Vol.2019-ARC-235 No.38
Vol.2019-SLDM-187 No.38
Vol.2019-EMB-50 No.38
2019/3/18

FPGAを用いた量子化DNNの推論ハードウェアの設計

山本 椋太[1,a)]　岡本 卓也[1]　本田 晋也[1]　張 天豫[2]　趙 茜[2]　中本 幸一[2]　若林 一敏[3]

概要：近年，DNN (Deep Neural Network) を用いた推論器の需要が高まっている．組込みシステムにおいても，FA の効率化や自動運転などの分野において，その需要が高まっている．組込みシステムにおいては，リアルタイム性やメモリ容量などの制約が強い場合が多く，CPU や GPU ではその制約を満足できない場合があり，FPGA を使用することが検討されている．本稿では，我々が現在開発を進めている FPGA に対する DNN (Deep Neural Network) 推論器のアクセラレータ生成フレームワーク (N3 フレームワーク) について述べ，また，3 値ニューラルネットワークを題材に，その高速化手法について述べる．

Design of Quantized DNN Inference Hardware on an FPGA Board

1. はじめに

近年，DNN (Deep Neural Network) を用いた推論器の需要が高まっている．組込みシステムにおいても，FA の効率化や自動運転などの分野において，その需要が高まっている．特に組込みシステムにおいては，リアルタイム性やメモリ容量などの制約が強い場合が多く，CPU や GPU ではその制約を満足できない場合がある．CPU では，低速になることが多く，高速化するためにメニーコアで環境を構築すると消費電力が課題となる．GPU では，組込み向け GPU プラットフォームの Jetson (Nvidia 社) などがリリースされているが，その利用方法によっては，実行性能に対して消費電力が大きいという報告がある [1].

そこで，専用ハードウェアを構築可能な，ASIC や FPGA を使用することが検討されている．機械学習においては，その研究が盛んであるため，日々新たなネットワークが提供されている．そのため，推論器のハードウェアも変化する可能性があり，設計変更の度に多大なコストが必要となる ASIC を用いることは現実的ではない．加えて，現在 FPGA ベンダである Xilinx 社が，DNN を用いたエッジコンピューティングに対する研究開発およびフレームワークの公開に積極的に取り組んでおり，FPGA を用いた DNN 推論のための環境が充実しつつある．

しかしながら，FPGA を用いた高速な DNN の推論ハードウェアの設計のために，ハードウェア記述言語に対する知識が必要となることから専門性が高くなり，AI 技術者にとってはその設計が非常に困難となる．FPGA を利用する上では，推論処理だけではなく入力データの受け取りや前処理，推論結果の利用処理などの DNN 以外の処理が必要となる．そのため，DNN の推論器を含めたシステム全体を容易に構築可能な開発フレームワークが求められている．

そこで，本稿では，我々が現在開発を進めている DNN 推論器のアクセラレータ生成フレームワークについて述べる．以降，本フレームワークを N3 フレームワークと呼称する．N3 フレームワークの概要を図 1 に示す．N3 フレームワークは，既存 DNN フレームワークの学習済みデータを用いて，推論器のための C ソースコードを生成する．生成された C ソースコードはシステムレベル設計環境の SystemBuilder[2] によって，ソフトウェア部とハードウェア部に分割される．ハードウェア部は推論処理を実行し，ソフトウェア部はハードウェア部を起動するために使用される．生成される設計既述が C 言語であるため，AI 技術者にとっても HDL に比べて取り扱いやすくなる．

加えて，N3 フレームワークにおける推論器の設計についても述べる．SystemBuilder によって設計を行う場合，ハードウェア部分は CyberWorkBench (以下，CWB) [3] によって高位合成 (High Level Synthesis, 以下，HLS と呼

[1] 名古屋大学
Nagoya University
[2] 兵庫県立大学
University of Hyogo
[3] 日本電気株式会社
NEC Corporation
[a)] muku@ertl.jp

