第5回 AIエッジコンテスト Vertical-Beach レポート

最終成果物概要

- 対象ボード: Ultra96v2
- 物体検出アルゴリズム: YOLOv4-tiny
- トラッキングアルゴリズム: ByteTrack
- HW構成: Xilinx DPU + RISCV
- DPU動作周波数: 150/300MHz
- RISCVアーキテクチャ: rv32imfac
- RISCV動作周波数: 150MHz
- RISCV独自追加命令: なし
- RISCVで実行される処理: 線形割当問題のハンガリアン法アルゴリズム
- 使用開発環境: Vivado/Vitis/Petalinux 2020.2, Vitis-AI v1.4, VexRiscv
- 最終成果物の公開予定リポジトリ: https://github.com/Vertical-Beach/ai-edge-contest-5
- 最終評価値: 0.2807344
- 最終評価結果可視化動画（限定公開）: https://youtu.be/xeXifLHZIno

最終成果物処理性能

	per frame[ms]	per test video[ms]
物体検出	51.66	7748
物体追跡	29.93	4489
全体処理	52.30	7845

- 後述の通り、マルチスレッド化により`全体処理時間 < 物体検出時間 + 物体追跡時間`となる。
- `全体処理時間`はメモリに動画のフレーム画像を読み込んでから、評価用のJSONファイルと等価な情報が生成されるまでの時間を指す。

開発方針

第4回までのAIエッジコンテストとは異なり、今回のコンテストではRISCVの使用が提出要件となることから、コンテスト開始当初から提出のハードルが高いことが明らかであった。 トラッキング処理の代表的な手法は物体検出と追跡処理を分けて行う`Tracking-by-Detection`方式と、それらを同時に行う`Joint detection and tracking`方式に分類される。これまでのコンテストでXilinx DPUを使った物体検出の実装の経験があること、RISCVでのDNNモデルの推論処理の実装の難しさなどの理由から、`Tracking-by-Detection`方式によるトラッキング処理を選択した。また、物体検出とトラッキング処理を分けて進めることで、チームメンバー2名で分担する作業の依存が少なくなった。

