5th AI Edge Contest / TFlite delegate と rv32emc を用いた実装

2022/02/27 s.yamashita

課題

- 車両前方カメラの撮影動画から物体の写る矩形領域を検出し、追跡するアルゴリズムを作成する
- RISC-Vをターゲットのプラットフォームに実装し、RISC-Vコアを物体追跡の処理の中で使用する

概要

- TensorFlow Lite (以下TFlite) は Google の提供する Mobile/Edge 向けの軽量な推論プラットフォームであり、外部アクセラレータに実行を委譲する delegate 機構を持っている。
- 今回の課題に対し、TFlite の delegate 機構を用いた FPGA アクセラレータを開発した。
(第４回コンテストで用いたものを修正して用いた)
- RISC-V は rv32emc に対応する CPU core を scratch から開発した。
- アクセラレータの実行制御と、物体検出結果からのトラッキング処理を１つの RISC-Vで行った。
- FPGA への実装は、アクセラレータ、RISC-V core とも SystemVerilog を用いた RTL 記述で行った。
- 推論ネットワークは TensorFlow ssd_mobilenet_v2_320x320_coco17_tpu-8 をベースに転移学習を行い、8bit 量子化を行った。
- アプリケーションは TFlite の python インターフェースを用い、RISC-Vでのトラッキング処理は C言語で開発した。
- 開発したアルゴリズムは、リーダーボード上の評価 0.0956744、Ultra96-V2 上での実行時間は 261 ms/frame となった。

推論ネットワーク

Object Detection のベースネットワークは TensorFlow2 Detection Model Zoo^1 から、軽量の "SSD MobileNet v2 320x320" を選択し、 今回の train_video 10 class で転移学習を行った。
転移学習は Google Colab の GPU インスタンスで行い、250k step まで学習した。
転移学習後のネットワークに更に Training 後の量子化を行って、8bit に量子化し、TFlite FlatBuffer 形式の graph に変換した。

以下は、TFlite の benchmark tool を ultra96v2 の CPU 上で実行し、今回のネットワークを分析した結果である。

^1 TensorFlow2 Detection Model Zoo

CPU での推論実行時間は約 383 ms であり、Conv2D, dwConv2D の２つの演算で 96.4% を占める。
この２つの演算を FPGA に delegate した。

```
Number of nodes executed: 102
===================== Summary by node type ========================
              [Node type]     [count]   [avg ms]    [avg %]    [cdf %]
                  CONV_2D          55    323.961    84.653%    84.653%
        DEPTHWISE_CONV_2D          17     45.074    11.778%    96.431%
TFLite_Detection_PostProcess        1      9.418     2.461%    98.892%
                     ADD           10      3.018     0.789%    99.681%
                QUANTIZE            1      0.870     0.227%    99.908%
                LOGISTIC            1      0.149     0.039%    99.947%
              DEQUANTIZE            2      0.113     0.030%    99.977%
           CONCATENATION            2      0.054     0.014%    99.991%
                 RESHAPE           13      0.035     0.009%   100.000%
                                  total  382.741 ms
```

このネットワークを python アプリから FPGA に delegate 実行する。

python の tflite delegate interface から、PL に実装した RISC-V コアに ネットワークの各nodeのパラメータを渡し、RISC-V コアがFPGA上のアクセラレータの設定、キック、終了処理などを行う。
Object Detection 結果は python 側に得られ、これを PL に実装した RISC-V コアに渡し、tracking 処理を行う。tracking 結果を python に返して 結果の json に反映する。
Tracking アルゴリズムは、Tracklet を保持し、新しい object Detection 結果との距離、矩形面積、輝度差 からマッチングを行う単純なものである

全体システム構成

図１にシステム構成を示す。
TFlite の Interpreter（推論プログラム）には外部のアクセラレータを接続する delegate API があり[^2]、これを用いて FPGA に演算を渡す。

