Vol.2021-ARC-244 No.43
Vol.2021-SLDM-194 No.43
Vol.2021-EMB-56 No.43
2021/3/26

PYNQ クラスタ上での ResNet の並列実装 (2020 年 2 月 17 日版)

福嶋 泰優[†1,a)] 飯塚 健介[†1] 天野 英晴[†1,b)]

概要：深層学習アプリケーションの実装では, 演算コストと電力コストを抑えるため, 省電力性と柔軟性に優れる FPGA（Field-Programmable Gate Array）がプラットフォームとしてよく選ばれている. しかし, 深層学習アプリケーションは計算量とパラメータ数が非常に大きく, 単一 FPGA での実装ではハイエンドで高価な FPGA を用いなければならない傾向にある. 特に畳み込みニューラルネットワークは計算量が膨大であり, その傾向はより顕著なものとなる.
我々の研究室では, M-KUBOS と呼ばれる価格性能比に優れた Zynq ボードに PYNQ（Python productivity for Zynq）と呼ばれるオープンソース・ソフトウェア・プラットフォームを導入し, 低コストかつ高性能な GTH シリアルリンクで接続することで構成される PYNQ クラスタの開発を行っている. PYNQ クラスタは MEC のサーバとして, 5G モバイルネットワークなどへの利用が期待される. 現在は 4 枚の M-KUBOS ボードを接続し, 群管理することでクラスタを構築している.
本稿では, 4 枚の M-KUBOS ボードを接続して形成した PYNQ クラスタに ResNet-18 の推論アクセラレータを実装する手法を提案する. ResNet-18 の各層の実行時間を求めることで, ボードごとの実行時間が可能な限り等しくなるよう 4 ボードに分割し, それぞれをパイプラインの 1 ステージとして並列処理を行うよう実装した. 分割以外に, 重みと特徴マップを量子化することでリソースを節約し, 畳み込み演算の多重ループを入力チャネルと出力チャネルでアンロールすることで演算を並列実行し, さらなる高速化を図った. また, ダブルバッファリングを用いることで, 計算に用いるパラメータを外部メモリからフェッチする時間を隠蔽するようにした.
本実装は, 158GOPS の性能, スループット 87.0FPS, 電力効率 3.21GOPS/W を達成し, GPU 実装との比較では性能, 電力効率で劣ったものの, CPU 実装との比較では性能は 1.16 倍, 電力効率は 6.55 倍となった.

Parallel Implementation of ResNet on PYNQ Cluster (version 2020/2/17)

1. 序論

1.1 本研究の背景と目的

近年の研究によって，深層学習が画像認識や自然言語処理などの認識タスクにおいて高い認識精度を実現可能であることが明らかになり，自動運転，自動翻訳，医療をはじめとした様々な分野への応用がすでに行われている．深層学習の中でも，画像認識の分野で高い認識精度を誇るのが畳み込みニューラルネットワーク（CNN: Convolutional Neural Network）である．しかし，CNN は計算量が膨大であり，汎用プロセッサでの実行ではレイテンシ，消費電力の増加が問題となる．

この問題を解決する方法の 1 つとして，ハードウェアレベルでの深層学習の高速化が注目されている．Domain Specific Architecture（DSA）と呼ばれる，特定のアプリケーションの実行のみに特化したハードウェアを設計することにより，低レイテンシ，低消費電力を実現することが可能である．この方針に基づいて，様々な深層学習アクセラレータが開発されている．深層学習アクセラレータとして，GPU や専用チップとともに有力な候補となっているのが FPGA（Field-Programable Gate Array）である．

FPGA はユーザが自分で回路を設計，構成することができるハードウェアである．FPGA は回路の再構成が可能

†1 現在，慶應義塾大学
Presently with Keio University
a) yasu@am.ics.keio.ac.jp
b) hunga@am.ics.keio.ac.jp

ⓒ 2021 Information Processing Society of Japan

であり，開発コストが低く抑えられるほか，ASIC 等の専用チップと比較して短期間での開発が可能であり，GPU と比較して消費電力が小さいなどの利点がある．これらの利点から，従来の IoT やエッジデバイスでの利用にとどまらず，MEC（Multi-access Edge Computing）を実現する 5G モバイルネットワークの基地局に設置する計算基盤や，データセンターへの導入が進んでいる．

しかし，強力な演算機能や膨大なオンチップメモリを備えた FPGA の多くは高額であるため利用しにくい他，単一 FPGA での開発では利用可能資源に限界がある．この問題を解決するために，我々の研究室では M-KUBOS[1] と呼ばれる価格性能比に優れた Zynq ボードに，PYNQ と呼ばれるオープンソース・ソフトウェア・プラットフォームを導入し，低コストかつ高性能な GTH シリアルリンクで接続することで構成される PYNQ クラスタの開発を進めている．

本稿ではこの PYNQ クラスタ上に代表的な CNN のひとつである ResNet-18 を分割して実装し，実機での演算性能や電力を計測し，性能評価を取ることで分割手法と実装の有用性について考察する．

