一般社団法人　電子情報通信学会
THE INSTITUTE OF ELECTRONICS,
INFORMATION AND COMMUNICATION ENGINEERS

信学技報
IEICE Technical Report
VLD2018-80,CPSY2018-90,RECONF2018-54(2019-01)

自動運転の実現に向けた画像処理アルゴリズムのFPGAによる実装

本田　紘規†　　ウェイカイジ†　　天野　英晴†

† 慶應義塾大学　〒 223–0061 神奈川県港北区日吉 3–14–1
E-mail:　†{kokihonda, wei, hunga}@am.ics.keio.ac.jp

あらまし　我々は FPT2018 で開かれた自動走行コンテストに PYNQ z-1 という FPGA ボードを用いて参加した。このコンテストは自動走行ロボットカーを FPGA を用いて自作し、ミニチュアの道路を完走することを目指したものである。そのためにはロボットカーに備え付けられたカメラから画像を読み込み、何らかの処理をした後、モータをコントロールしなけらばならない。本報告では PYNQ を用いることでカメラ画像の読み込みや、画像処理のためのプログラマブルロジック部への DMA による転送などが簡単に行えることを示す。普通、FPGA を用いてカメラ画像を読み取るためにはカメラ入力を画像に変換するモジュールが必要になる。しかし、PYNQ を使うことによりカメラを PYNQ に繋いだ後、コードを数行書くだけで画像を取得できる。さらに、PYNQ では DMA 転送するための煩雑な設定も必要ない。そして作成した IP はその IP への操作を Python でラップすることにより、メソッドのように使うことが出来る。また、2 種類の画像処理を FPGA に実装し、そのうちの一つである adaptive Threshold 処理では OpenCV と比べて、2.21 倍の性能向上を達成している。

キーワード　自動走行コンテスト、PYNQ、画像処理

Implementation of Image Processing Algorithm Aiming for Autonomous Car Using FPGA

Koki HONDA†, Kaijie WEI†, and Hideharu AMANO†

† Keio University　3–14–1 Hiyoshi, Kouhoku-ku, Yokohama, Kanagawa, 223–0061 Japan
E-mail:　†{kokihonda, wei, hunga}@am.ics.keio.ac.jp

1. はじめに

自動運転に対する社会の関心は高く、google やトヨタなどの大企業が自動運転技術に莫大な投資を行っている。そのような世の中の流れを受けて、FPGA による自動車の自動走行コンテストが FPT2018 で開催された。 [1] このコンテストの目的は、FPGA を使うことで画像処理の速度を自動走行に耐えうる基準まで高速化しようというものである。我々はこのコンテストに向けて PYNQ z-1 という FPGA ボードを使い、図 2 のような車を作成した。 [2] 本報告では、コンテストのための画像処理 IP の実装、評価と PYNQ の利点について述べる。

カメラベースの自動運転において道路の白線検出は重要である。白線を検出するための前処理として、画像のノイズ除去と二値化を行うことは一般的だ [3]。そこで我々はこれらを行うために median Blur と adaptive Threshold という処理を FPGA に実装した。

PYNQ z-1 ボードを図 3 に示す。PYNQ z-1 は PYNQ フレームワークを搭載した ZYNQ 7020 ボードである。PYNQ に関する説明は [4] に詳しい。PYNQ フレームワークは Python を使って FPGA を操作するためのフレームワークである。PYNQ フレームワークにより、ユーザーは Python のプログラムを Jupyter Notebooks に書き込むことで FPGA にアクセスすることができる。

Jupyter Notebooks とはコードを含めることのできるノートブックのようなものである。図 1 のようにウィンドウに Python コードを書くことでコードを実行することができる。さらに、PYNQ には PYNQ の機能を説明したノートブックが多く含まれており、ユーザーは PYNQ にアクセスするだけで PYNQ の機能を知ることができる。

一般に FPGA を使って設計することは難しいと言われている。我々は FPGA の設計の難しさは二つの部分があると考えている。一つは能率的な IP を設計することの難しさ、もう一つは IP を統合した後、それを実際に使えるようにするまでの難しさである。PYNQ は後者の難しさを和らげるものであ

