# 第5回AIエッジコンテスト 提出物実行方法

## SDカードのセットアップ

`sd.img`をSDカード(16GB以上)に書き込みます。ddコマンドまたはEtcherを使用して書き込みます。

```
sudo fdisk -l #SDのデバイスを確認, 以下の場合はsdc
sudo dd if=sd.img of=/dev/sdc bs=1M
```

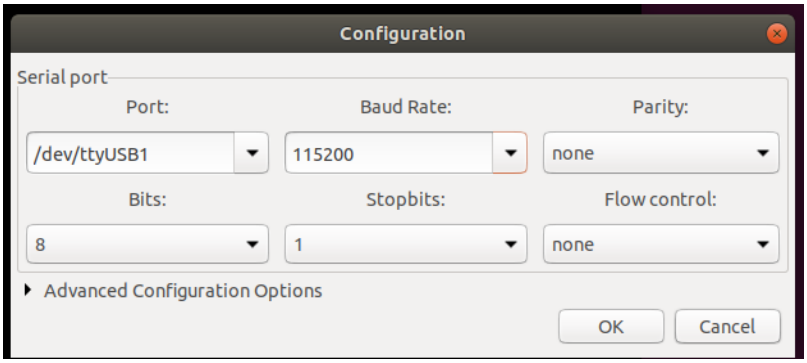
## Ultra96のセットアップ

SDカードをUltra96にセットし、SDブートモードにします。
UART-JTAGコネクタ・電源・USB-LANをUltra96に接続します。
UART-JTAGコネクタはホストPCとMicroUSBケーブルで接続し、USB-LANケーブルにLANケーブルを接続します。
※Ultra96のWiFiには対応していません。

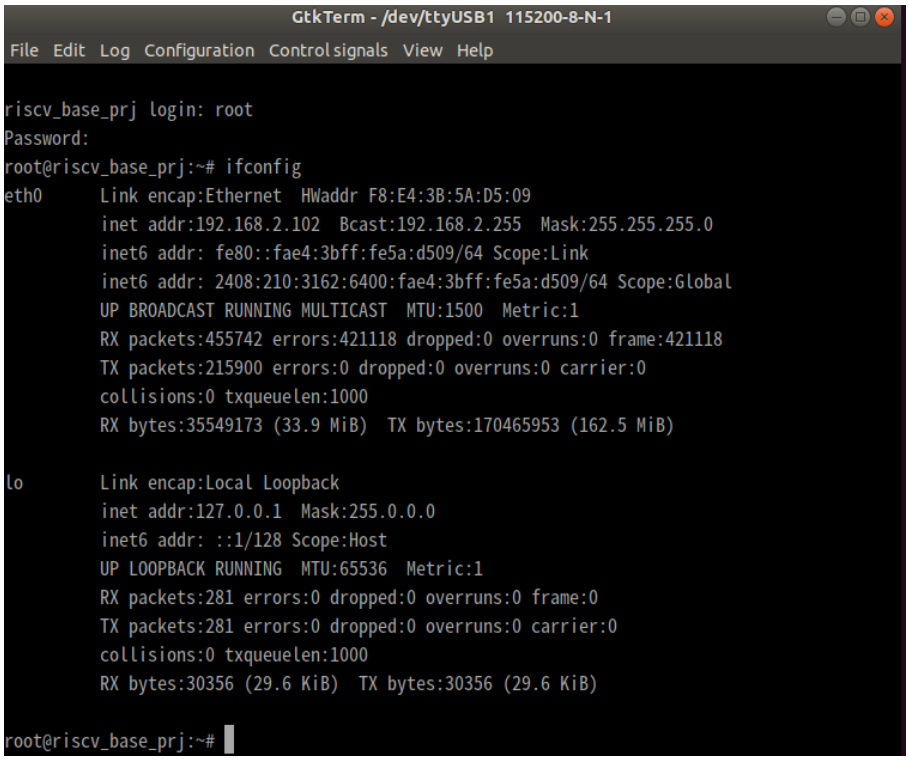
USB-LANはr8152またはAX88179が搭載されている以下の製品での動作確認をしています。
SDが正しく書き込めていない場合、ブート中にKernelPanicを起こしたりUSB-LANアダプタが使用できないといった事象を確認しています。

