバックプロパゲーションの演算もほぼ同じ形になり、演算量が同じことも特徴である。

○プーリング層

プーリング層は畳み込み層で検出した特徴の位置感度を落とし、位置普遍性を上げるために畳み込み層の後に設置される層である。プーリング演算にもいくつか種類があるがよく使われるサイズが 2x2 の Max プーリングについて説明する。

まず初めに、画像全体をプーリングのサイズで区切る。4x4 の入力画像に 2x2 のサイズでプーリングを行う場合、画像は 4 個の部分に区切られる。この後この区切られたマスの中の最大値を各マスでの出力値として出力する。プーリングによって出力される画像のサイズは 4 分の 1 のサイズになる。

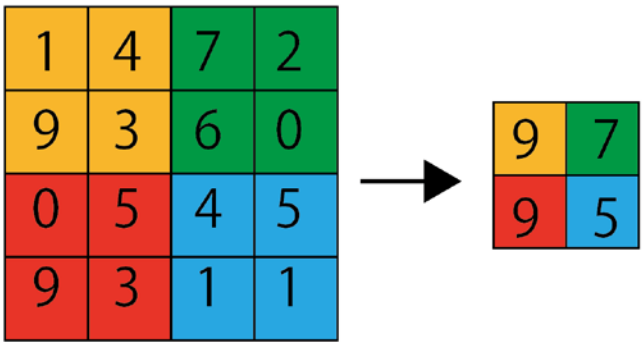

図 9　プーリング層

○活性化層

活性化層は入力画像に活性化関数を掛けたものを出力する。非線形な活性化関数を採用することで CNN 自体が非線形な関数に対応できる。CNN の活性化関数として ReLU（ランプ関数）が良く使われる。ReLU は以下の式で表される。

$$\boldsymbol{\varphi(x) = max(x, 0)} \tag{7}$$

○全結合層

全結合層は入力と出力のすべてが重みで結合している層で、CNN の終段付近に挿入され

る。畳み込み層で抽出した画像の特徴を元に全結合層で画像の分類を行っているとされる。

◯ソフトマックス層

ソフトマックス層はクラス分類を行う CNN で最終層に挿入される層で、CNN の出力ベクトルを、要素の和が 1 である、確率分布になるように正規化する。この層によって CNN の出力ベクトルの要素は、その要素と対応するラベルの該当確率として扱える。入力を x(x_0, x_1, x_2……x_n)、出力を y(y_0, y_1, y_2……y_n)とおくと以下の式で表せる。

$$\boldsymbol{y_j} = \frac{\boldsymbol{e^{x_j}}}{\sum \boldsymbol{e^{x_i}}} \tag{8}$$

◯クロスエントロピー

クロスエントロピーは誤差関数である。画像のクラス識別を行う CNN で多用されるためにここで紹介する。推論結果を y(y_1,y_2……y_n)、正解ラベルを l(l_1,l_2,……l_n)とするとクロスエントロピーは以下の式で表せる。

$$\mathbf{cross\ entrpy} = -\sum_{\boldsymbol{i=1}}^{\boldsymbol{n}} \boldsymbol{l_i} \log \boldsymbol{y_i} \tag{9}$$

2.3.2. 代表的なネットワークモデル

クラス識別に用いられる畳み込みニューラルネットワークは前段に数段の畳み込み層を持ち、後段には全結合層を持ち、最後にソフトマックス層を経てクラスごとの確率分布を出力する。層の構成や数がネットワークモデルによって異なる。代表的なネットワークモデルとして AlexNet [4]、VGG16 [17]、GoogleNet [18]がある。

AlexNet は ILSVRC 2012 で優勝したネットワークモデルであり。近年再びディープラーニングが脚光を浴びた主要因である記念碑的存在である。これは畳み込み層を 5 層、全結合層を 2 層持つ。畳み込み層のカーネルフィルタのサイズは図に示すように 11 から 3 と徐々に小さくなっていく。一方 VGG16 は畳み込み層を 13 層、全結合層を 3 層持つネットワークである。AlexNet と比べて畳み込み層のカーネルフィルタのサイズは 3 で統一されていて、その代わりに畳み込み層の数が多い。これは大きいカーネルフィルタのサイズを持つ畳み込み層は、複数の小さいカーネルフィルタのサイズを持つ畳み込み層と同等の機能を持つとされているからである。例えばカーネルフィルタのサイズが 5 の畳み込み層はカーネルフィルタのサイズが 3 の畳み込み層が 2 層重なっているものを同等とされる。

GoogLeNet は上記 2 つのものよりも新しいネットワークモデルであり、上記の 3 つよりも多くの層を持つ。

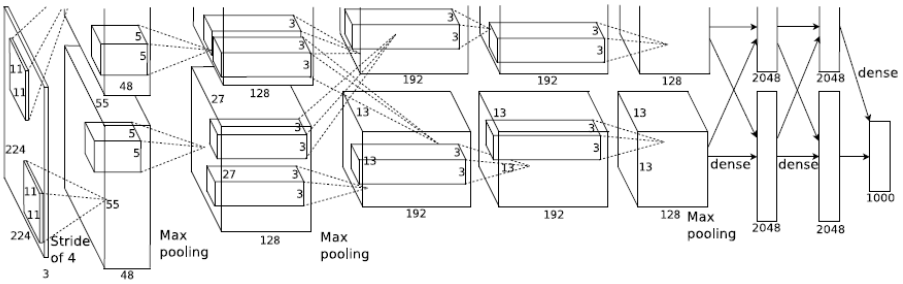

図 10　AlexNet のネットワークモデル [4]

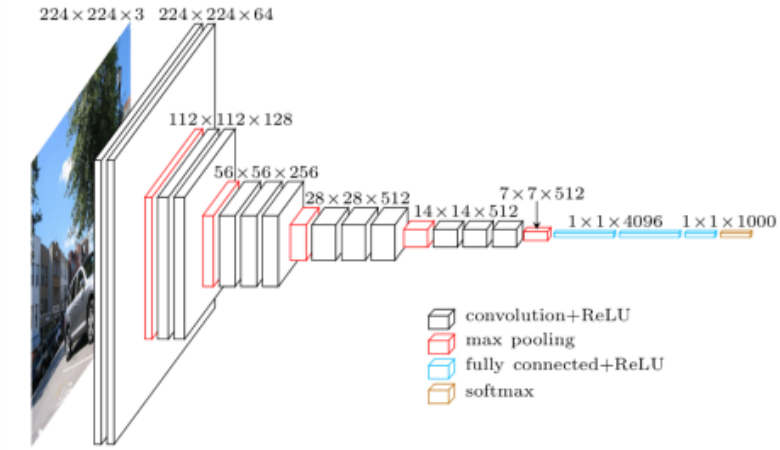

図 11　VGG16 のネットワークモデル　[48]より

2.4. CNN 演算の高速化

CNN 自体は膨大なパラメータを持ちまた演算量も多い。これに加えて学習を終了するまでに数十 epoch 掛かるために CNN の学習には多くの時間がかかる。例えば [14]では 1 つの GPU を使った場合に CNN の学習に 98 時間かかっている。

このため CNN の学習を高速化させるための手法が多数研究、開発されている。CNN の高速化の手法としては大きく分けてアクセラレーターの使用、並列化、ネットワークモデルの削減、低ビットでの演算の 4 つの手法がある。

2.4.1. アクセラレーターの使用

GPU（Graphics Processing Unit）は CPU に取り付けて、CPU の演算を部分的に肩代わりして高速に行うことで演算を加速化する装置である。CPU のような多様な命令セットやメモリ保護機能を持たないが、単純な機能のみのコアを大量に持つことによって高い並列演算性能を持つ。当初は名前の通りに画像処理を行う専用の装置であったが、GPGPU という一般の演算を高速化させるために使う手法が開発されて今では GPGPU 専用に設計された GPU もある。このような GPU を用いることで CPU よりもニューラルネットの演算を

70 倍も高速化できるという報告がある [8]。また同じく演算の高速化のために 1 つのマザーボードに複数の GPU を搭載してそれらを用いて演算を高速化させることもできる [19]。

このように GPU は高い演算性能を持つために TSUBAME をはじめとして、並列計算機システムにも導入されている [20]。

現在の GPU は単独で動作することは無く常に CPU からの命令で駆動されて動作する。つまり GPU で演算を行うプログラムを実行する場合でも、まず CPU でプログラムを起動して、CPU が GPU にカーネルと呼ばれる命令を発行することによって GPU がプログラムを実行する。

並列計算機に搭載される GPU 間でデータの転送を行う場合でも直接 GPU 同士が通信を行うのではなく、一度 GPU が CPU にデータを転送し CPU 同士でノード間通信を行う。しかしこれでは転送の遅延が大きく、転送帯域が狭くなるので CPU を経由せずに GPU 同士で直接データを転送する方式を NVIDIA が GPU ダイレクトとして開発している。

GPU 上でのプログラムのフレームワークとして CUDA [21] や OpenACC がある。Cuda は NVIDIA が開発している言語で OpenMP 等の様に C 言語や Fortrun で記述したプログラムにディレクティブを挿入することで GPU プログラムを記述する。生成しするプログラムの効率は良いが NVIDIA の GPU でしか使えない。一方 OpenACC も同様に C や fortran のプログラムにディレクティブを挿入する。

しかしこのような方式でプログラミングを行うと高い演算性能を発揮するためには GPU のハードウェア部分への理解と細かいチューニングが必要になり、このようなプログラムを作成するのは容易ではない。そこで専用のライブラリやフレームワークが多く存在する。CUDA 上で動く DeepLearning 用のライブラリとして cuDNN [22]がある。

2.4.2. 並列化

逐次実行では同時に 1 つの演算しか行えないが、1 つの CPU の中の複数のコアを同時に使用したり、複数の CPU を使用したりすることで、1 度に複数の演算を同時に行うことで演算を高速化して、実行時間を短縮することが出来る。また複数のプロセッサノードを搭載する並列計算機を使い、多数のプロセッサを使用して演算を行うことで処理の高速化を図れる。並列化の方法としてデータ並列とモデル並列の 2 つに分けられる。

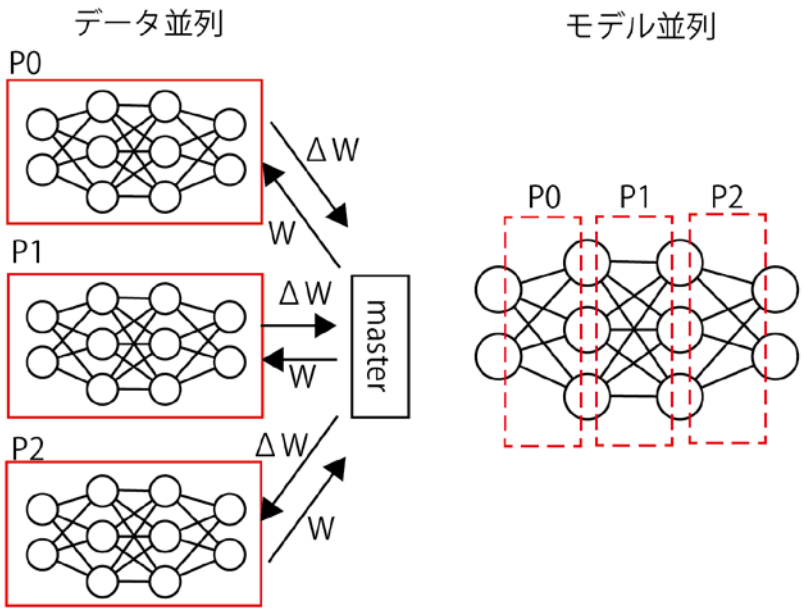

図 12　データ並列とモデル並列の模式図

①データ並列

現在実用的に良く使われているのがデータ並列である。これは各ノードにデータセットを分割して SIMD 的に実行する。目的とするネットワークモデルを各ノードに複製して各々が独立に推論、バックプロパゲーション、パラメータの更新値（ΔW）を求め、パラメータの更新値をノード間で交換して、その平均を求めて、その値でパラメータの更新を行う。こ

の方式で山崎らは 256 台のノードを用いて 1 台の場合と比べ 217 倍のパフォーマンスを得ている [23]。

データ並列の処理ではミニバッチ方式の学習が一般に行われる。各ノードは各々N 枚の

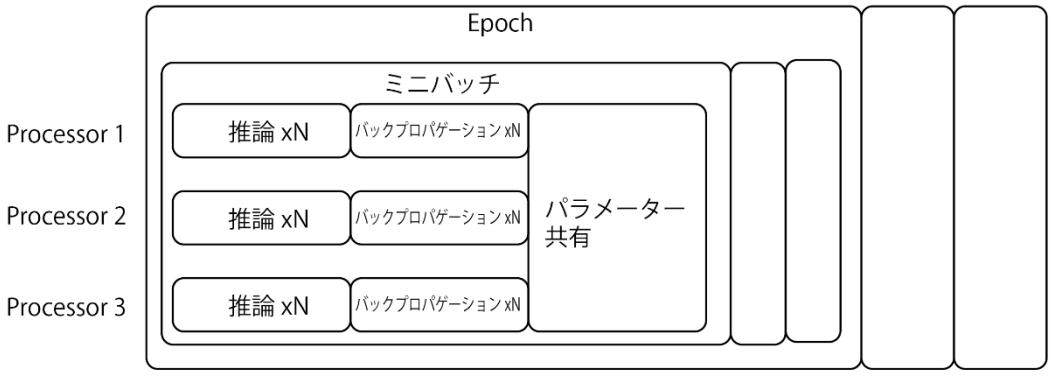

図 13　データ並列の流れ

画像を用いてミニバッチ処理をする（このような各ノードのミニバッチサイズをローカルミニバッチサイズと呼ぶ)。その後各ノードが計算したパラメータの更新値の平均を求めて、すべてのノードがその値でパラメータを更新する。このような手法のためにシステムのノード数が P 個である時にはシステム全体としてのミニバッチサイズは PN になり、これが実質的なミニバッチサイズになる（この値をグローバルミニバッチサイズと呼ぶ)。

このような仕組みであるために必ずミニバッチサイズはシステムのノード数よりも大きくなる。また、CNN のネットワークモデルのパラメータの量は膨大なためにパラメータ交換で通信するデータ量も多く、そのためにパラメータ交換には時間がかかる。パラメータ交換の間に各ノードは演算が出来ないために、演算器の利用率をある程度上げるためにはローカルミニバッチサイズもある程度大きくとる必要がある。

このように多くのノードを用いて高い並列度を達成しようとするとグローバルミニバッチのサイズもある程度大きくならなければならい。しかし前述したようにグローバルミニバッチのサイズを上げようとすると学習率をある程度大きくする必要があり、これは学習の収束率の悪化を招く。これに対応したのが [24] [9]である。 [24]では学習の手法として初期の段階で一般に使われる確率的勾配降下法ではなく RMSprop を用いてを 32K というグローバルミニバッチサイズでの学習を達成している。また [9]では通常 CNN のすべて層の演算で共通である学習率を層ごとに最適化する手法を用いている。このような工夫によって各々は 32K という大きなグローバルミニバッチサイズでの学習を達成している。

②モデル並列

モデル並列ではネットワークモデルを複数に分割し、各ノードが分割した部分の処理を行う。古くは [25]によってモデル並列用のバックプロパゲーションのアルゴリズムが発表さ

れていて古くから研究は行われてきたが、データ並列に比べて適用例は少ない。

Zhang らの研究では AlexNet を 4 つのノードに分割してモデル並列化を行っている [26]。畳み込み層 1 を 1 つ目のノードに、畳み込み層 2 を 2 つ目のノードに、畳み込み層 3 を 3 つ目のノードに、残りの層を 4 つ目のノードに分配することで約 4 倍の高速化を実現している。また [27]では AlexNet のネットワークモデルの枝分かれしている部分でネットワームモデルを分割し 2GPU で演算することによって約 1.6 倍の高速化を達成している。また [28]では 4 層の CNN の学習の演算を層ごとに 4 つの CPU に分割することで約 3 倍の高速化を実現している。

これらのモデル並列化ではノード数が多くなってもミニバッチのサイズが増えることは無い。しかしこれらのモデル並列化は図 14 の層に CNN の学習を層ごとに分割してパイプライン的に処理を行っているために以下のような欠点が存在する。