[^2] implementing_delegate

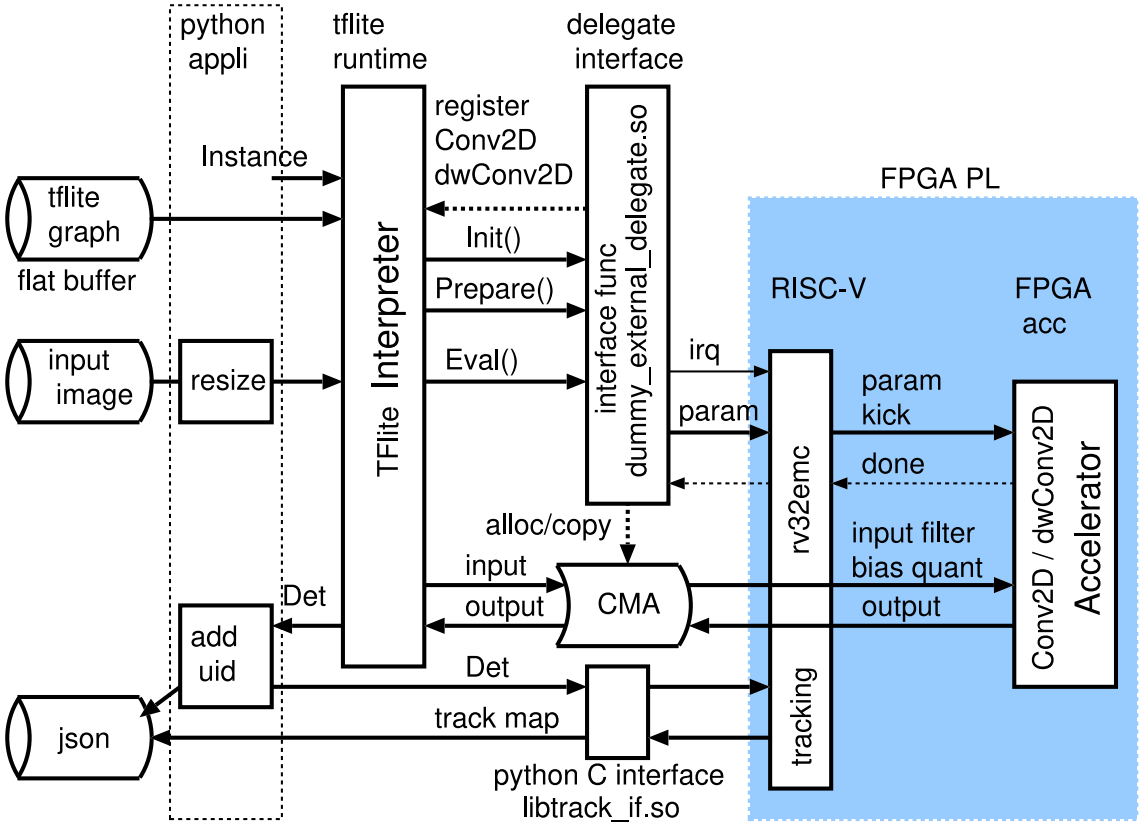

図１.システム構成

新たに作成したFPGAとのインターフェース関数 dummy_external_delegate.so を Interpreter にリンクする。
インターフェース関数は起動時にdelegate する演算種別 Conv2D, depthwiseConv2D の２つを登録する。
Interpreter を起動すると、FlatBuffer 形式の graph を実行し、登録した演算のみインターフェース関数に渡される。
インターフェース関数には Conv 演算のパラメータと、input, filter, bias, output の４つの Tensor へのポインターが渡されるので、パラメータとポインターを FPGA に渡して ハードウェアのシーケンサーを kick し、演算終了を待つ。
Tensor へのポインターは、Linux の仮想記憶領域にあり、直接FPGAからアクセスできない。
Interpreter起動時、インターフェース関数の Prepare() が一度だけ呼ばれるので、この中で予め Tensorデータを CMA 領域に copy しておき、この領域で FPGA とデータをやり取りする。
また、output channel ごとの量子化パラメータ quant を計算する必要があり、一度だけ計算して CMA 領域に保持する。
FPGA に渡した Conv演算パラメータは、FPGA 側に設けた RISC-V CPU で並列演算に対応した形に変換して Conv演算アクセラレータのシーケンサに与え、kick する。
FPGA の CPU は、シーケンサの終了を待って、output buffer に残ったデータを CMA 領域に flush して演算を終了し、アプリケーションに通知する。

