ディープラーニングアルゴリズムのハードウェア実装技術

Hardware Implementation Technology of Deep Learning Algorithms

山 野 龍 佑＊ Ryusuke YAMANO
戸 川 智 史＊ Satoshi TOGAWA
一 倉 孝 宏＊ Takahiro ICHIKURA
藤 澤 慎 也＊＊＊ Shinya FUJISAWA
内 野 浩 志＊ Hiroshi UCHINO
佐 藤 啓 介＊＊＊ Keisuke SATO
浅 井 佑 樹＊＊＊ Yuki ASAI
杉 田 克 行＊ Katsuyuki SUGITA
高前田 伸 也＊＊ Shinya TAKAMAEDA

要旨

Internet of Things（IoT）の普及に伴い，IoTデバイスに求められる機能はますます高度になっている。特に機械学習アルゴリズムを利用したソリューションの社会実装に対する期待が高まっており，スマートカメラを代表とする画像IoTデバイスへのDeep Learning（DL）推論機能の組み込みが急務である。

筆者らはField Programmable Gate Array（FPGA）を搭載した画像IoTデバイスにDL推論専用のハードウェアであるアクセラレーターを実装するため，学習済みDLモデルから回路を生成する高位合成コンパイラー「NNgen」[1), 2)]を共同開発してきた。NNgenを利用することにより，これまではハードウェア記述言語（HDL）を直接記述して開発を行った場合，数ヵ月から長ければ半年以上を要していた開発期間を数日にまで縮めることが可能となった。これにより，日進月歩のDLアルゴリズムの進化に追随しながら，ハードウェアの深い知識がないエンジニアでもFPGAを搭載した画像IoTデバイスにDLアクセラレーターを効率的に実装することが可能となった。さらに，NNgenはよりスピーディーな技術進化を目指してOpen Source Software（OSS）として公開しており，現在はコミュニティーによる開発が進められている。今後は世界中のユーザーから機能改善のアイデアや実装が提案されることが期待される。

本稿ではDLモデルから回路を生成するまでの一般的な流れを説明した後，NNgenのDLアクセラレーターの構成，DLアクセラレーターを生成する仕組み，及びハードウェアが専門でないエンジニアでもNNgenを利用できるようにするNNXTというツールを紹介する。

Abstract

With the spread of the Internet of Things (IoT), the functions required of IoT devices are becoming increasingly sophisticated. In particular, expectations are rising for the social implementation of solutions using machine learning algorithms. For those reasons incorporating Deep Learning (DL) inference functions into IoT imaging devices such as smart cameras are urgent need.

In order to implement a hardware accelerator dedicated to DL inference on an IoT imaging device equipped with a Field Programmable Gate Array (FPGA), the authors have jointly developed "NNgen" [1), 2)], a high-level synthesis compiler that generates a circuit from a trained DL model.

In comparison with directly writing a hardware description language (HDL), NNgen shortens the development period from several months or more than half a year to several days.

While keeping up with ever evolving DL algorithms, engineers can implement DL accelerators on IoT imaging devices equipped with FPGA, without deep knowledge of hardware. Furthermore, for speedy technological progress, NNgen has been released as Open Source Software (OSS) and is currently under development by the community. In the future, we expect that users all over the world will propose ideas and implementations for function improvement.

In this article, we first explain the general flow to generate DL inference circuit. Then we introduce the NNgen internal architecture and mechanism, and the tool NNXT for non-hardware engineers to use NNgen.

＊IoTサービス開発統括部　アーキテクチャ開発部
＊＊東京大学
＊＊＊IoTサービス開発統括部　エッジコントローラ開発部

1　はじめに

画像IoTデバイス上でのDL推論機能を実装するため，コニカミノルタはCPU，GPU，Application Specific Integrated Circuit（ASIC）はもちろんのことながらFPGAを活用したDL推論高速化技術の蓄積を進めてきた。CPUやGPUと比較して，FPGAやASICは消費電力当たりの演算性能が高いデバイスとして知られている[3]。また，機械学習を利用したIoTサービス開発／運用には学習データの収集やモデルの学習，更新等，一筋縄ではいかないノウハウが必要であるが，ASICはProof of Concept (PoC) やサービス運用時にアルゴリズムの改良やDLモデルの更新に対応することが難しい。FPGAであれば回路を変更することが可能なため，サービス開発／運用のフェーズを問わず適時改良を加えることが可能である。