HW構成

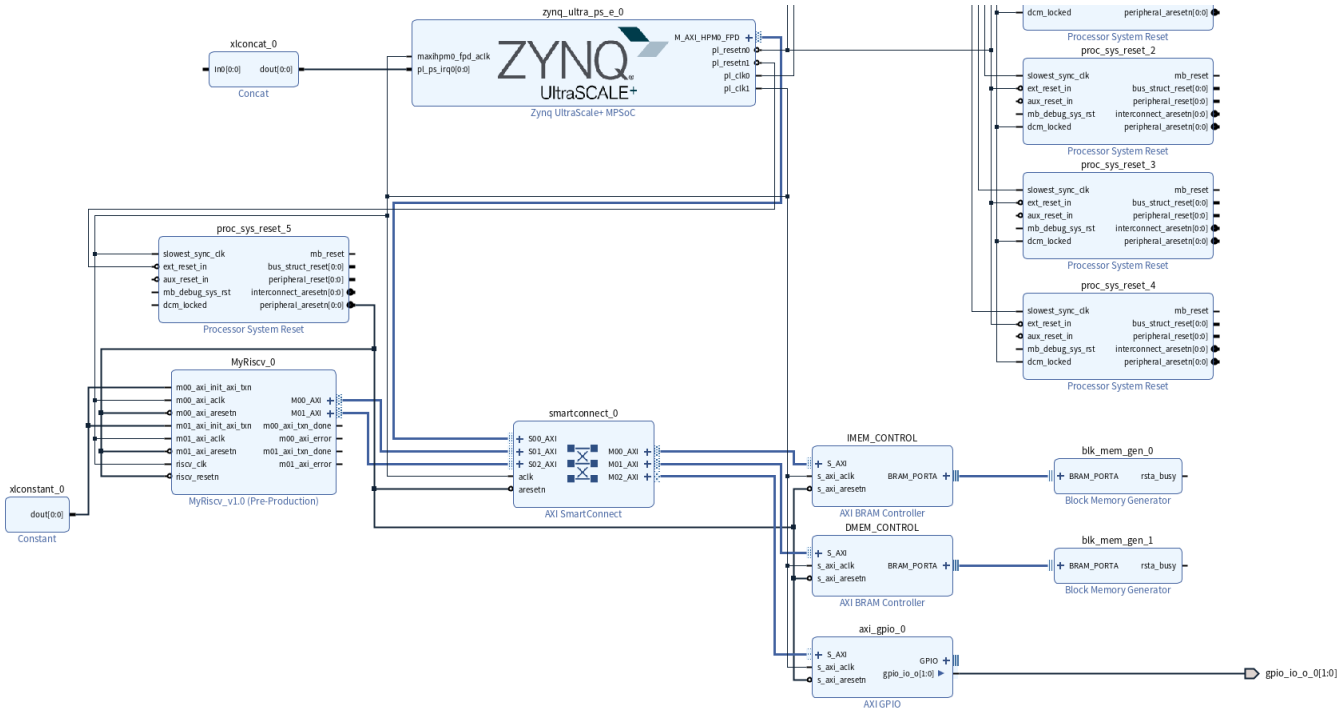
RISCVコア

コンテストの要件であるRISCVコアの実装には、コンテスト側から提供されるリファレンス環境をベースにした。 リファレンス環境で提供されるRISCVコアは、RISCVコアの実装としてVexRiscvを採用している。リ

ファレンスで提供されるコアのアーキテクチャは`rv32im`であり、浮動小数点演算命令に対応していなかった。

VexRiscvではプラグインを有効化することで様々なアーキテクチャのコアを生成することができる。`FPUPlugin`を有効化するなどのプラグインの追加を行うことで`rv32imfac`アーキテクチャのRISCVコアを生成した。リファレンス環境で提供されるRISCVコアは、RISCVコアの命令バスおよびデータバスをAXIプロトコルで接続する為に独自に実装されたVerilog HDLモジュール(`axi4lite_stream_if.v`)を使用していたが、FPUの追加に伴い、このモジュールを使用できなくなった。（FPUを使用するには命令バスとデータバスのプラグインにキャッシュ対応のものを使用する必要がありポート数が変更されるため。）独自モジュールを使用する変わりにVexRiscvの機能を用いて命令バス・データバスをAXIプロトコル化した。

RISCVコアのクロックは150Mhzの`pl_clk1`を与えて合成を行なったところ、タイミングに問題はなかった。以下にRISCVコアを搭載したFPGAブロックデザインおよびリソース使用率を示す。後述するXilinx DPUはこの時点では搭載されていない。リファレンス環境同様、RISCVコアの命令メモリ`IMEM`、データメモリ`DMEM`がBlockRAMとして作成し、AXIプロトコルで接続されている。これにより、ARM PSコアおよびRISCVコアの両方からIMEMとDMEMにアクセスすることができる。リファレンス環境から`IMEM`のデータサイズを64K、`DMEM`のデータサイズを128Kに拡張した。 なお、リファレンス環境ではRISCVコアのリセットを`AXI GPIO`経由で駆動していたが、RISCVコアのリセット時にAXIバスのリセットが駆動されず、2度目のリセット以降RISCVコアが正しく動作しない問題があった。このため設計したブロックデザインではRISCVコアおよびAXIバスのリセットをPSコアの`pl_resetn1`に接続している。



Utilization Post-Synthesis | **Post-Implementation**

Graph | **Table**

Resource	Utilization	Available	Utilization %
LUT	15797	70560	22.39
LUTRAM	2911	28800	10.11
FF	17233	141120	12.21
BRAM	52.50	216	24.31
DSP	7	360	1.94
IO	2	82	2.44
BUFG	1	196	0.51

Xilinx DPU

物体検出推論処理にはXilinx DPUを使用した。DPUはXilinxから提供されているIPコアであり、Vitis-AIを用いてDNNモデルをDPU向けのモデルに変換し、`VART(Vitis-AI Runtime)`を用いてARMコアからDPUを制御することができる。Vitis-AIを使用することで、少ない工数でDNNモデルの推論処理をFPGAにオフロードすることが可能である。DPUは処理性能とリソース使用率の異なるいくつかのコンフィグレーションを選択することができる。RISCVコアとDPUコアの両方を搭載する必要があるため、比較的リソース使用率の低いB1600を使用した。