Interpreter が推論ネットワークの Object Detection 結果を出した後、これを python の C-interface を介して FPGA の RISC-V に送り、RISC-V がトラッキング処理を行う。
トラッキング処理結果を再び python に返し、json に反映する。

rv32emc core のハードウェア

RISC-V の ISA のうち、組み込み用途向けの EMC (32bit 16 Register, Mul/Div, Compressed 命令) の構成で CPU core を開発した。[^3]
Fetch/Decode/Exec/MemoryAccess/Writeback の５段パイプライン構成とした。
分岐予測は実装していないが、プログラム/データメモリを全て FPGA の BRAM に置いているので、分岐のペナルティーは小さい。
クロスコンパイラは riscv-gnu-toolchain を用い、configure で rv32emc 対応に設定した。(gcc バージョンは 9.2.0)

```
$ ./configure --prefix=/opt/rv32e --disable-linux --with-arch=rv32emac --
with-abi=ilp32e
$ make newlib
```

rv32emc core の開発は、以下のような手順で行った。

1. CPUのハード構成を設計し、パイプラインまで正確にエミュレートした命令セットシミュレータ(ISS)をＣ言語で作成 → ISS でテストプログラムを実行
   risc-v のクロス gdb でのテストプログラム実行結果(実行トレース)をリファレンスとして ISS をデバッグ
   テストプログラムには、Reed-Solomon ECC、500桁の pi の計算、float 演算を含む計算(soft float のテスト)、など、ある程度複雑な演算処理を行って結果がはっきり現れるものを用いた
2. 作成した ISS をリファレンスとして HDL(SystemVerilog) を記述、ISS の実行トレースと HDL の論理シミュレーション(xsim) 結果をつき合わせてHDLをデバッグ
   ISS の実行トレースと突き合わせしやすい形のトレースを HDL の $display で出力しバグの追跡を行った
   また、ISS 用に作成した命令のテーブルを使用して HDL の命令テーブルを自動生成するようにして記述ミスなどを防いだ
3. テストプログラムで正しい実行結果が得られるようになったところで、割り込み機構を実装、uart でターミナルを接続、タイマー(mtime) でタイマー割り込みができるようにして、Vivado で論理合成を行いFPGAにロード、実機でターミナルをつないで動作確認
   ここで用いたFPGAは、Spartan-7 で 60MHz 動作 (Ultra-96v2 では 125MHz でMET)
4. モニタープログラムを組んで、テストプログラムをロード、実行できるようにして更に長時間の動作検証を行った

この環境で、coremark を実行してみたところ 2.402 (CMK/MHz) となったので、そこそこの性能が得られているようだ。

[^3] https://github.com/shin-yamashita/rv32emc

FPGAに実装したハードウエア構成

図２に FPGA PL 部に実装したアクセラレータのブロック構成を示す。
第４回コンテストで用いた回路を修正し、制御及びトラッキング処理用 CPU として前述の rv32emc CPU を用いた。

PS のアプリケーションは、AXI-APB bridge を介してRISC-V の RAM をアクセスすることができる。
RISC-V の実行バイナリを RAM にロードし、reset を解除することで CPU を起動する。
また、APB から 割り込みを発生し、アクセラレータの制御を kick する。
アクセラレータ制御を割り込み処理で行うことで、トラッキング処理を独立して並行処理できる。