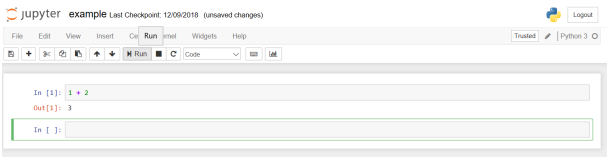

図 1　Jupyter Notebooks

る。PYNQ の生産性の高さを示す論文はいくつかある。 [5] は PYNQ を使って学生に授業で部分再構成可能な領域を持つビデオシステムを作らせることで PYNQ の使いやすさを示した。[6] はアプリケーション開発において PYNQ を使うことのインパクトを評価した。また PYNQ フレームワークを強化する研究もある。 [7] は PYNQ フレームワークに動的部分再構成の機能を加えた。本報告では自動走行コンテストにおいて PYNQ を使うことの利点を示したい。

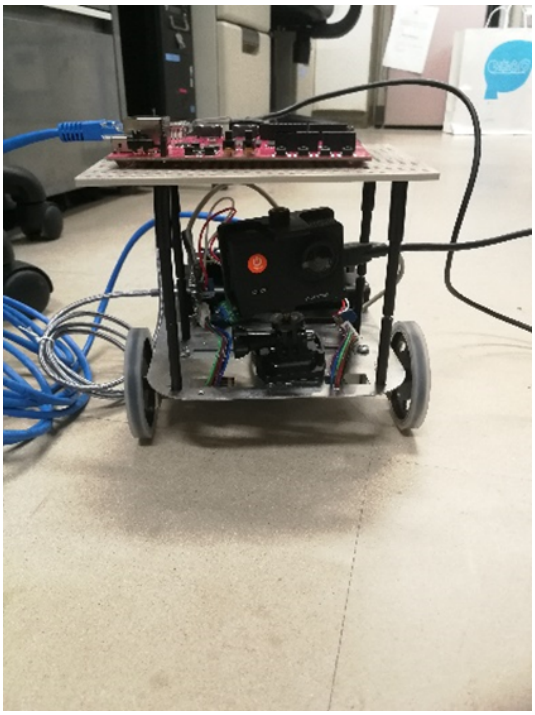

図 2　コンテストに向けて作成したロボットカー

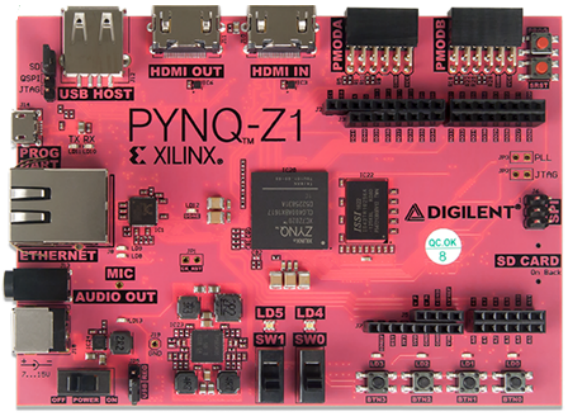

図 3　PYNQ z-1

2.　自動走行コンテスト

2.1　概　　要

ここではコンテストの概要を説明する。図 4 のようなミニチュアの道路上に FPGA により制御された自動車を走らせる。課題がいくつかあり、それをクリアするごとに得点が与えられる。合計得点が最も高いチームが優勝となる。課題は以下のようなものである。

- 道路をはみ出さずに走る
- 指示通りの道順で道路を走る
- 信号機を認識する
- 横断歩道を認識する
- 交差点の認識する
- 図 5 のような障害物の認識する
- 人（人形）の認識する

まずしなくてはならないのは道路の白線を検出し、そこから自動車の自己位置を算出することである。そこで我々は白線検出の前処理となるノイズ除去と二値化の処理を FPGA に実装した。