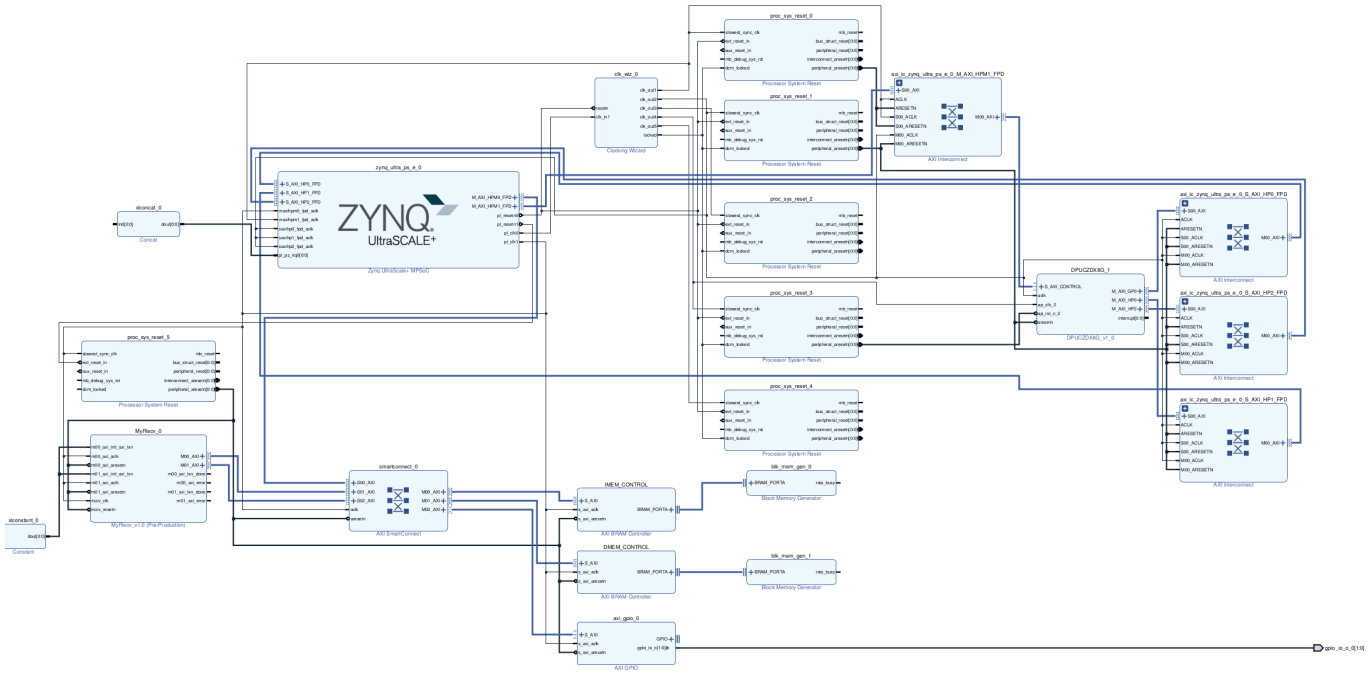
RISCV+DPUデザインの作成

Xilinx DPUを搭載するブロックデザインの生成には一般的にVitisフローを使用する。 Vitisフローではベースとなるプラットフォームデザインを選択し、その上でFPGAにオフロードしたい処理を「カーネル」として作成する。Vitisでの全体のシステムビルド時に、カーネルがベースのブロックデザインに自動的に追加され適宜クロック・リセット・データバスの配線が行われる。

上に示したRISCVコアおよびデータ・命令メモリの構成はVitisで自動的に行うことができない。そこで上に示したブロックデザインをVitisのベースデザインとし、その上でVitis上でDPUコアを追加するようにした。

DPUに与える周波数は150/300Mhzとした。（DPUにはベースとなるクロックと、その2倍のクロックの2つを与える。）下にVitisにより自動生成された、RISCV+DPUのブロックデザインおよびリソース使用率を示す。