しかしながら，アプリケーション実装にはソフトウェアのみならずHDLをはじめとする論理回路設計の知識が要求されるため，FPGAを利用したDL推論機能の実装は容易ではない。また，DLアルゴリズムの進歩は目覚ましく，サービスを継続する限りDLモデルの更新は必ず発生する。このような事情から人の手でHDLを記述して開発を行うのは工数的に現実的ではなく，高位合成コンパイラーを利用してHDLを自動生成することが一般的になりつつある。C/C++高位合成コンパイラーを利用することによりソフトウェアとしてアルゴリズムを開発し，C/C++ソースコードからHDLを生成することも可能となっているが，DLの研究開発分野においては，ソフトウェアではなくDLモデルの記述により成果が共有されるケースが多い。また，多くの場合Pythonで開発されており，C/C++高位合成コンパイラーを利用する場合，DLモデルを改めてC/C++表現として実装する工数が別途必要となるといった特有の問題がある。

上記の問題を解決するため，DLモデルからHDLを自動生成する高位合成ツール「NNgen」[1)2)]を共同開発してきた。Fig. 1は，NNgenを利用することにより，C/C++高位合成コンパイラーによる開発と比べて大幅に開発期間を短縮できることを示している。DLモデルからDLアクセラレーターを生成するツールはほかにも実用化されて一般的に利用可能になりつつあるが，NNgenでは回路規模がDLモデルの規模及び演算の並列度によって決定されるため，要求性能の制約及びハードウェアリソースと処理時間のトレードオフ関係から最適なアーキテクチャを探索することができる。また，回路を生成するツールの中にはベンダーロックインしているものもあり，その点NNgenは特定のFPGAベンダーによらず幅広いデバイスをターゲットにDLアクセラレーターを生成することができる。

しかしながら，研究開発者がNNgenを利用してDLアプリケーションを構築するためにはモデルの最適化や量子化，デバイスドライバー開発等の課題が残る。そこでコニカミノルタはDLモデルの軽量化，量子化，及びNNgenによるHDL生成までを自動化し，デバイスドライバー，アプリケーション開発のためのライブラリーをまとめたNNXTというツールを開発している。

本稿では，DLモデルからHDLを生成するフローとして，モデル最適化，量子化について説明した後，NNgenの全体構成，NNgenによるDLモデル構築方法，高速化の並列演算モデルを説明する。次にアプリケーションとして実装するためのデバイスドライバー及びライブラリーについて説明する。最後にOSS開発について今後の期待を述べ，まとめとする。

2　ハードウェア実装までの流れ

Fig. 2にハードウェア実装までの流れを示す。DLモデルによって，生成されるハードウェアやパラメーターに違いがあるものの，基本的なワークフローを図のように一般化することができる。コニカミノルタではビジネス適用へのTurn Around Time（TAT）短縮のため，これらの各工程を自動化するためのツール「NNXT」を開発した。NNXTはCaffe[4)]及びONNX[5)]フォーマットの学習済みモデルを入力として量子化したパラメーターとDLアクセラレーターのFPGA回路データ（bitstream）を自動的に生成する。以降の節で各工程について説明する。NNXTについては5章で説明する。

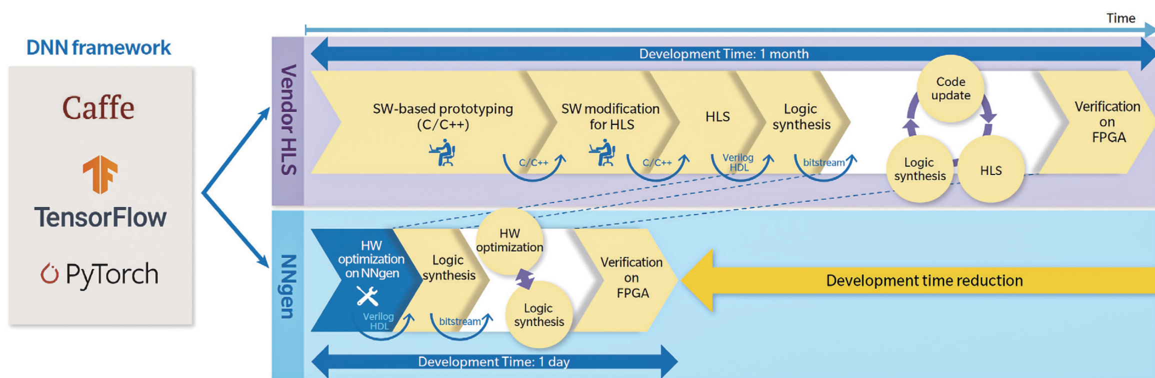

Fig. 1 Development flow comparison with vender HLS (High Level Synthesis).
By using NNgen, the development period can be greatly shortened compared to development using a C/C++ high-level synthesis compiler.