Vol.2019-ARC-235 No.38
Vol.2019-SLDM-187 No.38
Vol.2019-EMB-50 No.38
2019/3/18

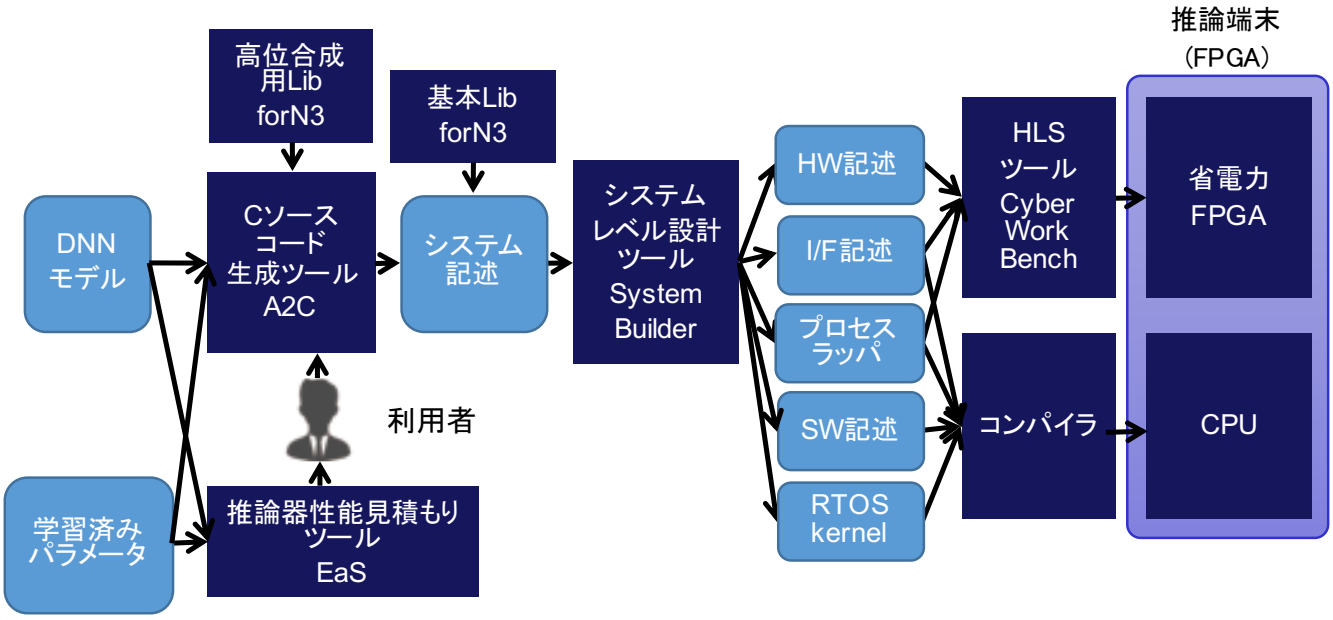


図 1　N3 フレームワークの概要

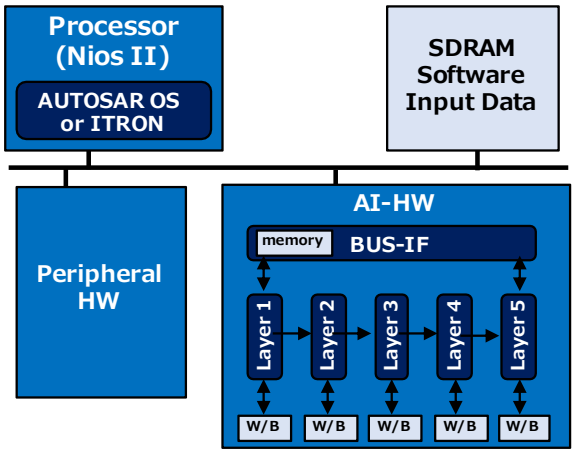


図 2　推論器の構成

ぶ) および論理合成によって生成される．HLS においては，ただ動作するだけの C ソースコードでは，効率的なハードウェアを生成できないことがある．そのため，HLS によって高速になるような記述方法も兼用する必要がある．また，ソフトウェア部分は RTOS のアプリケーションとして生成される．具体的には，AUTOSAR 準拠の RTOS や ITRON 準拠の RTOS を選択できる．

そして，HLS 向けの C ソースコードは，ただ動作するだけのコードでは効率の良いハードウェアが生成されるとは限らない．そのため，本稿では，特に weight と bias を 3 値化したニューラルネットワーク (Neural Network, 以下 NN と呼ぶ) を題材として，効率の良い推論器の設計方法について検討する．本稿では，図 2 に示すような構成を取ることとして説明を進める．

2. N3 フレームワークの概要

N3 フレームワークの概要は図 1 に示した通りである．本章では，図 1 における，システム記述が生成されるまでと，システム記述から実行されるまでに分割して説明する．

2.1 システム記述の生成

N3 フレームワークは，DNN の学習結果およびネットワーク定義ファイルを入力として，動作を開始する．

順に，フレームワークの動作の流れについて説明する．推論器性能見積もりツールの EaS について説明する．EaS は，NN のレイヤ構成や，パラメータ数，学習済みパラメータから，量子化の程度によってメモリ使用量や推論精度がどのように変化するかを推定する．学習済みパラメータは実数によって得られていることを想定しており，学習済みパラメータを K-means 法によってクラスタリングし，その結果から量子化を行う．その中から，利用者は適当だと考える精度を選択し，C プログラム生成ツール A2C に与える．このとき，生成可能な精度のリストをユーザに与えており，その中から任意の精度を選択して生成する．

次いで，C プログラム生成ツール A2C を説明する．A2C では，以下のファイルがシステム記述として生成される．

(1) 推論ハードウェアの C ソースコード
(2) 学習済みパラメータ (weight および bias) をまとめたヘッダファイル
(3) 推論ハードウェアを起動するために，CPU (NIOS II) 上で実行される C ソースコード
(4) システム定義 (System DeFinition, 以降， SDF と呼ぶ) ファイル

まず，推論ハードウェアの C ソースコードについて説明する．A2C は，様々な高速化手法を適用した C ソースコードのテンプレートを用意しておき，それに対して入出力特徴量画像の枚数とサイズ，出力特徴量画像の枚数の情報を埋め込むことで，推論ハードウェアの C ソースコードを生成する．使用するビット幅に合わせて推論ハードウェアも変化するため，最終的には精度ごとに推論ハードウェアのためのテンプレートファイルを用意し，それらに対して既述した NN の情報を埋め込むことで，精度の選択も可能となるようにする．