1.2 本報告の構成

本稿の構成を以下に示す．2 節ではまず，MEC および PYNQ クラスタについて説明する．3 節では，畳み込みニューラルネットワークおよび ResNet について説明する．4 節では，本稿に関連する研究を紹介する．5 節では演算特性に基づいた分割，並列化手法を提案，FPGA の設計と実装について述べる．6 節ではこの実装の性能評価を行い，最後に 7 節で結論を述べる．

2. 背景

2.1 MEC

Multi-access Edge Computing（MEC）は，5G モバイルネットワークの標準規格として ETSI（European Telecommunications Standards Institute）で標準化が進められている．MEC は複数のタイプのアクセス技術を含むアクセスネットワークのエッジにおいて，ユーザに近接した位置に IT サービス環境とクラウドコンピューティング機能を提供するものである．5G 無線技術の進歩を背景にしているが，5G に限定されず，WiFi，LPWA，有線ネットワーク環境などでも同様の概念が成立する．MEC のメリットをまとめると以下のようになる．

(1) エッジデバイスから近くの MEC へタスクのオフロードが可能．
(2) アプリケーションをローカル環境で動作させることが可能となり，応答時間やユーザの利便性が向上．
(3) クラウドとのネットワーク帯域を節約し，ネットワークの輻輳を軽減可能．

この特徴を利用し，工場制御，スマートシティのトラフィック管理，セキュリティ制御など，様々な用途が期待されている．また，これまで個人の端末内部で行っていた処理の重いタスクを，近くにある MEC にオフロードして実行するといった利用方法も考えられる．これらの用途には深層学習を用いた画像処理などの AI アプリケーションを用いることが十分考えられるため，MEC は深層学習のように計算量が非常に大きいアプリケーションにも対応する必要がある．

この MEC サーバには以下の要件がある．

(1) 建物の上に設置される基地局に設置するため，低消費電力・低コストであること．
(2) タイミングクリティカルジョブを実行するため，スケジューラビリティが高く，将来性があること．
(3) エッジデバイスからの直接要求にも対応できる強力で多様な I/O 能力を持つこと．

FPGA コンピューティングは，コストと消費電力に優れた MEC サーバとして有力な候補として注目されている．FPGA コンピューティングのもう一つの重要なメリットは，エッジからの要求を直接 I/O で受け付け，ハードワイヤードロジックにおいて一定の性能でジョブを実行することで，タイミングクリティカルなジョブに対応できることである．基地局への複数の要求を処理するためには，マルチ FPGA システムが特に有利である．

2.2 PYNQ クラスタ

Zynq とは Xilinx 社が提供する SoC の名称で，PS（Processing System）と呼ばれる ARM コアと，PL（Programmable Logic）と呼ばれる FPGA を搭載したデバイスである．PYNQ（Python productivity for Zynq）[2] は，元々は Xilinx が比較的小型の Zynq チップを搭載した PYNQ-Z1 用に開発したオープンソース・ソフトウェア・プラットフォームであり，PYNQ 環境上では Python や Jupyter Notebook が動作し，各種ライブラリを利用して AI アプリケーションを簡単に実装できることから，AI プラットフォームとして注目された．

Zynq は組み込みシステムでの利用を想定しており，PS の ARM 用のソフトウェアは PL のハードウェア開発と合わせて Xilinx 社の合成ツール Vivado により設計する．これにより生成したマシンコードとコンフィグレーションデータをまとめて Zynq ボードにダウンロードして実行する．この従来の使用方法は，PS と PL で実行される多数のタスクを Zynq クラスタに受け入れて分散実行しなければならない MEC には不向きである．つまり，MEC のための Zynq クラスタの使い方は，組み込みコンピューティングというよりは，FPGA-in-Cloud[3] に近いものになり，複数のボード同士を接続し，相互に通信しながら演算を行う方式となる．PYNQ は組み込み小型ボード向けに開発さ

表 1 M-KUBOS 主要スペック
Table 1 Key specifications of M-KUBOS.

項目	仕様
搭載 FPGA	XCZU19EG-2FFVC1760
メモリ	PS: 4GB DDR4-2400 PL: 1x 4GB DDR4-2400 SODIMM ソケット
I/O	4x GTY 4TX/4RX (max 28.125Gbps) 4x GTH 8TX (max 16.3Gbps) 4x GTH 8RX (max 16.3Gbps) USB3.0 × 1 USB-UART × 1 1Gb Ether(RJ45) DP1.2

れたが，Zynq でサポートされている Peta-Linux とは異なり，Ubuntu Linux 上に構築されており，Linux サーバ用のソフトウェアスタックを構築することができる．また，PL を制御するためのレイヤがオーバーレイとして統合されているため，PS を停止することなく PL をコンフィグレーションが可能であるほか，Python や Jupyter Notebook，C/C++を含む他のプログラミング言語とのインタフェースをとることができる．Python や Jupyter Notebook からは，ビットストリームの焼き込み，共有メモリへの書き込みやアプリケーションの実行などを行うことが可能である．