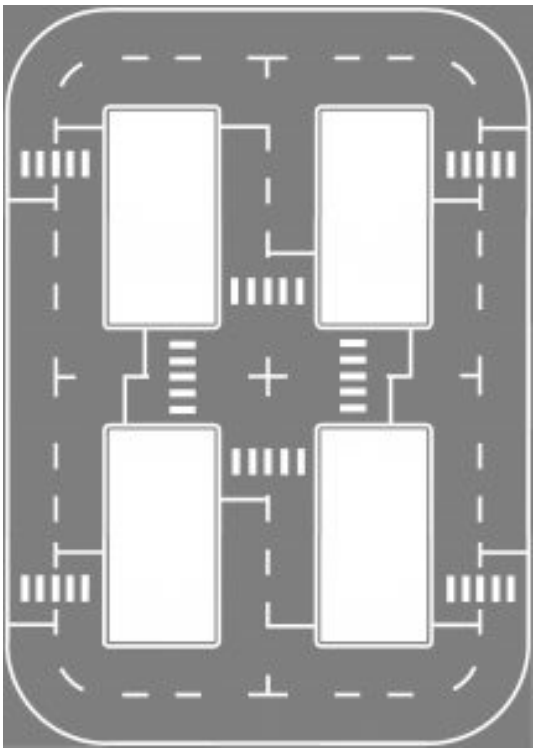

図 4　コンテストの道路

図 5　障　害　物

3.　PYNQ

PYNQ は Ubuntu Linux、Jupyter Notebooks, Python を組

み合わせたフレームワークである。ユーザーは Jupyter Notebooks を通じてボードにアクセスできる。普通、プロセッサから FPGA にアクセスするためには、Linux のカーネルドライバを使う。PYNQ フレームワークはこれらのドライバを Python API としてラップしている。このことにより Python コードによるボードへのアクセスが実現している。

PYNQ には Overlay という概念がある。Overlay とはソフトウェアでいうライブラリに相当するものである。Overlay は FPGA を設定する bitstream、そのハードウェアデザインの使用を容易にするための Python プログラム、使用可能な IP を決める Vivado デザイン tcl ファイルからなる。ユーザーは Overlay をダウンロードすることでそのハードウェアデザインを、Python を使って利用することができる。

4. 自動走行のためのハードウェアデザイン

PYNQ フレームワークには Base Overlay と呼ばれるのデザインがもともと含まれている。Base Overlay とは PYNQ の I/O が簡単に使えるように設計されたハードウェアデザインである。我々は Base Overlay を出発地点として設計を行った。

カメラベースの自動走行のためにはカメラから画像を受け取る必要がある。PYNQ の Base Overlay には HDMI 経由でカメラ画像を取り込む機能が含まれている。その機能を使うと、HDMI の設定を数行書けば、後は Jupyter Notebooks に 1 行書き込むだけで、画像を取得することができる。画像を取得した後は、何らかの処理をし、車の車輪のモータへの制御をする必要がある。我々の車のモータは PWM(pulse width modulation) で制御されるものなので、ハードウェアデザインに PWM モジュールを加えた。さらに、自動走行コンテストのために不要なものをハードウェアデザインから削除した。

下に Python コードによる自動走行プログラムの骨組みを示す。画像読み込みからモータへの制御までシンプルに書けていることが分かる。さらに、ハードウェアアクセラレータを Python のメソッドのように呼び出している点に注目してほしい。これはハードウェアのアクセスを Python のクラスでラップすることで実現している。

自動走行プログラムの骨組み

```
While True:
    # HDMI カメラからの読み込み
    frame = hdmi_in.readframe()
    # sw での処理
    out0 = cv2.cvtColor(frame,
                cv2.cv2.COLOR_RGB2GRAY)
    # hw での処理
    out1 = overlay.median_hw(out0)
    out2 = overlay.adaptive_hw(frame)
    # モータへの制御値を決める何らかの画像処理が続く

    #モータを制御する関数
    moter(control)
```

5. 画像処理 IP

我々は自動走行のための画像処理 IP を二つ設計した。median Blur と adaptive Threshold である。画像は DRAM から IP に転送し、IP から DRAM に書き戻すようにした。

PYNQ には DMA を使うためのライブラリがあり図 6 のように IP を DMA モジュールに繋げてさえいれば、簡単に DMA 転送を行うことができる。DMA 転送を行うためのコードを下に示す。

