4th AI Edge Contest / TFlite delegate を用いた実装

2021/2/13　s.yamashita

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。
- FPGA への実装は、主に SystemVerilog を用いた RTL 記述で行った。
- 推論ネットワークは TensorFlow deeplabv3 mobilenetv3 をベースに転移学習を行い、更に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て開発した。

推論ネットワーク

ベースのネットワークは TensorFlow DeepLab Model Zoo[^1] から、今回の課題に近い mobilenetv3_small_cityscapes_trainfine を選択した。
cityscape の class は 19 であるが、動作速度改善のため、今回の課題の評価対象 4 class + background の 5 class で転移学習を行った。また、信号器、歩行者の精度が上がらないので、ラベルの重みを調整した。
転移学習は Google Colab の GPU インスタンスで行った。
元のネットワークは入力サイズ 1025x2049 であるが、メモリオーバーで学習ができず、513x1025 に縮小して学習した。また、多くの追加オプションが必要であった。[^2]
転移学習後のネットワークに更に quantization aware training を行って、8bit に量子化し、TFlite FlatBuffer 形式の graph に変換した。
最終的に、精度は基準を達成できず、mIOU=0.51 (float)、0.47(uint8) に終わった。

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。
CPU での推論実行時間は約 633 ms であり、Conv2D, dwConv2D の2つの演算で 70.4% を占める。
今回はこの2つの演算を FPGA に delegate するシステムを開発した。

```
Number of nodes executed: 128  aarch64
============== Summary by node type ======================
        [Node type]  [count]  [avg ms]   [avg %]
           CONV_2D        45   285.908   45.176%  Conv2D,dwConv2Dで70.4%を
 DEPTHWISE_CONV_2D        11   159.624   25.222%  占める
   RESIZE_BILINEAR         5    80.827   12.771%  この演算を FPGA に
          ARG_MAX         1    55.767    8.812%  delegateする
               MUL        20    20.289    3.206%
        HARD_SWISH        18    13.774    2.176%
              MEAN         9     9.709    1.534%
               ADD        17     6.371    1.007%
   AVERAGE_POOL_2D         1     0.589    0.093%
  Misc Runtime Ops         1     0.019    0.003%
                   count=52  avg=632.942
```

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[^3]、これを用いて FPGA に演算を渡す。

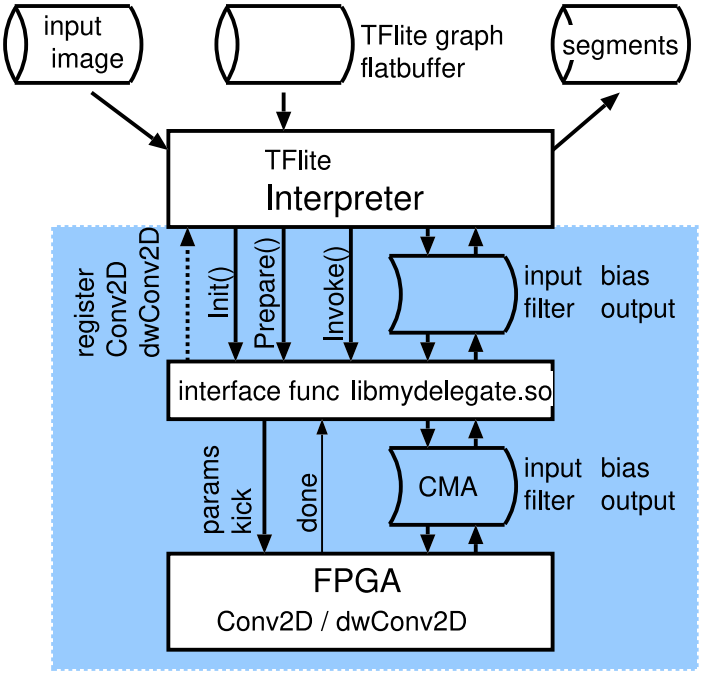


図1.システム構成

新たに作成したFPGAとのインターフェース関数 libmydelegate. 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ので、予め用意した CMA（連続メモリ）領域に一旦コピーしている。
インターフェース関数起動時 Prepare() が一度呼ばれるので、この中で予め Tensorのポインターを CMA 領域に alloc しているが、一部しか allocate できておらず、残った領域ではコピーが発生する。
FPGA に渡した Conv演算パラメータは、FPGA 側に設けた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。