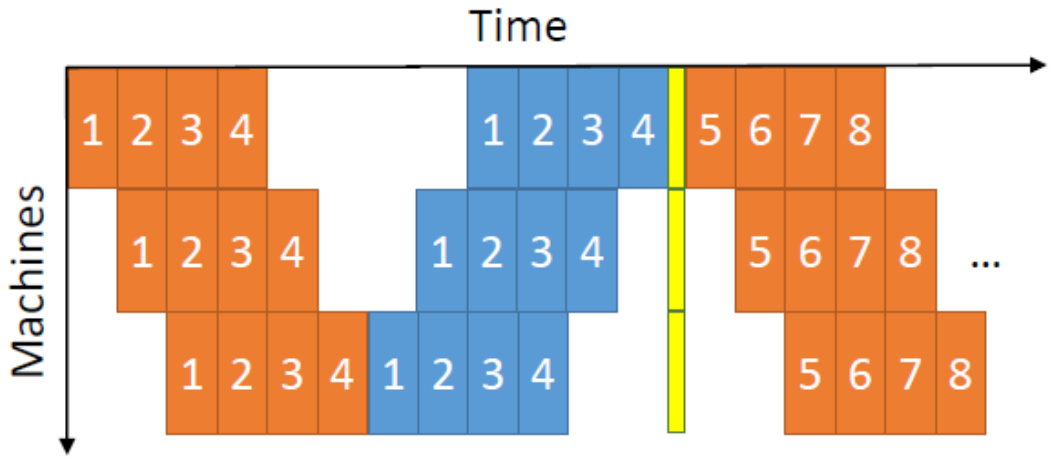

図 14　層単位でのモデル並列　[28]より

1. 層数による並列度の限界
   層ごとにネットワークモデルを分割するために必然的に層の数以上に演算を分割することはできない。例えば AlexNet は畳み込みが 5 層、全結合層が 2 層、VGG16 は畳み込み層が 11 層、全結合層が 5 層である。
2. 各層の演算量の不均衡
   CNN の各層の演算量は均一ではなく例えば AlexNet では最も演算量が高い層と低い層では 10 倍程度の演算量の差がある。これによって単純に 1 つの層に 1 つの演算器を割り当てた状態では各ノードでの演算量が不均衡になり、演算の能力の無駄が発生してしまう。

2.4.3. ネットワークモデルの削減

モデルの削減はネットワークモデルの無駄な部分の演算を省略するという手法である。一

般に使われているネットワークモデルは大量のパラメータをもち大量の演算を行うがそのすべてが必要不可欠であるわけではない。小さい重みの演算を省略する（これは層の重みを 0 にすることに相当する）エッジ刈りやそもそも演算全体に対する影響の小さいノードを特定してそのノードに関する演算を全て省略するノード刈りが研究されている [29]。しかし、これらによって演算量が減っても、削減した後の演算はいわば疎行列の演算に相当するために、これを CPU や GPU で効率よく演算することは難しい。

2.4.4. 低ビットでの演算

長いビット長での演算よりも、短いビット長の演算の方が少ないトランジスタ数で演算器を構成できる。そのため同じ面積のシリコンウェア上になら低ビットの演算のほうが多く詰められるために小ビットの演算の方が高速化出来る。

通常ディープラーニングの演算は 32bit で行われているが上記の様な高速化のために 32bit よりも短いビット長での演算の機運が高まっている。Tesla P100 は NVIDIA 社が発売するサーバー向けの GPU であり単精度浮動小数点(32 bit)でのピーク演算性能は 10.6TFLOPS であるが、半精度浮動小数点(16 bit)では 21.2TFLOPS と倍増する [30]。つまり単精度浮動小数点の演算を半精度に変えるだけで演算が 2 倍高速化される。

CPU や GPU ではあらかじめ回路に組み込まれているビット長でしか演算が出来ないが、FPGA では回路を自由にユーザーがコンフィギュレーションできる。そのため FPGA 上で様々な精度での CNN の演算の実装が行われている。低ビットで演算を実装すると演算性能は上がるが、精度が削減した分だけ演算の量子化誤差が大きくなり学習の効率が悪くなるため、その最適化が研究されている。 [31] [32] [33]。

2.5. 並列計算機

並列計算機とは複数のプロセッサノードがネットワークで接続された計算機で、複数のプロセッサで並列にタスクを処理することで高い演算性能を達成する。並列計算機はその性能ゆえに気象、航空工学、環境などの先端分野でのシミュレーションに使われている。また産業界でも CAE、バイオ、化学、金融の分野で高い精度の設計や予測をするために利用が進んでいる。

並列計算機の各プロセッサノードを接続するネットワークにはいくつかの種類があり、それによって通信性能が変わる。そのうちの 1 つがファットツリーで、この構成では同時通信帯域が高く確保できて、比較的通信の遅延も小さくなるが、大規模な並列計算機に適用することは難しい。一方大規模な計算機である「京」ではメッシュ構造のネットワークが採用されている [34]。

通常の計算機と並列計算機でのプログラミングでの違いの 1 つに並列計算機ではプロセッサノード間通信を記述する部分があげられる。MPI [35]はスーパーコンピューター等の

分散メモリシステムでノード間通信を行うためのライブラリであり、並列計算機でのノード間通信では通常 MPI が用いられる。MPI では 1 対 1 通信、集団通信に加えてバリア同期や時間計測など並列計算機に欠かせない機能をサポートしている。

1 対 1 通信は 1 つの送信ノードと 1 つの受信ノードの間でデータの転送が行われるという基本的な通信である。集団通信はいくつかの種類があるが、今回使用するものとして Bcast、Reduce、AllReduce について説明する。Bcast は 1 つの送信ノードと複数の受信ノードが存在し、その間で転送が行われる。送信ノードが送信するデータのすべてが、各々の受信ノードで受信されるので、転送後は送信ノードの持っているデータのコピーを各々の受信ノードが持つことになる。

Reduce は複数の送信ノードと 1 つの受信ノードが存在する演算である。各送信ノードの持っているデータすべてで特定の演算をした結果を送信ノードに転送する。例えば図のように Node1~4 が持っている Data1~4 で足し算の AllReduce をした場合は Data1+Data2+Data3+Data4 を受信ノードは受信する。

AllReduce は複数の送信ノードと複数の受信ノードが存在する転送で、Reduce では演算の結果を受信するノードは 1 つしかいなかったが、AllReduce では転送に参加した全ノードが演算結果を受信する。

以上 3 つの集団通信はもちろん複数の 1 対 1 通信でも実現は可能だが、ネットワークが集団通信をサポートしていれば複数の 1 対 1 通信を使用するよりも高速に転送ができる。例えば Bcast ではネットワークがマルチキャストを行うことでプロセッサノードのネットワークの性能を超えるデータを転送することが出来る。また AllReduce 演算も Reduce 演算と Bcast 演算の組み合わせでも実現できるが、ネットワークが Recursive Doubling 等のアルゴリズムに大王することでより高速に実行できる。

MPI そのものは通信 API の規格であり、実装としては有名なものとして Intel MPI、MPICH、OpenMPI がある。

MPI 自体は C 言語と Fortran 言語しかサポートしていないが、mpi の python ラッパーである mpi4py [36]を使うことで Python を使って並列計算機用のプログラムを記述することが出来る。

3.提案手法

前述したように CNN のデータ並列ではミニバッチサイズの制限に起因してデータ並列の並列度に上限が存在する。これが今後さらなる高速化を図るために並列度を上げていく際の障害となる。本節では既存のデータ並列に CNN のチャネル方向での並列化を使用するモデル並列を導入したハイブリッド並列を提案する。

3.1. ハイブリッド並列

データ並列では 1 つ 1 つのノードが CNN 全体の学習の計算を行うが、これにモデル並列を導入することでモデル並列・データ並列を混合したハイブリッド並列が行える。1 つの CNN をモデル並列により分担して学習をする複数のノードの集合を「セット」と呼ぶ。このセットを複数用意してそれらでデータ並列を行う。「セット」の内側では複数のノードで 1 つのネットワークモデルの演算を行うモデル並列を行うが、セット間ではデータ並列を行う。

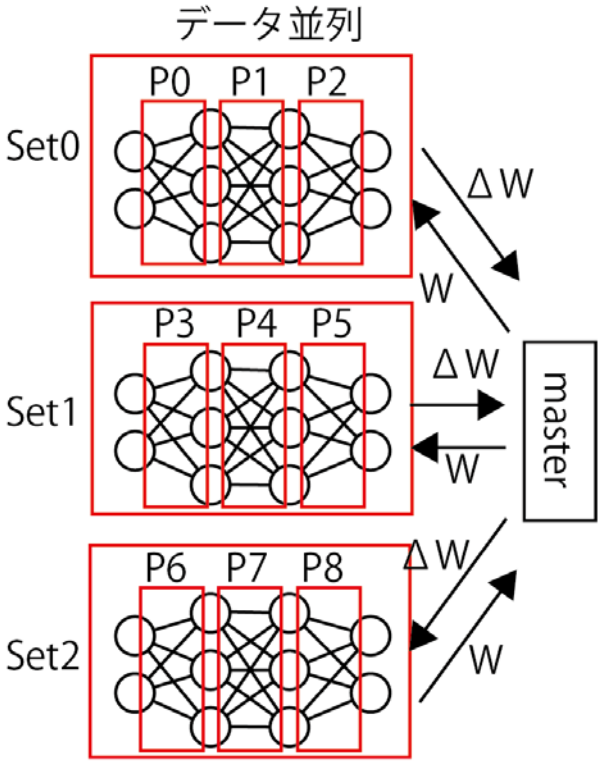

図 15　ハイブリッド並列

3.2. 使用するモデル並列化の検討

ハイブリッド並列化を行うにあたっては上記の CNN 学習の層ごとの並列化のみでは不十分である。まず全体の演算量のほとんどを占める畳み込み層については、一般的な CNN

では畳み込み層によって演算量は異なる。これを無視して層ごとに並列化をするとノードによって演算量にばらつきが生じ、これが性能低下の原因となってしまう。また、一般的なCNN では全体のほとんどのパラメータを 1 つの全結合層が持つ。そのために層ごとの並列化ではパラメータが一部の層に手中し、データ並列において必要になるパラメータ交換の速度向上は望めない。
そこで本報告では上記の 2 つの点に考慮して層ごとのモデル並列化に加え、チャネル方向での層の並列化を導入した。

3.2. CNN のチャネル方向並列化

前述したような理由により層単位での処理の分割だけでは不十分である。そのために本論文では CNN の各層をチャネル方向で処理を分割するチャネル方向並列化を導入する。本節では CNN を構成する主な層である畳み込み層と全結合層についてチャネル方向並列化の手法を検討する。

3.2.1. 畳み込み層の並列化

CNN では演算量の大部分は畳み込み層による。畳み込み層をチャネル方向に分割することによってバランスの良い分割が期待できる。

前述したように 1 チャネル入力（Input）と 1 つ畳み込みフィルタ（W）から 1 チャネルの出力（Output）を得る畳み込み演算を$\mathrm{Output} = \mathrm{w} \otimes \mathrm{Input}$と置くと、n チャネルの入力（$\mathrm{Input}_0$〜$\mathrm{Input}_{n-1}$）を受け取って m チャネルの出力（$\mathrm{Out}_0$〜$\mathrm{Out}_{m-1}$）を行う畳み込み層の演算は

$$\mathrm{Output}_j = \sum_{i=0}^{n} W_{j,i} \otimes Input_i \quad (0 \le j < m)$$

と表せる。ここではパラメータ j に従って畳み込み層の演算を分割することを出力チャネル方向の分割、パラメータ i に従うものを入力チャネル方向の分割と呼ぶ。これらどちらに従って畳み込み層を 2 つのノードに分割した場合も、演算量は 2 つのノードに均等に分配出来る。しかし通信データ量については異なる。

3.2.1. 入力チャネル方向での分割

入力チャネル方向で層の分割化を行うと、図 16 の様に、前の層からの入力はチャネル単位で分割して複数のノードに分配される。各ノードでは演算した結果は出力フィルタの部分値となる。各々のノードの出力の値を合計した値が本来の出力となる。また畳み込みフィルタも入力方向に分割した物を各々のノードが持つ。
この方式で演算を D 個に分割した場合は、1 ノードあたりの演算量は 1/D 倍、入力の通信量は 1/D 倍、出力の通信量は 1 倍になる。

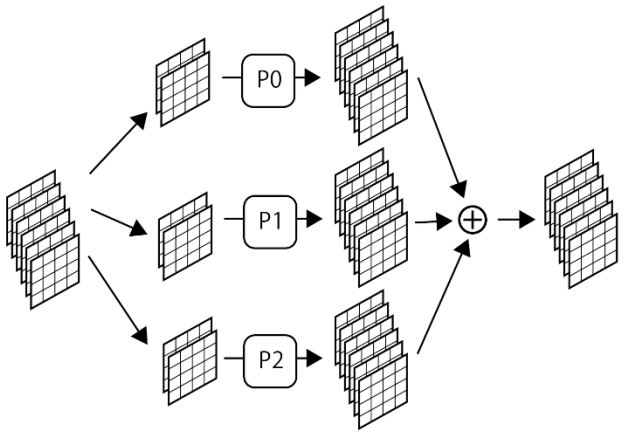

図 16　入力チャンネル方向での分割

3.2.2. 出力チャネル方向での分割

出力チャネル方向で分割する場合、図 17 の様に前の層からの入力は全てのノードに送られるされる。各ノードでの演算結果は出力の部分になるので、各ノードの演算結果を集約した物が本来の結果となる。また分割ノードは、カーネルフィルタの部分を持つ。

この方式では処理を D 分割すると 1 ノードあたりの入力は 1 倍、計算量は 1/D 倍、出力データ量は 1/D 倍になる。

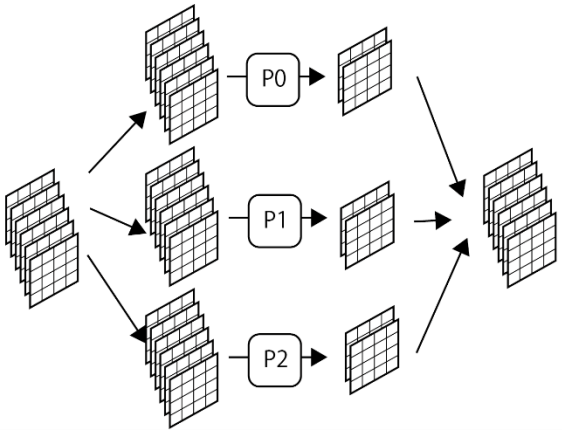

図 17　出力チャンネル方向での分割

3.2.3. 各方式の比較

これらのことをまとめると以下の表の様にまとめられる。

各方向に演算を D 分割した場合のノードあたりの演算量、パラメータ量、ノード間通信量を表 2 に示す。

表 2　各分割方式の比較（D 個のノードで並列化した場合）

分解方向	入力データサイズ	出力データサイズ
(1)入力 I	1/D	1
(2)出力 O	1	1/D

3.3. 全結合層のチャネル方向分割

全結合層の演算は単純な行列-ベクトル積となる。つまり n 要素の入力（in_0〜in_{n-1}）を受けとって m 要素の出力（out_0〜out_{m-1}）を行う全結合層の演算は

$$out_j = \sum_{i=0}^{n} w_{j,i} in_i \quad (0 \le j < m)$$

と表せる。この時、パラメータ j に従って全結合層の出力を分割することを出力チャネル方向での分割と呼ぶ。全結合層を分割することで、パラメータを複数のノードに分散させることが出来る。これによってノードあたりの交換すべきパラメータ量が減るためにパラメータ交換の処理を高速化することが出来る。

4. 実行時間予想モデル

本節では CNN の学習に掛かる処理時間をモデル化して同じ CNN の学習をデータ並列、ハイブリッド並列で実行した時の各々の処理時間を算出し、ハイブリッド並列の性能についての比較を行う。

4.1. 想定環境

VGG16 [37]を元に評価用に一部を単純化した、以下の図 18 に示す CNN を今回の評価の対象とした。サイズが 224x224 の画像が 64 チャネルある入力を受け取り、4 つの畳み込み層と 1 つの全結合層を経て 4096 次元の確率ベクトルを出力する。また各層での演算量、パラメータ量、入出力データ量は以下の表 3 のようになる。全結合層 cv1～cv3 では演算量が等しいが、cv4 の演算量はその半分になる。また全体の 97%ものパラメータが fc1 に集中している。

