AN02008: Reducing Memory Pressure in Neural Networks using Operator Splitting

Publication Date: 2026/4/8
Document Number: XM-015073-AN v1.0.0

IN THIS DOCUMENT

1 Prerequisites . . . 1
2 Introduction: the Size of a Neural Network . . . 1
3 Analyzing the Memory Requirements of MobileNet-V2 . . . 2
4 Making the Model Use Less Memory . . . 4
5 Running the Model on Hardware . . . 7
6 Summary . . . 7

This app-note discusses running neural networks on XCORE devices that require more space for tensors than is available on the device. The xcore-ai-tools provide a method for dealing with this, called *Operator Splitting*, which reduces the size of intermediate data stored without affecting the outcome of the computation.

1 Prerequisites

This document assumes that you have installed the xmos-ai-tools package and are familiar with its basic usage.

To install the required Python dependencies, run the following command in your project directory:

```
pip install -r requirements.txt
```

2 Introduction: the Size of a Neural Network

The size of a neural network typically encompasses two variables:

- The amount of memory required to store learned parameters
- The amount of memory required to store intermediate results

The former is a number that is often mentioned in papers on neural networks. For example, resnet50 has 25,583,592 learned parameters. Note that they are also known as *weights* or *trainable parameters*. Each learned parameters typically takes up a byte of space (assuming 8-bit quantisation), therefor ResNet50 requires approximately 25 MByte of space for learned parameters.

The second number is the amount of memory that is used to store data whilst it is processing the network. The network is typically organised as a sequence of operators, where each operator consumes and produces a tensor (a multi-dimensional vector). The input tensor may, for example, be 2D RGB image or a 1D sound sample. Intermediate tensor represents features that the Neural Network algorithm has inferred, and the final output is the desired classification or embedding.

The intermediate values are known as the activations. Unlike learned parameters, the activations are read and written and therefore need to be stored in a RAM memory. The maximum number of activations that need to be stored at any one time has to be less

than the amount of RAM memory available. Unlike weights, the number is dependent on the implementation of the neural network engine, so this is not a number that is typically published. As a first proxy one can use the largest tensor that is produced in the computation, but any forward connections in the network need to be added to that.

Given the an implementation of the neural network, one can calculate the memory required, and indeed the XMOS AI Tools can output the size required and a diagram depicting which part of the network requires this amount of memory.

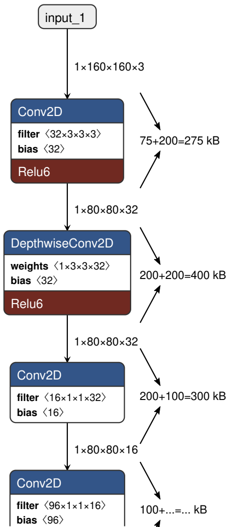

Fig. 1: First few layers of mobilenet V2

As an example we use a well known neural network that requires a large amount of memory: MobileNet-V2. It may be old and out of date, but it is well-known. We can look at a representation of the first few layers of the model as shown in *First few layers of mobilenet V2*. The graph (drawn by Netron) is annotated with the size of each tensor, and the total size of tensors going in and coming out of an operator. Assuming that the two cannot be overlapped, we can add these together to get a first idea how much memory will be required at least.

Looking a bit further down in the network we highlight the layers that include the first bypass in *First bypass in mobilenet V2*. Like before we have annotated the graph with the size of each tensor, and the total size of tensors being alive during a straightforward top-down execution of the graph (assuming zero overlap). The colours link the size calculation to each of the tensors.

3 Analyzing the Memory Requirements of MobileNet-V2

The particular flavour of MobileNet that we use has an input image of 160x160 and 1280 features in the last layer (an alpha of 1.0). We've chosen this network as it is too large to fit into memory. First we run the xcore-optimization tool on the file to establish what size memory is required:

```
% cd model
% xcore-opt -xp model --xcore-conv-err-threshold=2.1 mobilenetv2.tflite -o model.tflite -f params.flash
<unknown>:0: remark: Tensor arena size : 798120
%

% cd model
% xcore-opt -xp model --xcore-conv-err-threshold=2.1 mobilenetv2.tflite -o model.tflite -f params.flash
<unknown>:0: remark: Tensor arena size : 798120
%
```

Fig. 2: First bypass in mobilenet V2

798,352 bytes does not fit in memory of a single tile, even aside from the code and other information on the graph. We will need to reduce the size of the memory to something more manageable. As we have done the translation we can also quickly look at the size of the various parts of the model:

```
% wc -c  mobilenetv2.tflite params.flash model.tflite
 3991632 mobilenetv2.tflite
 3587136 params.flash
   45960 model.tflite
%
```

The input model that we started with was nearly 4 MBytes in size, which was split into a parameters file of just under 3.5 MBytes and a model of 46 kByte. The remaining 400 kByte comprised information that is not needed at run-time. The `params.flash` file will be stored in flash, the `model.tflite` size is an indication of the amount of memory needed to store the model information.

The next step is to run the graph optimizer again, but this time we request it to print a memory-map:

```
% xcore-opt mobilenetv2.tflite  --xcore-run-memory-analysis
MEMORY PLAN ANALYSIS
--------------------
conv_2d                0: 000......1111111111......................................... (275k), (400k)
depthwise_conv_2d      1: 2222222221111111111......................................... (400k), (400k)
conv_2d                2: 22222222233333.............................................. (300k), (300k)
conv_2d                3: .........33333................444444444444444444444444444444 (700k), (1216k)
pad                    4: 555555555555555555555555555555444444444444444444444444444444 (1216k), (1216k)
depthwise_conv_2d      5: 5555555555555555555555555555555666666....................... (766k), (766k)
[...]
```

This output tells shows us which part of the memory was occupied by which tensor after each step of the process. Each line represents a step of the process. The numbers at the end of the line states how many kilobytes were in use during this stage of the process, and the number of bytes that is needed including any fradmentation. The letters/numbers/dots specify what part of the memory was filled with a tensor, and which part was empty. For example, after step 3, the second 50% of memory was filled with tensor 4, and a small part of the first 50% with tensor 3. Tensors 3 and 4 together accounted for

700 kByte, but fragmentation meant that it needed an arena of 1216 kByte at this stage. After step 5, tensor 4 has been deleted.