このように，PYNQ を M-KUBOS の各ボードに導入することで，クラスタ構築の基盤として用いることができる．しかし，PYNQ は元は 1 ボード向けに開発されたが，M-KUBOS は MEC として用いるため，複数のボードを群管理する必要がある．本研究では，MEC で CNN を実行することを想定して，PYNQ を導入した複数枚の M-KUBOS ボードから構成される PYNQ クラスタに，ResNet を分割して実装し，性能評価を行った．

2.3 現在の PYNQ クラスタの構成

現在の PYNQ クラスタは，**図 1** に示すように 4 枚の M-KUBOS ボードを接続して構成されている．M-KUBOS ボードの主要スペックは**表 1** に示すとおりである．現在は 5 入力 5 出力のスイッチを使用しているが，クラスタのサイズが大きくなれば，簡単に 9 入力 9 出力まで拡張することができ，より多くのボードを密に接続することが可能である．表 1 に示した 4x GTY 4TX/4RX，4x GTH 8TX，4x GTH 8RX が，Samtec 社が提供する高速双方向リンクを実現する Firefly ケーブルを接続できる I/O であり，その性質上，現在のクラスタには 4 つの重複したリンク/スイッチが用意されている．

5 つのポートのうち 1 つを，Xilinx 社が提供しているインターフェイス IP である AXI Stream インタフェースを用いて，ユーザが設計したアプリケーションモジュールに接続し，他の 3 つのポートを用いて M-KUBOS ボード間にリングネットワークを構築する．使用していないもう 1 つのポートを用いてさらに大きなクラスタを形成することも可能である．

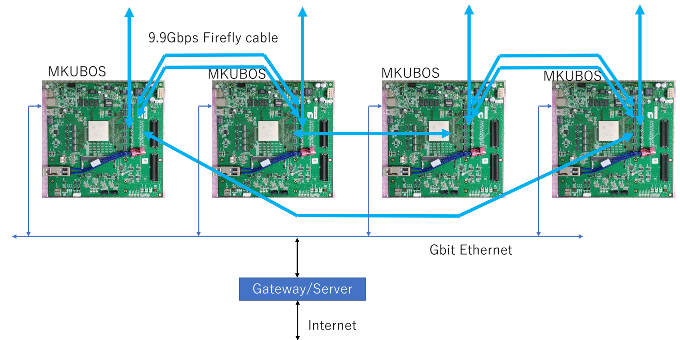

図 1 PYNQ クラスタの相互接続
Fig. 1 Interconnection of PYNQ Cluster.

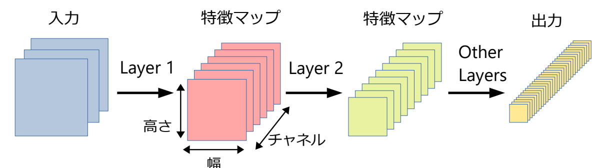

図 2 CNN の模式図
Fig. 2 Pattern Diagram of CNN.

3. 畳み込みニューラルネットワーク

ニューラルネットワークは，画像処理，音声認識など幅広いパターン認識問題に適した計算モデルであり，昨今多くの人工知能アプリケーションの有望な候補となっている．中でも，ニューラルネットワークの一種である畳み込みニューラルネットワーク（CNN: Convolutional Neural Network）は画像認識や物体検出などで高い精度を発揮することで知られ，自動運転における障害物検知などのサービスにも CNN が用いられるようになってきている．

図 2 に CNN の基本的な構造を示す．矢印で示される各層は入力や前の層からのデータに対し演算を行い，出力や次の層へ渡す結果を生成する．

CNN の途中の入出力は，画像の特徴点を抽出したものとなるため，これらの途中結果を特徴マップと呼ぶ．特に，それぞれの層の入力を入力特徴マップ，出力を出力特徴マップと呼ぶ．多くの層において，入力特徴マップに対

ⓒ 2021 Information Processing Society of Japan

して重み，バイアスと呼ばれるパラメータを用いて演算を行い，出力特徴マップを生成する．また，図 2 で示されるように，3 次元の特徴マップの各次元をそれぞれ幅，高さ，チャネルと呼ぶこととする．

CNN の層には畳み込み層，全結合層，プーリング層，ReLU などがあり，中でも畳み込み層の計算量は全体の大部分を占める．

3.1 ResNet

本稿で高速化の対象とした ResNet[4] は，2015 年の ILSVRC で優勝した CNN モデルであり，従来のモデルと比較して層が深いことが特徴である．CNN においては，層を深くすることでより複雑な特徴を抽出することができ，精度が向上すると考えられるが，それまでの CNN の研究においては層を深くしすぎると認識精度がかえって悪くなることが報告されていた．CNN では学習の際に活性化関数を微分して勾配を求めていくが，深い層にいくほど入力と出力の差が小さくなっていくため，層を深くしていくと勾配が非常に小さな値に収束してしまう．これを勾配消失問題とよび，CNN の層を深くできない要因となっていた．この問題を解決するために，通常は入力信号 x から $H(x)$ を出力として学習するところを，ResNet では残差 $F(x) = H(x) - x$ を出力として逆誤差伝搬による学習を行う．ResNet では**図 3** に示すような，いくつかの層をスキップするショートカットコネクションを導入したブロックを基本的な構造としている．これを残差ブロックとよぶ．この構造を何層にも渡って重ねていくことで，勾配の消失を防ぎつつ層の数を増やすことができる．

