第4回 AIエッジコンテスト レポート

チーム名 Vertical_Beach

lp6m, medalotte

1　開発フロー

DNN の HW アクセラレーションには，Xilinx 社から提供されている DPU（Deeplearning Processing Unit）コア [5] および統合開発環境である Vitis-AI を使用した．Vitis-AI は caffe，tensorflow 等の DNN フレームワークを用いて設計された DNN モデルを量子化し，DPU 向けにデプロイすることができる．[1]

2　DNN モデルの学習

2.1　DNN モデル

コンテストの課題であるセマンティックセグメンテーションを行う DNN モデルとして我々は resnet18-FPN を使用した．モデルは Xilinx 社から提供されるチュートリアル [4] に含まれるものを流用した．FPN（Feature Pyramid Network）[2] は，低解像だが意味的に強い (semantically strong) 特徴と高解像だが意味的に弱い (semantically weak) 特徴の両方を使用することで物体検出及び領域分割のタスクにおいて高い精度を挙げられることが知られている．図 1 に FPN の概要図を示す．

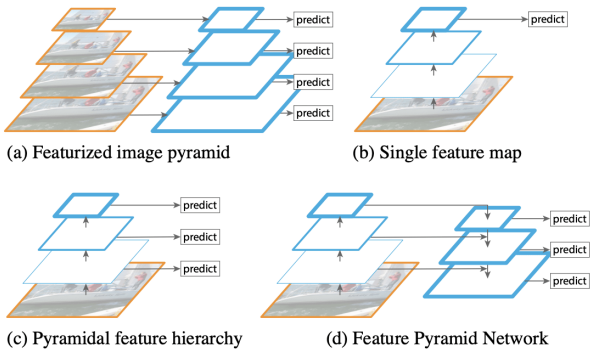


図 1: Feature Pyramid Network[2]

2.2　損失関数

参考にしたチュートリアルでは損失関数として SoftmaxWithCrossEntropy が使用されていた．コンテストで提供される学習画像を使用して学習を行ったが，テスト画像に対する mIoU スコアは 0.50 程度に留まり，処理速度部門における基準値である 0.60 を上回ることができなかった．そこで我々

[1] Vitis-AI v1.3 にて pytorch への一部対応が追加された．

は領域分割タスクにおいて精度を向上させる損失関数として提案されている Lovasz-Loss 関数 [1] を採用した．Lovasz-Loss 関数は，第 1 回 AI エッジコンテストのセグメンテーション部門において第 2 位のチームも使用していたことから [3]，精度向上に効果的であると考えた．Lovasz-Loss は予測領域と正解領域の IoU を指標とする Jaccard-Loss をさらに拡張したものであり，tensorflow と pytorch 向けに公式に実装が公開されている．流用したチュートリアルは Caffe を用いてモデルが定義されており，Caffe 上で Lovasz-Loss 関数を自前で実装するのは困難であった．モデルを pytorch に変換して pytorch 上で学習を行い，学習済みの重みを caffe 向けに変換することでこの問題を解消した．

2.3 学習時の解像度

推論を FPGA ボード上で高速に実行するには，入力画像サイズを小さくしても高い精度が出ることが望ましい．推論時と学習時の解像度が近いほうが精度が向上するのか，あるいは学習時に高解像度の情報を与えるほうが学習精度が向上するのかを検証した．学習時の入力解像度について以下の 2 つの方法を比較した．

- 512*1024 に Resize → 0.7 倍から 1.5 倍に Random Scaling → (256*256) に Random Cropping
- 400*800 に Resize → 0.7 倍から 1.2 倍に Random Scaling → (256*256) に Random Cropping

前者では与えられる解像度が (358,716)〜(768,1536) と比較的大きく，後者では (280,560)〜(480,960) と推論時に使用する解像度に近く小さい．比較結果を表 2 に示す．今回は後者の方法，すなわち推論時と学習時の解像度が近いほうが精度が僅かに向上する結果となった．

表 1: 学習時の入力解像度による精度の比較

	Low Resolution	High Resolution
320*640	0.6014	0.5829
352*704	0.6081	0.5972
384*768	0.6121	0.6084

2.4 学習結果

コンテストで提供される学習用画像 2243 枚の 8 割を学習用，2 割を検証用に分割し学習を行った．学習した際の学習曲線を図 2 に示す．解像度 480*960 の画像に対して GPU 上で推論を行った結果，