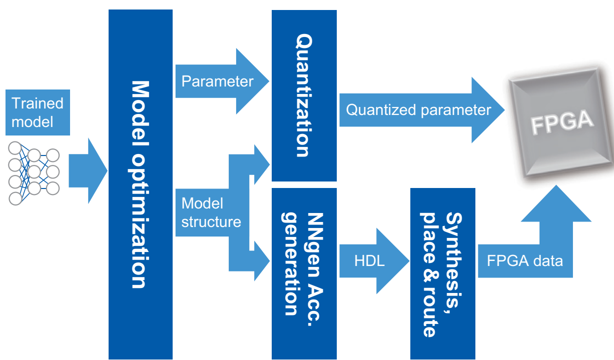

Fig. 2 Hardware implementation process.
We have developed the tool "NNXT" to automate each of these processes in order to shorten the Turn Around Time (TAT) for business applications.

2. 1　モデル最適化

モデル最適化の工程では，DLモデルを入力として2つの処理を行い，最適化したDLモデル構造とパラメーターを生成する。1つ目はモデル内にBatch Normalization層が含まれる場合にこの層の計算をConvolution層に統合する。この操作は演算回数削減による推論高速化と固定小数点演算の丸め誤差を抑制している。2つ目は入力データの値域が±1になるよう，値域に合わせた正規化処理を追加する。次節で説明する量子化処理では，小数点位置を考慮することなく計算できるように，計算単位毎の入力及び出力も値域が±1となるようにしている。そのため，最初の計算単位への入力となる入力データの値域を±1にするこの処理を行っている。

2. 2　量子化

DLアルゴリズム開発において学習時は浮動小数点演算を用いることが一般的である。組み込み機器への実装では推論時のパラメーターを量子化し整数演算に置き換える。これにより，パラメーターサイズの削減によるデータ転送の高速化，通信量削減による省電力化及び演算器を構成するリソース削減による並列度の向上を図っている。量子化工程では，最適化したDLモデルの構造とパラメーターを入力とした量子化処理を行い量子化されたパラメーターを生成する。量子化精度は精度重視の16bitと処理速度重視の8bitの2種類から選択可能である。量子化は一般的にPost Training Quantizationと呼ばれる方式で行う。アルゴリズムはコニカミノルタが独自開発したものを使用している。画像分類，物体認識及び関節点推定のモデルにおいて，16bitでは浮動小数点演算と同等の精度を確認している。8bitでは多少の劣化はあるものの，他社同等以上の精度を達成している。

2. 3　NNgenによるHDL生成

NNgenによるHDL生成の工程では，最適化したDLモデル構造を入力として，HDLを生成するPythonのソースコードを出力する。このプログラムには演算器の並列度やメモリーサイズなどHDL生成時に制約を加えるためのパラメーターが埋め込まれている。ユーザーは生成されたプログラムを実行することによりアプリケーションに最適なDLアクセラレーターのHDLを生成することができる。NNgenの詳細は次章で説明する。

2. 4　論理合成と配置配線

論理合成と配置配線の工程では，NNgenを用いて生成したHDLを入力として，bitstreamを生成する。bitstreamは，FPGAベンダーより提供されるツールを用いて，論理合成（Synthesis）と配置配線（Place & Route）を実行することで生成する。論理合成と配置配線の設定や実行方法，使用するFPGAのベンダーや種類及びFPGAが搭載されている環境によって異なる。それら全てに対応することは困難であるため，NNXTでは開発環境の構築やアプリケーション開発が比較的容易なXilinx社のZynq UltraScale+が使用されている以下のFPGAボードに対応している。

対応FPGAボード：
●Xilinx社 ZCU102, ZCU104
●Avnet社 Ultra96, Ultra96v2

3　NNgenについて

Fig. 3 にNNgenの概要図を示す。NNgenは基本機能としてONNXモデルから内部でDLモデルを構築するApplication Programming Interface（API）と，直接DLモデルを構築するAPIを提供している。

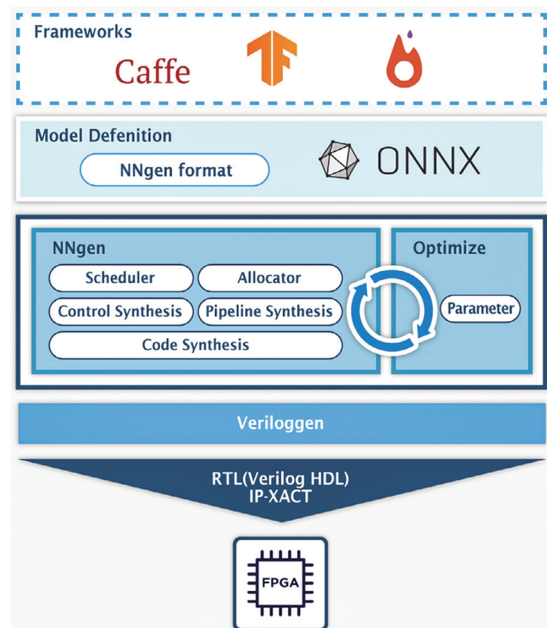

Fig. 3 NNgen overview.
Since NNgen can output the generated HDL as an IP-XACT format IP package, it can be easily incorporated into the system by utilizing the interconnect automatic wiring function and automatic address mapping function that are installed as standard in recent FPGA development tools.