ResNet は層の深さに関わらず，最初に畳み込み層と最大プーリング層が，最後に平均プーリング層と全結合層があり，その間にモデルに応じた数の残差ブロックを用意する設計となっていて，ResNet-18，ResNet-34，ResNet-50，ResNet-101，ResNet-152 などが一般的であるが，さらに層数を増やすことも可能である．このスケーラビリティの高さから，本研究では ResNet を対象とし，まずはモデルサイズが小さくデバッグを行いやすい ResNet-18 を実装することとした．

4. 関連研究

この章では本研究と関連性の高い，FPGA 上に実装された CNN アクセラレータに関する研究を紹介する．

4.1 FPGA での CNN 実装

FPGA ベースの CNN 実装には，リカレント構造とパイプライン構造の 2 通りの方法がある．リカレント構造は [5] のように，オンチップリソースをほぼフルに活用して統一された計算ユニットを構築し，異なる層間でユニットを共有する方法である．[6] など先行研究の多くの FPGA を用いた CNN アクセラレータは，畳み込み演算器（CLP: Convolutional Layer Processor）を 1 つしか持たない設計をしていることが多い．一方，パイプライン構造は [7] のように，ニューラルネットワークの各層を個別のパイプラインステージとして実装する．この方法では，層ごとに最適な設計を実装することが可能であるが，各層が利用できるリソースが制限される．本実装は ResNet の最初の畳み込み層のみ個別の設計を用意し，それ以外の畳み込み層には共通の設計を用いたリカレント構造となっている．

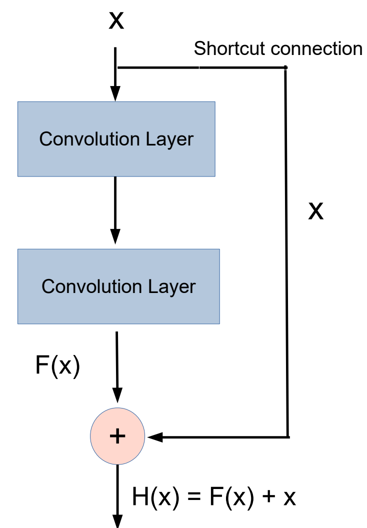

図 3 ResNet の基本構造
Fig. 3 Basic structure of ResNet.

4.2 CNN のマルチ FPGA への分割実装に関する研究

深層学習アクセラレータを単一 FPGA に実装する研究だけでなく，マルチ FPGA へ分割実装する研究も広く行われている．深層学習のネットワークモデルの巨大さは，1 枚の FPGA を用いてアクセラレータを実装するにはリソース不足に陥りやすく，全体の演算性能も制限されたものとなってしまいがちであるため，複数の FPGA を用いた実装を行うことは，FPGA のリソースを効率よく利用し，巨大な深層学習のネットワークをより低遅延，高エネルギー効率で実行するために望ましいといえる．

[8] では，複数枚の FPGA の設計空間探索問題を，1 枚の FPGA の設計空間探索の部分問題として扱うことで，効率の良い探索を行った．この手法を用いて設計したシステムの実行結果は，AlexNet と VGG-16 をベンチマークとして，6 枚の FPGA からなるシステムで CPU，GPU とそれぞれ比較して最大で約 21 倍，2 倍の電力効率を達成した．

[9] では，[8] の研究を踏まえて，FPGA 同士の接続網のトポロジーも考慮したアルゴリズムにより分割を行っ

Vol.2021-ARC-244 No.43
Vol.2021-SLDM-194 No.43
Vol.2021-EMB-56 No.43
2021/3/26

ている．基本的には [8] と同様に，動的計画法による設計空間探索を行う方針だが，この研究ではさらに，二分探索によって FPGA のリソースを割り当て，ループアンローリングのパラメータなどを決定している．この研究では ResNet-152 を FPGA の数を変え実装，評価し，最大 16 枚の FPGA を用いることで，GPU の 2 倍の性能向上を達成している．これら 2 つの研究は本研究と同様に，CNN の 1 つのネットワークを層と層の間で分割する手法であり，[9] は評価に ResNet を用いている点からも，本研究との比較を行いやすい．本研究は実装に Zynq ボードを用いた点と，手作業での分割実装を行った点において，これらの研究と異なる．

5. 畳み込み演算のパイプライン化手法と FPGA 実装

5.1 ResNet-18 の分割実装

本研究では，ResNet-18 を 4 枚の MKUBOS ボードに分割して実装することで，**図 4** に示すような，各ボードでの処理をパイプラインの 1 ステージとしたパイプライン処理を実現する．説明のために使用する各ボードをボード 0 から 3 とすると，ボード 0 の PS 部から入力データを入れ，ボード 0 から 3 まで PL 部で順番に処理を行って次のボードに途中結果を渡していき，ボード 3 の PS 部に最終的な結果を出力するような設計となっている．