次いで，学習済みパラメータをまとめたヘッダファイルについて説明する．学習済みパラメータは，EaS によって量子化されている前提である．量子化されたパラメータをパッキングしてヘッダファイルが生成される．このとき，各レイヤの weight と bias をそれぞれ配列として宣言し，CWB では ROM として扱われるように指定する．パッキングにおいては，メモリアクセス回数を削減する目的がある．これまでの先行研究 [4] から，メモリアクセスにおけるレイテンシが推論器のスループットを損なう要因として大きいことがわかっている．そのため，メモリアクセス回数を減らし，かつメモリ使用量も減らす目的でパッキングを行う．例えば，ヘッダファイルで宣言されているパラメータの配列が，符号なしの 32bit 整数型で宣言されていることとする．このとき，量子化後に各パラメータが 2bit である場合には，16 個のパラメータが配列の 1 つの要素に格納される．そのため，1 度のメモリアクセスで 16 個ものパラメータを得ることができるため，メモリアクセス回数を 1/16 に減らすことができる可能性がある．

そして，推論ハードウェアを起動するために，CPU 上で実行される C ソースコードについて説明する．このソースコードは，入力した画像をメモリに書き込み，推論結果を表示することができる．既述の通りだが，このソフトウェアは RTOS のアプリケーションとして生成される．このとき，AUTOSAR 準拠の RTOS を使用することができ，車載システムに対して親和性が高い．そのため，推論結果を利用した処理を追記することも可能である．ここでは，基本ライブラリ (基本 Lib forN3) を参照することができ，これを利用することでモータ制御などを容易に行うことができる [5], [6]．基本ライブラリとは別であるが，他にも，LCD 制御やタッチパネルの制御などのためのサンプルデザインを SystemBuilder が用意しているため，様々な入出力デバイスを駆動できる．

最後に，SDF ファイルについて述べる．SDF ファイルには，以下の記述が行われる．

- プロセス割当（プロセスをハードウェアとして配置するか，ソフトウェアとして配置するか）の定義
- チャネルの定義
- メモリの定義
- プロセスの定義

ここで，上記の定義について説明する．

SystemBuilder では，プロセス単位で処理を記述している．そのため，それらのプロセスをソフトウェアとして実装するか，それともハードウェアとして実装するかをプロセス割当として選択する必要がある．次いで，チャネルの定義では，通信プリミティブについて定義を行う．通信プリミティブは，プロセス間通信を行うためのインタフェースである．本研究では，SystemBuilder で使用可能な通信プリミティブのうち，BC (Blocking Channel) チャネルおよび MEM (Memory Access) チャネルを使用する．BC チャネルは，受信側が送信待ちをするチャネルで，MEM チャネルは，受信側が送信待ちをせず，SDF 上であらかじめ指定したメモリアドレスからの相対アドレスによってメモリアクセス可能なチャネルである．そして，メモリ定義では上述の MEM チャネルで使用するメモリのアドレスを定義する．最後に，プロセス定義では，プロセスが定義されているファイル名，およびエントリとなる関数名，および入出力それぞれのチャネルを書き入れる．

SDF ファイルを自動生成する際は，推論器をハードウェアとして指定し，推論器の起動および結果の受信部のみソフトウェアとして指定する．チャネル定義は，BC チャネルによって前段のレイヤを送信側，後段のレイヤを受信側として定義をする．メモリ定義では，基本的には weight および bias は内部メモリに保存されるように配置し，それらのサイズによっては外部メモリに保存されるようにする．

以上によって，システム記述が生成される．

2.2 システム記述の利用

2.1 において生成されたシステム記述を，SystemBuilder に与える．SystemBuilder の動作フローを，図 3 に示す．C ソースコードは，ハードウェアプロセス，ソフトウェアプロセスに分離され，それぞれ，HLS およびコンパイルのために必要なファイルが生成される．加えて，I/F としてチャネルの定義もハードウェアプロセスおよびソフトウェアプロセスから参照できるようにする．

その後，HLS およびコンパイルを行う．HLS については，FPGA ボードに合わせてクロック設定などは必要であるが，基本的にはオプションの変更なくボタンクリックによって HDL を生成することができる．コンパイルについても，Makefile が作成されているため，C ソースコードに誤りが存在しなければ make コマンドのみでビルドすることが可能である．

ここで，生成された HLS を Quartus II (Intel 社) によって論理合成するが，論理合成のためには一般に時間がかかる上に，実機実行では内部信号を確認することができないため，実機実行によるデバッグ作業は，コストがかかる．そこで，SystemBuilder では，コシュミレーションによっ