コニカミノルタでは2章で説明したモデル最適化及び量子化の処理を独自で開発しているため，後者のAPIを利用して最適化後のDLモデルから得られる情報を用いた独自のDLモデル構築機能を開発している。

NNgenは生成するHDLをIP-XACT形式のIPパッケージとして出力することが可能であるため，最近のFPGA開発ツールに標準搭載されているインターコネクト自動配線機能や自動アドレスマッピング機能を活用して容易にシステムに組み込むことが可能である。

3. 1　NNgenの構成

NNgenはVeriloggen[6) 7)]というVerilog HDLのメタプログラミングライブラリーを用いて構築している。Veriloggenを用いることにより，Verilog HDLをPython上のdomain-specific language（DSL）として表現することが可能である。また，VeriloggenはVerilog構文木の表現のみならずPythonの持つ表現力を生かしたより高位な表現で，Finite State Machine（FSM）やデータフローを実装可能なプログラミングパラダイムを導入している。特にNNgenでは，データフロー演算をPythonの式として記述し，演算パイプラインとしてハードウェアを構築することが可能なStreamモジュールがある。さらにStreamモジュールによって構築された各演算器及びメモリー，バス転送の調停をPythonの関数として記述し，コントローラーを構築することが可能なThreadモジュールを用いてDL推論に必要な畳み込み等の各種演算ユニットを構築している。

3. 2　高速化の並列演算モデル

Fig. 4にNNgenでの3×3畳み込み層の並列演算モデルを示す。図中（a）は並列度の指定を行わない場合の3×3畳み込み演算器の構成を示している。図からわかるようにカーネルはループアンローリングされ，常に並列で積和演算を行う構成となっている。カーネルをアンローリングして並列演算を行うために9-port RAMが必要となる。NNgenではFPGAのブロックRAMをダブルバッファリング付きの9-Bank RAMとして構築する。NNgenでは画像の入力チャネル（ich），高さ（height），幅（width）と出力チャネル（och）に対して演算の並列度を上げることができる。

各ループの並列演算はList 1の疑似コードで表現できる。各並列度の指定によりストリーム演算器及びRAMの構成が変化する。図中では各並列度を変更した場合の回路の増分は赤色で示す。par_col指定して並列度を上げる場合，図中（b）の構成となり，畳み込み用のストリーム演算器が内部で2ストリームになるよう複製される。それに伴い，och出力用のRAMも並列でデータを受け取れるように複製される。