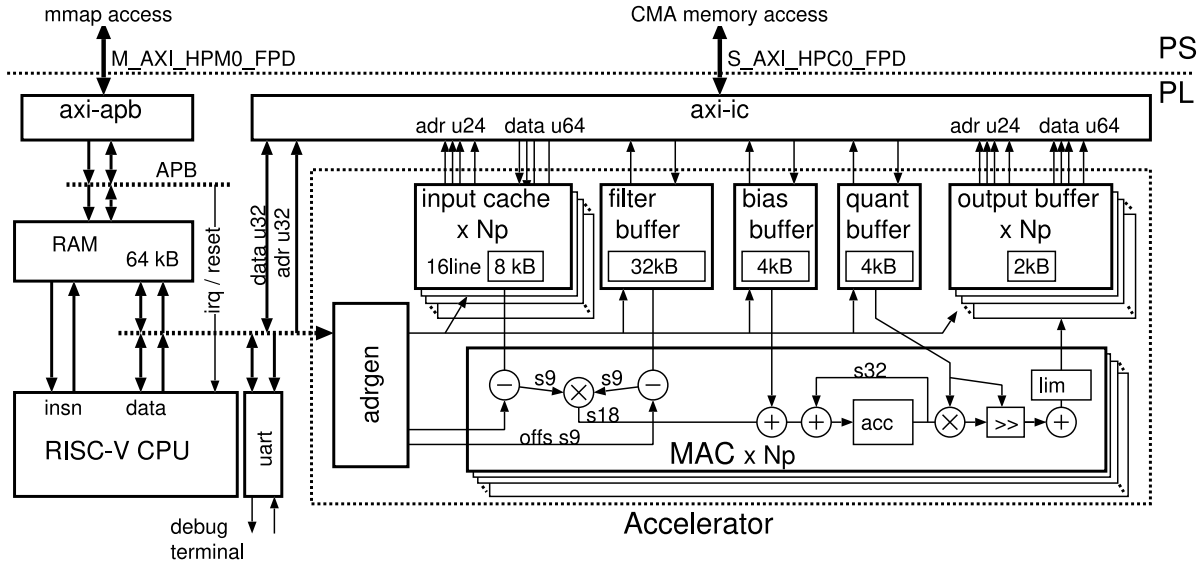

図２. PL ブロック図

Accelerator 部では、Tensor data のアドレスを発生する adrgen からのアドレスにしたがって Tensor data にアクセスする。
畳み込み演算回路(MAC)で input tensor と filter tensor を乗算、累算し、bias を加えたのち8bitに量子化し、output tensor に出力する。 Conv2D と dwConv2D の２種の演算で MAC 回路は共通であり、アドレス発生パターンが異なるのみである。
input, filter, bias, quant, output の5つのデータアクセスは、S_AXI_HPC0_FPD ポートを介して PS の CMA 領域をアクセスする。
MAC は、Np 個並列に実装されており、input, output のデータアクセスには Np 個並列にキャッシュメモリを設けた。filter, bias, quant は全ての MAC 演算で共通なので、それぞれ１個のキャッシュメモリを設けた。
filter, bias, quant, output はほぼ連続アクセスなので、FIFOバッファのような構造で良いが、input は畳み込みのタップで大きくアドレスが飛ぶので、16ラインのキャッシュメモリ構造にした。(図３.input-cache ブロック図 参照)

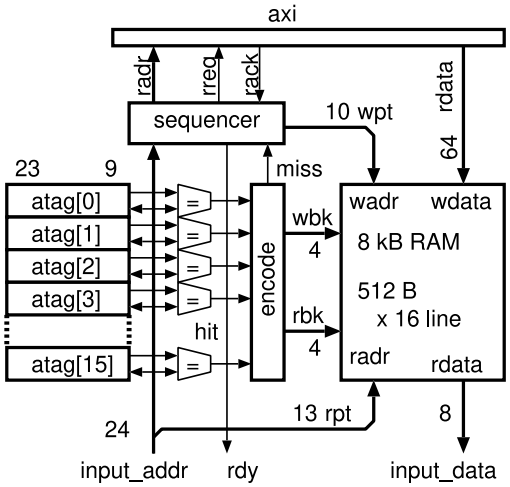

図３. input-cache ブロック図

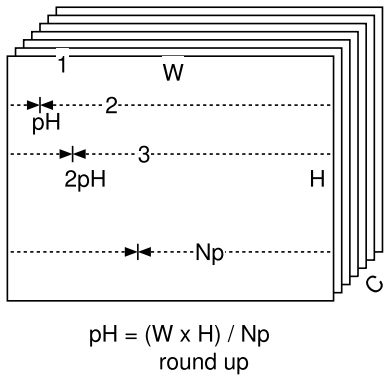

図４. Np 分割

並列化のアドレス分割は、図４に示すように、output Tensor の W x H を Np 分割する。C を共通にすることで、filter, bias, quant のキャッシュが１系統ですむ。 ただし、W x H < Np のときは並列数は W x H に制限される。

インターフェース関数から渡される演算パラメータは、M_AXI_HPM0_FPD ポート axi-apb ブリッジを介し、アクセラレータ制御用 RISC-V CPU に渡される。
Np個並列の input/output それぞれに異なるアドレス発生を行うため、CPU は adrgen ブロックに Np種類のアドレスを計算し、設定を行う。
並列に実装するアドレス発生回路をできるだけ簡単化してハード規模を抑えるために、CPU にアドレス計算を分担するようにした。また、この CPU にはシリアルターミナルを接続できるようにし、デバッグおよび実行時間の観測などにも活用した。