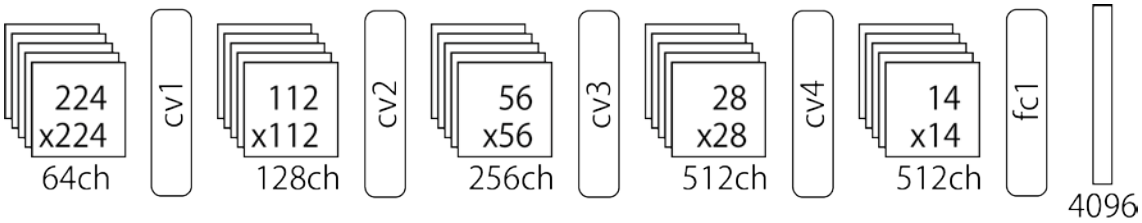

図 18　評価用ネットワークモデル

表 3　評価用ネットワークモデルの各処理量

	演算量（flop）	Param（B）	Input（B）	Output (B)
cv1(L1)	7.40G	0.29M	12.8M	6.42M
cv2(L2)	7.40G	1.18M	6.42M	3.21M
cv3(L3)	7.40G	4.72M	3.21M	1.60M
cv4(L4)	3.70G	9.43M	1.60M	0.40M
fc1(L5)	0.21G	411M	0.40M	4096

またこの CNN の学習の並列化を実行する並列計算機については以下の性能を仮定した。これらの値はのちの評価実験で用いる ReedBush の実効性能を参考にした。

表 4　想定する並列計算機の性能

ノードあたり演算性能	4T FLOPS
ノードあたり	400G B/s

メモリアクセス性能	
ネットワーク通信性能	4GB/s
ネットワーク接続	Fat tree

4.2. 並列化シナリオ

比較条件としてデータ並列と、3 種類のモデル並列を用いたハイブリッド並列の計 4 通りのシナリオを今回考慮した。各ハイブリッド並列での 1 セット構成を以下に示す。

① データ並列

② ハイブリッド並列 1（5 ノード）

1 セットは 5 つのノードからなる。CNN を層単位で 5 つに分割した。

③ ハイブリッド並列 2（6 ノード）

1 セットは 6 つのノードからなる。全結合層である fc1 のみチャネル方向に 2 分割し、全結合層は層ごとの分割のみを行った。

④ ハイブリッド並列 3（9 ノード）

1 セットは 9 つのノードからなる。畳み込み層の内 cv1～3 をチャネル方向に 2 分割し、cv4 は層ごとの分割化のみ、全結合層 fc1 もチャネル方向に 2 分割した。

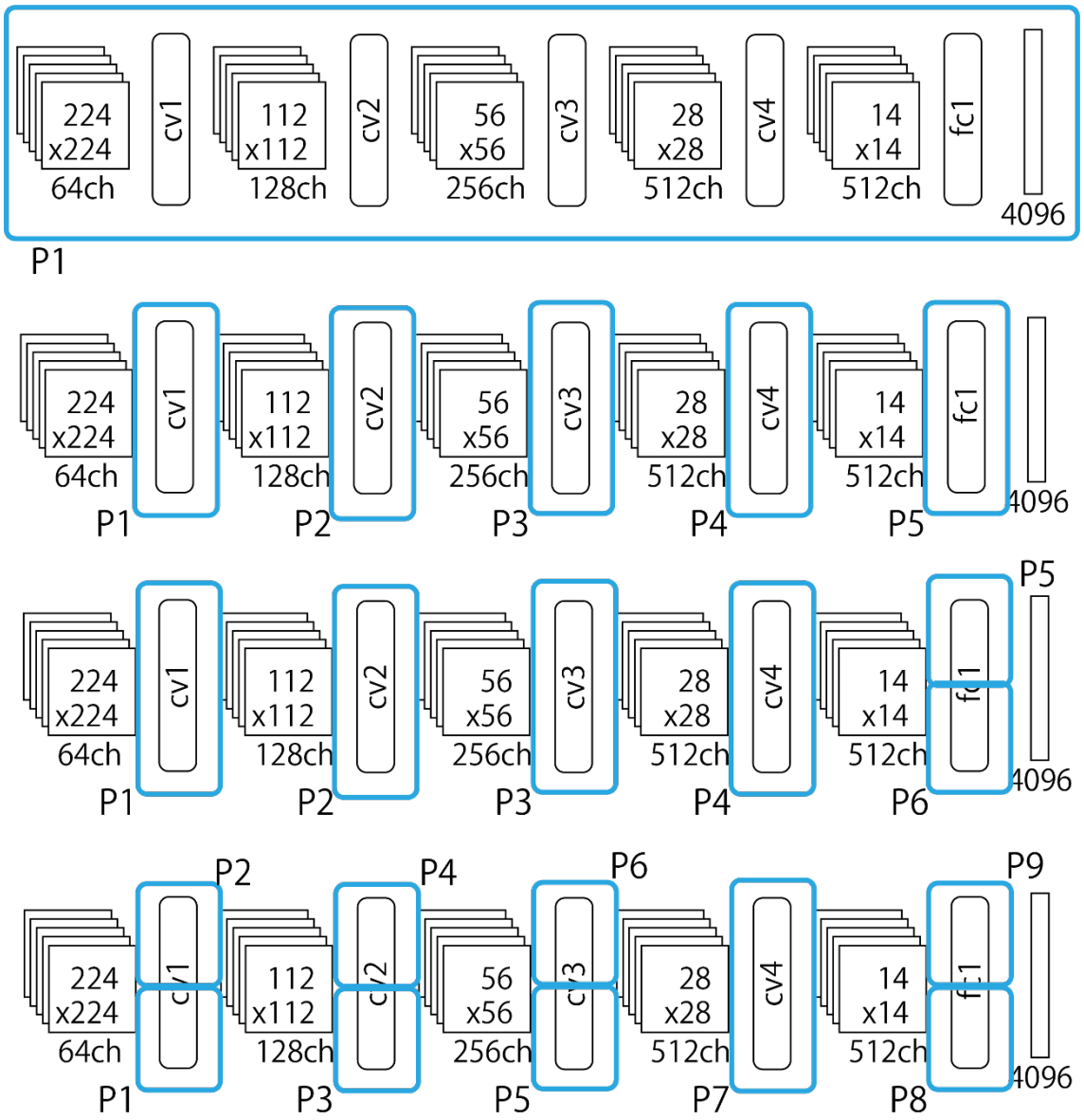

図 19　並列化シナリオ一覧

4.3. データ並列の実行時間モデル

データ並列の処理は図 20 にあるように 3 つの部分に分かれる。各々の実行時間をモデル化してその和をデータ並列における実行時間とする。以下ローカルミニバッチサイズを N として 1 回のミニバッチ処理を行う時間について考慮する。

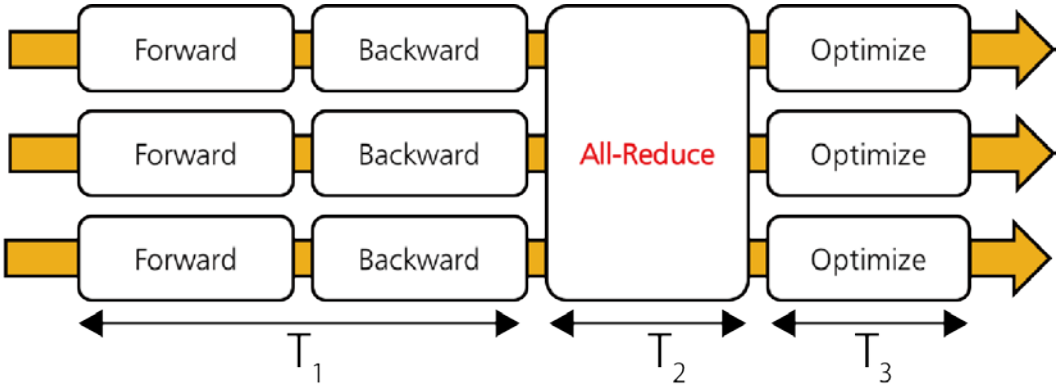

図 20　データ並列の流れ

まず、CNN の各層での処理時間だが、畳み込み層では演算が隘路となるので層 Li における処理時間 T_{Li} は層の演算量 $Copm_{Li}$ とノードの演算性能 Pnode を用いて$T_{Li} = \frac{Comp_{Li}}{Pnode}$と表せる。一方畳み込み層ではメモリ隘路となる。よって層 Li における処理時間 Ti は層のパラメータ量を $Param_{Li}$、ノードのメモリアクセス性能を Mnode と置くと$T_{Li} = \frac{Param_{Li}}{Mnode}$と表せる。

データ並列で各ノードが独立に学習を行う時間(T_1)では N 回の推論と N 回のバックプロパゲーションを行う。推論とバックプロパゲーションの演算量は同じなので以下の式で表される。

$$T_1 = (T_{cv1} + T_{cv2} + T_{cv3} + T_{cv4} + T_{cv5}) \times 2N$$

パラメータ交換を行う部分(T_2)は CNN の全体のパラメータのサイズとネットワーク転送性能(NWnode とおく)から以下の式の様に求められる。

$$T_2 = \frac{Param_{cv1} + Param_{cv2} + Param_{cv3} + Param_{cv4} + Param_{cv5}}{NWnode}$$

最後のパラメータ更新の時間(T_3)はパラメータ更新の演算量は推論やバックプロパゲーションと同じであるので以下の式で表される。

$$T_3 = (T_{cv1} + T_{cv2} + T_{cv3} + T_{cv4} + T_{cv5})$$

4.4. ハイブリッド並列の実行時間モデル

ハイブリッド並列でも図 21 に示すようにモデル並列と同様に 3 つの部分に分けることが出来る。パラメータ交換の時間とパラメータ更新の時間は比較的簡単に求められるが、各ノードが独立に学習を行う部分では、推論とバックプロパゲーションを複数のノード間でパイプラインを組んで処理を行うので実行時間の計算は複雑になる。

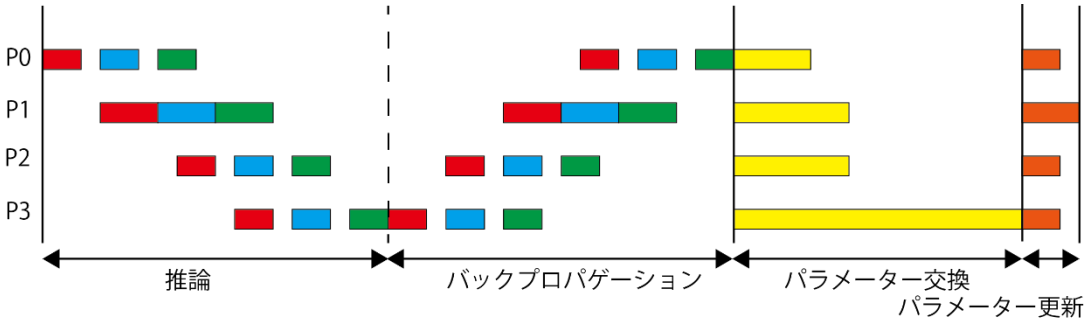

図 21　ハイブリッド並列の 1 セットの流れ

推論とバックプロパゲーションは対称な演算なのでここでは推論部分のみに注目する。まずノード 1 で 1 番目のデータの L1 での処理を行う、処理が終わったら出力を次の層を担当するノードに送って、転送終了後ノード 2 でデータ 1 についての L2 でのデータの処理を行い、終わり次第出力を次のノードに送信する。またノード 1 ではデータ 1 の処理が終わり次第、速やかにデータ 2 の処理に移る。この時に重要になる指標として interval と latency がある（図 22）。interval はデータ 1 の処理を始めてから次のデータであるデータ 2 の処理を開始するための間隔でスループットの逆数である。これは以下の式で表せる。

$$\text{interval} = \max(T_{cv1}, T_{cv2}, T_{cv3}, T_{cv4}, T_{cv1})$$

latency はノード 1 がデータ 1 の処理を始めてから最後のノードがデータ 1 の処理を終えるまでの時間である。層 i から層 j への通信時間を U_{ij} とすると

$$\text{latency} = T_{cv1} + T_{cv2} + T_{cv3} + T_{cv4} + T_{cv1} + U_{12} + U_{23} + U_{34} + U_{45}$$

と表せる。チャネル方向に層を並列化した場合は層の処理時間である T_{Li} との時間が並列化した分だけ小さくなるそのために interval と latency の両方が小さくなる影響を及ぼす。この 2 つの値を用いてハイブリッド並列で各セットが独立にミニバッチ学習を行う時間(T_1)は

$$T_1 = (\text{interval} \times (\text{N} - 1) + \text{latency}) \times 2$$

と表せる。

層間並列化のみの場合は図 22（上）の様に 1 番処理時間の長いノードが interval を決めて、他のノードでは処理をせずに待機する時間が発生する。この時に一番処理が重い層を図 22（下）の様に 2 つのノードに 2 分割をすると interval に対して支配的だったノードの実行時間が短くなるために interval も短縮される。ノードの分割の仕方を工夫することで、ノードが処理をせずに待機する時間を最小化できる。

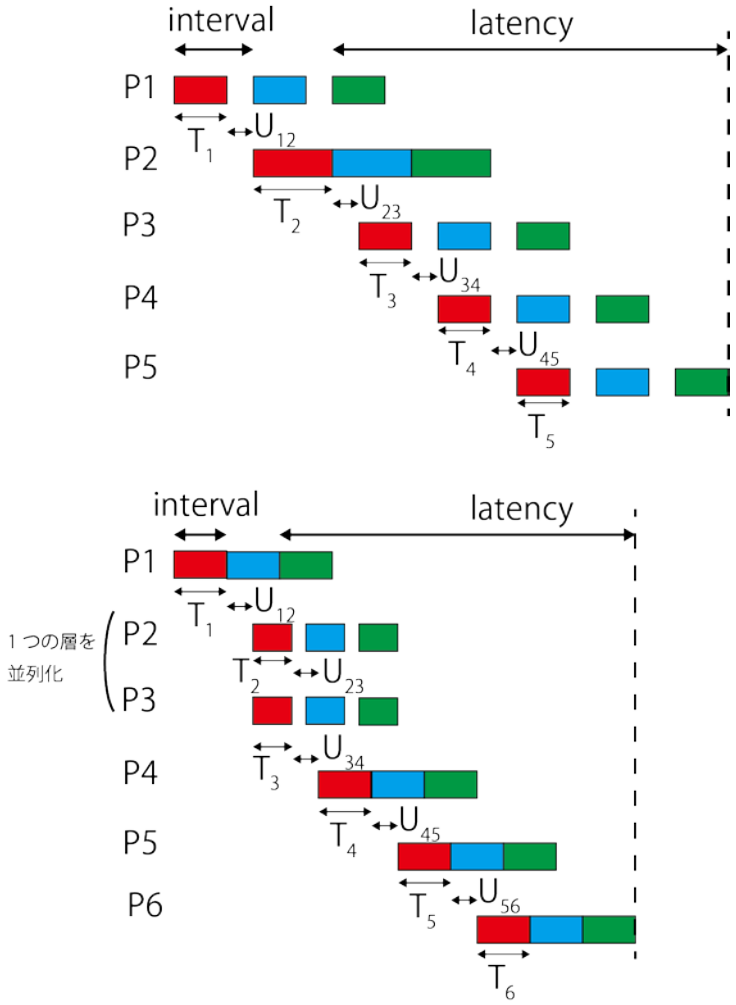

図 22　推論のパイプライン処理
（上）は層単位のみの分割
（下）第 2 層にチャネル単位の分割を導入

パラメータ交換を行う時間（T_2）はセットの中の各ノードが別々にパラメータ交換を行うので

$$T_2 = \frac{\max(Param_{cv1}, Param_{cv2}, Param_{cv3}, Param_{cv4}, Param_{fc1})}{NWnode}$$