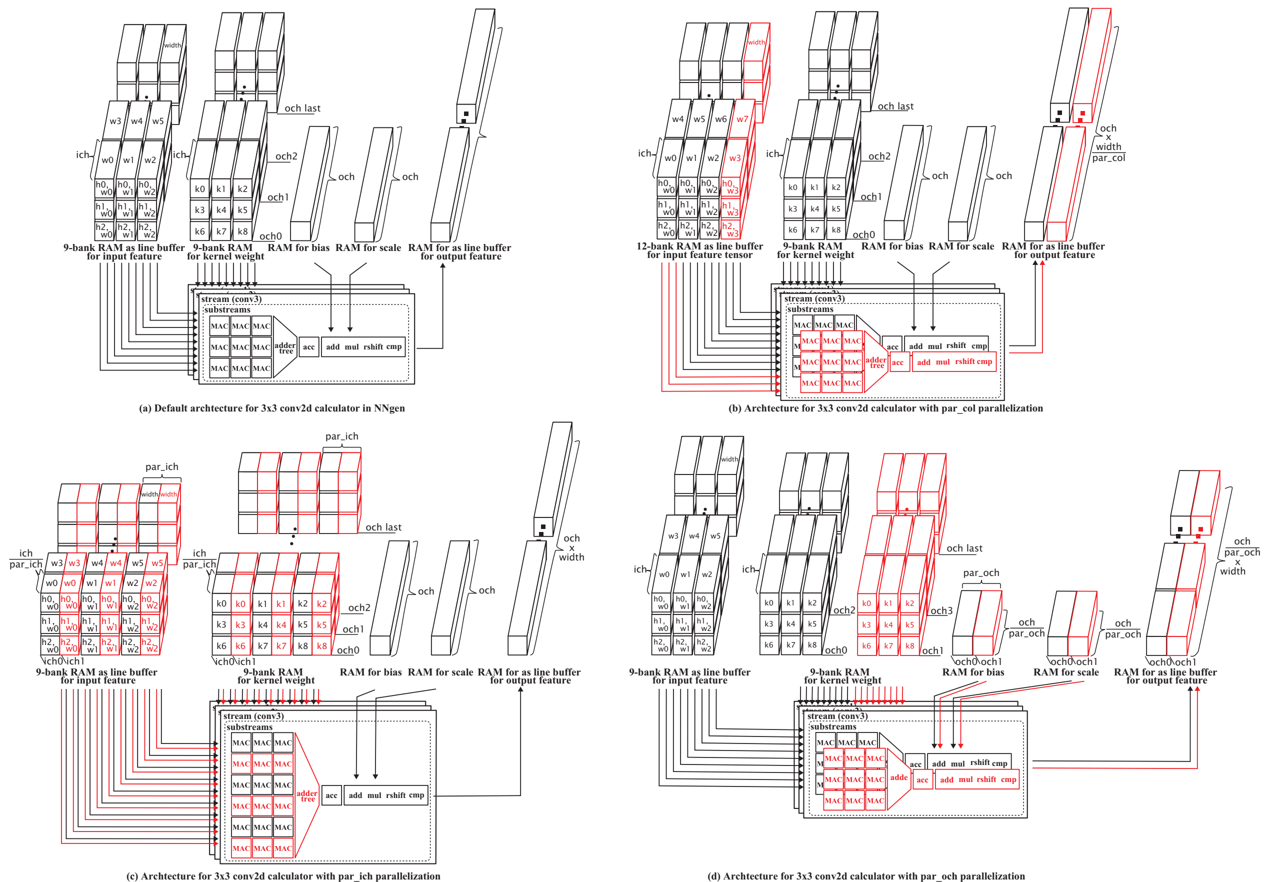

Fig. 4 Parallel computing model.
The 3×3 convolutional layer parallel computing model. The configuration of the stream calculator and RAM changes depending on the designation of the degree of parallelism. In the figure, the increment in the circuit when the degree of parallelism is changed is shown in red.

par_rowを指定した場合はpar_colではwidth方向に増えたインデックスがheight方向に増えるのみであるため説明は省略する。par_ichを指定して並列度を上げる場合，図中（c）の構成となる。9-Bank RAMはich方向に出力を増やすためにバス幅を増やし，複数chを同時に出力できる構成となる。そのうえで，ストリーム演算器が増えるのではなく，増えた分のichデータを処理するためにストリーム内部の積和演算器を増やしadder-treeも大きくして一度に畳み込めるデータ数を増やす。

List 1 Pseudocode representing convolution calculator for parallel compute.

```
0   conv2d(int* X, int* K, int* Y){
1   /* The following loops consume each clock cycles */
2    for(int o=0; o <    och/par_och; o=o+par_och)
3    for(int h=0; h < height/par_row; h=h+par_row)
4    for(int w=0; w <  width/par_col; w=w+par_col)
5    for(int i=0; i <    ich/par_ich; i=i+par_ich)
6   /* The following loops are unrooled
                       and consume just one cycle  */
7     for(int po=0; po<par_och; po++)
8     for(int ph=0; ph<par_row; ph++)
9     for(int pw=0; pw<par_col; pw++)
10    for(int pi=0; pi<par_ich; pi++)
11    for(int kh=0; kh<khsize;kh++) for(int kw=0; kw<kwsize;kw++
12     Y[height*width*(o + po) + width*(h + ph) + (w + pw)]
13     += X[widht*(h+ph+kh)+ich*(w+pw+kw)+(i+pi)]
14        * K[(o+po)*khsize*kwsize + kh*kwsize + kw] }
```

par_ochを指定して並列度を上げる場合，図中（d）の構成となる。ochを並列で出力するために重みデータが入っている9-Bank RAMとストリーム演算器が複製され，同時に各ochに対応したbias及びscale値を格納するRAMも複製される。

それぞれの並列度は複合して指定することが可能であり，並列数はpar_och × par_row × par_col × par_ich × khsize × kwsizeとして表現できる。演算対象データのサイズによって最適な並列度の組み合わせが変わってくるため，FPGAへの論理合成トライアルを行い各DLモデルに最適な構成を探索する。