Accelerator 部の開発手順

図5 に今回の開発で行ったワークフローを示す。

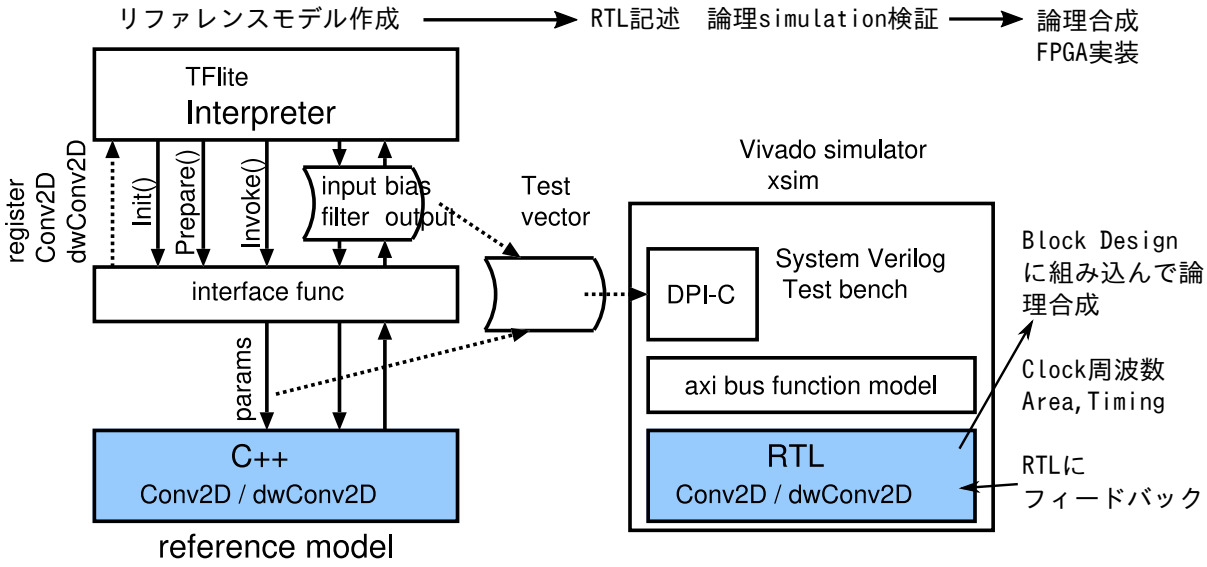

図5. システム開発手順

リファレンスモデル作成

TFlite の delegate API を調査し、インターフェースを把握し、アプリケーションを実装できるようにする。
FPGA に持っていく前に delegate で Conv2D,depthwiseConv2D の reference 実装を C++ で記述。推論の実行を delegate して結果の一致を確認してリファレンスモデルとする。
TF lite のソース^4には、各演算の reference 記述があるので、これを見ることでパラメータの意味、演算順序などがわかる。
ハード化を意識しながら Conv2D,depthwiseConv2D を１つのルーチンにまとめ、パラメータで切り替えられるように変更する。
FPGA での並列化の分割方法を考え、ハード化に適した形の演算順序に整え、 更に累算器 acc の丸めを精度に影響の出ない程度に簡単化した。
このソフトをハード実装のリファレンスとし、ハード検証用のテストベクタ（Conv パラメータと ４つの Tensor データのセット）を作成した。

^4 tensorflow

RTL記述、simulation 検証

C++ で実装した reference を参照しながら SystemVerilog で RTL 記述する。
論理シミュレーションには Vivado xsim を用いた。
PS/PL 間 interface は axi bus であり、プロトコルに違反するとハングアップにつながる。この部分の設計は axi bus function model を用いて simulation 検証しながら行った。
（FPGAの部屋「AXI4 Slave Bus Functional Model のVerilog HDL版２」^5 を使わせていただきました)
全体を結合しての検証は、axi bus function model に DPI-C を介して検証用のテストベクタをアクセスする機構を加えて一致検証できるようにし、バグ解析の効率化を行った。