と表される。
最後のパラメータのアップデートを行う時間は

$$T_3 = \max(T_{cv1}, T_{cv2}, T_{cv3}, T_{cv4}, T_{fc1})$$

と表せる。

4.5. 比較

1 ノードあたりのグローバルミニバッチサイズを変更して各方式での処理時間の理論値

を求めた。その結果を図 23 に示す。この値が小さいほど（グラフでは左側に当たる）、ミニバッチサイズが小さく、並列度が高い（使用しているノード数が多い）状況を表している。ハイブリッド 1 方式（hv1:5 nodes）は層単位のモデル並列化のみを使用している方式だが、グラフの全域においてデータ並列（data）よりも時間がかかっている。ハイブリッド並列 3（hv3:0 nodes）はグラフの領域においてはデータ並列よりも処理時間が短く高速に動作している。

特に 1 ノードあたりのグローバルミニバッチサイズが小さい値でデータ並列との処理時間の差が大きい、これは現在以上の大規模な並列度で実行しようとすればするほど、提案方式のハイブリッド並列に優位性があることを示している。

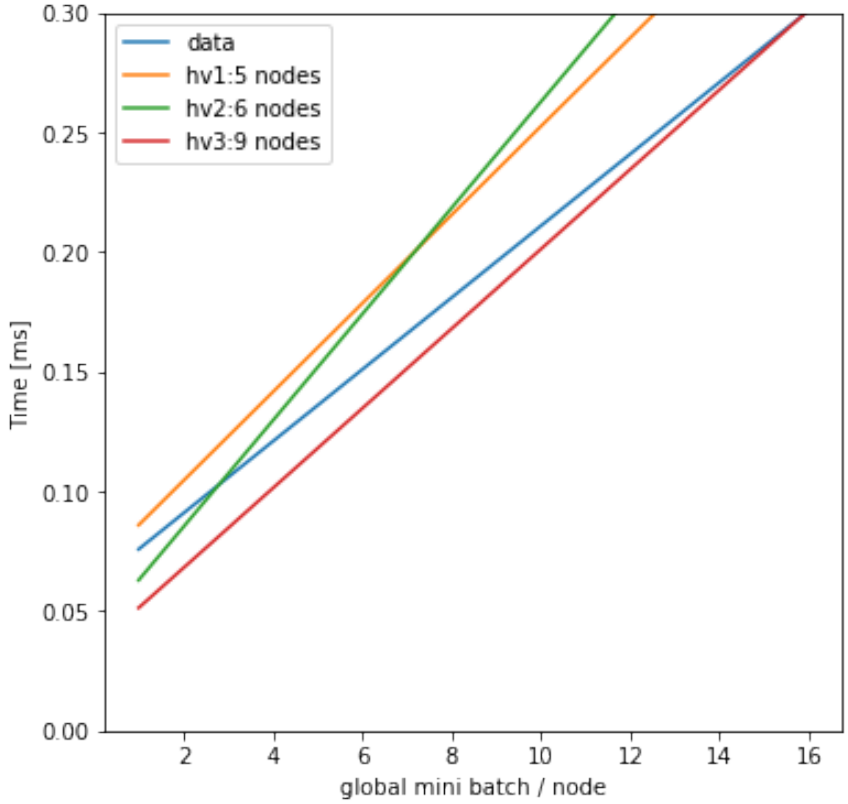

図 23　各並列化方式の処理時間

5. 超広帯域インターコネクトを想定した同期型アーキテクチャーと通信方式

前節のようなハイブリッド並列化ではチャネル方向並列化を用いるが、これにより各ノードでは小さいサイズでの演算が行われる。またデータ並列ではミニバッチ毎に 1 回のノード間通信しか必要としないが、本提案手法ではデータ毎にノード間通信が発生する。このような小さい演算サイズ、細粒度での通信は現在の一般的な並列計算機のアーキテクチャーとは異なるものである。

また現在半導体の微細化の速度の低下によりムーアの法則の限界が近づきつつある。そのためにシングルプロセッサの演算性能の向上は限界に達しつつある。しかし広域網用光通信技術の導入により超広帯域な光インターコネクトを並列計算機に適用する機運が高まっている。これによって並列計算機のネットワーク性能が爆発的に向上することが期待され、これを十分に活用することで並列計算機の性能向上を図ることが考えられる。しかし、既存の通信方式にそのまま超広帯域通信を適用するといくつかの問題が起こる。そのため、超広帯域通信を活用するためには従来方式とは違う新しい方式を採用する必要がある。

まず、今後並列計算機間での通信に適用する可能性が高まっている光による超広帯域インターコネクトについて説明し、次にこれを既存の通信方式に適用したときの問題について述べる。その後に将来の超広帯域インターコネクトを見据えた新しい通信方式を提案する。

5.1. 超広帯域光インターコネクト

広域網ではすでに超広帯域な光通信技術が普及している。現在はコストや大きさの問題があるために並列計算機に適用するのは難しいが、将来的には導入が期待されている。本節では超広帯域な通信に必要な伝送と交換の技術について述べる。

5.1.1. 波長多重化通信

1 つのファイバーコア上で異なる複数の波長を用いたり、様々な変調方式を用いたりすることで超広帯域な通信が出来る。広域網用向けには 1 波長あたり 680Gbps の通信を 46 波長で多重化させてコアあたり 31.3Tbps の通信を行うという研究がある [38]。

このような通信は広域網では使われているが、光源のコストや消費エネルギーの問題で並列計算機への応用には困難がある。解決策として共通の光源から各通信ノードに光源を供給する光バンク方式がある [39]。この方式では光源はシステム全体で 1 つしか必要ない。またプロセッサと光トランシーバを別のパッケージとして設計するとそれらの間をプリント基板上の配線で接続するが、この部分のピン数の制限により帯域が制限される。これはシリコンフォトニクス技術で光トランシーバを製作し、プロセッサと同じパッケージに組み込むことで解決できる [40]。

5.1.2. 光サーキットスイッチ

光ネットワークを形成するには光インターコネクトに加えて光交換技術が不可欠である。しかし波長多重化通信による Tbps クラスの信号を O-E、E-O 変換をして電気的にパケットスイッチすることはコストや消費エネルギーの問題で難しい。

光サーキットスイッチでは光信号を光信号のまま交換を行うために Tbps クラスの信号の交換に適している。また途中で電気に変換するステップを踏まないので通信レイテンシーが小さく一定に抑えられる。

このような光スイッチでは経路を切り替えるために数μs〜数 10ms の時間がかかる [41]。そのために経路の切り替えを高頻度に行うと通信不可能な時間の割合が上がり有効な通信帯域が落ちてしまう。

5.2. 超広帯域インターコネクト用で従来型通信方式を用いる際の問題点

前述のような超広帯域光通信を既存の通信方式に適用出来れば、現状の資産を活用したまま並列計算機の性能向上を図ることが出来る。しかし、現実にはいくつかの問題が存在する。これらは CPU によるソフトウェア処理とメモリ間コピーを前提とした通信という従来型通信方式の構成に原因である。

5.2.1. バッファを使用した従来型通信とフローコントロール信号の役割

従来型の通信を図 24 に示す。通信が発生した時には

(1) 送信側の CPU は送信したいデータをメインメモリのバッファに転送する。
(2) 送信側の NIC が送信側バッファのデータを読んで転送を開始する。
(3) 受信側の NIC はデータを受信したらそれを受信バッファに格納する。
(4) 受信側の CPU はメインメモリの受信バッファからデータを読み取り適宜処理を行う。

この時にフローコントロール信号は送信側と受信側の転送量を調停する機能を持つ。受信側の処理能力を超えて送信側がデータを送信したら受信側のバッファが溢れてデータを取りこぼしてしまう。これを防ぐために受信側のバッファが溢れそうになると受信側から送信側にフローコントロール信号を送り、(2)の段階で送信を抑制する。

このように従来型の通信は CPU による処理とバッファを介した通信であることとデータとは逆方向のフローコントロール信号があることで特徴づけられる。

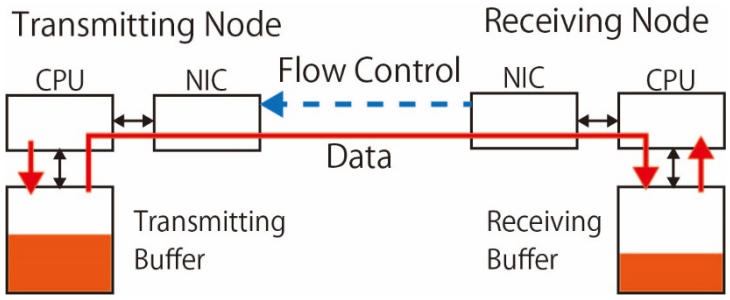

図 24　バッファを用いた通信

5.2.2. 受信処理の間隔の問題

前述したように受信側の CPU は受信バッファから間欠的にデータを読み出すので、少なくともある読み出しから次の読み出しの間に受信するデータを全て格納できる容量のバッファを持つ必要がある。通常オフチップの DRAM を使えば十分なバッファを構築することが出来るが、ここでは通信の帯域が非常に大きいことを想定しているために DRAM の使用は困難である。

現在、広帯域な DRAM の規格として HBM2 がある。これでも転送帯域は 256GB/s ほどで、前節で述べたような光インターコネクトと比べると帯域は小さい。このため光インターコネクトから受信したデータは CPU 上のオンチップの SRAM にバッファリングをする必要がある。

しかし、オンチップの SRAM 上には高々数十 MB 程度のバッファを確保するのがせいぜいである。最新の CPU である Intel Core i7-6950X のキャッシュは 25MB である。このために超広帯域な光通信のバッファをオンチップの SRAM 上に構築すると、受信側のプロセッサがデータを短い間隔で処理する必要がある。例として 32Tbps の通信を 25MB のメモリでバッファをすることを考えると少なくとも少なくとも約 6μs 毎に受信データの処理をする必要がある。しかし現在の一般的な CPU によるソフトウェア処理では、割り込みレイテンシーやコンテキストスイッチに数十μs かかることや [42]、一般的に CPU 上で実行される OS である Linux のスケジューリングの粒度が細かくても 1ms 単位であることを考えるとこのような受信データの処理は難しい。

5.3. FPGA

FPGA(Field-Programmable Gate Array)は製造後に内部回路をプログラム可能な素子である。ASIC に比べてチップあたりのコストが高い、動作周波数が低いという欠点はあるものの現場レベルで回路を変更できることや開発コストが低いために近年利用される機会が増えている。FPGA でコンピューティングを行うと処理を行うタスクに対して最適なプロセッサを FPGA 上で構成して処理を行える。

FPGA では内部で処理のパイプラインを自由に設計が出来るために畳み込み層を実装す

る場合でも演算の対象とする演算のサイズが大きい場合でも小さい場合でもほぼ同程度の演算性能を持つ演算器を構成できると考えられる。

FPGA 上には数種類の回路資源があり、IP コアはその処理内容に応じて一定数の回路資源を必要とする。必要とする回路資源量が少なくなればなるほど 1 つの FPGA 上に多くの該当する IP コアを実装することが出来るので大きな演算サイズに対応できると言える。

FPGA 上に実装する回路は基本的には HDL（Hardware Description Language）で IP コアを記述する。近年では効率的な開発のために C 言語などのプログラミング言語に近い言語で高レイヤの記述を行い、それを HDL に変換する高位合成を用いた開発が行われている。

5.4. 超広帯域インターコネクトを見据えた通信方式の概要

前節で述べたように CPU によるソフトウェア処理とバッファ・フローコントロールを用いる通信に超広帯域インターコネクトを適用することには難点がある。そこで超広帯域通信に適合する同期型アーキテクチャーを本節では提案する。

まず目的とする演算をデータフローグラフで表し、その演算を高速で行う巨大な論理回路を設計する。この時に回路が 1 つの半導体チップに載る規模よりも大きくなってしまうと複数の半導体チップに分割して実装する必要がある。本方式では最初に製作したデータフローグラフのノードを単位としてその境界で回路を分割実装する。

5.3.1. 高速に処理を行う大規模回路の複数チップへの実装

前節で述べたように超広帯域インターコネクトを適用すると時間的に細かい粒度での処理が必要になるので CPU によるソフトウェア処理ではなく、専用回路による処理を行う。

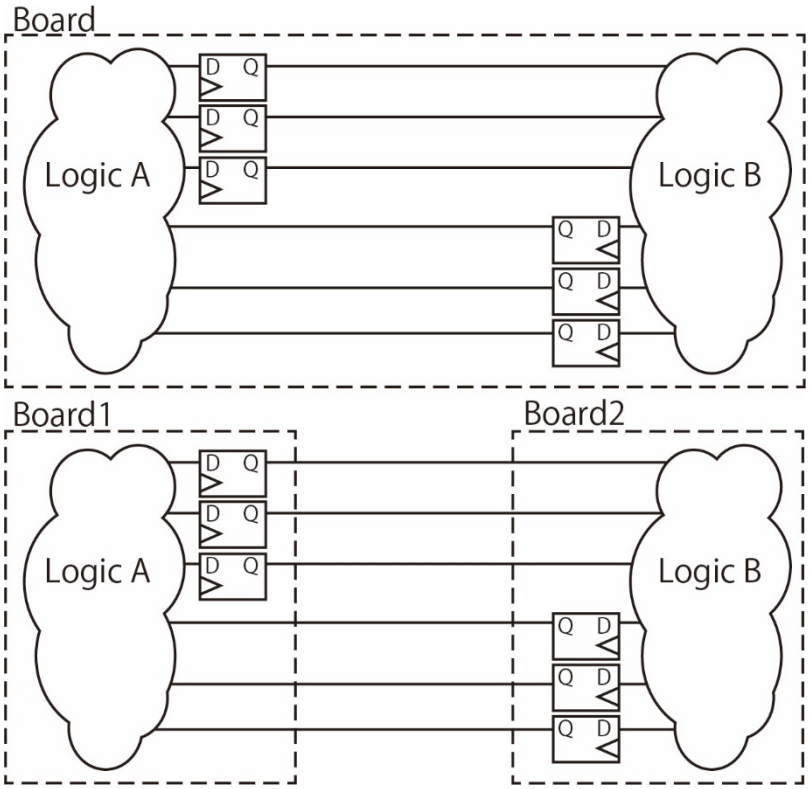

図 25　回路の分割実装
（上）1 チップでの実装　（下）分割実装

まず目的とする処理を高速に演算する大規模な回路を設計する。一般に演算量の多い処理では、演算器を数多く並べて演算の並列度を上げることで高速な処理を行うことが出来る。つまり大規模な回路で実装することで処理能力の高い回路が出来る。通常は回路の大きさが 1 枚のチップに載る大きさに制限されるが、本方式では回路を複数のチップに分割実装をすることで 1 チップに収まらないような大規模な回路を実装する。最も原始的な分割の仕方を図 25 に表す。このようにロジック回路のフリップフロップの出力の部分で分割することで元の回路をそのままで分割実装できる。

5.4.2. 回路の複数チップへの実装における問題

大規模な回路を複数のチップに分割して実装した場合にチップ間の信号線の数の問題と動作周波数が低下する問題の 2 つが発生する [43]。また複数チップ間での同期の問題も発生する。1 つ目の問題は、例えば 2 つのチップの間に 100 本の信号線が必要な場合には各々のチップに 100 本の物理的な I/O ピン必要になるという問題である。実装する回路が大規模になるにつれてチップ間の信号線の本数も増し、チップの持つ物理的な I/O の数よりも多くの信号線が必要となることは容易に起こりえる。先行研究 [43]では回路間の信号線の伝達に高速シリアル I/O を用いることで解決している。送信側でパラレルな信号線をシリア

ル化して受信側でパラレルに戻す。この方式を採用することで物理 I/O ピンの数以上の信号線の伝達をチップ間で行っている。本方式ではチップ間を超広帯域インターコネクトで接続するためにより多くの信号線を分割面で扱うことが出来る。
2 つ目の問題は回路を分割するとチップ間通信の遅延の分だけ回路遅延が増加して、結果として回路の動作周波数が下がるという問題である。この問題のために先行研究では 20MHz 程度と本来のチップの性能と比べるとだいぶ低い周波数で駆動している。
本研究では高速な処理を目的としているためにこの問題を解決しなければならない。そもそもこの問題は元の回路をそのまま分割実装していることで発生する。本報告では使用する回路に制限をつけて上記のような遅延が生じても演算結果が変わらない回路のみを使用することで、この問題を解決する。詳細は次節以降で述べる。
3 つ目の問題は元の回路は同期式であるために、分割した回路も同期式で動かす必要があることである。そのためには同一のクロックをシステム全体に分配する必要がある。