3. 3　DLモデル実装方法

List 2に浅い畳み込みネットワークをNNgenで実装するサンプルコードを示す。サンプルコードについて前半1–22行と後半23–45行目に分けて説明する。

NNgenはDLモデルを構築するためにtensorflow1.0[8)]の関数を参考にしたDLモデル定義関数を用意している。コードの前半はそれらの関数を利用してDLモデルを構築している。1行目はNNgenのモジュール名をngに置き換えている。4–22行目でDLモデル構築を行っている。4行目は畳み込みネットワークへの入力データの定義である。6–7行目は畳み込み演算に必要な重みデータとbiasデータの定義である。9行目は4行目で定義した入力データと6–7行目で定義したデータを入力とした畳み込み演算の定義である。畳み込み演算の定義と同時にアクティベーション関数についても引数内でReLUを指定している。これらの引数の内容はtensorflow1.0のconv2d関数の引数とほぼ同じ意味を持つ。12–13行目は畳み込み演算の結果を入力としたmax pooling演算の定義である。15–22行目は6–13行目と同様に13行目max pooling演算の結果を入力とした畳み込み演算，そして再度max pooling演算を行うネットワークを定義している。ここまでのコードで畳み込み層とmax pooling層を2層ずつ持つ畳み込みネットワークの定義が完了する。最終的にDLアクセラレーターの出力として受け取りたい層を44–45行目のng.to_ipxact関数にリストとして与えることにより，HDLが生成される。サンプルコード内では21–22行目のmax poolingの演算結果を最終出力として受け取りたいので，変数pool2_layerをリストとして与えている。

List 2 Sample code to generate a DL accelerator for a shallow convolutional network.
It is possible to generate HDL for DL accelerator with code only up to the 22nd line. However, as NNgen accelerates each defined layer, it is possible to specify an option to create a circuit performs parallel calculation via the attribute member function.

```
1   import nngen as ng
2
3   # CNN definition
4   input_layer = ng.placeholder(ng.int8,
5                                shape=[1, 224, 224, 3])
6   conv1_filter = ng.variable(ng.int8, shape=[64, 3, 3, 3])
7   conv1_bias   = ng.variable(ng.int32, shape=[64])
8   conv1_layer  = ng.conv2d(
9       input_layer, conv1_filter, strides=[1, 1, 1, 1],
10      bias=conv1_bias, act_func=ng.relu,
11  )
12  pool1_layer  = ng.max_pool(
13      conv1_layer, ksize=[1, 2, 2, 1], strides=[1, 2, 2, 1])
14
15  conv2_filter = ng.variable(ng.int8, shape=[64, 3, 3, 64])
16  conv2_bias   = ng.variable(ng.int32, shape=[64])
17  conv2_layer  = ng.conv2d(
18      pool1_layer, conv2_filter, strides=[1, 1, 1, 1],
19      bias=conv2_bias, act_func=ng.relu,
20  )
21  pool2_layer  = ng.max_pool(
22      conv2_layer, ksize=[1, 2, 2, 1], strides=[1, 2, 2, 1])

23  # Attribute (Parallelism)
24  conv1_layer.attribute(par_ich=2, par_och=2)
25  pool1_layer.attribute(par=2)
26  conv2_layer.attribute(par_ich=2, par_och=2)
27  pool2_layer.attribute(par=2)
28
29  # Attribute (RAM size)
30  conv1_layer.attribute(
31      input_ram_size =1024,
32      filter_ram_size=1024,
33      bias_ram_size  =1024,
34      scale_ram_size =1024,
35      out_ram_size   =2048,
36  )
37  conv2_layer.attribute(
38      input_ram_size =1024,
39      filter_ram_size=1024,
40      bias_ram_size  =1024,
41      scale_ram_size =1024,
42      out_ram_size   =2048,
43  )
44  m = ng.to_ipxact([pool2_layer,],
45                   "accelerator_ip")
```

次に23–43行目の説明をする。22行目までのコードだけでDLアクセラレーターのHDLを生成することが可能であるが，NNgenは定義した各層に対して高速化を行うため，attributeメンバー関数を介して並列計算を行う回路を生成させるオプションを指定することができる。24行目では最初に定義した畳み込み層に並列数としてich方向に2並列，och方向に2並列で演算を行う回路を生成するように指定している。これにより最初の3×3畳み込み演算は3×3×2×2=36並列で積和演算を行うストリーム演算器として構築される。また，畳み込み層の様にローカルメモリーを多用する層についてはメモリーサイズを制限するサイズ指定オプションを記述することが可能である。これによりFPGAリソースで不足しがちなブロックRAMを制限した回路を生成することができる。この際，入力データあるいは出力データのラインメモリー，重みデータのメモリーのいずれかについて実際のデータが乗りきらないサイズになった場合，データ転送回数が増えるが，各メモリーの足りるサイズにまでochのループを自動で分割することができる。