PS 部と PL 部の間は AXI4-Lite インターフェイスを通じて 32bit 幅のデータを受け渡す．本実装では 32bit 幅を使って 32bit の浮動小数点数 1 つを送受信している．ボード間で受け渡すデータは AXI4-Stream インターフェイスを通じて 170bit 幅のデータを STDM（Static Time Division Multiplexing）スイッチに送ることで次のボードに転送される．本実装では 170bit のうち 128bit がデータ部分となっており，他は STDM スイッチでの送受信時に必要な部分である．データ部分の 128bit 幅には 16bit の固定小数点数 8 つをまとめて送受信している．また，各ボードはそれぞれ重みデータを保持するための DDR DRAM を備えており，AXI4 インタフェースを通じて 128bit 幅のデータをやりとりする．ここでも 128bit 幅に 16bit の固定小数点数 8 つをまとめて送受信している．

パイプライン処理では実行時間が最も長いステージによって全体の性能が決定されるため，各ボードの実行時間が可能な限り一定になるようにする．そのために各層の処理に要する時間を求め，そこから最適な分割を決定する．しかし，最初の畳み込み層は他の畳み込み層と設計の共有ができず，出力特徴マップのサイズが非常に大きいため，オンチップメモリリソースを占有してしまう．そのため最初の畳み込み層と最大プーリング層は他の畳み込み層と同じボードに割り当てることができない．したがって，残り 3 枚のボードで他の畳み込み層を分割し，最後のボードには平均プーリング層，全結合層を割り当てる．平均プーリング層と全結合層は，入出力のサイズが小さいことと，計算に必要なリソースが少ないことから，畳み込み層と同じボードに割り当てても問題ないと判断した．

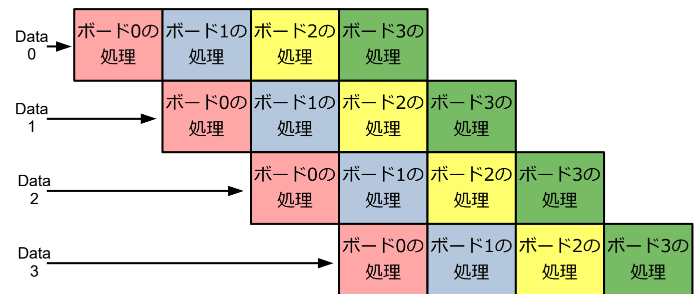

図 4 4 ボードのパイプライン処理
Fig. 4 4 board pipeline processing.

最初の畳み込み層と最大プーリング層以外を 3 枚のボードに分割するにあたって，まず最後のボードに平均プーリング層と全結合層を割り当てる．次に残りの畳み込み層を各ボードの処理時間ができる限り等しくなるよう分割して配置する．そのためにまず，Xilinx 社が提供する高位合成ツール Vivado HLS で高位合成を行い，各層の実行時間の見積もりから分割を決める．その後，PYNQ クラスタ上で実際に動作させ，実行時間を測定し，分割の微調整を行う．

1 枚目のボードに実装した畳み込み層と最大プーリング層は，他のボードと比較して実行時間がやや短い程度だったため，本実装ではそれ以上の積極的な高速化を行わなかった．より多くのボードを用いた実装によって，他のボードの実行時間が短縮された場合には，この部分の高速化について考える必要がある．

5.2 量子化

本実装では，32bit の浮動小数点数で学習を行ったモデルから，16bit の固定小数点数への量子化を重み，特徴マップの両方に行った．量子化手法は線形量子化を採用し，浮動小数点での重み，特徴マップの分布を調べることで，オーバーフローが発生しないような整数部のビット幅を選択した．量子化によってメモリバンド幅，オンチップメモリリソースの要求を低減することができるほか，32bit 浮動小数点数どうしの乗算には DSP ユニットが 3 つ必要となるのに対し，16bit 固定小数点数どうしの乗算は DSP ユニット 1 つで行われるため，量子化を行うことで FPGA が持つ DSP ユニットを有効に利用することができる．また，[10]

ⓒ 2021 Information Processing Society of Japan

表 2 本実装と浮動小数点モデルの認識精度の比較
Table 2 Comparison of recognition accuracy between this implementation and the floating point model.

	Float32	Fixed16
Top-1 Accuracy	78.6%	78.5% (-0.1%)
Top-5 Accuracy	93.4%	93.2% (-0.2%)

によると，16bit 幅への線形量子化では，認識精度はほとんど低下しない．

本実装と浮動小数点モデルの精度比較を**表 2** に示す．ImageNet の画像 1000 枚を用いて ResNet-18 による推論を行い，Top-1 と Top-5 の認識精度を比較した．ここで，Top-1，Top-5 とは，それぞれ推論結果の上位 1 位または 5 位までのうちに正解のクラスが含まれているかという精度の指標であり，CNN の認識精度を評価する際には多くの場合この指標が用いられる．Float32 は元の浮動小数点モデル，Fixed16 は 16bit 固定小数点への量子化を行った本実装の精度である．これにより，本実装における 16bit 固定小数点への量子化でもほとんど精度の低下が起こらないことを確認できた．