5.4.3. データフローグラフを用いた回路設計

データフローグラフは、演算の半順序関係を表す循環構造の無い有向グラフである。グラフのノードが演算を示し、エッジがデータの転送を示す。
データフローグラフはあくまでも半順序関係しか規定しないので並列化できる部分が多々ある。そのためにデータフローグラフで記述された処理は並列計算と親和性が高い。本研究ではまずデータフローグラフを用いて回路を設計し、それから分割実装することを検討する。
今回は例として以下の図 26 のような 4 つのノードを持つデータフローグラフを実装することを考える。
これらの 4 つの演算ノードは独立に設計することが出来るので、これらをまず別々の回路として設計し、それらを接続することで全体の回路を得る。
この時に各回路の演算の遅延が一定になるように設計する必要がある。つまり演算に DRAM や CPU によるソフトウェア処理を使用することが出来ない。もし遅延が一定にならないと図 26 の Node D の様にデータが合流するノードに設置するバッファの容量を確定することはできない。これについては次節で記す。

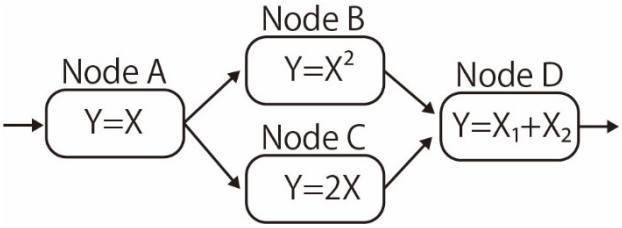

図 26　データフローグラフ

5.4.4. 遅延の一定な演算回路

本方式では通常とは違い遅延が一定な演算回路で回路を構成する。演算の遅延とは回路に前段からデータを入力し始めてから後段に結果を出力し始めるまでの時間である。例えば以下の図 27 のタイミングチャートで表される Input を入力に持ち、Output を出力に持つ演算ノードについて考えると。Data1 の入力が始まってから Data1 の出力が始まるまでに 2 クロック掛かるので、演算の遅延は 2 クロックといえる。

通常の演算器はデータの保存に DRAM を使ったり、if 文などで処理が条件分岐したりするために演算の遅延が一定にならない。しかし今回の分割実装するにあたって遅延が一定にならない演算器を使用すると分割実装をするにあたって難点が発生する。これの詳細については次節で後述する。このために本方式では遅延が一定になる演算器のみで演算器を構成する。

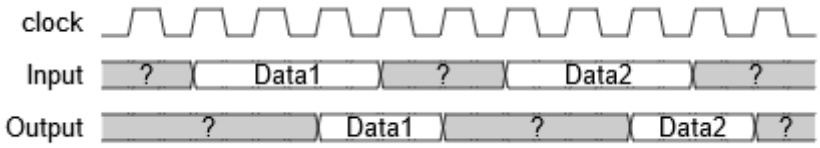

図 27　演算の遅延

5.4.5. データフローグラフを用いた分割実装

前節で製作した回路を図 28（上）のようにノード単位で 4 つのチップに分割実装する。この時に Node D では 2 つのノードから受け取るデータを使って演算を行うが、対応するデータを同じタイミングで受信するとは限らない、そこで、2 つの経路の演算と通信の遅延の差を FIFO で補償する。チップ間の通信はサーキットスイッチを用いたネットワークを経由して行われるために、チップ間通信の遅延は光トランシーバと通信のための回路の遅延と通信路の遅延の和で決まり、並列計算機の構成が決まれば一定になる。本方式では前節では演算の遅延が一定になる演算器しか使わないので、演算と通信の遅延もすべて一定になる。よって FIFO の段数も決定的に求められる。

この分割実装した回路全体と等価になる回路を図 28（下）に示す。灰色のフリップフロップは通信の遅延に対応し、黒の物は上記の FIFO の遅延に対応する。

これら 2 つのフリップフロップは全体を 1 つのチップ実装した時には生じないために分割実装した結果は厳密には 1 チップでの実装とゲートレベルで等価にならない。しかしデータフローグラフをもとに実装して循環構造が無いために、通信の遅延がどんな値でも、機能的には同じになる。

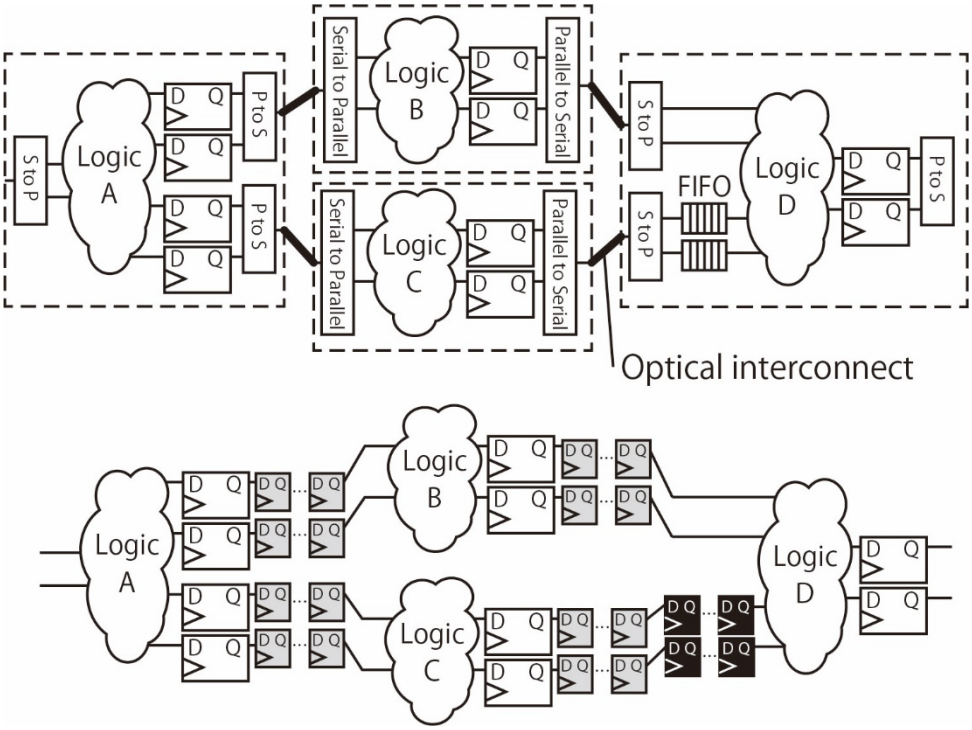

図 28　DFG を用いた分割実装
（上）複数チップへの分割実装
（下）分割実装と等価な回路

5.4.6. クロック分配

通常このような回路は同期回路として設計されるために、分割した時にはすべてのチップに同じクロックを分配しなければならない。例え同じ周波数の水晶発振器でも個体差や熱によるドリフトなどのために、各々のクロックの周波数は多少ずれてしまう。このずれが積み重なることによりチップ間での同期がずれてしまう。そのためにシステム全体で 1 つの同じクロック源を使う必要がある。

今回はチップが複数のボードに分かれることを想定している。そのためにクロック源のあるボードと多数の演算ボードをケーブルでつないでクロックを分配する。

全てのチップのクロックが共通でクロック周波数のずれが無いために、適切に設計を行えば従来型の通信で発生したような受信側の能力以上に送信側がデータを送ってしまうということは発生しない、そのためにこの方式では受信データを一旦ためておくためのバッファやフローコントロールは必要ない。

5.5. 提案方式での CNN への応用

本節では上記の同期型アーキテクチャーの CNN への応用について検討する。CNN の内畳み込み層は層の内部ではチャネルごとに独立した演算になっているために畳み込みの演算はデータフローグラフとして表せる。例えば 3ch の特徴マップを入力して 3ch の特徴マップを出力する畳み込み演算は以下の図 29 のようにデータフローグラフとして表せる。図の下のデータフローグラフの CV は 1ch の入出力の畳み込みをするノード、Add は 3ch の入力を足して出力するノードである。また畳み込み演算はアルゴリズム的に演算の手順が一定で演算の遅延を固定値と出来る。

このように畳み込み演算はデータフローグラフとして表せるので上述したように同期型アーキテクチャーで実行することが出来る。

本論文での提案方式であるハイブリッド並列では小さい演算サイズの演算が発生し、細かい粒度でのノード間通信が発生してしまう。小さい演算サイズの演算に対しては FPGA で演算器を構成することが解決策である。FPGA では目的の演算に対して最適なパイプラインを組むことが出来るために小さな演算サイズの演算でも FPGA 上の演算資源を効率的に使用することが出来る。

まず細粒度の通信に対しては方法式の同期型アーキテクチャーで解決が出来る。きわめて低いレイテンシーで処理や通信を行うことによって既存の CPU による並列演算器における処理や通信のオーバーヘッドが極めて小さくなり、細粒度の通信を行うことによる通信性能の低下を避けれれる。

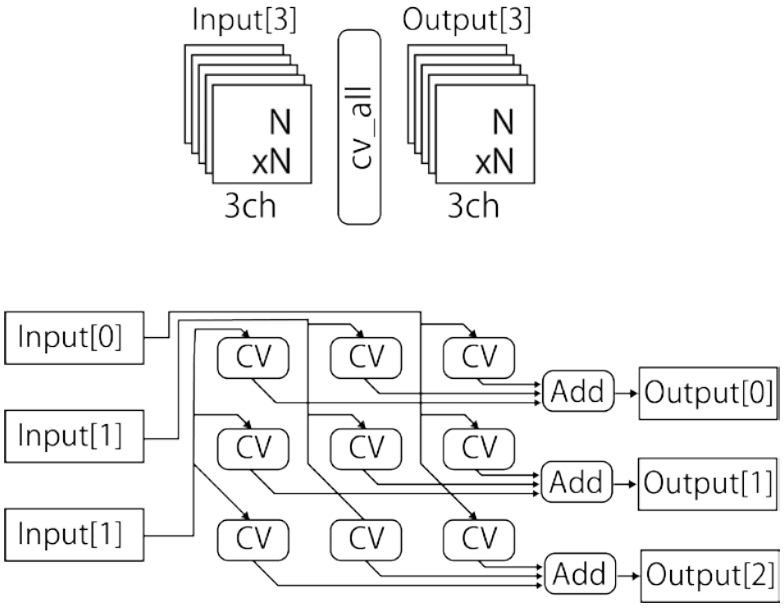

図 29　畳み込み演算のデータフローグラフ

5.6. Flow-in-Cloud

Flow-in-Cloud (FiC)は NEDO「IoT 推進のための横断技術開発プロジェクト」の、「省電

力 AI エンジンと異種エンジン統合クラウドによる人工知能プラットフォーム」で開発している並列計算機であり、多数の GPU、FPGA 等の異種ノードとメモリノードを高速低コストの光ネットワークで接続し、専用ソフトにより統合制御する。この大規模システムでは処理内容に応じて資源を柔軟に接続することが出来る。このシステム上で CNN を実装する開発が現在進められている [44] [45]。

このシステムでは本節で提案する同期型アーキテクチャーを実装することが可能である。これによって、今後さらなる CNN の高速化が図れる可能性がある。

6. 評価実験

本章ではまず GPU での畳み込み演算の実効性能を検証した。次に本論文で提案しているハイブリッド並列化で使用しているチャネル方向の並列化の検証をするために、並列計算機上で CNN の畳み込み層のチャネル方向への分割を実行し、並列化により処理の高速化を検証した。

最後に FPGA 上での畳み込み層の実装を行った。

6.1. 評価環境

本章の実験の内、実験 1-1、1-2、2 は ReedBush 上で行った。また実験 3 は FPGA 上で行っている。

6.1.1.ReedBush

ReedBush は東京大学情報基盤センターが運営するスーパーコンピューターである [46]。ReedBush は 3 つのクラスタからなり、それぞれは ReedBush-U、ReedBush-H、ReedBush-L という。それぞれのクラスタで使われているノードの基本的な構成は同じだが、ネットワークの構成や GPU の搭載数に違いがある。

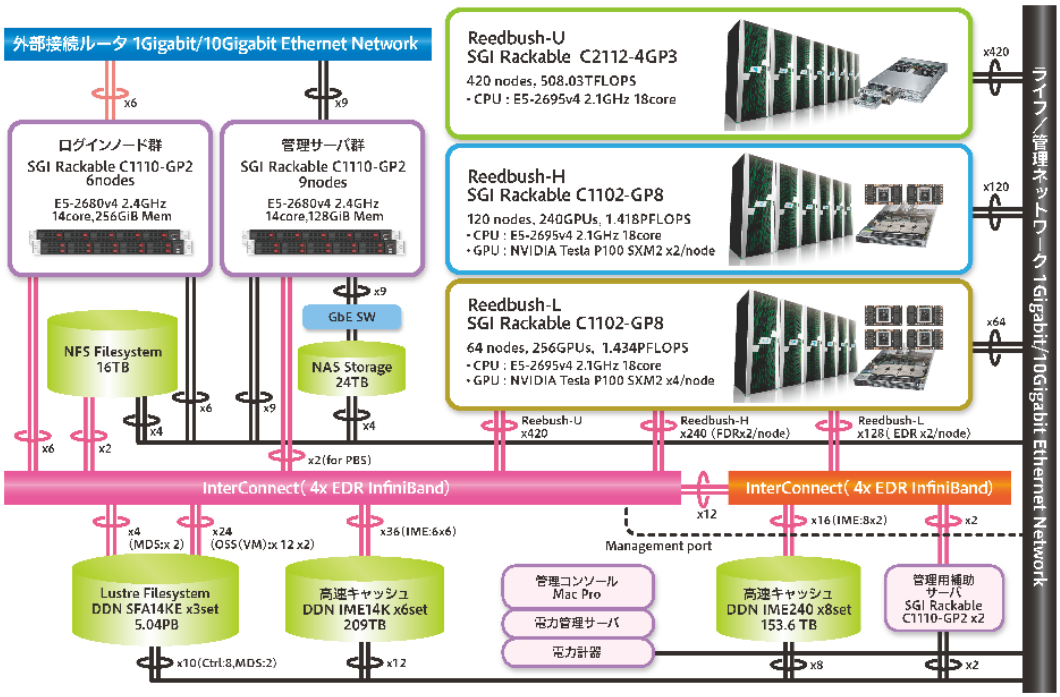

図 30　ReedBush の概要 [46]

ReedBush では多数の演算ノードが高速な InfiniBand で結合している。どのノードも同

じ CPU を搭載するが、ReedBush-U では GPU がなく、ReedBush-H では GPU を 2 つ、ReedBush-L では GPU を 4 つノードあたり搭載する。また ReedBush-U、L ではノード間をつなぐネットワークは InfiniBandEDRx4 だが、ReedBush-H では InfiniBand FDRx2 である。どのネットワークもクラスタ内では各ノード間の接続がファットツリーになっていてフルバイセクションな帯域を持つ。つまり 2 つのノード間ではそのノードの距離や他のノードの通信に関わらず、一定の速度での通信が可能である構成である。このためにノードへの割り当て方による実行時間の影響がほぼ発生しない。

今回は演算に GPU を用いるために ReedBush-H を用いた。

表 5　ReedBush-h の性能

CPU 性能	プロセッサ名	Intel Xeon E5-2695v4(Broadwell-EP)
	プロセッサ数（コア数）	2(36)
	周波数	2.1GHz（最大 3.3GHz）
	理論演算性能	1209.6GFlops
	メモリ容量	256GB
	メモリ帯域幅	153.6GB/s
GPU 性能	プロセッサ名	NVIDIA Testa P100(Pascal)
	コア数	56
	メモリ容量	16GB
	メモリ帯域幅	732GB/s
	理論演算性能	5.3TFlops
	CPU-GPU 接続	PCIe Gen3 x16（16GB/s）
インターコネクト性能	InfiniBand FDR 4x 2 リンク　（56Gbps x2）	