4　性能評価

コニカミノルタはNNXT及びNNgenを利用して物体検出アルゴリズムであるyolov3 tiny[9]の高速化を実現した。組み込み向けプロセッサー，ARM Cortex A53 4コア 1.2GHz上で動作させる場合，約0.3fpsで動作する。FPGA上のDLアクセラレーターを利用した高速化の評価を次に示す。Table 1 論理合成条件およびNNgenの並列度の設定，Table 2 に量子化後のyolov3 tinyの精度及び速度性能，Table 3 にリソース使用量を示す。

Table 1 Evaluation conditions.
Logic synthesis conditions and settings for NNgen degree of parallelism.

Synthesis tool	Xilinx Vivado 2018.3
Target FPGA	Avnet Ultra96v2 (Xilinx ZU3EG)
Frequency	300 MHz
AXI4 bus width	64 bit
Data type	8 bit
Parallelization	288 (3x3 conv2d, par ich=8, par och=4)

Table 2 Inference performance evaluation.
Accuracy and speed performance of the yolov3 tiny algorithm after quantization. When operating on an ARM Cortex A53 4-core 1.2 GHz, it operates at about 0.3 fps (before quantization), while it achieves an operating speed of 18 fps (after quantization) on the FPGA.

DL model	yolov3 tiny
mAP-50	20.2
mAP-75	8.8
Inference speed	18 fps

Table 3 Resource usage.

Total LUT	54783
Total FF	102650
RAMB18	122
RAMB36	16
DSP48	320

5　NNXTについて

DLアクセラレーターを用いたシステム開発にはDLモデルからのハードウェア生成のみならず，量子化，デバイスドライバー開発，ライブラリー開発等の課題がある。コニカミノルタではそれらの課題を解決するための開発支援ツール「NNXT」を開発し，ハードウェアの深い知識がないエンジニアがDLを活用したアプリケーションを組み込むことができる環境を提供している。本章では，NNXTの構成要素について述べる。

5. 1　ソフトウェアスタック

DLアクセラレーターの制御にはレジスター操作・メモリー操作を伴うため，不適切な操作はシステムをクラッシュさせる。そこでハードウェア技術者でなくともDLアクセラレーターを気軽に活用できる環境を構築するために，DLアクセラレーターを制御するデバイスドライバー・ライブラリーを実装した。Fig. 5 に，NNXTが提供するソフトウェアスタックを示す。

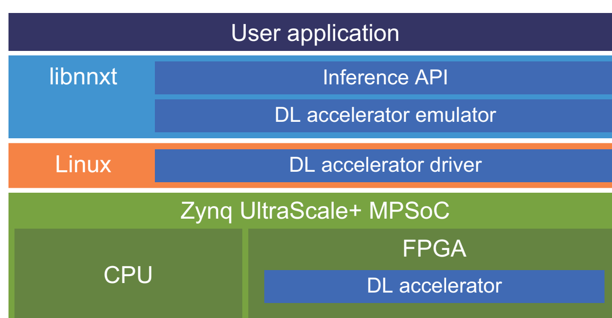

Fig. 5 Software stack for using DL accelerator.
As the library has a mode that emulates the calculation of the DL accelerator on the CPU, development is possible without an actual FPGA.

デバイスドライバー・ライブラリーに要求される操作は以下である。

●アルゴリズム差し替え
・bitstream書き換え
・パラメーター転送
●推論
・入力データ転送
・推論開始要求
・推論終了通知
・出力データ転送

NNXTはこれらの操作を隠蔽，抽象化したデバイスドライバー・ライブラリーを提供している。NNXTでは特に利用頻度の高い2つのFPGA製品シリーズ向けに対応している。ライブラリーにはDLアクセラレーターの演算をCPUでエミュレートするモードを備えており，FPGAの実機がなくとも開発が可能である。CPUによるエミュ

レーションは推論速度が遅いが，アルゴリズム開発および量子化演算の精度検証に利用することができる。このためFPGA実機を持たないアルゴリズム開発者やソフトウェア開発者が円滑に開発を進めるのに役立てている。

5.2 ハードウェア生成の自動化

NNXTは2章で述べた一連のハードウェア化作業を自動化する機能を備えている。モデルの違いにより生じる細かな作業の違いをツールで吸収することで，一連のプロセスを自動化した。