また、ARMコアで動作させるPetalinuxシステムも、Vitisフローにおいて自動的にビルドされる。



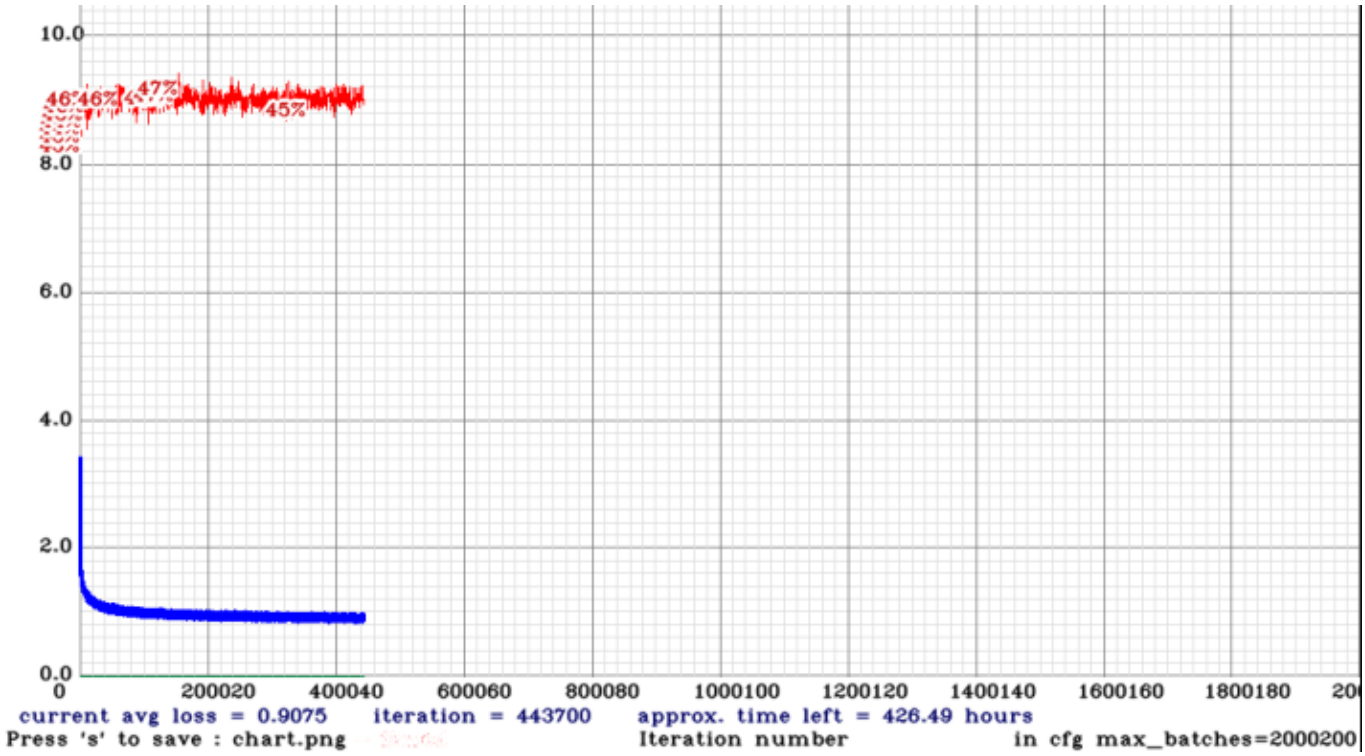
Resource	Utilization	Available	Utilization %
LUT	49038	70560	69.50
LUTRAM	6120	28800	21.25
FF	69973	141120	49.58
BRAM	211.50	216	97.92
DSP	289	360	80.28
IO	2	82	2.44
BUFG	6	196	3.06
MMCM	1	3	33.33