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。

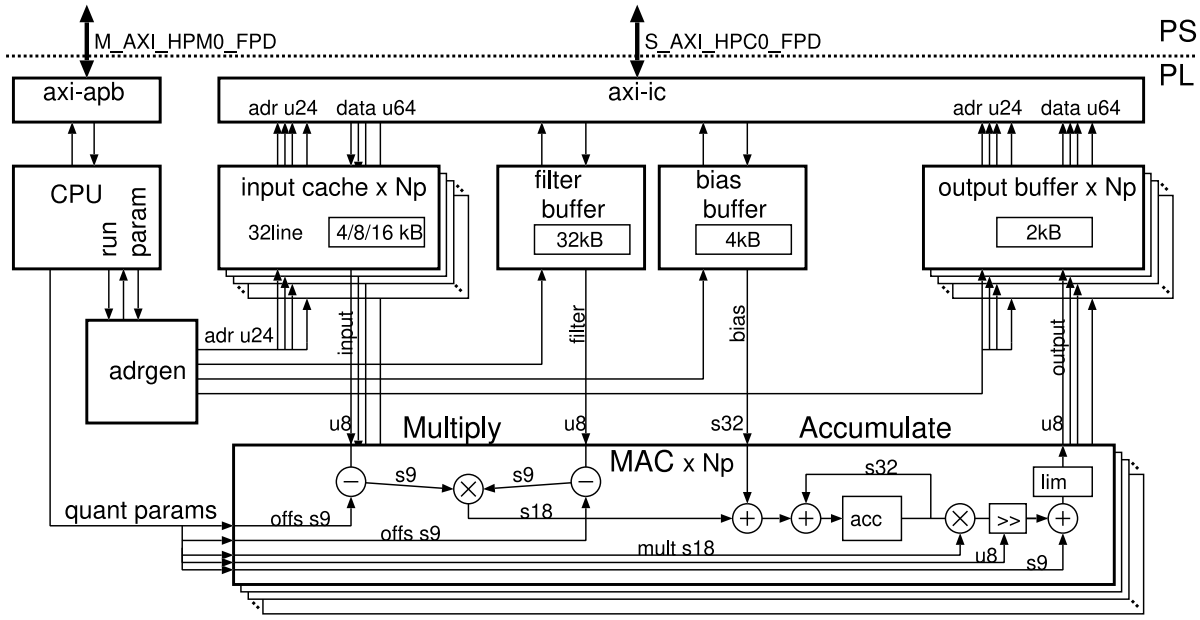


図2. PL ブロック図

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。

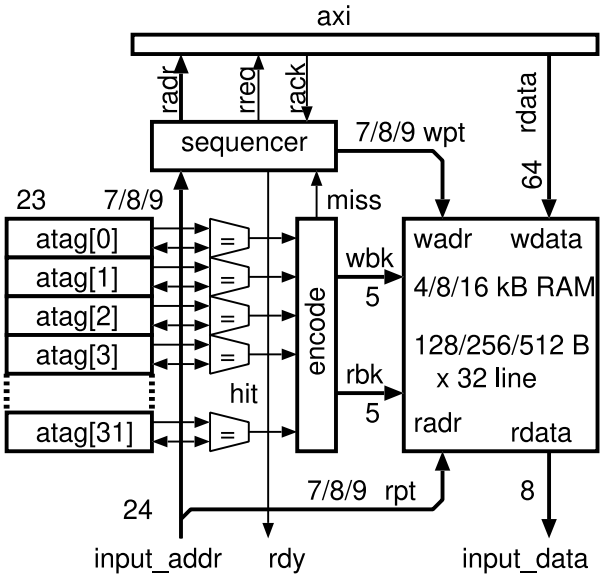
アドレス発生回路 adrgen も2種の演算で共通であり、与えるパラメータで演算を切り替えている。

input, filter, bias, output の4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 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, 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、32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)