表 2: 損失関数による mIoU スコア比較

SoftmaxWithCrossEntropy	0.5093
Lovasz Loss	0.6224

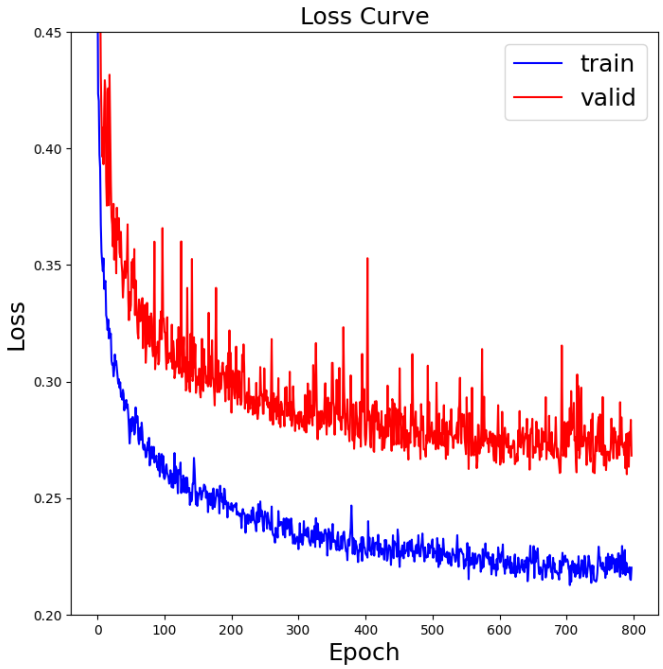


図 2: Lovasz-Loss Curve

mIoU スコアは 0.62 となり，Lovasz-Loss 関数を使用したことで精度が大幅に向上した．

3　ハードウェア最適化

Xilinx 社から提供される DPU IP コアは，画素や入出力チャネルに対する並列数が異なる複数の種類の IP コアが提供されている．より並列数の高い IP コアを使用することで処理性能が向上するが，回路規模および消費電力が増加する．また，DPU コアの一部のレイヤーのサポートを無効にすることでリソース使用率を低減することができる．

コンテストの評価ボードである Ultra96V2 に搭載可能な DPU コアとして B2304 を採用した．デフォルトで有効になっている DepthWiseConvolution および Pool Average のレイヤーは今回設計したモデルでは必要ないため無効にした．

さらに，DPU の動作周波数を高めることにより，DPU における推論実行時間を短縮することができる．論理合成のストラテジを Flow_AreaOptimized_high，配置配線のストラテジを performance_ExtraTimingOpt に変更することで動作周波数を 150MHz/300MHz から 200MHz/400MHz[2]に高めてもタイミング制約を満たし，FPGA ビットストリームの生成を行うことができた．

表 3 に動作周波数と入力画像サイズごとの DPU における推論実行時間・およびスコアを示す．なお，表に示すスコアは 2.3 章における前者の設定で学習を行った際のスコアであり，最終評価に使用した重みによるスコアとは異なる．DPU の推論時間は入力画像サイズに概ね比例し，動作周波数を向上させることによって推論処理が約 1.2 倍高速化されることがわかる．

表 3: 各入力画像サイズ・動作周波数における推論時間とスコア

Image Size	DPUTask [ms]		Score
	150/300MHz	200/400MHz	
256*512	35	30	0.539
320*640	60	55	0.579
352*704	72	65	0.593
384*768	81	70	0.608
480*960	128	110	0.616

4　ソフトウェア最適化

本コンテストでは 1216*1936 の車載カメラ画像の Semantic Segmentation を行うことが課題として設定されている．具体的には，入力画像の各ピクセルに対して 4 カテゴリ (乗用車，歩行者，信号，車道・駐車場) の分類を行った結果を出力する推論アプリケーションの実装が求められる．

我々の実装では，入力画像の縮小 (1216*1936 → 320*640) ，および，正規化を行った後に DPU による推論を行う．また，DPU による推論の結果は，ラベル付け，および，入力画像と同サイズへの拡大 (320*640 → 1216*1936) を行うことで出力画像となる．ところで，本コンテストのルールでは，入力画像のメモリ上へのロード，および，出力画像のファイル出力は推論処理時間に含めない．よって，上記の前処理 (縮小・正規化)，推論 (DPU 実行)，後処理 (ラベル付け・拡大) の 3 つの処理が推論処理時間計測の対象となる．