5.3 畳み込み演算の高速化

本研究において ResNet-18 を高速化するために用いた，畳み込み演算を高速化する手法について述べる．説明のために，入力特徴マップのチャネル数を ICH，高さを IH，幅を IW とし，出力特徴マップのチャネル数を OCH，高さを OH，幅を OW とする．また，カーネルサイズは K とする．畳み込み演算は，入力特徴マップ $IH \times IW \times ICH$ と，$OCH \times ICH \times K \times K$ 個の重みから，出力特徴マップ $OH \times OW \times OCH$ を生成する操作であり，その演算量は式 1 で表される．

$$\text{conv_operations} = OH \times OW \times K \times K \times OCH \times ICH \quad (1)$$

この演算を FPGA 上で並列化して実行することで高速化を実現する．並列化は，入出力チャネルのループをアンロールすることで実現した．カーネルループをアンロールして並列度を上げる手法も存在するが，本研究では行わなかった．入出力チャネルでアンロールする場合，最初の畳み込み層以外の畳み込み層は入出力チャネルが 2 の累乗となっているため，入力チャネル，出力チャネルをそれぞれ 2 の累乗で分割することでアンローリング係数を選択する．一方，カーネルループをアンロールする場合，ResNet-18 においては 3×3 のループをアンロールすることになる．これだけでは並列度が小さすぎるため，カーネルループのアンロールに加えて入出力チャネルのアンロールも行う必要がある．すなわち，カーネルループのアンロールは，入出力チャネルのアンロールからさらに並列度を大きくするために用いる手法といえる．しかし，ResNet-18 の最初の畳み込み層以外の入出力チャネル数は 64 以上となっているため，畳み込み層に単一の設計を用いた場合でも最大で 64×64 の並列度を計算上では実現可能だが，これは今回実装に用いた FPGA ボードが持つ DSP ユニットの数 1968 を超えている．よって，本研究においてはカーネルループをアンロールする余地はないと考え，入出力チャネルのループのみをアンロールした．

入出力チャネルのアンローリング係数を決定するための設計空間探索アルゴリズムをアルゴリズム 1 に示す．このアルゴリズムでは，使用可能な DSP の数を $To \times Ti$ が超えないよう，To と Ti を交互に 2 倍していく．これにしたがって入力チャネルアンローリング係数 Ti，出力チャネルアンローリング係数 To を求め，これらの係数に従い $To \times Ti$ 個の乗算を並列実行する．

Algorithm 1 設計空間探索アルゴリズム

```
Set available DSP: DSP_NUM
Input Channel Size: ICH
Output Channel Size: OCH
To ⇐ 1
Ti ⇐ 1
while To × Ti × 2 ≤ DSP_NUM do
   if Ti < To  &&  Ti × 2 ≤ ICH then
      Ti  × = 2;
   else if To ×2 ≤ OCH then
      To  × = 2;
   else
      break;
   end if
end while
```

また，計算に必要な重みは DDR DRAM からフェッチする必要がある．本実装では，重みのフェッチに要する時間を隠蔽するためにダブルバッファリングを行い，重みを保持するバッファを 2 つ用意し，交互に計算に用いるようにした．これによって，片方のバッファのデータを用いて計算を行っている間に，もう片方のバッファを用いて次の計算に必要な重みのフェッチが可能となり，処理時間は計算時間と重みのフェッチに要する時間を足したものではなく，どちらかより時間を要する方のみの処理時間と同等になる．これまで述べた方針から，畳み込み演算のアルゴリズムはアルゴリズム 2 のようになる．本研究において実装を行った環境である Vivado HLS が提供するプラグマを挿入することによりデータフロー処理，パイプライン処理，ループアンロールを行う．このように，重みを $To \times Ti$ 個使って計算を行うと同時に，次の計算に使う $To \times Ti$ 個の重みを DDR DRAM からフェッチする．

6. 評価

6.1 実験結果

本研究では，Xilinx 社が提供する高位合成ツール Vivado HLS 2019.1.3 で C++で記述したコードから IP を生成し，

Algorithm 2 畳み込み演算

```
Require: input:IW × IH × ICH; output :OW ×OH × OCH;
  weight_buf :To ×Ti
Ensure: output
  for to = 0; to < OCH/To; to++ do
    for ti = 0; ti < ICH/Ti; ti++ do
      for each(ky, kx) within(K, K) do
        pragma HLS DATAFLOW
        load_weight(weight_buf, ddr)
        for each(h, w) within(OH, OW) do
          pragma HLS PIPELINE
          for too = 0; too < To; too++ do
            pragma HLS UNROLL
            for tii = 0; tii < Ti; tii++ do
              pragma HLS UNROLL
              output[w][h][to × To + too] += input[S ×
              w + kx − P][S × h + ky − P][ti × Ti + tii] ×
              weight_buf[too × Ti + tii]
            end for
          end for
        end for
      end for
    end for
  end for
```

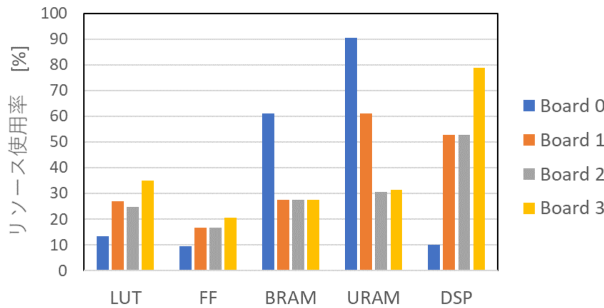

図 5 各ボードのリソース使用量とその割合
Fig. 5 Resource usage of each board and its percentage.