^5 AXI4 Slave Bus Functional Model のVerilog HDL版２

論理合成、FPGA 実装

作成した RTL module を Vivado の Block Design ツールに読み込んで Zynq と接続した。
(図6. Block Design　色付きのブロックがRTLブロック)
Np 、キャッシュサイズ、クロック周波数などパラメータを変えて論理合成を繰り返し、エリア及びタイミングが MET できるように RTL 記述にフィードバックした。

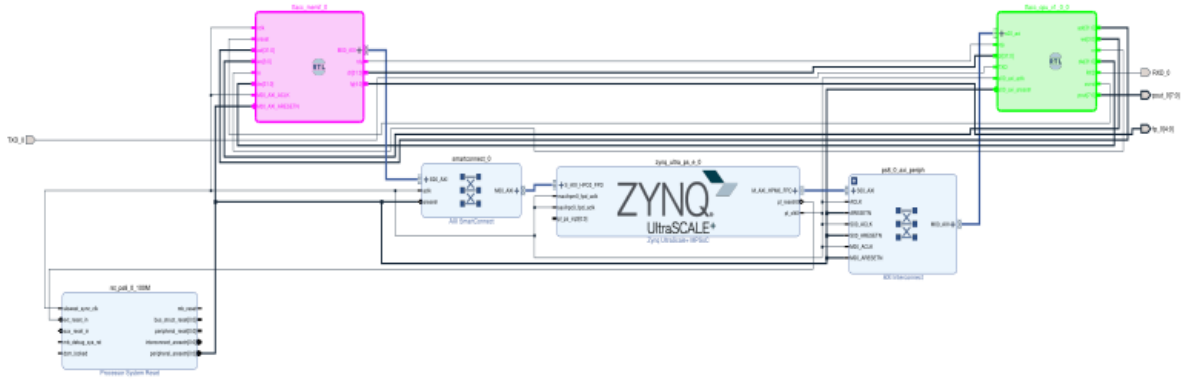

図6. Block Design

図7 は最終的な FPGA のリソース使用率である。並列数 Np = 42、クロック周波数 = 125 MHz とした。

Resource	Utilization	Available	Utilization %
LUT	63426	70560	89.89
LUTRAM	1416	28800	4.92
FF	36744	141120	26.04
BRAM	204	216	94.44
DSP	263	360	73.06
IO	15	82	18.29
BUFG	7	196	3.57

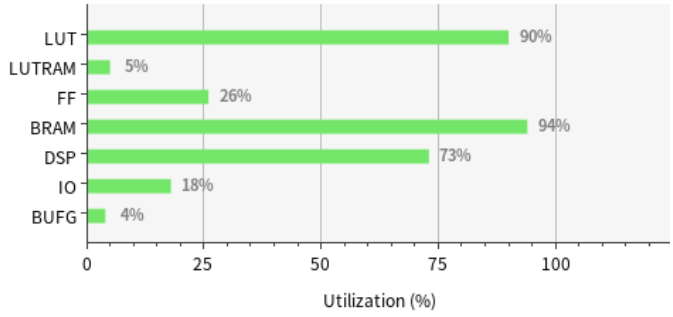

図7. Utilization

アプリケーション

今回、推論の実行アプリケーションは python で作成した。
Google は、FlatBuffer に変換した推論グラフを高速に実行する軽量の tflite_runtime[^6] を提供している。
ultra96v2 にも pip で導入できる。
新たに作成した delegate インターフェースは、ダイナミックリンクライブラリ(dummy_external_delegate.so)として tflite_runtime とリンクすることで実行の delegate が行われる。
tflite_runtime の python インターフェースでは interpreter のインスタンス時に dummy_external_delegate.so を指定してリンクするだけである。

```
import tflite_runtime.interpreter as tflite
# Instantiate interpreter
interpreter = tflite.Interpreter(model_path=model,
      experimental_delegates=[tflite.load_delegate(
                  '{path-to}/dummy_external_delegate.so.1')] )
interpreter.allocate_tensors()
# Get input and output tensors.
input_details  = interpreter.get_input_details()
output_details = interpreter.get_output_details()
# set image
interpreter.set_tensor(input_details[0]['index'], np.uint8(image))
# Invoke
interpreter.invoke()
# Output inference result
score = interpreter.get_tensor(output_details[0]['index'])[0]
boxes = interpreter.get_tensor(output_details[1]['index'])[0]
num   = interpreter.get_tensor(output_details[2]['index'])[0]
classes = interpreter.get_tensor(output_details[3]['index'])[0]
```