6.1.2. Chainer

Chainer [47]は PFN が公開しているディープラーニング用のフレームワークで、Python 上で実装されている。CNN を構成する畳み込み層や全結合層などの層に対応するクラスが実装されていて、そのメソッドでフォワード側の処理やバック側の処理を行うことが出来る。ディープラーニングの多数の層のつながりも、Chainer の各クラスの入出力を接続することで表現できる。また Chainer は GPU 実行にも対応しており、CPU 実行用のコードに少しのコードを加えるだけで GPU 上で実行することが出来る。

6.2. 実験 1：基本的性能評価

6.2.1. 実験 1-1：演算性能評価

計算機の演算性能には上限がありこれは理論演算性能（ピーク演算性能）と呼ばれる。一般には演算器の動作クロックと 1 クロックあたりに実行できる演算の数の積で表される。科学技術計算では多くの計算は浮動小数点の演算であり、計算機の演算性能は浮動小数点の演算が 1 秒あたりに何回出来るかで表す。この数値の単位として FLOPS (Floating-point Operations Per Second)が使われる。

しかし演算器は常にこの理論最大性能が発揮できるわけではない。実際の演算をした時の演算性能をその演算の実行演算性能と呼ぶ。演算によってはその演算の性質により理論最大性能より低い値に性能の上限が発生する。また演算のサイズが小さい場合は演算器の並列度を十分に発揮できないために大きなサイズの演算よりも演算性能が低くなる。

本節では CNN の畳み込みの演算を GPU 上で実行してその実行性能を評価する。畳み込み層の演算には重要なものとして 4 種類のパラメータがある。カーネルフィルタのサイズ、特徴マップのサイズ、入力チャネル数、出力チャネル数である。これらはネットワークモデルを定めれば決まる値ではあるが、目的に応じて多少変化させることは可能である。また本論文で提案するチャネル方向での畳み込み層の分割を行うと入力チャネルと出力チャネルの数は、分割の仕方によって元の数の整数分の 1 の値になる。

入力チャネルの数、出力チャネルの数が変化するにあたって、実行演算性能がどう変化するかを評価した。畳み込み層のその他のパラメータは AlexNet の第 2 層を想定して設定した。

表 6　畳み込み層の条件

カーネルフィルタのサイズ	5
特徴マップのサイズ	62
入力チャネルのサイズ	1,2,4,8,16,32,64,128,256,1024,2048,4096
入力チャネルのサイズ	1,2,4,8,16,32,64,128,256,1024,2048,4096
パディング	1
ストライド	無し

畳み込み層の推論の演算量は以下の式で表される。ただし K をカーネルフィルタのサイズ、F を特徴マップのサイズ、M を入力のチャネル数、N を出力のチャネル数とする。

$$\boldsymbol{2K^2F^2MN\ (FLOP)} \tag{10}$$

処理に必要な時間が t であった場合、その演算の実行演算性能は以下の式になる。

$$\frac{\boldsymbol{2K^2F^2MN}}{\boldsymbol{t}}\ \boldsymbol{(FLOPS)} \tag{11}$$

演算はReedBushに搭載されているNVIDIA P100を用いて行った。畳み込み演算はChainerの convolution2D()を用いて実行する。実行時間は MPI.Wtime()関数を用いて測定した。MPI.Wtime()はCPU上での実行するの関数なので、今回のGPU上で実行される畳み込み演算をの実行時間の計測時間を直接測ることはできない。そこで今回はCPUからGPUに入力データの送信を開始してから、GPUが演算を終了させた後のCPUに出力データの転送が終わるまでをGPUの演算時間として、これを計測する。

また対象とする関数は最初の数回の実行は GPU 側のセットアップ等のためにそれ以外の実行に比べて非常に時間がかかる。この影響を除くために対象とする実行を60回繰り返した場合の11〜60回目の50回の平均を計測値とする。

本章での最終的な目的である畳み込み層のチャネル方向での分割での演算速度が向上するには、チャネルのサイズが変わるにつれて演算時間も変化しないとその効果は確認できない。どの程度のチャネルサイズならこの効果を発揮できる見込みがあるのかを本実験では検証する。

○結果

実験結果のうち実行時間を表 7 に示す。

表 7　実験 1-1 結果（実行時間）

time(ms)		in channel												
		1	2	4	8	16	32	64	128	256	512	1024	2048	4096
out channel	1	0.48	0.42	0.42	0.50	0.51	0.53	0.56	0.66	0.93	1.64	3.18	6.58	13.90
	2	0.43	0.51	0.51	0.51	0.52	0.53	0.57	0.71	0.93	1.65	3.19	6.61	13.86
	4	0.44	0.51	0.52	0.52	0.53	0.54	0.58	0.71	0.94	1.65	3.19	6.62	13.70
	8	0.52	0.53	0.53	0.54	0.54	0.56	0.59	0.73	0.95	1.66	3.19	6.63	13.73
	16	0.54	0.55	0.55	0.56	0.57	0.58	0.62	0.75	0.98	1.70	3.24	6.66	13.92
	32	0.58	0.59	0.59	0.63	0.62	0.63	0.65	0.80	1.03	1.75	3.27	6.71	14.00
	64	0.66	0.66	0.66	0.66	0.69	0.72	0.78	0.91	1.28	2.12	3.92	7.88	16.08
	128	0.71	0.77	0.78	0.78	0.80	0.83	0.96	1.26	1.69	2.86	5.32	10.62	21.39
	256	0.95	0.90	0.91	0.92	0.92	1.04	1.28	1.80	2.67	4.30	7.94	15.74	31.65
	512	1.36	1.36	1.36	1.39	1.47	1.69	2.04	2.63	4.36	7.28	13.03	25.38	50.54
	1024	2.25	2.28	2.31	2.38	2.53	2.83	3.47	4.77	7.39	13.27	23.88	45.58	89.21
	2048	2.69	2.71	2.76	2.94	3.21	3.77	5.10	7.56	12.59	22.72	44.45	85.07	166.97
	4096	33.08	33.16	33.28	33.53	34.03	35.14	37.32	41.56	50.19	67.54	102.96	170.86	313.36

実験結果のうち演算性能を以下の表 8 に示す。

表 8　実験 1-1 結果（演算性能）

演算性能 (GFLOPS)		in channel												
		1	2	4	8	16	32	64	128	256	512	1024	2048	4096
out channel	1	0.28	0.68	1.36	2.16	4.32	8.34	15.9	28.51	41.31	47.11	48.43	47.12	44.7
	2	0.66	1.07	2.16	4.25	8.37	16.63	31.08	51.9	82.82	93.36	97.17	93.8	89.34
	4	1.29	2.14	4.23	8.49	16.78	32.93	62.56	104.24	164.58	186.83	194.76	187.26	181.36
	8	2.11	4.12	8.3	16.33	32.48	64.45	122.15	203.26	324	372.32	389.52	373.54	361.45
	16	4.07	8.04	16	32.01	61.96	121.74	235.37	394.54	627.04	727.65	765.6	744.38	711.02
	32	7.63	15.05	30.39	60.69	115.59	227.46	453.48	755.58	1201.6	1416.1	1518.5	1476.6	1416.1
	64	13.71	27.23	55.15	109.84	216.61	407.66	764.93	1328.1	1925.5	2337.8	2541.6	2517	2467.3
	128	26.63	47.74	95.03	189.12	367.95	723.6	1298.6	1913.4	2936.5	3464.6	3726.4	3742.6	3698.8
	256	39.51	84.63	168.34	335.98	666.78	1210	1929.8	2733.4	3697.2	4596.5	5009.4	5016.1	5013.8
	512	56.61	112.8	230.11	443.8	841.2	1456.1	2424.8	3783.4	4526.2	5437.3	6095.2	6249.2	6274.6
	1024	67.4	133.86	263.96	510.95	967.32	1735.3	2856.6	4138.9	5362.5	5964.1	6621.7	6948.3	7111.9
	2048	114.22	226.52	445.09	837.13	1542.6	2703.9	3863.7	5228	6301.6	6975.4	7135.8	7447.2	7600.2
	4096	18.75	37.38	74.41	147.85	291.47	563.17	1062.6	1908.6	3161.4	4702.3	6167	7425	8094.5

○考察

チャネル総数と演算性能をプロットしたグラフを図 31 に示す。チャネル総数は入力チャネルと出力チャネルの積で求められて、演算量と比例する。チャネル総数がおよそ 20000 程度までは演算量と演算性能が比例して伸びていく。この領域を演算時間で見ると演算時間がおよそ 0.5ms 程度で変わらない。これは GPU の並列度を十分に使い切れていないので、演算のサイズが変化しても GPU の演算器の使用率が変わるだけで、演算時間は変わらないためと考えられる。

また演算サイズが約 20000 を超えたあたりから演算性能の伸びが緩やかになり約 10TFLOPS の値に漸近していく。今回使用している GPU である P100 の理論最大性能は 10.6TFLOPS であり、演算のサイズが大きいと理論最大性能に近い性能が発揮できることが確認できた。

また、図 31 を見ると同じチャネル総数でも演算性能に大きな差があることが分かる。例えば出力チャネルが 2、入力チャネルが 4096 の演算と、同じく 64、128 の演算はチャネル総数が 8192 という点では同じだが、演算性能は前者が 89.34GFLOPS で後者は 1328.1GFLOPS と大きく差がある。全体的な傾向を見ると入力チャネル数と出力チャネル数が同程度の場合の演算の方が高速であるということが分かる。

およそチャネル総数が 20000 を超えて、入力と出力チャネル数が近い値の範囲なら畳み込み層のチャネル方向分割による高速化が期待できると言える。

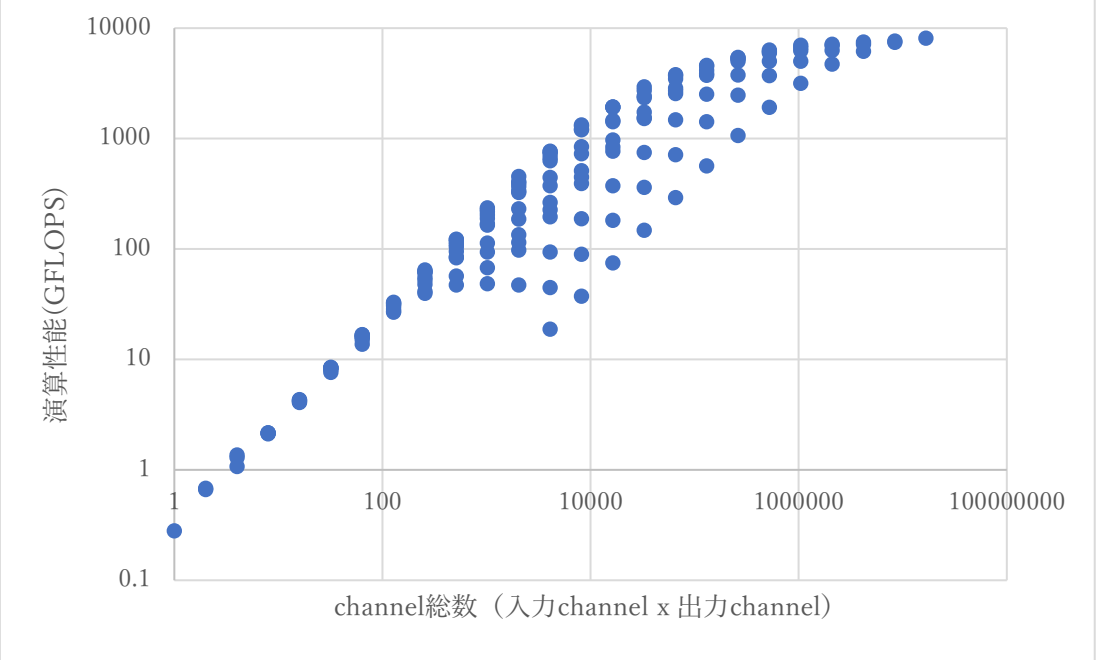

図 31　畳み込み層のチャネル総数と演算性能　（x 軸、y 軸共に対数軸）

6.2.2. 実験 1-2：通信性能評価

ReedBush-L では各ノードは Infiniband FDRx4 を 2 リンク持ちファットツリーで接続されていて、フルバイセクションな接続帯域を持つ。そのために理論通信性能として片道 112Gbps を持つ。しかしバッファや OS 等の影響でその最大性能が出せるわけではなく、実際の最大通信性能はいくらか最大性能よりも低い値になる。

また実際の通信時間に掛かる時間は通信データ量を通信性能で割った値で求められるが、転送するデータサイズが小さい場合は転送の効率が悪くなり、それ以上に時間がかかるという傾向がある。本章では ReedBush-L 上の 2 ノードを用いて 1 対 1 通信を行い実際に通信にかかる時間を実測した。

2 ノードの片方（Node 0）がまず Send()で任意のデータサイズのデータを送信する。もう片方のノード（Node 1）は Recv()でデータを受信したら、受信したデータを Send()でまた Node 0 に送信する。そして Node 0 は Recv()でデータを受信する。Node 0 がデータを送信してから、データを受信するまでの時間を Wtime()関数を用いて計測する。転送データサイズを S、転送時間を t とすると転送帯域は以下の式で求められる。

$$\mathbf{BW}\left[\frac{\boldsymbol{B}}{\boldsymbol{s}}\right] = \frac{\mathbf{2S}}{\mathbf{t}} \tag{12}$$

通信を始めてからの数回は環境の設定などの都合で転送に本来よりも多くの時間がかかる。この影響を排除するために各データサイズで 10 回の通信を行い、後半の 5 回の平均値を測

定値をした。評価の条件を表 9 に示す。

表 9　評価条件

使用システム	ReedBush
使用関数	MPI.Send()、MPI.Recv()
転送データサイズ	4B~512MB
接続	Infiiniband FDRx4 2 リンク
時間計測	MPI.Wtime()

○結果

実験結果を表 10 に示す。また転送サイズと転送帯域をプロットしたグラフを図 32 に示す。転送データサイズや 4KB までは転送時間は 5μ秒程度で変わらない。4KB を超えると転送データサイズが大きくなるにしたがって転送帯域も伸びてきて、2MB を超えたあたりで帯域が約 10GB/s に漸近する。

AlenNet 層間では 1MB 程度の特徴マップが転送される。これを Reedbush のノード間転送を用いて行った場合は、最大性能まではいかないが数 GB/s 程度の転送帯域で転送できることが検証できた。

表 10　実験 1-2 結果

転送サイズ	転送時間(ms)	転送帯域(GB/s)
4	0.0045	0.0008
8	0.0042	0.0018
16	0.0044	0.0034
32	0.0046	0.0065
64	0.0045	0.0132
128	0.0050	0.0237
256	0.0049	0.0488
512	0.0049	0.0976
1K	0.0052	0.1818
2K	0.0056	0.3386
4K	0.0065	0.5872
8K	0.0080	0.9552
16K	0.0096	1.5950
32K	0.0121	2.5222

64K	0.0179	3.4077
128K	0.0269	4.5410
256K	0.0384	6.3652
512K	0.0619	7.8883
1M	0.1073	9.1048
2M	0.2007	9.7321
4M	0.3890	10.0408
8M	0.7620	10.2528
16M	1.5084	10.3588
32M	2.9933	10.4400
64M	5.9810	10.4497
128M	11.9455	10.4642
256M	23.8717	10.4727
512M	47.7540	10.4703