同じく Xilinx 社が提供する合成ツール Vivado 2019.1.3 で配置配線することで実装を行った．配置配線後の，各ボードのリソース使用量とその割合を**図 5** に示す．ボード 0 の DSP 使用率が低い理由は 5.1 節で述べたように演算の並列化を積極的に行わなかったためだが，ボード 1，ボード 2 においても DSP の使用率が 50%強に留まっている．本研究で用いた並列化手法では，これ以上並列度を大きくすると DSP の必要数が 2 倍となり，使用率が 100%を超えて配置配線が不可能となってしまうためである．DSP の使用率は設計の性能に直結するため，異なる並列化手法を取り入れ DSP の使用率を改善することが今後の課題である．

次に，各ボードの実行時間と消費電力を**表 3** に示す．各ボードの実行時間をできるだけ揃えるために，まず Vivado HLS の合成結果から各層の実行クロックサイクル数を確認し，どこで分割するかを決める．次に実機での実行時間を確認して微調整を行うことでボード間の実行時間の差を可能な限り小さくした．FPGA の消費電力は，Vivado の合成結果から確認した．

表 3 各ボードの評価
Table 3 Evalation of each board.

	Board 0	Board 1	Board 2	Board 3
実行時間 (msec)	8.7	10.5	11.5	9.3
消費電力 (W)	11.7	12.1	12.4	13.0

表 4 CPU，GPU との性能比較
Table 4 Performance comparison with CPU and GPU.

	CPU	GPU	本実装
Device	AMD Ryzen Threadripper 3960X	NVIDIA GeForce RTX 2080 Ti	Xilinx Zynq UltraScale+
動作周波数 (MHz)	3800	1350	100
ソフトウェア実装	PyTorch 1.7.1 + Python3.6.8	PyTorch 1.7.1 + Python3.6.8 + CUDA 10.1	Xilinx Vivado HLS 2019.1.3
演算精度	Float32	Float32	Fixed16
レイテンシ (msec)	13.4	2.8	11.5
性能 (GOPS)	136	649	158
電力 (W)	TDP 280	83	49.2
電力効率 (GOPS /W)	0.49	7.82	3.21

パイプライン処理では最も実行時間が長い部分で性能が決定されるため，本実装では 11.5msec ごとに 1 つのデータを処理することが可能である．このため，スループットは $1/0.0115 = 87.0$ FPS となり，一般的なフレームレートのひとつである 60FPS の映像を処理する十分な能力があることがいえる．また，ResNet-18 の総演算量は 1.816×10^9 であるため，性能は $1.816 \times 10^9/0.0115 = 158$ GOPS となる．電力効率は性能と総消費電力から求まり，本実装は 3.21 GOPS/W を達成した．

6.2 性能比較

本実装の実行時間，消費電力，電力効率を CPU，GPU と比較した結果を**表 4** に示す．比較対象として，CPU は AMD Ryzen Threadripper 3960X を，GPU は NVIDIA GeForce RTX 2080 Ti を用いた．CPU は 24 スレッドで動作させた．CPU，GPU ともに，深層学習フレームワーク PyTorch を用い，学習済みの ResNet-18 モデルで推論を実行し，評価を行った．

GPU の消費電力の測定は NVIDIA System Management Interface [11] を利用して行った．GPU が搭載されたマシンの CLI からインターフェイスを起動し，実行時と待機時の電力の差分を実行時の消費電力とした．

表 4 に示した通り，本実装は CPU の深層学習フレームワーク実装に対して 1.16 倍の高速化，6.55 倍の電力効率を達成したが，GPU の深層学習フレームワーク実装に対しては電力効率でも劣る結果となった．ただし，GPU の動作には CPU も必要となるため，実際の消費電力はさらに大きくなり，本実装との電力効率の差は小さくなる．

マルチ FPGA に CNN を実装する関連研究 [9] との比較を**表 5** に示す．本研究と同様に複数枚の FPGA ボードに ResNet を実装しており，層数や用いた FPGA ボードの違

ⓒ 2021 Information Processing Society of Japan

表 5 関連研究との性能比較
Table 5 Performance comparison with related works.