[^6] tflite_runtime

RISC-V と推論実行アプリのシーケンス。

今回実装したトラッキングアルゴリズムはおよそ以下のようなものである

1. Object Detection 結果 1 frame 分に、一旦ユニークな id をつけて、tracking 処理に渡す。
2. tracking 処理では Tracklet を保持し、Tracklet と Detection 結果の距離などの評価指標でマッチングを行う。
3. マッチした Object に一旦つけた id を Tracklet が保持した id にすげ替える。
4. マッチしなかった Tracklet は一旦保持し、数フレーム連続でマッチしない場合破棄する。
5. マッチしなかった Object を新たに Tracklet に加える。

これを python と RISC-V で分担して行う。
Object Detection の結果(score/box/class)は python 側に得られるので、これを RISC-V に渡す必要がある。
データを渡す手段として、CMA 領域に場所を確保し、一旦 python - C インターフェースで C プログラムにデータを渡し、C の中で CMA 領域にデータをコピーするようにした。
その後 C から RISC-V に tracking 処理を kick し、RISC-V が CMA 領域のデータをもとに tracking 処理を行う。
処理が終わるのを待って処理結果のマッチング情報を APB 経由で読み、python - C インターフェースで python に戻す。
python で json に反映する。

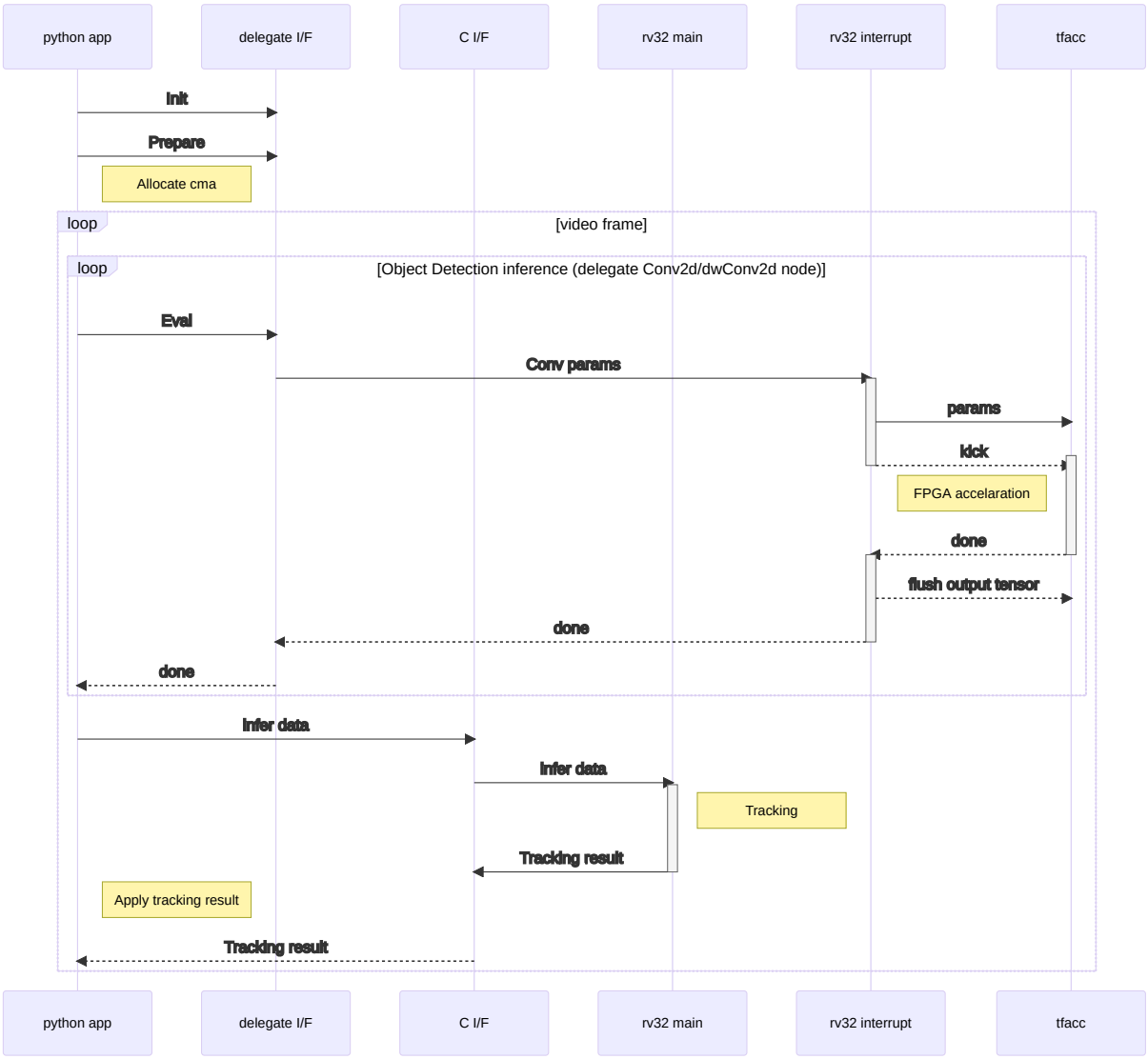

図8. 推論実行アプリシーケンス

実行結果

MAC並列数 Np 、input-cache サイズ、クロック周波数は、RTLで組んだ回路がタイミングMETできる最大の並列数、最大の周波数を探って、以下の表のように決定した。

MAC 並列数 Np	FPGA MAC 演算クロック	input-cache 転送単位(1ラインサイズ)