コニカミノルタでは早い時期からDL研究開発に取り組んできたため，初期からDLアルゴリズム開発で利用されているDLフレームワークCaffeを活用しており，ソフトウェア資源の蓄積がある。DLアルゴリズム開発においてOSS版のCaffeにはない演算機能やアルゴリズムについては，独自にCaffeを拡張することにより機能を実現してきた。NNXTでは派生したCaffe環境や今後の新たなDLフレームワークで構築されたDLモデルの処理を自動化フローに取り入れることを可能にするため，モデルの処理を行う環境のDocker[10]対応を実現した。また，NNXTの配布にもDockerを積極活用している。NNXT自体もDockerイメージとして配布を行うことで，統一したユーザー環境を少ない手間で構築することを可能にした。

なお, NNXT用のコンテナーとCaffe用のコンテナーの2つを協調動作させる必要がある。複数コンテナーの協調動作には一般的に「Docker Compose」というツールが利用される。しかしNNXTではユーザー環境に依存する項目が増えることを避けるために，「Docker Outside Of Docker」と呼ばれる，コンテナー内からさらに別のコンテナーを起動する手法を用いた。

5.3 Webアプリ化と社内展開

更にNNXTではコマンドラインツールの利用に不慣れな開発者のアプリケーション開発を容易にするため，NNXT環境をクラウド上に構築し，Webアプリケーションとしても提供している。Webアプリケーションは初心者向けの位置付けとして提供している。そのため，設定可能な項目を利用頻度の高いオプションに限定するなどUIを簡素化し，アプリケーション開発までのフローを可能な限り短縮することを可能にしている。コニカミノルタでは，このWebアプリケーションを用いたDLアクセラレーターに関する社内研修コースを開講しており，社内事業部向けの周知・活用を推進している。

6 まとめ

ハードウェアの世界においてはDLアルゴリズムの進化に追随しながらDLアクセラレーターを短期間で開発し，なおかつ継続して改良を行うというのは資金的，工数的な観点から現実的ではなかった。本稿ではNNgenとNNXTを利用することにより，短いTATでソフトウェアとハードウェアを縦断したアプリケーション開発を行うことが可能となることを示した。特にNNXTを利用することで，DLアルゴリズムをFPGAに実装するほとんどのフローを自動化することができ，DLアルゴリズムの研究者をはじめ，利用者の間口を広げることが可能となった。また，一般的な物体検出アルゴリズムに適応した場合の性能評価を行い，CPUへの実装と比べて大幅な性能向上が可能であることを示した。

NNgenはDLアクセラレーターの性能向上，及び新たなDLアルゴリズムへの応用に向けた改良を目指しOSSとして開発を続けている。今後もアカデミアと協力しながらNNgen開発の成果を社会に還元し，コニカミノルタとしてますますDLテクノロジーの社会実装を加速させていきたい。

●参考文献

1） Takamaeda-Yamazaki, Shinya. “NNgen: A Fully-Customizable Hardware Synthesis Compiler for Deep Neural Network”. https://github.com/NNgen/nngen.
2） 高前田 伸也，藤澤 慎也，藤崎 修一，“ディープニューラルネットワークのモデル特化ハードウェア合成コンパイラ”. 第2回機械学習工学研究会（MLSE夏合宿2019）論文集 pp.24–29
3） Ovtcharov, Kalin, et al. “Accelerating deep convolutional neural networks using specialized hardware.” Microsoft Research Whitepaper 2.11. (2015)
4） Jia, Yangqing et al. “Caffe: Convolutional Architecture for Fast Feature Embedding”. arXiv preprint arXiv: 1408. 5093, (2014). https://caffe.berkeleyvision.org
5） Bai, Junjie et al. (2019) “ONNX: Open Neural Network Exchange”. https://onnx.ai
6） Shinya Takamaeda-Yamazaki. “Pyverilog: A Python-based Hardware Design Processing Toolkit for Verilog HDL”, 11th International Symposium on Applied Reconfigurable Computing (ARC 2015) (Poster), Lecture Notes in Computer Science, Vol.9040/2015, pp.451–460. (2015)
7） Takamaeda-Yamazaki, Shinya. “Veriloggen: A Mixed-Paradigm Hardware Construction Framework”. https://github.com/PyHDI/veriloggen
8） Martín Abadi et al. “TensorFlow: Large-Scale Machine Learning on Heterogeneous Systems” (2015). https://www.tensorflow.org/
9） Redmon, Joseph and Farhadi, Ali. “YOLOv3: An Incremental Improvement” (2018). https://pjreddie.com/darknet/yolo/
10） Merkel, Dirk. “Docker: lightweight linux containers for consistent development and deployment” Linux journal, 239, p.2, (2014)