	本実装（4 ボード）	[9]（4 ボード）	[9]（16 ボード）
Device	Xilinx Zynq UltraScale+	Xilinx Vertex UltraScale	Xilinx Vertex UltraScale
動作周波数（MHz）	100	150	150
CNN モデル	ResNet-18	ResNet-152	ResNet-152
演算精度	Fixed16	Fixed16	Fixed16
性能 （GOPS）	158	62.9	256.6

いはあるものの，比較対象として妥当といえる．4 ボードの実装と比較すると本実装は 2.5 倍の性能を達成している．また，[9] の 4 ボードと 16 ボードの性能を比較すると，FPGA の枚数に比例して性能が向上していることがわかる．ここから，本研究においても今後 PYNQ クラスタを拡張してより多くの M-KUBOS ボードを用いることで，モデルを巨大化しても性能向上を図れると考えられる．

7. 結論

本稿では，M-KUBOS ボード 4 枚から構成される PYNQ クラスタ上において，ResNet-18 を分割実装し，評価を行った．ボード 1 枚の処理をパイプラインの 1 ステージとしてパイプライン処理を行うことにより，ResNet-18 を 4 並列で動作させ，スループットを向上させた．また、16bit 固定小数点への量子化，ループアンローリング，ダブルバッファリングなどの手法を用いることにより，さらなる高速化を図った．

本実装はスループット 87.0FPS，性能 158GOPS，電力効率 3.21GOPS/W を達成し，GPU の深層学習フレームワーク実装との比較では性能，電力効率で劣ったものの，CPU の深層学習フレームワーク実装との比較では，1.16 倍の高速化，6.55 倍の電力効率を達成した．

現状，DSP の使用率の改善，さらなる量子化，その他の CNN の高速化手法を用いることなど，性能向上の余地は大きい．具体的には，入出力チャネル以外のループをアンロールすることで DSP の使用率を改善することが考えられる．量子化については，8bit 整数値への量子化を行い，認識精度の低下なしで高速化を行うことが課題となる．ループアンローリング以外の畳み込み層を高速化する方法を実装するための検討も必要となる．また，PYNQ クラスタが拡張されれば，本実装を拡張することで，さらに多くのボードを用いた実装を行い，評価を行うことも容易である．そして，ResNet は最大 152 層となる複数のモデルが存在するため，今後段階的な拡張を行うことも考えられる．分割手法に関してもさらなる研究が必要であり，手作業で行っている分割の自動化などが今後の課題といえる．

謝辞 本研究は科学技術振興機構戦略的研究推進事業（JST），CREST，JPMJCR19K1 の支援を受けたものです．

参考文献

[1] PALTEK, “FPGA computing platform M-KUBOS,” https://www.paltek.co.jp/design/original/m-kubos/ (accessed 2021-1-20).

[2] Xilinx Inc, “PYNQ - Python productivity for Zynq - Home,” http://www.pynq.io/ (accessed 2021-1-22), 2019.

[3] S. Byma, J.G. Steffan, H. Bannazadeh, A.L. Garcia, and P. Chow, “FPGAs in the Cloud: Booting Virtualized Hardware Accelerators with OpenStack,” 2014 IEEE 22nd Annual International Symposium on Field-Programmable Custom Computing Machines, pp.109-116, May 2014.

[4] K. He, X. Zhang, S. Ren, and J. Sun, “Deep Residual Learning for Image Recognition,” 2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR), pp.770-778, June 2016.

[5] H. Sharma, J. Park, D. Mahajan, E. Amaro, J.K. Kim, C. Shao, A. Mishra, and H. Esmaeilzadeh, “From High-Level Deep Neural Models to FPGAs,” 2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO), pp.1-12, 2016.

[6] C. Zhang, P. Li, G. Sun, Y. Guan, B. Xiao, and J. Cong, “Optimizing FPGA-based Accelerator Design for Deep Convolutional Neural Networks,” Proceedings of the 2015 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays, pp.161–170, New York, NY, USA, 2015, ACM.

[7] X. Zhang, J. Wang, C. Zhu, Y. Lin, J. Xiong, W. Hwu, and D. Chen, “DNNBuilder: an Automated Tool for Building High-Performance DNN Hardware Accelerators for FPGAs,” 2018 IEEE/ACM International Conference on Computer-Aided Design (ICCAD), pp.1-8, 2018.

[8] C. Zhang, D. Wu, J. Sun, G. Sun, G. Luo, and J. Cong, “Energy-Efficient CNN Implementation on a Deeply Pipelined FPGA Cluster,” Proceedings of the 2016 International Symposium on Low Power Electronics and Design, p.326–331, New York, NY, USA, 2016, Association for Computing Machinery.

[9] W. Zhang, J. Zhang, M. Shen, G. Luo, and N. Xiao, “An Efficient Mapping Approach to Large-Scale DNNs on Multi-FPGA Architectures,” 2019 Design, Automation Test in Europe Conference Exhibition (DATE), pp.1241-1244, March 2019.

[10] K. Guo, S. Zeng, J. Yu, Y. Wang, and H. Yang, “A Survey of FPGA-Based Neural Network Inference Accelerators,” ACM Trans. Reconfigurable Technol. Syst., vol.12, no.1, March 2019.

[11] “NVIDIA System Management Interface,” https://developer.nvidia.com/nvidia-system-management-interface (accessed: 2021-01-22).

ⓒ 2021 Information Processing Society of Japan