我々はこれらの処理のマルチスレッド実装を行った．DPU による推論の実行中に，次の画像の前

[2]DPU コアはベース周波数に加えてその 2 倍の周波数のクロックを DSP に接続するため，動作周波数はこのような表記とした．

処理，および，前の画像の後処理を行うことで，シーケンシャルに処理する場合に比べて大幅な高速化が見込まれる．

本章では，はじめに上記の処理をシングルスレッド実装する場合についての説明を行う．その後，マルチスレッド実装についての説明を行う．

4.1　シングルスレッド実装

図 3 に我々の推論処理をシングルスレッドで実装する場合のフローチャートを示す．

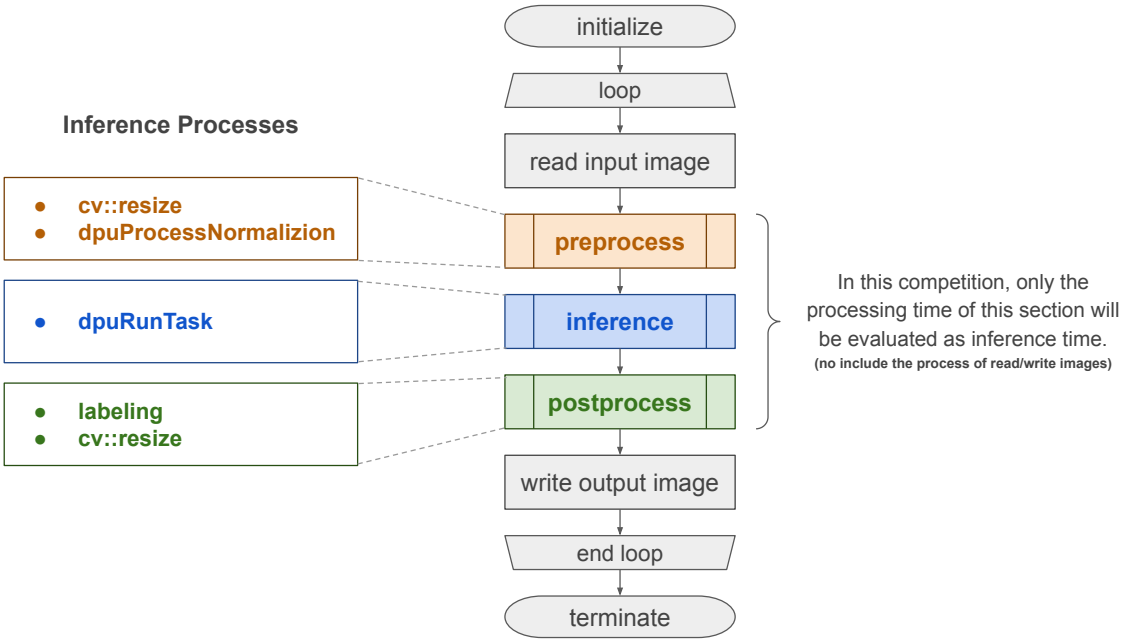


図 3: シングルスレッド実行時

シングルスレッド実装を行う場合，画像を 1 枚読み込み，推論を行い，推論結果を書き込む処理をテスト画像全てに対して繰り返す実装が考えられる．前処理 (preprocess)，推論処理 (inference)，後処理 (postprocess) の実行区間が推論処理時間計測の対象である．前処理では，OpenCV の関数 cv::resize を用いて入力画像の縮小を行い，Vitis-AI の DPU ライブラリの関数 dpuProcessNormalizion によって各画素値の正規化を行う．推論処理では，DPU による推論を行う関数 dpuRunTask を実行する．後処理では，推論結果を参照してラベル画像を生成し，その後，関数 cv::resize を用いてラベル画像の拡大を行う．