32本のアドレスタグ atag と input_addr を比較して、全て不一致の時に axi bus にリクエストし、PS のメモリから 1 ライン分のデータを読んで cache fill し、atag を更新する。この間メモリアクセス待ちとなる。
1 ラインの転送データ数はパラメータで 128/256/512 byteを選べるようにした。

図3. input-cache ブロック図

mobilenet-v3 には 5x5 の畳み込みがあるので、離れた 25 アドレスのアクセスが起こる。
input-cache のメモリサイズは、アクセス効率に関わるが、並列数 Np と FPGA の RAM リソース、FPGAの配線性、動作速度など様々なトレードオフとなる。

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 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。

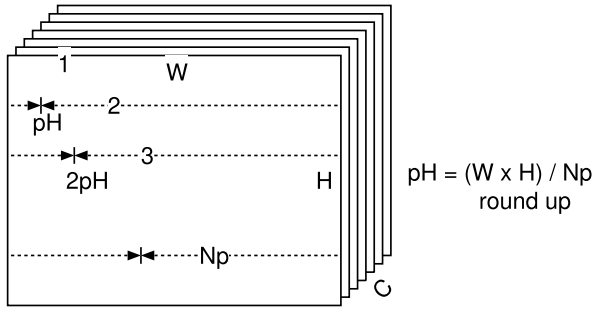


図4. Np 分割

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 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。