- https://www.amazon.co.jp/gp/product/B07MK6DJ6M/
- https://www.planex.co.jp/products/usb-lan1000r/spec.shtml

Ultra96の電源ボタンを押して電源を駆動し、TeraTermやgtktermなどUSBシリアル接続が可能なアプリケーションを起動します。 BandRateは115200を設定してください。以下にgtktermでの接続画面を示します。




ブートが問題なく成功し、シリアル接続ができればユーザ名`root`パスワード`root`でログインしてください。 `ifconfig`でUSB-LAN経由で割り当てられたIPアドレスを確認します。以下の例では`192.168.2.102`




が割り当てられています。

以降はホストPCからのssh接続で作業を続行できます。

```
ssh root@192.168.2.102 #password: root
```

## 入力用データセットの準備

Ultra96で動作するトラッキングアプリケーションは、mp4形式の動画入力に対応していないため、事前にホストPCでavi形式に変換する必要があります。ここではffmpegを使用した変換の手順を示します。テスト動画は再配布禁止のためSDイメージに含まれていません。変換時にffmpegに`-vcodec mjpeg`を指定することに注意してください。指定しない場合、生成されたファイルは拡張子はaviですがコーデックがmp4のためアプリケーションから読み込むことができません。

```
cd <parent of 'test_videos' directory>
mkdir test_videos_avi
cd test_videos_avi
ffmpeg  -i ../test_videos/test_00.mp4 -vcodec mjpeg test_00.avi #00~73
```

変換完了後、`test_videos_avi`ファイルをUltra96の`/home/root/`以下に配置します。

```
cd ..
scp -r test_videos_avi root@192.168.2.102:/home/root
```

## トラッキングライブラリのビルド

ビルド済みライブラリ`libbytetrack.so`が`sd.img`には含まれているので、再ビルドする際にのみ必要です。 Ultra96上で作業を行います。

```
cd ~
cd ByteTrack-cpp-ai-edge-contest-5
mkdir build && cd build
cmake -DRISCV=ON -DDPU=ON ..
make -j
```

## トラッキングアプリケーションの実行

トラッキングアプリケーションのディレクトリに移動します。

```
cd ~/ByteTrack-cpp-ai-edge-contest-5/app/generate_submit_file
```

`run.bash`はアプリケーションのビルド・実行・評価用JSONファイルの生成の一連の処理を行います。
`run.bash`の実行コマンドは以下の通りです。
Usage1は物体検出の推論を実行せずに、JSONファイルから読み込みトラッキングのみを実行します。
Usage2はDPU,RISCVを使用して物体検出およびトラッキング処理の両方を実行します。

```
Usage 1: ./run.bash <videos directory path> <video name prefix> json
<detection results directory path>
Usage 2: ./run.bash <videos directory path> <video name prefix> dpu
<modelconfig .prototxt> <modelfile .xmodel>
```

テストデータに対する実行コマンドは以下の通りです。

```
./run.bash \
~/test_videos_avi \
test \
dpu \
~/demo_yolov4/yolov4_tiny_conf0.1.prototxt \
~/yolov4_tiny_signate/yolov4_tiny_signate.xmodel
```

※`<modelconfig .prototxt>`にはファイル名が`conf0.1`のものを使用してください。`conf0.3`,`conf0.5`を使用すると期待通りの評価値が得られません。

実行が完了すると、以下のような実行時間のサマリが表示されます。1フレームごと・動画ごとの処理時間が表示されます。
レポートで詳しく説明していますが、マルチスレッド処理により全体処理時間は物体検出・トラッキング処理の合計時間よりも小さくなっています。

```

```

```
time summary:
inference : 59.97ms/frame 8995.00ms/video
tracking : 60.64ms/frame 9096.00ms/video
inference + tracking : 61.57ms/frame 9235.00ms/video
```