4.2　マルチスレッド実装

今回我々が実装したマルチスレッド実装の概要を図 4 に示す．

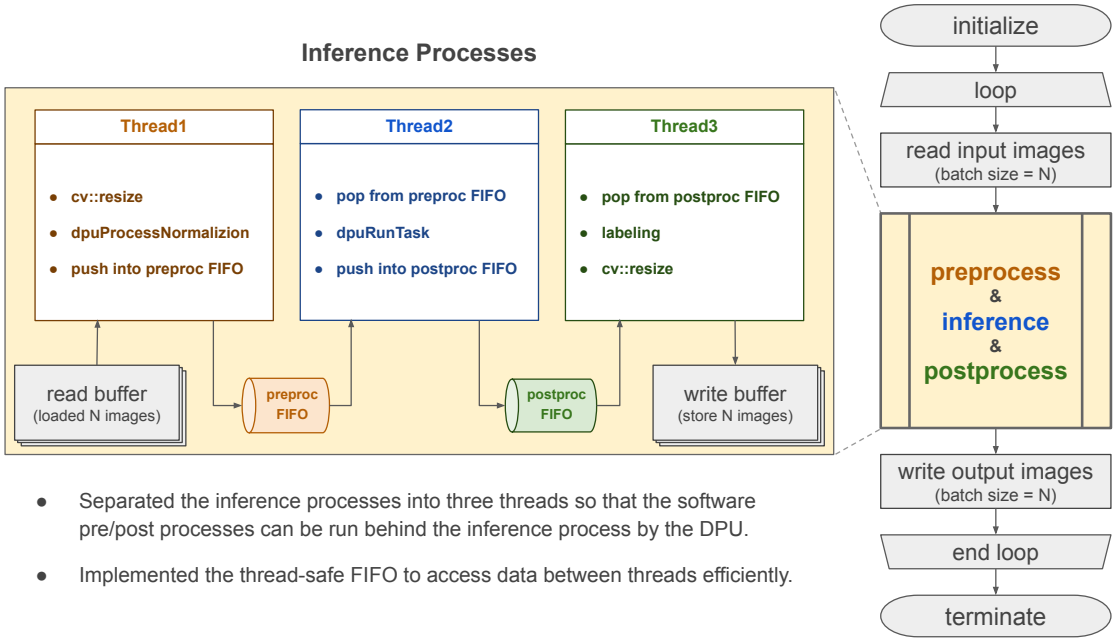


図 4: マルチスレッド実行時

我々のマルチスレッド実装では，前処理，推論処理，後処理をそれぞれ別スレッド (Thread1, Thread2, Thread3) で実行する．スレッドセーフな FIFO クラスを実装し，これを用いてスレッド間でデータを受け渡す実装を行った．前処理を行うスレッドで処理した入力を DPU による推論処理を行うスレッドに渡す preproc FIFO と，DPU による推論処理を行うスレッドで得られた出力を後処理を行うスレッドに渡す postproc FIFO の 2 つを用いる．また，マルチスレッド実装では 1 枚ずつ推論を行うのではなく，複数枚の画像のバッチ処理を行う．ここでは一度のバッチ処理に使用する入力画像の枚数を N とする．このような実装により，前処理，推論処理，後処理が擬似的にパイプライン実行され，実行速度の向上が期待される．

図 5 にマルチスレッド実装の処理のシーケンス図を示す．

N 枚の画像をメモリ上に読み込み，推論を行った後に結果をファイル出力するまでの処理のシーケンスを示している．3 つのスレッドが生成される時刻を t_0 ，3 つのスレッドの全てが終了する時刻を t_1 とするとき，本コンテストにおいて評価対象となる画像 1 枚あたりの推論処理時間 $t_{\text{inference}}$ は式 1 のように計算される．