Name	CLB LUTs (70560)	CLB Registers (141120)	CARRY8 (8820)	F7 Muxes (35280)	F8 Muxes (17640)	CLB (8820)	LUT as Logic (70560)	LUT as Memory (28800)	Block RAM Tile (216)	DSPs (360)	Bonded IOB (82)	HDIOB_M (12)	HDIOB_S (12)	GLOBAL CLOCK BUFFERs (196)	MMCM (3)	PS8 (1)
N design_1_wrapper	49038	69973	988	1642	26	8752	42918	6120	211.5	289	2	1	1	6	1	1
design_1_i (design_1)	49038	69973	988	1642	26	8752	42918	6120	211.5	289	0	0	0	6	1	1
axi_gpio_0 (design_1_axi_gpio_0_0)	36	56	0	0	0	19	36	0	0	0	0	0	0	0	0	0
axi_ic_zynq_ultra_ps_e_0_M_AXI_HPM1_FPD (design_	1566	2364	2	3	0	472	1197	369	0	0	0	0	0	0	0	0
axi_ic_zynq_ultra_ps_e_0_S_AXI_HP0_FPD (design_1_	182	381	0	32	0	57	165	17	0	0	0	0	0	0	0	0
axi_ic_zynq_ultra_ps_e_0_S_AXI_HP1_FPD (design_1_	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
axi_ic_zynq_ultra_ps_e_0_S_AXI_HP2_FPD (design_1_	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
blk_mem_gen_0 (design_1_blk_mem_gen_0_0)	4	5	0	0	0	4	3	1	16	0	0	0	0	0	0	0
blk_mem_gen_1 (design_1_blk_mem_gen_0_1)	4	5	0	0	0	4	3	1	32	0	0	0	0	0	0	0
clk_wiz_0 (design_1_clk_wiz_0_0)	0	0	0	0	0	0	0	0	0	0	0	0	0	3	1	0
DMEM_CONTROL (design_1_axi_bram_ctrl_0_1)	151	171	0	0	0	82	151	0	0	0	0	0	0	0	0	0
DPUCZDX8G_1 (design_1_DPUCZDX8G_1_0)	31463	49926	864	1571	26	6299	28641	2822	159	282	0	0	0	1	0	0
IMEM_CONTROL (design_1_axi_bram_ctrl_0_0)	147	170	0	1	0	74	147	0	0	0	0	0	0	0	0	0
MyRiscv_0 (design_1_MyRiscv_0_0)	3776	3411	118	35	0	683	3508	268	4.5	7	0	0	0	0	0	0
proc_sys_reset_0 (design_1_proc_sys_reset_0_0)	14	33	0	0	0	7	13	1	0	0	0	0	0	0	0	0
proc_sys_reset_1 (design_1_proc_sys_reset_0_1)	16	33	0	0	0	6	15	1	0	0	0	0	0	0	0	0
proc_sys_reset_5 (design_1_proc_sys_reset_5_0)	14	33	0	0	0	8	13	1	0	0	0	0	0	0	0	0
smartconnect_0 (design_1_smartconnect_0_0)	11673	13385	4	0	0	2086	9034	2639	0	0	0	0	0	0	0	0
zynq_ultra_ps_e_0 (design_1_zynq_ultra_ps_e_0_0)	0	0	0	0	0	0	0	0	0	0	0	0	0	2	0	1

物体検出処理

物体検出処理のDNNモデルとしてtiny-YOLOv4を採用した。採用した理由としては比較的軽量なモデルであること、第3回AIエッジコンテストで多数の参加者が使用していたtiny-YOLOv3にくらべて推論精度が向上していることが挙げられる。コンテストの題材が同じ第3回AIエッジコンテストでは、入賞者は精度向上のために入力画像の解像度を元動画とほぼ解像度としていたが、今回はエッジデバイスでの高速な推論を実現する必要があったため、入力解像度は416*416とした。

tiny-YOLOv4の学習はオリジナルのリポジトリを参考に行なった。コンテストで与えられている学習画像は少ないため、同じ交通データセットであるBDD100Kを使用して最初の学習を行い、途中からSIGNATEのデータセットを使用して学習を行なった。学習過程でのlossおよびmAPカーブを以下に示す。400000iterationを超えたところで学習は打ち切った。学習中のmAPが最大になったのは150000iterationを過ぎたあたりで、mAPの値は47.1であった。なお、tiny-YOLOv3も同様に学習を行なったが、mAPのベストスコアは36.0でありtiny-YOLOv4のほうが精度が高いことが確認された。