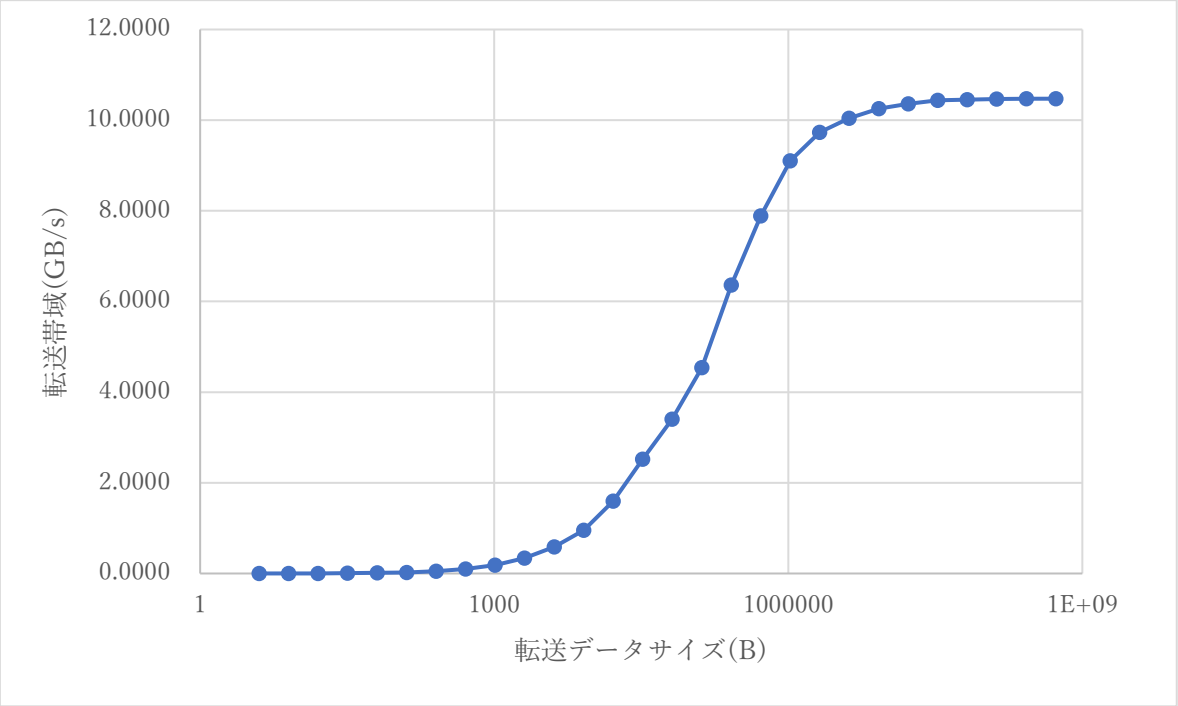

図 32　転送データサイズと転送帯域　（x 軸のみ対数軸）

6.3. 実験 2：畳み込み層のチャネル方向並列化の実行性能評価

本節ではチャネル方向での層内並列化による処理の速度向上を実証するために CNN の推論を実際に並列計算機上で実行を行った。本実験で用いた CNN のネットワークモデルを図 33 に示す。このネットワークモデルは AlexNet を基本としていて 8 つの層を持ち、L1 から L5 までは畳み込み層で、L5 から L8 は畳み込み層である。今回はチャネル方向での畳み込み層の演算の並列化の効果を実証するためにして畳み込み層 L2 の入力、出力チャネル数を増加させた。実験 1-1 の結果を参照して、入力・出力チャネル共に 1024 とした。この

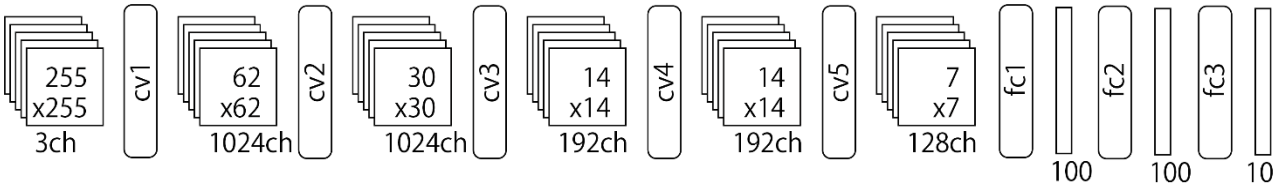

図 33　本実験で使用するネットワークモデル

層をチャネル方向に並列化することで演算の高速化を図る。

本実験では図 34 の様に L1 を 1 つのノードに、L3〜L8 で別の 1 つのノードに割り当て、L は実験条件に従って 1 つまたは複数のノードに分割実装する。本節では L2 について並列化無し、出力チャネルで 2 分割・4 分割、入力チャネルで 2 分割・4 分割の 5 通りで実装を行い実験した。

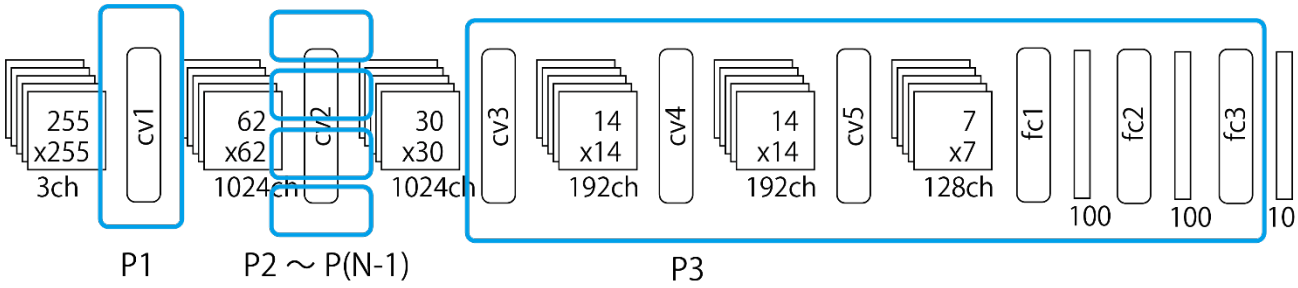

図 34　ノードの割り当て

各条件の試行では各ノードの処理前後のタイムスタンプを取得して、実際に演算が行われているタイミングを取得した。各分割での推論の演算を 60 枚の画像に対して行い、51 枚目から 60 枚目までの 10 枚の画像の処理のタイムスタンプを記録した。最初の数回は GPU のバッファのセットアップ等のために通常よりも多くの時間がかかる。この影響を排除するために上記のような測定方法を取った。

各条件での処理のタイムスタンプから処理のインターバルとスループットを求めた。インターバルとは処理の間隔であり、ある画像の処理を始めてから、次の画像の処理を始めるまでの時間である。スループットはインターバルの逆数である。これはそのシステムが 1 秒間にいくつのデータを処理できるかを表し、スループットが大きいと性能が高いと言える。

ノード間通信にはMPIを用いた。MPIには1対1通信と集合通信があるが、今回はすべて1対1通信のIsend()とRecv()を用いて実装を行った。

処理を高速に行うために演算と転送をオーバーラップさせるために送信側ではノンブロッキング送信関数であるIsend()を用いた。バッファのつまりが発生する可能性があるために適宜wait()を挿入した。例えば分割が無い場合などでは前段のL1を計算しているノードの処理時間よりも後段のL2を計算しているノードの処理時間の方が長い。そのため、L2での演算が終わらないうちにL1が次々と演算を終えて特徴マップを送信することでL2を担当する演算ノードの受信バッファを圧迫してL2を担当するノードの演算性能を下げる恐れがある。これを防止するために各 MPI の送信を始める時に、前回の送信が終わるまでのWait()関数を挿入した。

○結果

計測した結果の各条件でのスループットを以下の表 11 に示す。条件の欄の値の数字の組(n,n)は出力チャネル方向に n 分割、入力チャネル方向に m 分割していることを表している。高速化率は分割無しの場合と比べてのスループットの高速化の割合を示す。

表 11　実験2：結果

条件		スループット (data/s)	高速化率
分割無し	(1,1)	32.1	1.00
2分割	(1,2)	41.5	1.29
	(2,1)	42.4	1.32
4分割	(1,4)	47.2	1.47
	(4,1)	54.3	1.69

また各条件での処理のタイミングチャートを代表して分割無しの場合を図 35 に、出力チャネルで2分割した場合を図 36 に示す。各タイミングチャートではx軸が時間を表してy軸方向はノードを示す。例えば分割無しの場合を（図 35）ではNode 0、Node 1、Node 2の3つのノードを用いている。このタイミングチャーチトでは10データ分の処理を表示している。同じデータの処理は同じ色で示している。Node 0 の 0 秒付近の緑のバーは data1 の L1 層の処理を表している。バーの長さは処理時間を表す。Node 1 の 0.06～0.09 秒付近の緑のバーは data1 の L2 層の処理を表している。また Node 2 の 0.09 秒付近の緑のバーは data1 の L3～L8 層の処理を表している。縦方向にずれている 5 つのバーはそれぞれ L3～L8 に対応している。

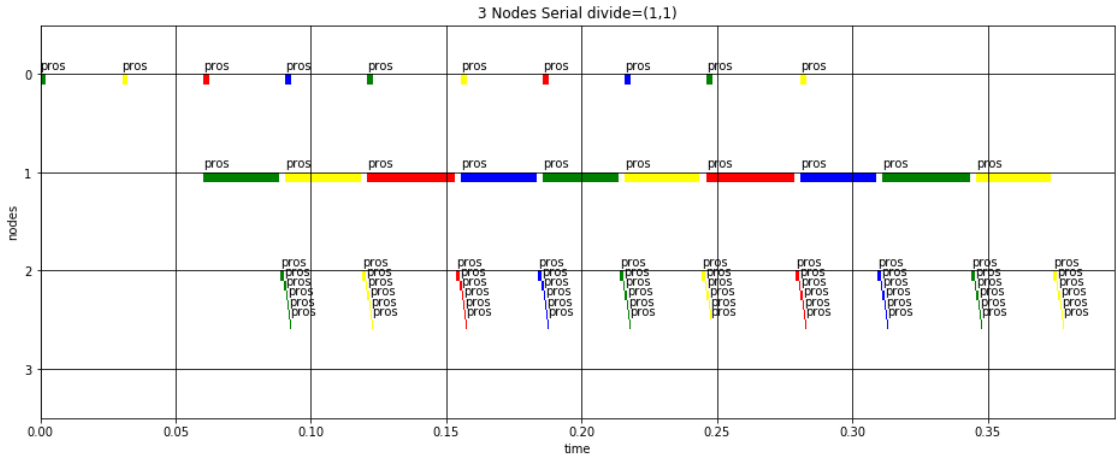

図 35　分割無しの場合のタイミングチャート

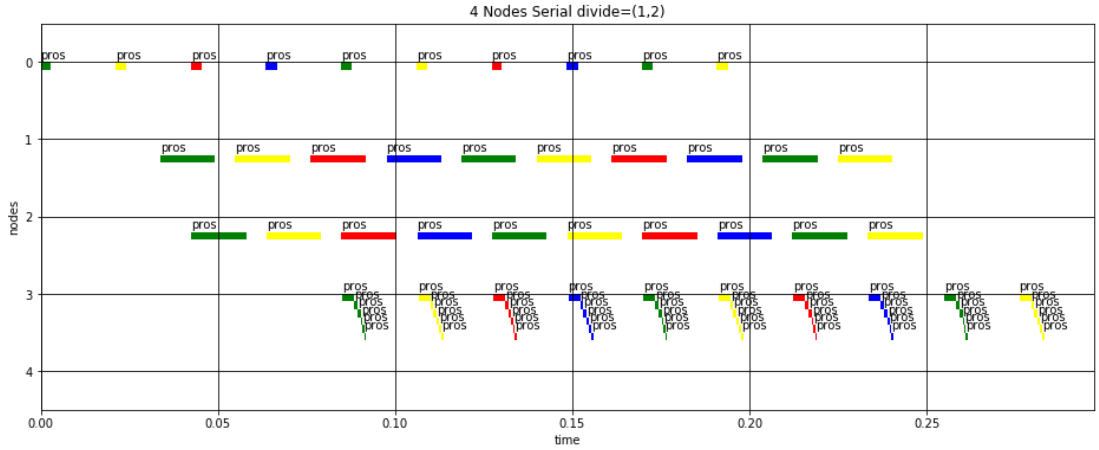

図 36　出力チャネルで 2 分割した場合のタイミングチャート

○考察

表 11 の条件(2,1)では 1.29 倍の高速化を、条件(4,1)では 1.69 倍の高速化を達成したために、チャネル方向並列によって演算の高速化を図れるということが分かる。タイミングチャートを見ると分割無しの時（図 35）の L2 層の演算がボトルネックになっており interval に対して支配的で、他のノードは待機している時間が長い。一方出力チャンネルで 2 分割した場合（図 36）でも L2 層の演算が支配的であることは変わらないが、演算を分割したことによって演算時間が長くなり、相対的に見て他のノードが待機している時間が短くなっていることが分かる。

しかし想定していた程の高速化の効果を得ることはできなかった。今回の環境では理想的には L2 層の演算を 2 分割すればインターバルは 1/2 になり、スループットは 2 倍になるはずである。しかし 2 分割では 1.29 倍にとどまる。これは実験 1-1 が示すように GPU を用いた演算では演算サイズが小さくなると演算性能が低下してしまう事の影響、また処

理時間モデルでは考慮していなかった各ノードが通信などの演算以外の処理を行う時間が必要であることの 2 つが原因として考えられる。

6.4. 実験 3

本節では FPGA 上に高位合成で畳み込み演算器を実装して、実行性能と実装に必要な回路資源量を比較した。FPGA はコンフィグレーションすることで内部に自由な演算器を構成できる素子である。これによって目的とする処理に最適なパイプラインを組むことが出来るために、本論文で対象としているような演算サイズの小さな演算に対しても高い演算性能を発揮することが可能である。

実験条件を以下の表に示す。実行間隔は実行間隔(clock/data)の値は演算性能(flops)の値の逆数に比例する値であり、大きければ大きいほど演算のスループットは低下する。

表 12　実験 3　実験条件

高位合成ツール	Vivado HLS 2016.4
対称 FPGA	Xilinx UVC108(XCVU095)
実行周波数	100MHz
実行演算	畳み込み演算
入力チャネルサイズ	8
出力チャネルサイズ	8
実行間隔	1 clock/data～ 18 clock/data

○結果

実験の結果を以下の図 37 に示す。また DSP の使用資源量に注目した表を表 13 に示す。

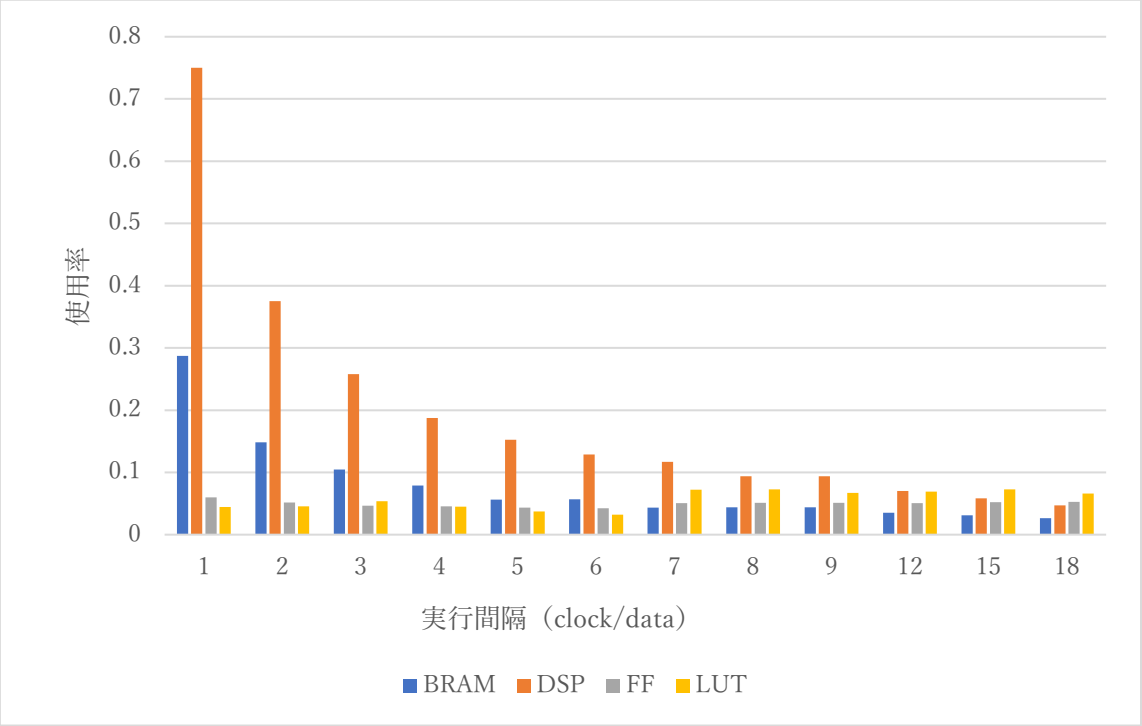

図 37　実行間隔と必要資源量

表 13　DSP の資源使用量と削減率

実行間隔（ckock/data）	使用量	使用率	削減率
1	576	0.750	1.000
2	288	0.375	2.000
3	198	0.258	2.909
4	144	0.188	4.000
5	117	0.152	4.923
6	99	0.129	5.818
7	90	0.117	6.400
8	72	0.094	8.000
9	72	0.094	8.000
12	54	0.070	10.667
15	45	0.059	12.800
18	36	0.047	16.000