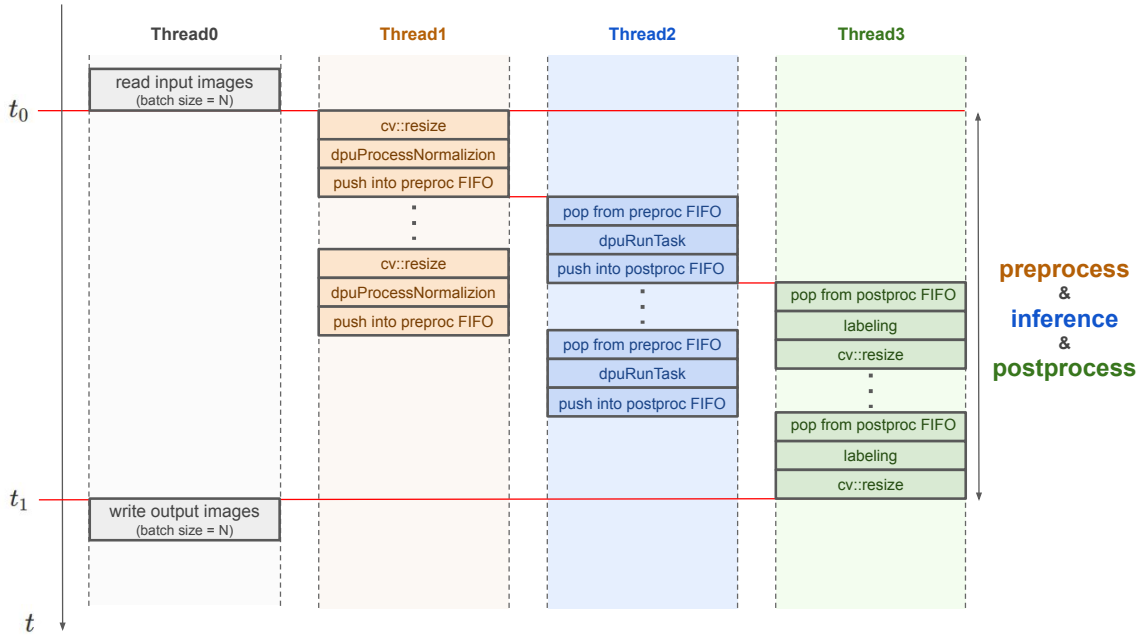


図 5: シーケンス図

$$t_{\text{inference}} = \frac{t_1 - t_0}{N} \tag{1}$$

ここで，シングルスレッド処理における前処理の縮小の処理時間を $t_{\text{pre\_resize}}$ ，正規化の処理時間を $t_{\text{pre\_normalize}}$ ，DPU による推論の実行時間を t_{dpu} ，後処理のラベル付けの処理時間を $t_{\text{post\_labeling}}$ ，拡大の処理時間を $t_{\text{post\_resize}}$ とする．また，preproc FIFO からデータを取得する処理の処理時間を t_{pop} ，postproc FIFO にデータを入れる処理の処理時間を t_{push} とする．このとき，画像 1 枚あたりの推論処理時間の予測値 $t^*_{\text{inference}}$ は式 2 ように表される．

$$t^*_{\text{inference}} = t_{\max} + \frac{t_{\text{pre}} + t_{\text{post}}}{N} + t_\alpha \tag{2}$$

$$\text{where} \quad t_{\max} = \max\{t_{\text{pre}}, t_{\text{pop}} + t_{\text{dpu}} + t_{\text{push}}, t_{\text{post}}\},$$

$$t_{\text{pre}} = t_{\text{pre\_resize}} + t_{\text{pre\_normalize}},$$

$$t_{\text{post}} = t_{\text{post\_labeling}} + t_{\text{post\_resize}}$$

ここで， t_α はソフトウェア処理のマルチスレッド化による処理時間のオーバーヘッドを表している．我々の実装において，画像 1 枚あたりの推論処理時間は，前処理，DPU による推論処理，後処

理のいずれかの処理時間に律速することが分かる．

次に，前処理，DPU による推論処理，後処理のそれぞれの具体的な実装を説明する．ここで示すソースコードは説明のためにいずれも変数名や実装の簡略化等を行っている．実際のソースコードは後日公開予定のリポジトリ[3]を参照されたい．

4.2.1　前処理

前述の通り，前処理では，cv::resize を用いて入力画像の縮小を行い (Listing 1, Line 6)，Vitis-AI の DPU ライブラリの関数 dpuProcessNormalizion によって各画素値の正規化を行う (Listing 1, Line 8-10)．

Listing 1: do_preprocess()

```
1  void do_preprocess() {
2    cv::Mat in(in_size, CV_8UC3);
3    auto r_buf = read_buffer.begin();
4    auto end = false;
5    while (!end) {
6      cv::resize(*r_buf, in, in.size(), 0, 0, cv::INTER_LINEAR);
7      end = (bool)(r_buf == read_buffer.end())
8      preproc_fifo.write(ok, end, [&](PreprocFIFOElementType& dst) -> void {
9        dpuProcessNormalizion(dst.data(), in.data, in.rows, in.cols, in_mean,
             in_scale_fix, in.step1());
10     });
11     r_buf++;
12   }
13 }
```

我々が実装した FIFO クラスは read/write の処理を lambda 式で記述するようになっている．これは，mutex を持つ FIFO のバッファを直接参照する記述をクラス定義のスコープ外で行うためである．このようにすることで FIFO 操作の処理を簡潔に記述することが出来る．FIFO クラスの実装は本レポートの最後に付録として添付している (Listing 4, Line 16-127)．

4.2.2　DPU による推論処理

DPU による推論処理では，preproc FIFO から DPU の入力バッファにデータをコピーし (Listing 2, Line 7)，関数 dpuRunTask の実行による推論後 (Listing 2, Line 9)，DPU の出力バッファから postproc FIFO に結果をコピーする (Listing 2, Line 12)．