DMA 転送を行うコード

```
xlnk = Xlnk()
# DRAM の一続きのメモリ領域の確保
in_buffer = xlnk.cma_array(shape=(height, width),
                    dtype=np.uint8, cacheable=1)
out_buffer = xlnk.cma_array(shape=(height, width),
                    dtype=np.uint8, cacheable=1)

#  DMA の in_buffer からの読み込みと out_buffer への書き込みのスタート
dma.sendchannel.transfer(in_buffer)
dma.recvchannel.transfer(out_buffer)

# ハードウェアアクセラレータの起動
hw_accel.write(0x00,0x81)

# 完了を待つ
dma.sendchannel.wait()
dma.recvchannel.wait()
```

しかし、PYNQ は IP の設計そのものを容易にするわけではない。そこで我々は [8] を基に IP を作成した。ここには効率的な 2D フィルターの HLS C コードが載っていて、インターフェイスとそれに伴う変更のみでベースを作成することができた。画像処理関数のインターフェイスは [9] の下にあった axis2xfMat と xfMat2axis という関数を使ったので xfMat にした。下にトップ関数のコードを示す。

```
void adaptive_threshold_accel
(axis_t *src, axis_t *dst, int src_rows,
int src_cols, int dst_rows, int dst_cols) {

#pragma HLS INTERFACE axis port=src
#pragma HLS INTERFACE axis port=dst
#pragma HLS INTERFACE s_axilite port=src_rows
#pragma HLS INTERFACE s_axilite port=src_cols
#pragma HLS INTERFACE s_axilite port=dst_rows
#pragma HLS INTERFACE s_axilite port=dst_cols
#pragma HLS INTERFACE s_axilite port=return
```

```
xf::Mat<TYPE, HEIGHT, WIDTH, NPC1>
                    _src(src_rows, src_cols);
xf::Mat<TYPE, HEIGHT, WIDTH, NPC1>
                    _dst(dst_rows, dst_cols);
#pragma HLS stream variable=_src.data depth=150
#pragma HLS stream variable=_dst.data depth=150

#pragma HLS dataflow

axis2xfMat(_src, src, src_rows, src_cols);

adptive_threshold(_src, _dst);

xfMat2axis(_dst, dst, dst_rows, dst_cols);

}
```

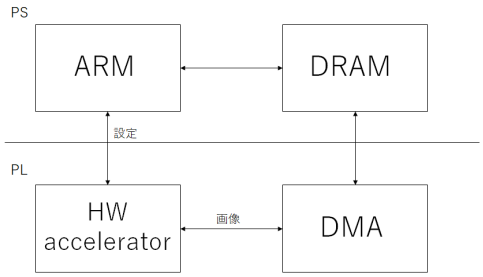

図 6　DMA 転送のためのハードウェアデザイン

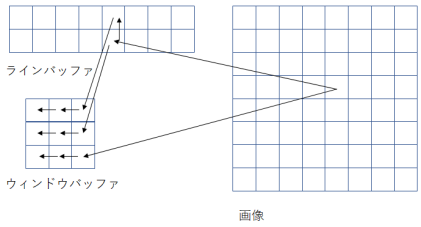

図 7　ラインバッファとウィンドウバッファの動作

5.1　median Blur

median Blur は画像のノイズ除去のために行われる。median Blur は画像に 2D フィルターを適用する。そしてフィルター中心のピクセルはそのフィルターのピクセルの中央値となるピクセルに置き替えられる。

2D フィルターの実現のためにラインバッファとウィンドウバッファを使用した。ラインバッファは 2D フィルターのために一時的にピクセルを蓄えておくバッファで、ウィンドウバッファは 2D フィルターの計算のために使うピクセルを保存しているバッファである。ラインバッファは BRAM、ウィンドウバッファは LUT で実装されている。ラインバッファとウィンドウバッファの動きを図 7 に示す。また中央値を求めるためのソートアルゴリズムは [10] に載っていた図 8 のハードウェアデザインを見ながら作成した。