Vol.2019-ARC-235 No.38
Vol.2019-SLDM-187 No.38
Vol.2019-EMB-50 No.38
2019/3/18

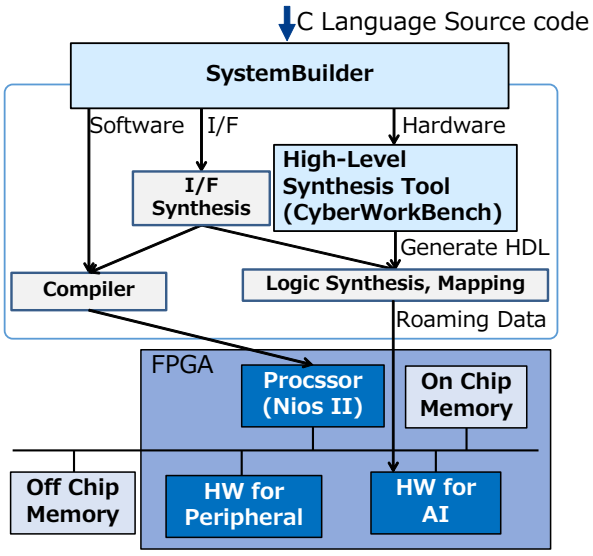


図 3　推論器の構成

て動作確認を行うことができる．コシュミレーションの際には，Questa Sim (Mentor 社) および ISS の Skyeye を利用する．

最後に，以上によって正常に動作することを確認したハードウェアを FPGA 上に構築すると，実行準備は完了である．

2.3　現在のフレームワークの開発状況

現在，EaS や A2C の改良を行っている．EaS においては，メモリ使用量の表示を現在行っていないため，その機能の追加を行う予定である．また，A2C に関しては，さらに量子化の程度を変えた場合に効率の良い C ソースコードの検討を行っているところであり，C ソースコードの拡充を行っている．

また，SystemBuilder の改良を行っている．推論器を実装する際にはメモリアクセスによるレイテンシを改善することによってスループットが向上するとわかっている．そのため，PFBC (Prefetch FIFO BC) チャネルを新たに実装した．メモリの読み込み時のバースト転送を実現したため，メモリアクセス回数を更に低減できることを期待している．加えて，現状の SystemBuilder は Intel 社製の FPGA に対応可能だが，Xilinx 社製の FPGA には対応することができない．今後，Xilinx 社製の FPGA にも対応を進め，利用可能なハードウェアプラットフォームを増やすことを検討している．加えて，割込み要求によって AI のハードウェアを駆動できるように検討を進めており，将来的に実装する予定である．

最後に，CWB は，現在，DNN 向けライブラリの開発を進めており，DNN の演算に特化したハードウェアを容易に利用できるようにしているところである．

3.　設計事例

本稿では，設計事例として 3 値ニューラルネットワーク (Ternary Weight Networks, 以下，TWN と呼ぶ)[7] の推論プログラムを用いる．TWN を用いる理由は，すでに 2 値ニューラルネットワーク (Binarized Neural Network, BNN と呼ぶ) [8] についての高速化についての検討は進んでいるためである．

3.1　3 値ニューラルネットワークの概要

TWN は，-1, 0, 1 の 3 つの値を weight に持つネットワークである [7]．BNN にくらべて，扱うことが可能な数が 1 種類多いため，BNN よりも精度を向上できるというメリットがある．しかしながら，浮動小数点数を用いた推論と比較して，精度が低下することが問題としてあげられる [7]．特に，ネットワーク規模が多くなるにつれて精度の低下は顕著になる [7]．

ここで，各レイヤの動作について説明する．畳み込みレイヤは，その名の通り，特徴量画像に対して畳み込み演算を行う．このとき，フィルタは一辺の長さをカーネルサイズとしており，ストライドの分だけスライドして積和演算を行う．畳み込みレイヤは，入力特徴量画像の枚数と出力特徴量画像の枚数がいずれも 1 枚以上である．各出力特徴量画像の演算のためには，同じ入力特徴量画像の activation を使い回すため，この箇所は並列演算可能であると考えられる．加えて，畳み込み演算が可能な量の入力特徴量画像を受け取った時点で演算を開始できる．ここで，畳み込み演算の際の weight はパッキングされた状態でメモリに格納されているが，ベースラインの時点ではその取得を毎回行っており，全く効率化されていない．

プーリングレイヤは，その種類によって演算がことなる．

- 最大値プーリングは，カーネル内の activation の内最大値を取る．
- 平均値プーリングは，カーネル内の activation の平均を取る．

本稿では，プーリングと書いた場合に最大値プーリングを指すこととする．プーリングレイヤも畳み込み演算と同様の並列化を施すことが出来る可能性がある．

FC レイヤは，すべての入力 activation 使って，各出力の activation を計算する．そのため，すべての入力特徴量画像が集まるまでは演算を開始できず，並列化を施すことが困難である．