[3] https://github.com/Vertical-Beach/ai-edge-contest4

Listing 2: do_inference()

```
1  void do_inference() {
2    constexpr auto in_size = sizeof(int8_t) * IN_IMG_W * IN_IMG_H * IN_IMG_C;
3    constexpr auto out_size = sizeof(int8_t) * OUT_IMG_W * OUT_IMG_H * NOF_CLASS;
4    auto end = false;
5    while (!end) {
6      preproc_fifo.read(ok, [&](const PreprocFIFOElementType& src) -> void {
7        std::memcpy(in_addr, src.data(), in_size);
8      });
9      dpuRunTask(task_conv_1);
10     end = preproc_fifo.neverReadNextElement();
11     postproc_fifo.write(ok, end, [&](PostprocFIFOElementType& dst) -> void {
12       std::memcpy(dst.data(), out_addr, out_size);
13     });
14   }
15 }
```

4.2.3　後処理

後処理では，推論結果を参照してラベル画像を生成し (Listing 3, Line 6-13)，その後，関数 cv::resize を用いてラベル画像の拡大を行う (Listing 3, Line 15).

Listing 3: do_postprocess()

```
1  void do_postprocess() {
2    cv::Mat out(out_size, CV_8UC1);
3    auto w_buf = write_buffer.begin();
4    while (w_buf != write_buffer.end()) {
5      postproc_fifo.read(ok, [&](const PostprocFIFOElementType& dst) -> void {
6        auto offset = dst.data();
7        for (int ri = 0; ri < out.rows; ri++) {
8          for (int ci = 0; ci < out.cols; ci++) {
9            const auto max_itr = std::max_element(offset, offset + NOF_CLASS);
10           out.at<uint8_t>(ri, ci) = (uint8_t)(std::distance(offset, max_itr));
11           offset += NOF_CLASS;
12         }
13       }
14     });
15     cv::resize(out, *w_buf, (*w_buf).size(), 0, 0, cv::INTER_NEAREST);
16     w_buf++;
17   }
18   if (!postproc_fifo.neverReadNextElement()) {
19     throw std::runtime_error("[ERROR] The data is still in the postproc FIFO.");
20   }
21 }
```

ここで，本推論アプリケーションの最終的な出力となる 1216*1936 のラベル画像は，各ピクセルに 0 から 5 の値が格納されたグレイスケール画像であることに注意されたい．この 0 から 5 の値は順に「乗用車」,「歩行者」,「信号」,「車道・駐車場」,「その他」に相当する．

5　性能評価

コンテストから提供されるテスト画像 649 枚に対して Ultra96-V2 ボード上で推論を実行した．実行時間の評価における設定項目は以下の通りである．

DPU: B2304@200/400MHz

推論画像サイズ: 320*640

N **(一度に処理する画像の数):** 130

5.1　実行時間

テスト用画像 649 枚に対する推論処理時間の平均を計測した．はじめにシングルスレッド実装における詳細な処理時間を表 4 に示す．

表 4: シングルスレッド実装における平均処理時間

name	elapsed time (ms)
$t_{\text{pre\_resize}}$	9.36
$t_{\text{pre\_normalize}}$	1.98
t_{pop}	0.63
t_{dpu}	54.1
t_{push}	1.09
$t_{\text{post\_labeling}}$	6.93
$t_{\text{post\_resize}}$	5.25

シングルスレッド実装では，前処理の処理時間 ($t_{\text{pre\_resize}} + t_{\text{pre\_normalize}}$) は 11.334ms，DPU によるによる推論の処理時間 (t_{dpu}) は 54.134ms，後処理の処理時間 ($t_{\text{post\_labeling}} + t_{\text{post\_resize}}$) は 12.183ms であった．すなわち，シングルスレッド実装の推論処理時間の平均は 77.651ms であることが分かる．