42	125 MHz	512 byte

処理時間

処理時間は上記の条件で推論実行に約 261 ms を要した。
RISC-V によるトラッキング処理は約 0.2 ms と小さい。

	前処理(resize)	Object Detection	トラッキング処理
処理時間(ms)	17.9	242.6	0.2

推論処理時間詳細

推論時間は、PS の CPU と FPGA/RISC-V で分担されており、FPGAの実行時間を FPGA内に実装したカウンタで測定した。
図9に結果を示す。

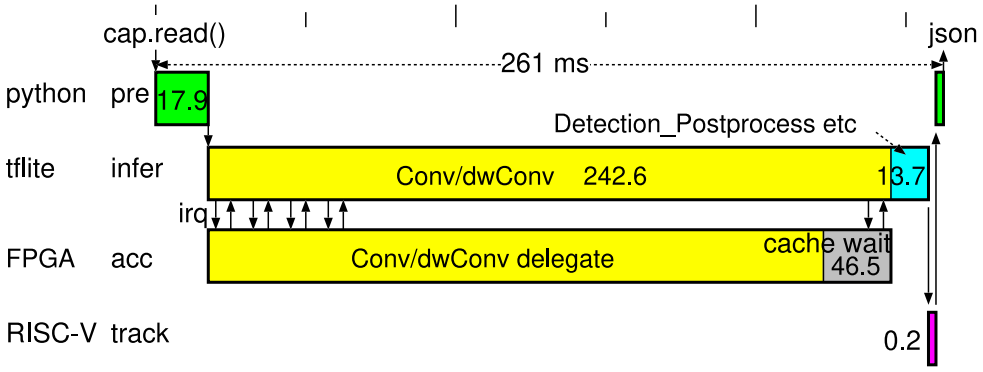

図9. 推論処理時間

グラフの Conv/dwConv(黄色) 部分、畳み込み演算そのものの実行時間が支配的である。

まとめ、考察

- TensorFlow Lite の delegate 機構を用いた FPGA アクセラレータを開発し、動作を確認することができた。
- RISC-V (rv32emc) を RTL で実装し、FPGAに組み込んで、アクセラレータ制御とトラッキング処理に用いて動作を確認した。
- リーダーボードの評価値は 0.0956744 と低いが、mobilenet_v2 の転移学習結果の検出精度が低い様で、トラッキングアルゴリズムも簡易なためトラッキングが継続できない様子が見られる。
  centernet が軽量で精度が良さそうであったが、現在 TF2 では量子化に対応できていないので採用できなかった。
- 速度は 261 ms/frame と遅い。MACの並列数が 42 にとどまることが大きい。今回の並列化は１次元であるが、output チャンネルも 4~8 並列として２次元の並列化で FPGA のDSPリソースの活用を行うのが効果が大きいと思われる。
  今回初期化処理 Prepare() で filter Tensor を CMA にコピーしているので、チャンネル並列に合わせた filter 順序に並べ替えることで、最小限のハード追加で２次元化できそうである。