3.2　対象ソースコード

対象ソースコードについて，A2C のために作成した，特に高速化手法を適用していない C ソースコードをベースラインとする．本ソースコードのネットワーク構造は，

Vol.2019-ARC-235 No.38
Vol.2019-SLDM-187 No.38
Vol.2019-EMB-50 No.38
2019/3/18

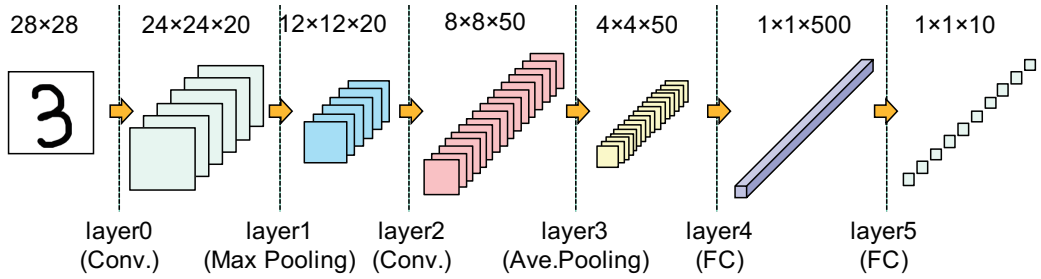


図 4　LeNet の構成

LeNet[9] である．LeNet のネットワーク構造を，図 4 に示す．

ベースラインのソースコードでは，畳み込みレイヤが 6 重ループ (外から順に，出力特徴量画像の枚数，入力特徴量画像の高さ，入力特徴量画像の幅，入力特徴量画像の枚数，カーネルサイズの高さ，カーネルサイズの幅)，プーリングレイヤが 5 重ループ (外から順に，出力特徴量画像の枚数，入力特徴量画像の幅，入力特徴量画像の枚数，カーネルサイズの高さ，カーネルサイズの幅) で構成されている．また，FC (Fully Connected) レイヤは 2 重ループ (外から順に，出力特徴量画像の数，入力特徴量画像の数) で構成されている．

また，各入出力特徴量画像は内部メモリに保存され，weight および bias は定数配列として宣言している．各 activation は整数値である．

各レイヤ間は，BC チャネルによって接続されており，前段のレイヤが後段のレイヤを起動する形で，レイヤ毎に推論処理が進行していく．最初のレイヤ (layer0) は，ソフトウェア側から起動される．

また，スループットの評価を行うため，6 枚の推論を行うようにしている．ベースラインでは，1 枚目の推論が終わってから 2 枚目の推論が始めるようにしている．

4. 高速化のための設計

本章では，ベースラインに加えた変更として，以下の 5 点について説明する．

(1) レイヤレベルのパイプライン化
(2) レイヤ間通信の FIFO 化
(3) layer2 のプロセス多重化
(4) layer2 のフィルタ処理の効率化

これらの方法について，ベースラインの C ソースコードに対して，レイヤレベルのパイプライン化を適用し，レイヤレベルのパイプライン化が適用された C ソースコードに対して，レイヤ間通信の FIFO 化を適用した．加えて，レイヤ間通信の FIFO 化が適用された C ソースコードに対して畳み込み演算に対するシフトレジスタを適用しレイヤ間通信の FIFO 化を行った C ソースコードに対して，layer2 のプロセス多重化および layer2 のフィルタ処理の効率化を行った．

表 1　使用した FPGA の概要

Logic Elements [LEs]	228,000
Embedded Memory [Kbits]	17,133
Clock [MHz]	50

また，本稿において使用した FPGA の概要は，表 1 に示すとおりであり，開発環境は以下の通りである．

- HLS ツール: CyberWorkBench 6.1 Enterprise
- 論理合成ツール: Quartus II 18.1
- コシュミレーション環境: Questa Sim 10.7c

本稿では性能評価にあたってスループット [fps] と，回路面積 [LEs] を用いる．回路面積においては，NiosII の面積 (8,115 [LEs]) を除いたものを記載している．スループットについては，コシュミレーションの結果から算出している．具体的には，Questa Sim による波形出力における，layer0 の開始地点から layer5 の終了地点を測定区間とした．実機による測定結果ではない理由は，実機にて実行することができない手法が存在するためである．また，回路面積に関しては Quartus II のサマリから，組み合わせ論理回路の素子数 (Combinational ALUTs) を用いている．

なお，ベースラインのソースコードについて評価を行った結果は，図 7 の (1) の通りである．

4.1 レイヤレベルのパイプライン化

この手法については，先行研究 [4] においても取り組まれていることである．

2.1 において説明したとおり，現状ではソフトウェアが推論ハードウェアを起動している．1 枚目の推論が完了後，ソフトウェアが 2 枚目の推論を開始しているが，これでは，スループットが向上しない．

そこで，レイヤレベルのパイプライン化を行い，スループットの向上を目指す．すなわち，最初のレイヤである layer0 の処理の完了した後すぐに，次の画像に対する推論の開始を要求するように変更する．これによって，次の layer1 の処理が行われている間，layer0 の処理も実行される．

表 2　各レイヤの 1 枚入力時のレイテンシ

レイヤ名	レイヤ種別	レイテンシ [ms]
layer0	畳み込み	29.2
layer1	プーリング	1.04
layer2	畳み込み	192
layer3	プーリング	0.241
layer4	FC	40.0
layer5	FC	0.495

しかし，layer0 の出力と layer1 の入力は同一であるため，layer1 がすべての入力をメモリから読み込む前に，layer0 の出力によってメモリの値が上書きされては正しく演算できなくなる．そのため，特徴量画像の保存先をもう 2 つ用意し，layer0 は一方に書き込み，layer1 は他方から読み込む．これらの 2 つのメモリの役割を交互に入れ替えて使用することで現在読み込みを行っているレイヤの入力が，不正に上書きされないよう対策した．

本実装においては，図 7 の (2) のような評価となった．ベースラインと比べると，スループットは 1.29 倍となり，回路面積は 1.95 倍増加した，現在，6 レイヤ構成であるのに対して，パイプライン化前と比べてスループットの向上率が低い．その理由は，各レイヤのレイテンシのばらつきがあげられる．各レイヤの 1 枚入力時のレイテンシを表 2 に示す．この表をみると，layer2 が最もレイテンシが大きく，layer4，layer0 と合わせて桁違いのレイテンシとなっている．特に layer2 は，最小レイテンシである layer3 と比べて約 797 倍の処理時間を必要とする．この結果から，layer2 がボトルネックになっていることは明らかであり，layer2 の高速化が最も重要であると考えられる．