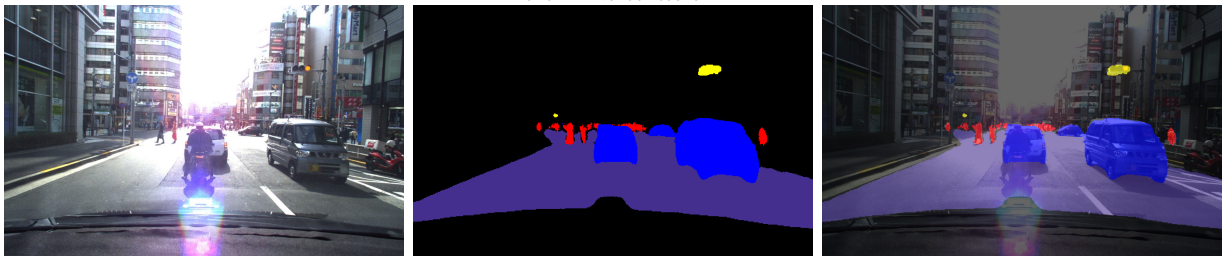
マルチスレッド実装では，推論処理時間の平均 ($t_{\text{inference}}$) が 57.928ms となった．処理のマルチスレッド化により約 25%の高速化を達成したことが分かる．ここで，マルチスレッド化によるオーバー

ヘッド t_α は 1.93ms であった．この値は理想的なパイプライン処理との処理時間の差を示す．

5.2 実行結果

図 6 に入力画像と推論結果，および推論結果を入力画像に合成したものの 1 例を示す．

図 6: 推論結果



推論結果を SIGNATE に投稿した結果，mIoU スコアは 0.6014857 となった．

5.3 リソース使用率

B2304 DPU を第 3 章で説明したとおりのコンフィグレーションで実装した場合の DPU および回路全体のリソース使用率を表 5.3 に示す．DPU IP を含んだ Vivado プロジェクトを Vitis が自動生成する際にベースとするプラットフォームプロジェクトに AXI GPIO や UART などの IP が含まれているため，今回のコンテストにおいては不要な IP も含まれている．

表 5: リソース使用率

	Total LUT	Logic LUT	LUTRAM	SRL	FF	RAMB36	RAMB18	DSP
DPU	35175	31982	1618	1575	61635	147	29	290
ALL	50414	45964	2450	2000	80647	151	29	290

5.4 消費電力

Ultra96V2 ボードに供給される DC 電源プラグに流れる電流から，消費電力を計測した．計測値はアイドル時 9.49W，推論時平均 11.52W，推論時ピーク 12.27W となった．

参考文献

[1] Maxim Berman, Amal Rannen Triki, and Matthew B Blaschko. The lovász-softmax loss: A tractable surrogate for the optimization of the intersection-over-union measure in neural networks. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 4413–4421, 2018.

[2] Tsung-Yi Lin, Piotr Dollár, Ross B. Girshick, Kaiming He, Bharath Hariharan, and Serge J. Belongie. Feature pyramid networks for object detection. *CoRR*, abs/1612.03144, 2016.

[3] 森 大輝）MTLLAB（谷合 廣紀. 第1回 ai エッジコンテストレポート. https://static.signate.jp/competitions/138/summaries/AIEdgeContest_Segmentation_2_MTLLAB.pdf.

[4] Xilinx. Ml-caffe-segmentation-tutorial. https://github.com/Xilinx/Vitis-AI-Tutorials/tree/ML-Caffe-Segmentation-Tutorial.

[5] Xilinx. Zynq dpu v3.1 product guide. https://www.xilinx.com/support/documentation/ip_documentation/dpu/v3_1/pg338-dpu.pdf.