○考察

実験の結果が示すように実行間隔を大きくするとそれに従って各資源の消費量も低くな

る。今回の実装では DSP の使用率が隘路となるので、DSP の使用率と削減率を表した物が表 13 である。削減率は実行間隔が 1 の時と比べて、どれだけ資源の使用量が減ったかを表す。この表から今回実験した範囲では資源の消費の削減率は実行間隔とほぼ同程度になることが分かる。つまり FPGA 上に畳み込み演算の演算器を構成した時に FPGA 上の回路の演算性能と必要な資源量はほぼ比例する。

今回の実験から実行間隔を変えて資源の消費量を減らすことで大きな総チャンネル数に対応できるように回路を変更しても、全体としての演算性能はほぼ変化しなく、仮定通りに演算性能をほぼ一定に保ったまま自由に演算器を構成出来ることが分かる。

7. 考察

実験 1-1、実験 2 は GPU 上で畳み込み演算を実行する評価実験である。しかし実験 1 では畳み込み演算の演算サイズが小さい領域では演算性能が極端に下がってしまう。GPU は内部の高い並列性によって高い演算性能を発揮する。つまり畳み込み演算の演算サイズが小さい場合ははこの GPU の並列性を十分に発揮できないために演算性能が下がってしまうと考えられる。実験 2 では高速化は確認できたが、想定していたほどの高速化は達成できなかった。これはやはり上記の GPU の演算性能の傾向に起因するところが大きいと言える。このように GPU はチャンネル方向並列化に適していない。

一方 FPGA ではチャンネル方向での並列化について大きなペナルティーは発生しない。これによってさらなる高速化のための CNN の高速化のためには既存のシステムの様に GPU を使うのではなく、FPGA を使うことが最適と言える。

CNN の学習の処理においては演算と通信の 2 つの部分が重要になってくるが今回は演算の部分の評価にとどまった。本方式ではモデル並列を行うために細粒度の通信が発生するが、これは既存の CPU-GPU をベースとしたアーキテクチャーではオーバーヘッドが大きく FPGA での専用回路のほうが通信のレイテンシーにおいて有利と考えられる。この部分の評価が今後の課題と言える。

8. 結論

本論文ではまず CNN の並列化による高速化の技術としてデータ並列を紹介した。そしてデータ並列ではグローバルミニバッチサイズの制限に起因する並列度の限界が発生することを説明した。

提案手法としてデータ並列とチャンネル方向の並列化を導入したモデル並列を組み合わせるハイブリッド並列を提案した。データ並列とハイブリッド並列の各々について処理時間をモデル化して比較して特にミニバッチサイズが小さく並列度が高い状況でハイブリッド並列の方がデータ並列よりも処理時間が短くなることを示すことで、ハイブリッド並列の優位性を説明した。

またハイブリッド並列では小さい演算サイズの演算と細粒度の通信が発生する。これは既存の並列演算器が苦手とする処理で、このような処理に適した同期型アーキテクチャーを提案した。

次に GPU 上と FPGA 上で CNN の畳み込み演算を実装したが、GPU では本手法であるチャネル方向の分割によって極端に性能が低下して、推論の演算を実際に並列計算機で実行しても処理時間モデル程高速化することが出来ないことが判明した。

FPGA では高い演算性能を保ったまま小さい演算サイズの畳み込み演算器を作成できる。よってチャネル方向並列化を採用する提案手法であるハイブリッド並列は FPGA での実装に適している。

FiC のような FPGA を用いた並列計算機で同期型アーキテクチャーを用いてハイブリッド並列の実装を行い、演算の高速化を実証することが今後の課題として挙げられる。

謝辞

本研究を進めるにあたり、多くの人々にお世話になりました。

指導教官である工藤知宏教授には研究を進めるうえで多くの助言、指導をいただきました。加えて研究発表の際には様々なサポートをしていただきました。心より感謝申し上げます。

産業技術総合研究所の高野了成氏、池上努氏、大内真一氏には研究の内容についてミーティングで議論をしていただき、様々な意見を頂きましたことを深く感謝申し上げます。

また、本研究を進めるにあたって，多くのご助言・ご意見を賜りました本学情報基盤センターの中山雅哉順教授、小川剛史准教授、佐藤周行准教授、関谷勇司准教授、中村文隆助教、中村遼助教に深く感謝申し上げます。

最後に，大学・大学院と 6 年間も学ぶ機会を与えてくれた両親にこの場をお借りして，深く心より感謝いたします。

発表文献

国内会議（査読無し）

1. 赤沼 領大, 高野 了成, 工藤 知宏. “超広帯域インターコネクトを想定した同期型アーキテクチャーと通信方式の検討”, 電子情報通信学会, 信学技報, Vol. 117, no. 153, pp. 117-122, 秋田アトリオンビル(秋田県), 2017 年 7 月
2. 赤沼 領大, 高野 了成, 工藤 知宏. “CNN の学習におけるチャネル方向並列化の提案”, 電子情報通信学会, ひめぎんホール(愛媛県), 2018 年 3 月（発表予定）

参照文献

[1] Lowe, D., "Object recognition from local scale-invariant features," in *IEEE ICCV*, 1999.

[2] Dalal, N; Triggs, B, “Histograms of Oriented Gradients for Human Detection,” 著: *CVPR*, 2005.

[3] J. Deng, W. Dong, R. Socher, L.-J. Li, K. Li, and L. Fei-Fei., "ImageNet: A Large-Scale Hierarchical Image Database," in *CVPR09*, 2009.

[4] A. Krizhevsky, I. Sutskever, and G. E. Hinton., "ImageNet classification with deep convolutional neural networks," in *NIPS*, 2012.

[5] L.Deng, "The MNIST Database of Handwritten Digit Images for Machine Learning Research," *IEEE Signal Processing Magazine,* vol. 29, no. 6, pp. 141-142, 2012.

[6] R. Girshick, J. Donahue, T. Darrell, U. C. Berke-ley, and J. Malik., "Rich feature hierarchies for accurate object detection and semantic segmen-tation," in *IEEE CVPR*, 2014.

[7] Fei-Fei., A. Karpathy and L., "Deep Visual-Semantic Alignments for Generating Image De-scriptions," in *IEEE CVPR*, 2015.

[8] Rajat Raina, Anand Madhavan, Andrew Y. Ng, "Large-scale deep unsupervised learning using graphics processors," in *ICML*, 2009.

[9] Yang You, Igor Gitman, Boris Ginsburg, “Large Batch Training of Convolutional Networks,” 2017. https://arxiv.org/abs/1708.03888.

[10] Haruki Mori, Tetsuya Youkawa, Shintaro Izumi, Masahiko Yoshimoto, Hiroshi Kawaguchi, "A layer-block-wise pipeline for memory and bandwidth reduction in distributed deep learning," in *IEEE MLSP*, 2017.

[11] 岡谷貴之, 深層学習, 講談社, 2015.

[12] Diederik P. Kingma, Jimmy Ba, “Adam: A Method for Stochastic Optimization,” 2014. https://arxiv.org/abs/1412.6980.

[13] Y. LeCun, L. Bottou, Y. Bengio, and P. Haffner, "Gradient-based learning applied to document recognition," *Proceedings of the IEEE,* vol. 86, no. 11, pp. 2278-2324, 1998.

[14] Alex Krizhevsky, “One weird trick for parallelizing convolutional neural networks,” 2014. https://arxiv.org/abs/1404.5997.

[15] Fukushima., K., "Neocognitron: a self organizing neural network model for a mechanism of pat-tern recognition unaffected by shift in position," *Biological Cybernetics,* vol. 36, pp. 193-202, 1980.

[16] J. Cong and B. Xiao, “Minimizing computation in convolutional neural networks.,” 著: *Artificial Neural Networks and Machine Learning – ICANN*, 2014.

[17] Zisserman, K. Simonyan and A., “Very Deep Convolutional Networks for Large-Scale Image Recoginition,” *ICLR*, 2015.

[18] C. Szegedy, W. Liu, Y. Jia, P. Sermanet, S. Reed, D. Anguelov, D. Erhan, V. Vanhoucke, and A. Rabinovich, “Going Deeper with Convolutions,” 著: *IEEE CVPR*, 2015.

[19] Omry Yadan, Keith Adams, Yaniv Taigman, Marc’Aurelio Ranzato, “Multi-GPU Training of ConvNets,” https://arxiv.org/pdf/1312.5853.pdf.

[20] 東京工業大学 学術国際情報センター, “東京工業大学が世界初の大規模 GPGPU コンピューティング基盤を大規模スーパコンピュータ基盤 TSUBAME 上に実現,” 2008. http://www.gsic.titech.ac.jp/contents/news.html.ja?page=News/2008/1119/01, Novem-.

[21] NVIDIA, “CUDA C Programming Guide,” http://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html.

[22] NVIDIA, “cuDNN developer guide,” http://docs.nvidia.com/deeplearning/sdk/cudnn-developer-guide/index.html.

[23] 山崎 雅文, 笠置 明彦, 田原 司睦, 中平 直司, "MPI を用いた Deep Learning 処理高速化の提案," *研究報告ハイパフォーマンスコンピューティング,* vol. 155, no. 6, pp. 1-8, 2016.

[24] T. Akiba, S. Suzuki, K. Fukuda, “Extremely Large Minibatch SGD: Training ResNet-50 on ImageNet in 15 Minutes,” 2017. https://arxiv.org/abs/1711.04325.

[25] Yoon, Hyunsoo, Jong H. Nang, and S. R. Maeng, “A distributed backpropagation algorithm of neural networks on distributed-memory multiprocessors,” 著: *Frontiers of Massively Parallel Computation*, 1990.

[26] C. Zhang, D. Wu, J. Sun, G. Sun, G. Luo, J. Cong, “Energy-Efficient CNN Implementation on a Deeply Pipelined FPGA Cluster,” *ISLPED*, 2016.

[27] Yadan, O., Adams, K., Taigman, Y., Ranzato, M. A., “Multi-GPU Training of ConvNets,” 2013. https://arxiv.org/pdf/1312.5853.pdf.

[28] Alexander L. Gaunt, Matthew A. Johnson, Maik Riechert, Daniel Tarlow, Ryota

Tomioka, Dimitrios Vytiniotis, Sam Webster, “AMPNet: Asynchronous Model-Parallel Training for Dynamic Neural Networks,” 2017. https://arxiv.org/abs/1705.09786.

[29] Song Han, Jeff Pool, John Tran, William Dally, “Learning both Weights and Connections for Efficient Neural Networks,” *NIPS*, 2015.

[30] NVIDIA, “Pascal Architecture Whitepaper,” http://www.nvidia.com/object/pascal-architecture-whitepaper.html.

[31] J. Qiu, J. Wang, S. Yao, K. Guo, B. Li, E. Zhou, J. Yu, T. Tang, N. Xu, S. Song, Y. Wang, H. Yang, “Going Deeper with Embedded FPGA Platform for Convolutional Neural Network,” 著: *Proceedings of the 2016 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, 2016.

[32] Matthieu Courbariaux, Itay Hubara, Daniel Soudry, Ran El-Yaniv, Yoshua Bengio, “Binarized Neural Networks: Training Deep Neural Networks with Weights and Activations Constrained to +1 or -1,” 2016. https://arxiv.org/abs/1602.02830.

[33] Urs Koster, Tristan J. Webb, Xin Wang, Marcel Nassar, Arjun K. Bansal, William H. Constable, O?uz H. Elibol, Scott Gray, Stewart Hall, Luke Hornof, Amir Khosrowshahi, Carey Kloss, Ruby J. Pai, Naveen Rao, “Flexpoint: An Adaptive Numerical Format for Efficient Training of Deep Neural Networks,” 2017. https://arxiv.org/abs/1711.02213.

[34] 松本幸, et al, "MPI_Allreduce の 「京」 上での実装と評価," *研究報告計算機アーキテクチャ,* vol. 6, pp. 1-10, 2011.

[35] Gropp, W., Lusk, E., Doss, N., & Skjellum, A., "A high-performance, portable implementation of the MPI message passing interface standard," *Parallel computing,,* vol. 22, no. 6, pp. 789-828, 1996.

[36] L.Dalcin, “MPI for Python,” http://mpi4py.scipy.org/docs/.

[37] Simonyan, K., & Zisserman, A., “Very Deep Convolutional Networks for Large-Scale Image Recoginition,” 2015.

[38] "毎秒 1 ペタビット容量で世界最長 200 km 超の長距離空間多重光伝送実験に成功," *NTT 技術ジャーナル,* vol. 26, no. 6, pp. 60-62, 2017.

[39] T. Inoue, T. Kurosu, K. Ishii, H. Kuwatsuka, and S. Namiki, “Exabit Optical Network Based on Optical Comb Distribution for High-Performance Datacenters: Challenges and Strategies,” 著: *Frontiers in Optics/ Laser Science*, 2015.

[40] Akinori HAYAKAWA, Hiroji, EBE, Chen Yanfei, Toshihiko MORI, "Silicon Photonics Optical Transceiver for High-speed, High-density and Low Power Consumption LSI

Interconnect," *Magazine FUJITSU,* vol. 66, no. 5, pp. 19-26, 2015.

[41] S. N. Kiyo Ishii, "Toward exa-scale photonic switch system for the future datacenter (invited paper)," *Tenth IEEE/ACM International Symposium on Networks-on-Chip (NOCS),* 2016.

[42] Y. L. W. Z. F. F. H. X. Jianfeng He, "Real-Time Optimization and Application of the Embedded ARM-Linux Scheduling Policy," *Information Technology, Computer Engineering and Management Sciences (ICM),* pp. 134-138, 2011.

[43] 村瀬 大, 高木 大智, 尼崎 太樹, 久我 守弘, 飯田 全広, 末吉 敏則, "高速シリアル光インターコネクトを用いた FPGA 分割実装," *研究報告システムと LSI の設計技術（SLDM）,* vol. 178, no. 6, pp. 1-6, Jan 2017.

[44] 武者千嵯、工藤知宏、鯉渕道紘、天野英晴, "マルチ FPGA 上での CNN の実装," *信学技報,* vol. 116, no. 417, 2017.

[45] 工藤 知宏, 高野 了成, 天野 英晴, 鯉渕 道紘, 松谷 宏紀, 塙 敏博, 池上 努, 須崎 有康, 田中 哲, 赤沼 領大, 並木 周, 田浦 健次朗, "データの流れに着目した異種エンジン統合クラウドシステム Flow in Cloud (コンピュータシステム)," *信学技報,* 第巻 117, 第 153, pp. 1-5, 2017.

[46] 東京大学情報基盤センター, "Reedbush スーパーコンピュータシステムの紹介," https://www.cc.u-tokyo.ac.jp/system/reedbush/reedbush_intro.html.

[47] Tokui, S., Oono, K., Hido, S., & Clayton, J, "Chainer: a next-generation open source framework for deep learning. In Proceedings of workshop on machine learning," 著: *NIPS,* 2015.

[48] "VGG in TensorFlow," http://www.cs.toronto.edu/~frossard/post/vgg16/.

[49] C. Zhang, P. Li, G. Sun, Y. Guan, B. Xiao, J. Cong, "Optimizing FPGA-based Accelerator Design for Deep Convolutional Neural Networks," 著: *Proceedings of the 2015 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, 2015.

[50] WildML. http://www.wildml.com/.

[51] Alex Krizhevsky, "One weird trick for parallelizing convolutional neural networks," https://arxiv.org/abs/1404.5997.

[52] Sano, Kentaro, Yoshiaki Hatsuda, and Satoru Yamamoto, "Multi-FPGA accelerator for scalable stencil computation with constant memory bandwidth," *IEEE Transactions on Parallel and Distributed Systems,* vol. 25, no. 3, pp. 695-705, 2014.

[53] "Deep learning for complete beginners: convolutional neural networks with keras," https://cambridgespark.com/content/tutorials/convolutional-neural-networks-with-

keras/index.html.

[54] 中山英樹, "深層畳み込みニューラルネットワークによる画像特徴抽出と転移学習," *信学技報,* vol. 146, no. 55-59, p. 115, 2015.