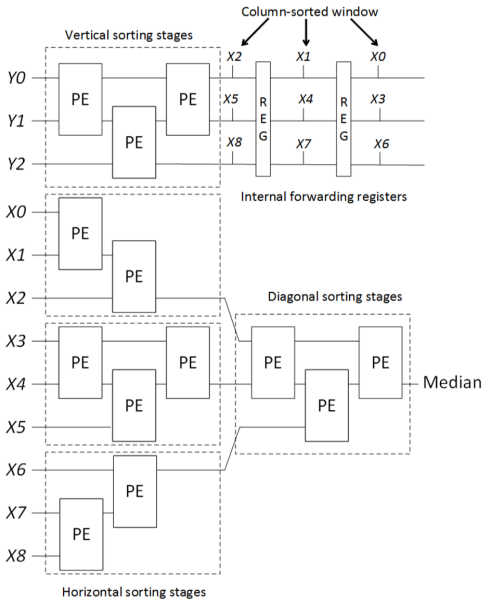

図 8　ソートデザイン [10]

5.2　adaptive Threshold

Adaptive threshold は二値化をする処理である。Adaptive threshold は近傍のピクセルをもとに閾値を決めるので、グローバルな閾値をもつ threshold と比べて光に対して少しロバストになる。Adaptive threshold もまた 2D フィルターを適用する。閾値の決め方はいくつかの種類があるが、ここではフィルター内のピクセルの平均を取りそれを閾値とするようにした。図 9 でフィルタ処理の動作の例を示す。

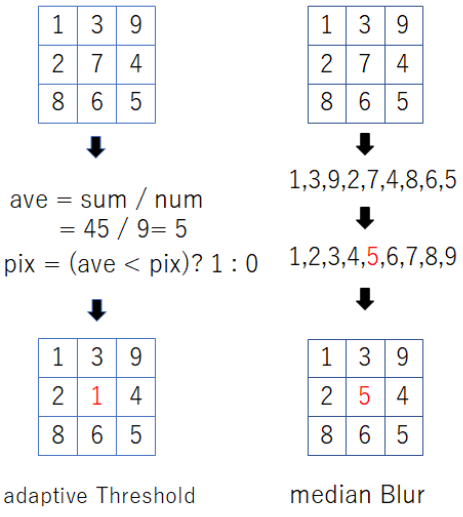

図 9　フィルタ処理の例

6. 評　　価

作成した画像処理 IP の性能を比較するために PYNQ の Jupyter Notebooks 上で、OpenCV、ハードウェア、ナイーブな Python の 3 種類の実行時間を測定した。実行時間測定には Jupyter Notebooks の timeit マジックを用いた。

6.1 セッティング

実験の環境を示す。使用した PYNQ ボードは Dual core の Cortex A9 プロセッサと、512MB の DDR3 メモリを備えている。プロセッサは 667MHz で動作する。FPGA 部分は Artix-7 ファミリーであり、アクセラレータは 100Mhz で動作する設定にした。処理する画像は 640x480 のグレースケールであり、フィルターのサイズは 3x3 である。IP は Vivado HLS 2017.4 を使って作成した。

6.2 画像処理 IP の評価

median Blur と adaptive Threshold の実行時間の表を表 1、表 2、表 3 に示す。我々は OpenCV をベースラインとして採用した。OpenCV での基礎的な関数は高度に最適化されており、その CPU で出せる最高性能に近いと考えて良いだろう。

ハードウェアでの処理は OpenCV での処理と比べて adaptive Threshold で 2.21 倍のスピードアップを達成した。

median Blur では 0.67 倍だった。これはアルゴリズム自体の効率性の差が原因だと考えられる。OpenCV の median Blur は隣り合ったフィルターの共通部分の並びを保持するなどの工夫がされている一方、我々のコードはただ 9 個のピクセルをソートして中央値を求めるという処理を繰り返しているだけだからである。

さらに、median Blur と adaptive Threshold を続けて処理した場合を考える。ハードウェアは 1 つに IP に統合した。OpenCV では 2 つ関数の処理時間を足した時間より遅くなったが、FPGA では実行時間はほとんど変わらなかった。FPGA の実行時間がほとんど変わらなかったのは、DMA の相対的なオーバーヘッドが減ったことと、Vivado の dataflow プラグマにより前の関数の実行をまたずに次の関数が実行できたことが原因だと考えられる。この結果は、処理をまとめればまとめるほど FPGA は画像処理を効率的に行えるということを示唆している。