付録

Listing 4: MultiThreadFIFO

```
template<typename T>
class ObjWithMtx {
public:
  T obj;
  ObjWithMtx() = default;
  ObjWithMtx(const T& _obj) : obj(_obj) {}
  void operator =(const ObjWithMtx& obj) = delete;
  ObjWithMtx(const ObjWithMtx& obj) = delete;
  void lock() { mtx_.lock(); }
  bool try_lock() { return mtx_.try_lock(); }
  void unlock() { mtx_.unlock(); }
private:
  std::mutex mtx_;
};

template<typename T, size_t D>
class MultiThreadFIFO {
public:
  MultiThreadFIFO(const uint32_t& sleep_t_us = 100) :
    sleep_t_us_(sleep_t_us) {
    init();
  }

  void operator =(const MultiThreadFIFO& obj) = delete;
  MultiThreadFIFO(const MultiThreadFIFO& obj) = delete;

  void init() {
    std::unique_lock<std::mutex> lock_w_func(w_func_guard_, std::try_to_lock);
    std::unique_lock<std::mutex> lock_r_func(r_func_guard_, std::try_to_lock);
    if (!lock_w_func.owns_lock() || !lock_r_func.owns_lock()) {
      throw std::runtime_error("[ERROR] Initialization of the FIFO failed.");
    }
    for (auto& state : fifo_state_) {
      std::lock_guard<ObjWithMtx<ElementState>> lock_state(state);
      state.obj = ElementState::INVALID;
    }
    r_idx_ = 0;
    w_idx_ = 0;
  }

  void write(ObjWithMtx<bool>& no_abnormality, const bool& is_last, std::function
      <void(T&)> write_func) {
    std::unique_lock<std::mutex> lock_w_func(w_func_guard_, std::try_to_lock);
    if (!lock_w_func.owns_lock()) {
```

```
44      throw std::runtime_error("[ERROR] The write function can't be called at the
            same time from multiple threads.");
45    }
46    while (true) {
47      {
48        std::lock_guard<ObjWithMtx<ElementState>> lock_state(fifo_state_[w_idx_]);
49        if (fifo_state_[w_idx_].obj == ElementState::INVALID) {
50          break;
51        } else {
52          std::lock_guard<ObjWithMtx<bool>> lock(no_abnormality);
53          if (no_abnormality.obj == false) {
54            throw std::runtime_error("[ERROR] Terminate write process.");
55          }
56        }
57      }
58      std::this_thread::sleep_for(std::chrono::microseconds(sleep_t_us_));
59    }
60    {
61      std::lock_guard<ObjWithMtx<T>> lock_fifo(fifo_[w_idx_]);
62      write_func(fifo_[w_idx_].obj);
63    }
64    {
65      std::lock_guard<ObjWithMtx<ElementState>> lock_state(fifo_state_[w_idx_]);
66      fifo_state_[w_idx_].obj = is_last ? ElementState::VALID_LAST : ElementState
            ::VALID;
67    }
68    incrementIdx(w_idx_);
69  }
70
71  void read(ObjWithMtx<bool>& no_abnormality, std::function<void(const T&)>
        read_func) {
72    std::unique_lock<std::mutex> lock_r_func(r_func_guard_, std::try_to_lock);
73    if (!lock_r_func.owns_lock()) {
74      throw std::runtime_error("[ERROR] The read function can't be called at the
            same time from multiple threads.");
75    }
76    while (true) {
77      {
78        std::lock_guard<ObjWithMtx<ElementState>> lock_state(fifo_state_[r_idx_]);
79        if (fifo_state_[r_idx_].obj == ElementState::VALID ||
80            fifo_state_[r_idx_].obj == ElementState::VALID_LAST) {
81          break;
82        } else {
83          std::lock_guard<ObjWithMtx<bool>> lock(no_abnormality);
84          if (no_abnormality.obj == false) {
85            throw std::runtime_error("[ERROR] Terminate read process.");
86          }
87        }
```

```
88        }
89        std::this_thread::sleep_for(std::chrono::microseconds(sleep_t_us_));
90      }
91      {
92        std::lock_guard<ObjWithMtx<T>> lock_fifo(fifo_[r_idx_]);
93        read_func(fifo_[r_idx_].obj);
94      }
95      {
96        std::lock_guard<ObjWithMtx<ElementState>> lock_state(fifo_state_[r_idx_]);
97        if (fifo_state_[r_idx_].obj == ElementState::VALID) {
98            fifo_state_[r_idx_].obj = ElementState::INVALID;
99          incrementIdx(r_idx_);
100       } else {
101         fifo_state_[r_idx_].obj = ElementState::INVALID_LAST;
102       }
103     }
104   }
105
106   bool neverReadNextElement() {
107     std::unique_lock<std::mutex> lock_r_func(r_func_guard_, std::try_to_lock);
108     if (!lock_r_func.owns_lock()) {
109       throw std::runtime_error("[ERROR] The read function can't be called at the
              same time from multiple threads.");
110     }
111     std::lock_guard<ObjWithMtx<ElementState>> lock_state(fifo_state_[r_idx_]);
112     return (fifo_state_[r_idx_].obj == ElementState::INVALID_LAST);
113   }
114
115 private:
116   enum class ElementState { VALID, VALID_LAST, INVALID, INVALID_LAST };
117
118   void incrementIdx(size_t& idx) const {
119     idx = (idx < D - 1) ? idx + 1 : 0;
120   }
121
122   const uint32_t sleep_t_us_;
123   std::array<ObjWithMtx<T>, D> fifo_;
124   std::array<ObjWithMtx<ElementState>, D> fifo_state_;
125   std::mutex r_func_guard_, w_func_guard_;
126   size_t r_idx_{0}, w_idx_{0};
127 };
```