システム開発手順

図5 に今回の開発で行ったワークフローを示す。

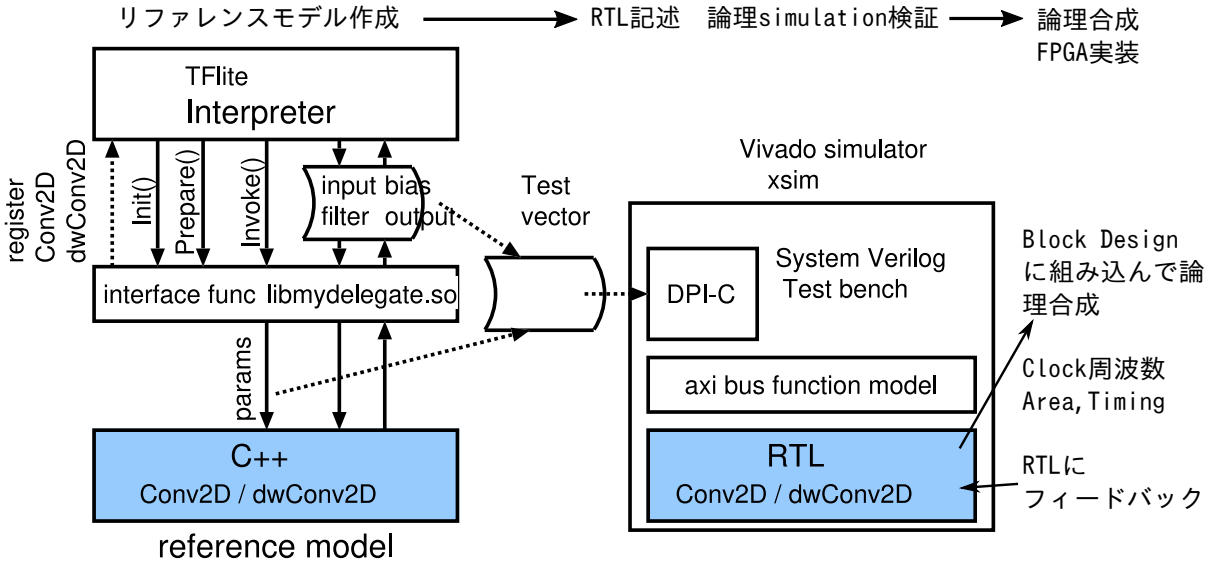


図5. システム開発手順

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。

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。

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。

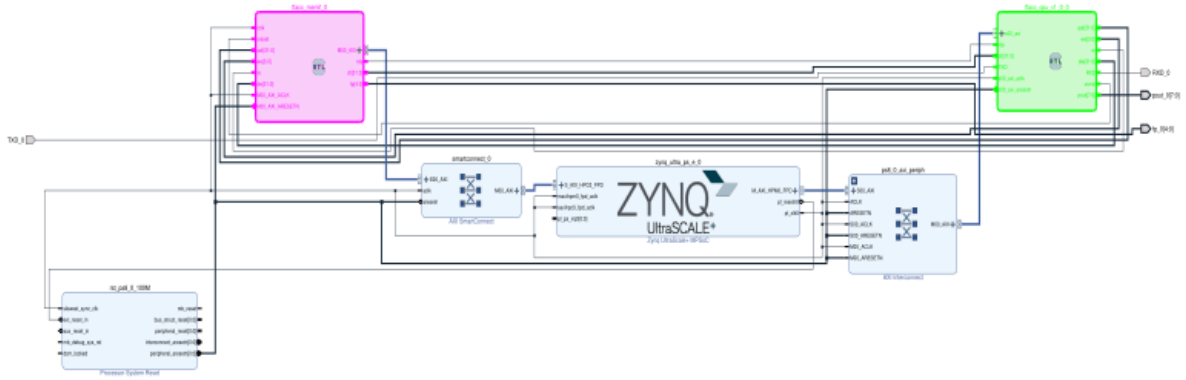


図6. Block Design

図7 は最終的な FPGA のリソース使用率である。並列数 Np = 32、クロック周波数 = 150 MHz とした。
LUT 78.8%, BRAM 97% となり、これ以上の高速化は難しい。