IP を作る際、HLS コードの書き方により性能が大きく変化した。前に述べたように PYNQ は IP の設計そのものを容易にはしない。おそらく、本当はもっと高性能を出せるはずであり、高性能な IP を容易に生成するための技術も重要だろう。

さらに、自動走行コンテストに関して、実行時間がクリティカルな箇所は画像の座標変換と白線検出なのだが、どちらも、画像と同じくらいの大きさの記憶領域が必要となり、単純な実装ではローエンドな FPGA ボードに載せることが出来なかった。

表 1　median Blur においての OpenCV、ハードウェア、ナイーブな Python の比較

方法	時間 (ms)	スピードアップ
OpenCV	47.1	1.00x
ハードウェア (Ours)	60.4	0.67x
ナイーブな Python	23700	0.0060x

表 2　adaptive Threshold においての OpenCV、ハードウェア、ナイーブな Python の比較

方法	時間 (ms)	スピードアップ
OpenCV	15	1.00x
ハードウェア (Ours)	6.77	2.21x
ナイーブな Python	70000	0.00021x

表 3　median Blur と adaptive Threshold を続けて行った場合の OpenCV、ハードウェア、ナイーブな Python の比較

方法	時間 (ms)	スピードアップ
OpenCV	60.6	1.00x
ハードウェア (Ours)	60.3	1.00x
ナイーブな Python	57200	0.0011x

7. ま と め

本原稿では自動走行コンテストに向けた PYNQ ボードを使った画像処理 IP の作成を行った。画像処理 IP は OpenCV と比べて adaptive Threshold の処理で 2.21 倍の性能を達成した。また PYNQ を使うことによりカメラからの画像の取り込みや、作成した IP への DMA による転送が容易に行うことができた。さらに、Overlay をダウンロードすることで、ソフトウェアプログラムのようにハードウェアアクセラレータが利用できることを示した。このような Overlay が増えればソフトウェアライブラリのように気軽にハードウェアアクセラレータが利用できるになるだろう。

文　　献

[1] “FPT2018 FPGA Design Competition,” 2018. https://wwp.shizuoka.ac.jp/fpt-design-contest/

[2] W. Kaijie, K. Honda, and H. Amano, “Fpga design for autonomous vehicle driving using binarized neural networks,” 2018 International Conference on Field-Programmable Technology (FPT), pp.428–431, 2018.

[3] A. Borkar, M. Hayes, M.T. Smith, and S. Pankanti, “A layered approach to robust lane detection at night,” 2009 IEEE Workshop on Computational Intelligence in Vehicles and Vehicular Systems, pp.51–57, March 2009.

[4] “PYNQ-Introduction,” 2018. https://pynq.readthedocs.io/

[5] B. Hutchings and M. Wirthlin, “Rapid implementation of a partially reconfigurable video system with pynq,” 2017 27th International Conference on Field Programmable Logic and Applications (FPL), pp.1–8, Sep. 2017.

[6] A.G. Schmidt, G. Weisz, and M. French, “Evaluating rapid application development with python for heterogeneous processor-based fpgas,” 2017 IEEE 25th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM), pp.121–124, April 2017.

[7] B. Janen, P. Zimprich, and M. Hbner, “A dynamic partial reconfigurable overlay concept for pynq,” 2017 27th International Conference on Field Programmable Logic and Applications (FPL), pp.1–4, Sep. 2017.

[8] R. Kastner, J. Matai, and S. Neuendorffer, Parallel Programming for FPGAs, ArXiv e-print, May 2018.

[9] “PYNQ-DL,” 2018. https://github.com/Xilinx/PYNQ-DL

[10] A. Sanny and V.K. Prasanna, “Energy-efficient median filter on fpga,” 2013 International Conference on Reconfigurable Computing and FPGAs (ReConFig), pp.1–8, Dec. 2013.