4.2　レイヤ間通信の FIFO 化

この手法についても，先行研究 [4] においても取り組まれていることである．

この手法は，大きく 2 点の変更が含まれる．

- レイヤ間の特徴量画像の授受を FIFO によって行う．
- layer0 の畳み込み演算をシフトレジスタによって行う．

まず，レイヤ間の通信の FIFO 化について述べる．これまで，特徴量画像はメモリに一旦書き込んでその後メモリから読み込むことで実現していた．これを FIFO に変更することで，特徴量画像のためのメモリアクセスを大幅に低減できる．しかし，前段の入力を複数回使い回す layer2 と layer4，layer5 は FIFO 通信にこの時点で変更できない．layer0 に関しても，出力特徴量画像の枚数分だけ入力特徴量画像を使い回すが，これについては，layer0 起動時に入力画像を出力特徴量画像の枚数分だけ繰り返し送ることで FIFO 通信を実現した．

layer0 を高速化するため，前レイヤからの入力を保存するラインバッファおよびフィルタ処理を行うシフトレジスタを実装した．これにより，layer0 の演算が高速化された．ボトルネックである layer2 も シフトレジスタの実装を行いたいが，この時点では大きな変更となるため，実施していない．

本実装においては，図 7 の (3) のような評価となった．パイプライン化した場合と比べると，スループットは 1.21 倍となり，回路面積は 1.23 倍増加した．やはり，ボトルネックである layer2 のレイテンシがあまり改善されていないこともあり，あまり高速化できていない結果となった．そのため，4.3 以降では，layer2 の高速化に注力して設計を検討する．

4.3　layer2 のプロセス多重化

layer2 は，出力特徴量画像の枚数が 50 枚であるため，layer2 のプロセスを 50 個用意し，それぞれのプロセスがそれぞれの出力特徴量画像を出力するようにした．各プロセスが，送信順序を守るよう，layer2 と layer3 の間には layer2 の出力の順序を整えるプロセスを配置した．これにより，演算自体はすべて終了していて，単に通信オーバヘッドのみが遅延の要因になる．

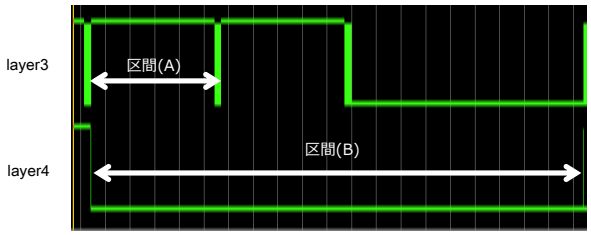
本実装においては，以下のような評価となった．本実装においては，図 7 の (4) のような評価となった．FIFO 化した場合と比べると，スループットは 4.00 倍となり，回路面積は 27.5 倍増加した．確かに，明らかな高速化を行うことができたが回路面積が StratixIV の上限を超えた．組み合わせ論理回路の素子数だけで言えば足りているが，それ以外の使用率を含めると，回路面積を超過する．

どの程度 layer2 を高速化できているか確認を行った．FIFO の実装では，1 つのレイヤがどの程度のレイテンシで実行したかを数値上で確認することが困難であるため，波形をもとに確認を行った．Questa Sim 上で確認した波形を図 5 に示す．波形について，区間 (A) は 1 枚の入力画像に対する layer0 から layer3 までのレイテンシであり，10.1[ms] である．また，区間 (B) は 1 枚の入力画像に対する layer4 のレイテンシであり，40.0[ms] である．明らかに区間 (A) は区間 (B) よりも短いことがわかる．このことから，ボトルネックのレイヤが layer2 から layer4 に入れ替わる程の高速化を施すことができている．そのため，さらなる高速化のためには，FC レイヤである layer4 を高速化する必要がある．

また，今後の課題として，プロセスの複製数を出力特徴量画像の枚数の半分にまで減らすことがあげられる．その変更を行う目的は，layer4 のレイテンシに対して later2 以前は 4 倍弱の余裕があるため，高速化のために回路面積を無駄に消費している可能性がある．すなわち，プロセス数が半減してもこのスループットを実現できる可能性があり，この場合は面積を超過せずに実装できる可能性もある．

以上のように，FIFO 化した場合よりは明らかに高速だが，実機で動作させることができないため，面積を低減し

Vol.2019-ARC-235 No.38
Vol.2019-SLDM-187 No.38
Vol.2019-EMB-50 No.38
2019/3/18



※ 電圧レベルが Low の間，処理を実行中である．

図 5 プロセス多重化の際の測定波形（抜粋）

て実機でも動作させられるような手法を 4.4 にて説明する．

4.4 layer2 のフィルタ処理の効率化

フィルタ処理の効率化について，説明する．4.3 では，プロセスを多重化したが，本節においては，プロセス中の一部のみ多重化することで効率化を図る．

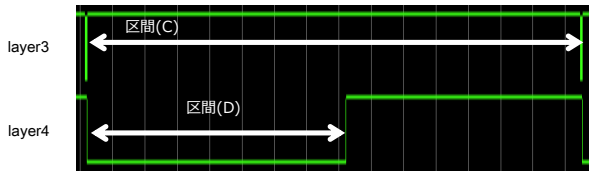
具体的には，4.2 において述べた，layer2 に対するシフトレジスタの実装を検討する．そこで，50 個のラインバッファとシフトレジスタを用意するように実装した．この結果，以下のような評価となった．本実装においては，図 7 の (5) のような評価となった．