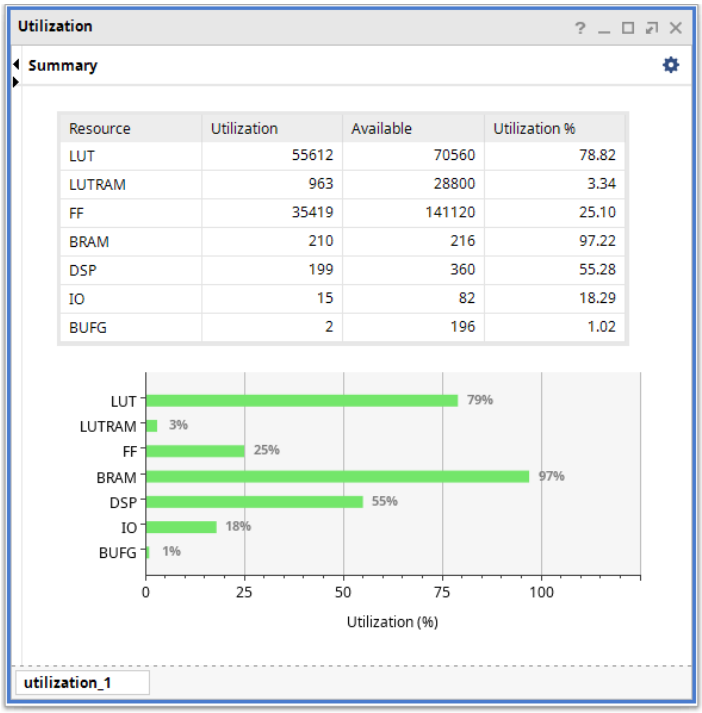


図7. Utilization

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(libmydelegate. 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 libmydelegate. 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}/libmydelegate.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
segments = interpreter.get_tensor(output_details[0]['index'])[0]
```

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)
32	150 MHz	512 byte

処理時間

処理時間は上記の条件で推論実行に約 410 ms を要した。

	前処理(resize etc)	推論	後処理(resize etc)
処理時間(ms)	17.67	410	6.69

推論処理時間詳細

推論時間は、PS の CPU と FPGA 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、分析した。
図8に結果を示す。

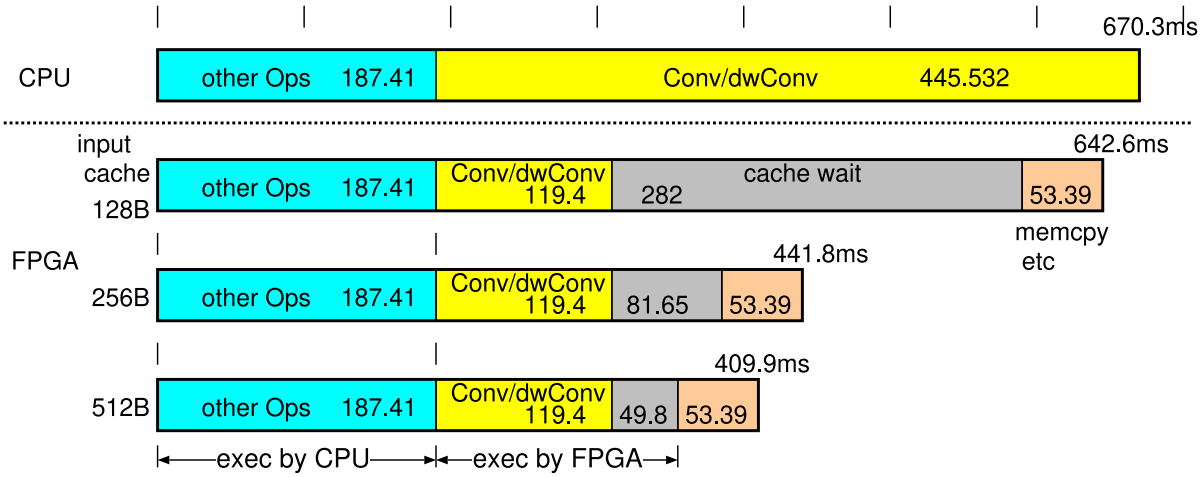


図8. 推論処理時間

グラフの Conv/dwConv(黄色) 部分は、畳み込み演算そのものの実行時間であり、CPU 実行に対して FPGA 実行が4倍弱の高速化が出来た。ただし、input-cache のキャッシュミスによるメモリアクセス待ち時間と、VM CMA 間の memcpy などのオーバーヘッドが加わり、最終的に -260ms 程度の高速化になった。

FPGA の実行時間の内、メモリアクセスの待ち時間が多くを占めており、これに input-cache の1ラインサイズが大きく影響することがわかった。

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、CPU 実行に対して -260ms 程度の高速化が得られた。
- FPGA の実行時間の内、特に input cache のメモリアクセス待ち時間が多くを占めることがわかった。
  input data のアクセス効率を上げる工夫が有効である。画像アクセスの性質を利用し、プリフェッチするなどが考えられる。
- 今回の方式では、TensorFlow Lite 用に開発したネットワークを加工することなくそのまま実行することができるのがメリットである。
  ただし、実装した Conv2D/dwConv2D の2つの演算のみ高速化される。
- 演算の種類を増やすには、それぞれの演算に応じた回路を実装するか、より汎用性をもたせた演算回路を開発する必要がある。これは、回路規模と演算スピードのトレードオフになるので、容易ではない。
  たとえば今回のネットで3番目に時間のかかっている resize_bilinear は、2x2 の畳み込み演算であるので MAC回路はそのまま使えそうだが、filter 係数を output address から演算して求める回路を新たに設けることになる。
  ArgMax、HardSwish などの演算は全く異なる構造になるので、演算回路を並列に持つことになる。