最終的に生成された`predictions.json`が評価用のJSONファイルとなります。

## tiny-YOLOv4単体アプリケーションの実行

ここではtiny-YOLOv4単体アプリケーションの実行方法について説明します。
画像に対する推論処理、または動画に対する推論処理を実行してJSONファイルとして保存することができます。

```
cd demo_yolov4
sh build.sh
./demo_yolov4 \
./yolov4_tiny_conf0.5.prototxt \
../yolov4_tiny_signate/yolov4_tiny_signate.xmodel \
./test.jpg \
image
```

- 出力結果

```
./demo_yolov4 ./yolov4_tiny_conf0.5.prototxt
../yolov4_tiny_signate/yolov4_tiny_signate.xmodel ./test.jpg  image
./yolov4_tiny_conf0.5.prototxt
../yolov4_tiny_signate/yolov4_tiny_signate.xmodel ./test.jpgModel
Initialize Done
pedestrian 0.997999 742.968 847.408 585.005 754.543
pedestrian 0.997482 1565.91 1670.35 622.344 791.883
pedestrian 0.996782 1115.28 1219.72 580.114 797.806
pedestrian 0.995807 1271.53 1352.86 587.657 757.196
pedestrian 0.993249 1039.02 1120.36 590.477 760.015
pedestrian 0.993233 426.126 507.464 575.575 745.113
pedestrian 0.990519 1521.68 1557.93 596.448 683.082
pedestrian 0.988856 1445.64 1467.62 597.649 665.12
pedestrian 0.985038 1378.46 1429.03 593.256 723.378
pedestrian 0.976868 530.265 552.248 555.434 607.981
pedestrian 0.923061 1831.89 1913.23 631.774 801.312
pedestrian 0.851889 952.121 1017.04 566.77 733.85
pedestrian 0.8173 890.1 971.438 575.575 745.113
pedestrian 0.729961 620.609 701.947 563.076 732.614
pedestrian 0.679071 828.476 893.398 574.704 704.826
```

結果は`result.jpg`として保存されます。

## RISCV単体アプリケーションの実行

ここではRISCVにオフロードした処理のテストを行うアプリケーションの実行方法について説明します。
線形割当問題のハンガリアン法アルゴリズムをRISCVで実行し、入力に対し期待する出力が得られるかを確認します。 クロスコンパイル方法などについては`riscv_test/README.md`を参考にしてください。
主なファイルは以下の通りです。

- host/main.cpp: riscvで実行されるハンガリアン法アルゴリズム
- edge/main.cpp: riscv実行・入出力チェックを行うプログラム
- edge/riscv_imm.c: `host`側でコンパイルされたRISCV向け命令データ
- edge/data/: 入出力用データ

```
cd riscv_test
cd edge
g++ main.cpp riscv_imm.c
./a.out
```

以下のように出力され、正常に動作していることが確認できます。

```
n: 30
set input:433[microsec]
      run:873[microsec]
Attempt 0 Success
n: 30
set input:115[microsec]
      run:854[microsec]
Attempt 1 Success
...
Attempt 606 Success
ALL PASS
```

# 付録

Ultra96への各種ライブラリ・ツールのインストール作業を説明します。
ここでの作業は`sd.img`では既に行われているので実行する必要はありません。

## VART(Vitis-AI runtime)のインストール

Vitis-AI v1.4からVARTをダウンロードします。

```
cd VART
./target_vart_setup_2020.2.sh #エラーが出る場合、rpm --nodepsを追加
```

## Eigenのインストール

```
export EIGEN_VERSION=3.3.9
mkdir -p /tmp/eigen && cd /tmp/eigen && \
    wget https://gitlab.com/libeigen/eigen/-/archive/3.3.9/eigen-3.3.9.zip
&& \
    unzip eigen-${EIGEN_VERSION}.zip -d . && \
    mkdir /tmp/eigen/eigen-${EIGEN_VERSION}/build && cd
/tmp/eigen/eigen-${EIGEN_VERSION}/build/ && \
    cmake .. && \
    make install && \
    cd /tmp && rm -rf eigen
```

## CMakeのインストール

```
git clone -b release --depth=1 https://github.com/Kitware/CMake.git && cd
CMake && \
     ./bootstrap && make -j "$(nproc)" && make install && \
     cd ../ && rm -rf CMake
```