しかしながら，ラインバッファやシフトレジスタは，出力特徴量画像の枚数分を用意する必要がない．そのため，ラインバッファやシフトレジスタを一元化し，積和演算の結果を保存する配列だけを 50 個用意した．この積和演算の結果は 50 並列で行われる．この結果，以下のような評価となった．本実装においては，図 7 の (5)’ のような評価となった．ラインバッファやシフトレジスタの一元化前よりもスループットが向上した．しかし，積和演算を並列化したことにより，回路面積が増大した．FIFO 化した場合と比べると，スループットは 2.01 倍となり，回路面積は 1.96 倍増加した．また，プロセス多重化を施した実装と比較すると，スループットは 0.502 倍となり，回路面積は 7.23 倍減少した．

FIFO 化した場合は，スループットの向上率と回路面積増加率が非常に近い値となっているが，プロセス多重化を施した場合は，スループットの向上率と回路面積増加率に比較的大きな差がある．そのため，FIFO 化した場合には，面積あたりのスループットで考えたときの効率が同程度であり，プロセス多重化よりも本実装のほうが，効率が良いと言える．

本実装は，回路面積がプロセス多重化を施した実装よりも明らかに小さく，充分に StratixIV 上で動作する．

4.3 と同様，どの程度 layer2 を高速化できているか確認を行った．Questa Sim 上で確認した波形を図 6 に示す．波形について，区間 (C) は 1 枚の入力画像に対する layer0 から layer3 までのレイテンシであり，76.4[ms] である．また，区間 (D) は 1 枚の入力画像に対する layer4 のレイテンシであり，40.0[ms] である．つまり，区間 (C) は区間 (D) よりも長いことがわかり，ボトルネックのレイヤが layer2 のまま変わっていないことがわかる．そのため，既述のプロセス多重化によって高速化することで，layer4 がボトルネックに変化する可能性がある．



※ 電圧レベルが Low の間，処理を実行中である．

図 6 フィルタ処理の効率化の際の測定波形（抜粋）

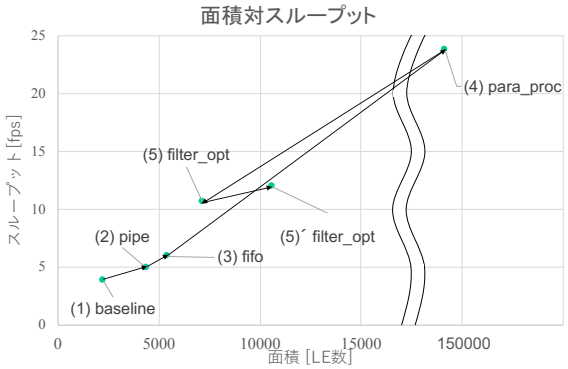


図 7 各実装の回路面積に対するスループットのまとめ

4.5 結果に対する考察と今後の方針

まず，それぞれの実装について，ここまでの結果を図 7 にまとめる．

FPGA ボードに実装可能な実装の中で最もスループットが高いものは，(5)’ の改善されたフィルタ処理の効率化である．また，最もスループットが小さいが，面積が小さいものは，(1) のベースラインの実装となった．(5)’ は，(1) と比べて，スループットは約 3.1 倍大きく，面積も約 4.7 倍大きいという結果になった．このことから，今回の推論プログラムに対しては，(1) のほうが，面積に対してスループットの効率が良いと言える．

また，(4) については，回路面積が非常に大きく，現状では FPGA ボード上で動作させられないが，ボトルネックのレイヤに対する多重化は TWN の推論プログラムに対しても効果的であるということを示唆している．また，回路面積が増大し過ぎる問題に対処するべく，部分的な多重化については検討を進める必要がある．

ここで，スループットを重視するか，回路面積を重視するかはターゲットボード次第であり，柔軟にこれらを選択できるよう，今後も効率の良い実装の検討進める予定である．効率の良い実装として，たとえば，先行研究 [4] では CWB による高速化を取り入れている．CWB におけるループアンローリングや，ループフォールディングは，高

Vol.2019-ARC-235 No.38
Vol.2019-SLDM-187 No.38
Vol.2019-EMB-50 No.38
2019/3/18

速化に寄与する可能性が非常に高いため，今後適用を検討する．

現在，TWN をベースに推論プログラムを作成していたが，activation を量子化していないため，演算があまり高速化できない問題がある．精度とのトレードオフになる可能性が高いが， activation も 3 値化すれば，BNN とほぼ同じ演算を行うことができるため，全体的な高速化を見込める．

本稿で提案した手法を含め，今後ハードウェアのパターンは非常に多くなることが予想される．そのため，高位合成や論理合成を行わずに性能見積もりができる方法についても今後検討を進め，EaS に統合することを検討する．