Vitis-AIでDPU向けにtiny-YOLOv4を変換するにあたり、以下の工夫を行なった。オリジナルのtiny-YOLOv4はdarknetフレームワークを用いているが、Vitis-AIのDPU向けの変換可能な入力フレームワークはTensorflow, Caffe, PyTorchでありdarknetには直接対応していない。Vitis-AIではdarknetのモデルをCaffeのフレームワークに変換するためのスクリプトが公開されている。ただし、このスクリプトをそのまま使用するとtiny-YOLOv4の中間層に含まれるgroupレイヤがDPUで処理できないために、DPUで実行されるモデルが分割されてしまう。モデルが分割されると、groupレイヤの処理はARMコアで実行されるため、ARMコアとDPUコアでの通信が推論処理の前後だけでなく間でも必要になり推論時間が増加してしまう。この問題を解決するために、groupレイヤを演算が等価なconvolutionレイヤに置き換えるように変換スクリプトを修正した。修正した結果、tiny-YOLOv4の推論処理はすべてDPU上で実行されるようになった。

トラッキング処理

RISCVへの処理のオフロード

トラッキング処理に含まれる`lapjv(Linear Assignment Problem solver using Jonker-Volgenant algorithm)`関数をRISCV上で実行することにした。

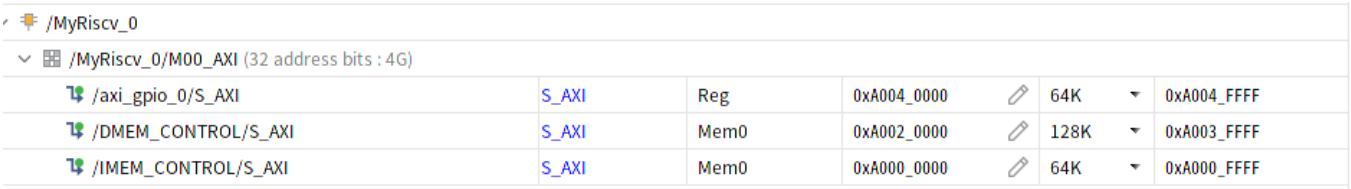
RISCV上で実行する処理に`lapjv`関数を選択したのは、元のByteTrackの実装がSTLを使用しておらず、RISCVにオフロードしやすそうであったこと、乗算や除算が含まれておらずRISCVコアでの処理もある程度の速度が見込めることが理由として挙げられる。

RISCVコアへの処理のオフロードは以下の手順で行なった。 まず、元の実装からオフロードする処理を切り出してRISCV向けのクロスコンパイラを使用してコンパイルする。クロスコンパイラにはcross-NGを使用した。`cross-NG`ではRISCVを含めた様々なアーキテクチャのCPU向けのクロスコンパイラを生成することができる。 次にRISCVコア向けに命令メモリ`IMEM`にセットすべき命令列を作成する。RISCVのスタートアップ処理やリンクに必要なファイルはリファレンス環境のものを参考にした。

RISCVコアへオフロードした処理のARMコアからの実行手順は以下のようになる：

1. `IMEM`に命令列をセット
2. `DMEM`にRISCV向けの入力データをセット
3. `pl_resetn1`をリセット・RISCVでの処理が実行開始される
4. RISCVの処理完了を待機
5. `DMEM`にあるRISCVの処理結果を取得

2.および5.では入出力に使用する`DMEM`のアドレスをプログラム内で固定することでARMコアとRISCVコアでの入出力を簡単に行なっている。 説明のために、Vivadoで設定した`DMEM Controller`の使用するアドレス空間を以下に示す。



画像に示すとおり、`DMEM Controller`は`0xA002_0000~0xA003_FFFF`に割り当てられている。この128Kのデータ領域のうち前半`0xA002_0000~0xA002_FFFF`をRISCVのプログラム実行時使用領域、後半`0xA003_0000~0xA003_FFFF`を入出力のデータ配置領域とした。命令列作成時のリンカスクリプトには`DMEM`の領域を半分の64を指定することで後半の領域を使用されないようにした。

```
MEMORY {
  ROM (rx) : ORIGIN = 0xA0000000, LENGTH = 64K  /*start from 0xA0000000 to
0xA000FFFF*/
  RAM (wx) : ORIGIN = 0xA0020000, LENGTH = 64K  /*start from 0xA0020000 to
0xA0002FFF*/
}
```