5. 関連研究

Xilinx 社は，HLS ツールの Vivado HLS と連携して動作する DNN 開発環境として，FINN [10] や FINN-R [11] を発表している．パラメータのビット精度を指定して学習を行うことができ，FPGA への実装までをサポートしている．また，Xilinx 社は，PYNQ (PYNQ: Python productivity for ZYNQ) [*1] を発表している．これは，ハードウェアプラットフォームであるが，開発環境も公開されており，Python 言語を利用して FPGA による DNN 推論することが可能である．また，Jupyter notebook によりブラウザ上から容易に FPGA による DNN 推論を実行できる．

また，Nakahara らは，GUINNESS (GUI based Neural Network Environment SyntheSizer) を開発した [12][13]．GUINNESS も Chainer による学習フロントエンドを提供しており，学習から高位合成用 C++ソースコードの生成までをサポートしている．生成される C++のソースコードは，SDSoC 向けであり，Linux 上で動作する．本稿では，Linux ではなく，小規模かつ低オーバヘッドな RTOS と推論器が連携して動作する．そのため，リアルタイム性が要求されるシステムや車載システムにおいては，我々のフレームワークの方が適している可能性がある．

6. おわりに

本稿では，我々が開発を進めている DNN 推論アクセラレータ生成フレームワークである，N3 フレームワークについて概説した．現在，実用が容易なフレームワークとなることを目指して，N3 フレームワークの開発を進めている．

また，効率の良いハードウェアの設計のための変更について TWN を題材に例示した．その結果，先行研究 [4] で示されている高速化手法の効果を TWN においても確認することができた．また，新たにプロセスを特徴量画像の枚数分だけ用意する手法や，シフトレジスタを複数用意することによる並列化を試み，これについてもその効果を確認することができた．今後，先行研究 [4] の手法と合わせて，HLS ツールによるハードウェアの最適化を適用し，さらに面積やスループットの面からより効率の良いハードウェアを，開発者が DNN の推論器を利用する際のコンテキストに合わせて選択できるよう，検討を進めていく．

謝辞 本研究の一部は，国立研究開発法人新エネルギー・産業技術総合開発機構（NEDO）の委託業務の結果得られたものです．

[*1] http://www.pynq.io/

参考文献

[1] Nurvitadhi, E., Sheffield, D., Sim, J., Mishra, A., Venkatesh, G. and Marr, D.: Accelerating Binarized Neural Networks: Comparison of FPGA, CPU, GPU, and ASIC, *FPT 2016*, IEEE, pp. 77–84 (2016).

[2] 本田晋也，冨山宏之，高田広章：システムレベル設計環境: SystemBuilder，電子情報通信学会論文誌， Vol. 88, No. 2, pp. 163–174 (2005).

[3] Wakabayashi, K.: CyberWorkBench: integrated design environment based on C-based behavior synthesis and verification, *VLSI Design, Automation and Test, 2005.(VLSI-TSA-DAT). 2005 IEEE VLSI-TSA International Symposium on*, IEEE, pp. 173–176 (2005).

[4] 岡本卓也，山本椋太，本田晋也：DNN の推論器向け高位合成用 C 記述の検討，VLSI 設計技術研究会，電子情報通信学会技術研究報告 (2018).

[5] Ando, Y., Ishida, Y., Honda, S., Takada, H. and Edahiro, M.: Automatic Synthesis of Inter-heterogeneous-processor Communication for Programmable System-on-chip, *IPSJ Transactions on System LSI Design Methodology*, Vol. 8, pp. 95–99 (2015).

[6] Ando, Y., Honda, S., Takada, H. and Edahiro, M.: System-level Design Method for Control Systems with Hardware-implemented Interrupt Handler, *Journal of Information Processing*, Vol. 23, No. 5, pp. 532–541 (2015).

[7] Li, F. and Liu, B.: Ternary Weight Networks, *arXiv preprint arXiv:1605.04711* (2016).

[8] Courbariaux, M., Hubara, I., Soudry, D., El-Yaniv, R. and Bengio, Y.: Binarized neural networks: Training deep neural networks with weights and activations constrained to+ 1 or-1 (2016).

[9] Gu, J., Wang, Z., Kuen, J., Ma, L., Shahroudy, A., Shuai, B., Liu, T., Wang, X., Wang, G., Cai, J. et al.: Recent advances in convolutional neural networks, *Pattern Recognition* (2017).

[10] Umuroglu, Y., Fraser, N. J., Gambardella, G., Blott, M., Leong, P., Jahre, M. and Vissers, K.: FINN: A Framework for Fast, Scalable Binarized Neural Network Inference, *FPGA 2017*, ACM, pp. 65–74 (2017).

[11] Blott, M., Preußer, T. B., Fraser, N. J., Gambardella, G., KennethO' brien, Umuroglu, Y., Leeser, M., Vissers, K. : FINN-R: An End-to-End Deep-Learning Framework for Fast Exploration of Quantized Neural Networks, *TRETS 2018*, Vol. 11, No. 3, p. 16 (2018).

[12] Yonekawa, H. and Nakahara, H.: On-chip memory based binarized convolutional deep neural network applying batch normalization free technique on an FPGA, *IPDPSW 2017*, IEEE, pp. 98–105 (2017).

[13] Nakahara, H., Fujii, T. and Sato, S.: A fully connected layer elimination for a binarizec convolutional neural network on an FPGA, *FPL 2017*, IEEE, pp. 1–4 (2017).