下記にARMコアで実行されるコード、およびRISCV向けにコンパイルされるコードの一部を示す。`0xA0030000`をオフセットとして入出力のデータにアクセスしていることがわかる。

- ARMコアでのRISCVの実行コード

```
#define DMEM_OFFSET 1024*16 //64K offset
// /dev/uio0 is AXI DMEM BRAM Controller
int uio0_fd = open("/dev/uio0", O_RDWR | O_SYNC);
volatile int* DMEM_BASE = (int*) mmap(NULL, 0x20000, PROT_READ|PROT_WRITE,
MAP_SHARED, uio0_fd, 0);
```

```
//set input
DMEM_BASE[DMEM_OFFSET+0] = n;
volatile float* DMEM_BASE_FLOAT = (volatile float*) DMEM_BASE;
for(int i = 0; i < n; i++){
    for(int j = 0; j < n; j++){
        DMEM_BASE_FLOAT[DMEM_OFFSET+i*n+j+1] = cost[i][j];
    }
}
//set incomplete flag
DMEM_BASE[DMEM_OFFSET+(1+n*n+n*2)] = 0;
//start RISCV
reset_pl_resetn0();
//wait completion
while(1){
    bool endflag = DMEM_BASE[DMEM_OFFSET+(1+n*n+n*2)] == n*2;
    if(endflag) break;
    usleep(1);
}
//get output
volatile int* riscv_x = &DMEM_BASE[DMEM_OFFSET+1+n*n];
volatile int* riscv_y = &DMEM_BASE[DMEM_OFFSET+1+n*n+n];
```

- RISCV向けにコンパイルされるコード

```
#define DMEM_BASE  (0xA0030000)
#define REGINT(address) *(volatile int*)(address)
#define REGINTPOINT(address) (volatile int*)(address)
#define REGFLOAT(address) *(volatile float*)(address)
int main(){
    int n = REGINT(DMEM_BASE);
    //start flag
    REGINT(DMEM_BASE+4*(1+n*n+n*2)) = n;

    volatile float cost[N_MAX*N_MAX];
    for(int i = 0; i < n*n; i++) cost[i] = REGFLOAT(DMEM_BASE+4*(1+i));
    volatile int* x = REGINTPOINT(DMEM_BASE+4*(1+n*n));
    volatile int* y = REGINTPOINT(DMEM_BASE+4*(1+n*n+n));
    int ret = lapjv_internal(n, cost, x, y);

    //end flag
    REGINT(DMEM_BASE+4*(1+n*n+n*2)) = n*2;
    while(1){
    }
    return 1;
}
```

また、4.のRISCVの処理完了待機の実現には割り込みよりも簡単なポーリング方式を使用した。特定のアドレス（上記コードでは`DMEM_BASE[DMEM_OFFSET+(1+n*n+n*2)]`）が完了を示す値になるまでARM側で待機している。

マルチスレッド化

評価用アプリケーションでは、入力動画の各フレーム画像に対して物体検出処理およびトラッキング処理を順に実行する。物体検出処理ではFPGA上のDPUコアの実行完了待ちが処理時間の大半を占めており、ARMコアが使用されていない時間が長い。我々はこれらの処理のマルチスレッド実装を行なった。トラッキング処理実行中に次フレームの物体検出を実行することで、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。

以下にマルチスレッド化前後での1フレームの画像に対する処理性能を示す。

	multithread[ms]	**sequential[ms]**
物体検出	51.66	49.05
物体追跡	29.93	27.93
全体処理	52.30	77.08

マルチスレッド化により、全体処理が77.08/52.30=**1.47**倍高速になった。 物体検出とトラッキング処理のそれぞれの時間がマルチスレッド化によって遅くなっているのはCPUの負荷がシーケンシャル実行時よりも大きいからであると考えられる。