You may notice that the maximum size has gone up to 1.2 MByte - do not worry, in order to print a map it has switched off all optimisations, and normally the pad operator (the fifth line) would have been done in place saving a lot of memory.

4 Making the Model Use Less Memory

In order to make the model use less memory, we are going to chop the graph up into smaller sections. We do this by instructing the xformer to split some of the operators into multiple parts that are executed sequentially, avoiding the need for large intermediate results.

The first thing to do is to look at the memory-map in detail. Below we show the first 15 layers - all subsequent layers are much smaller:

```
% xcore-opt mobilenetv2.tflite  --xcore-run-memory-analysis
MEMORY PLAN ANALYSIS
--------------------
conv_2d               0: 000......1111111111......................................... (275k), (400k)
depthwise_conv_2d     1: 2222222221111111111......................................... (400k), (400k)
conv_2d               2: 22222222233333.............................................. (300k), (300k)
conv_2d               3: .........33333................444444444444444444444444444444 (700k), (1216k)
pad                   4: 555555555555555555555555555555444444444444444444444444444444 (1216k), (1216k)
depthwise_conv_2d     5: 5555555555555555555555555555556666666....................... (766k), (766k)
conv_2d               6: ......................77......6666666....................... (188k), (766k)
conv_2d               7: ...........888888888887 7.................................... (263k), (488k)
depthwise_conv_2d     8: 999999999998888888888877.................................... (488k), (488k)
conv_2d               9: 99999999999a..........77.................................... (300k), (488k)
add                  10: b..........a..........77.................................... (113k), (488k)
conv_2d              11: b..........ccccccccccc...................................... (263k), (462k)
pad                  12: dddddddddddccccccccccc...................................... (462k), (462k)
depthwise_conv_2d    13: ddddddddddddeee.............................................. (293k), (293k)
conv_2d              14: ...........eee.............................................. (69k), (293k)
conv_2d              15: ...gggg..................................................... (88k), (175k)
[...]

OPERATION SPLIT ANALYSIS (BETA FEATURE)
---------------------------------------
The compiler suggests the following configuration for operation splitting:
Start op is 0, End op is 14 and Number of Splits is 10
Pass the following options to the compiler to get started:
--xcore-op-split-tensor-arena=1 --xcore-op-split-top-op=0 --xcore-op-split-bottom-op=14 --xcore-op-split-num-
↪splits=10
```

From the printed memory-map, we can see that the main pressure is around lines 3, 4, and 5. We can also see that the analyser suggests to split from op 0 to 14 inclusive.

The first thing we can try is to use the suggested split, and request the optimiser to split at these levels:

```
xcore-opt --xcore-op-split-tensor-arena=1 -tc 5 \
          --xcore-op-split-top-op=0 --xcore-op-split-bottom-op=14 \
          --xcore-op-split-num-splits=10 \
          --xcore-conv-channelwise-split-size=80000 \
          -xp model --xcore-conv-err-threshold=2.1 \
          mobilenetv2.tflite -o model.tflite -f weights.flash
```

These options can now be inspected in turn:

- `--xcore-op-split-tensor-arena=1` this instructs the optimiser to split the tensor arena.
- `--xcore-op-split-top-op=0 --xcore-op-split-bottom-op=14` This instructs the optimiser to split it from operator 0 to 14 inclusive.
- `--xcore-op-split-num-splits=10` this instructs the optimizer to chop the operations into 10 chunks. More chunks leads to more overhead but better potential memory savings.
- `--xcore-conv-channelwise-split-size=80000` instructs the optimizer to split the very large fully connected layer at the bottoms into chunks of 80,000 each. The default is 100,000.

The output of the memory map now has improved to be 308,848 bytes. This will fit in memory.

Splitting operators has two disadvantages: it increases the size of the program (the number of operators), and it may have to replicate computations. When a number of layers are split and computeted independently, there may be computations on the edge of the split that need to be performed twice; once in each split. This area may grow when more layers are split. This may make it beneficial to split two groups of layers separately, and allow the graph to "regroup" inbetween.

If we're looking at the above memory map in more detail, we can see that there are two narrow parts in teh memory map: between 6 and 7 and between 10 and 11:

```
[...]
depthwise_conv_2d      5: 5555555555555555555555555555555556666666...................... (766k), (766k)
conv_2d                6: .....................77......6666666...................... (188k), (766k)
conv_2d                7: ..........8888888888877................................... (263k), (488k)
[...]
add                   10: b.........a.........77..................................... (113k), (488k)
conv_2d               11: b.........ccccccccccc...................................... (263k), (462k)
pad                   12: ddddddddddcccccccccccc..................................... (462k), (462k)
[...]
```

Between 6 and 7 only tensor **7** is alive; which accounts for a very small part of the arena; and between 10 and 11 only tensor **b** is alive, which again accounts for a very small part.

The automated splitter does not yet look at these but we can try what happens when we split in three separate part; from 0 to 6, 7 to 10, and 11 to 14:

```
xcore-opt --xcore-op-split-tensor-arena -tc 5 \
          --xcore-op-split-top-op=0,7,11 --xcore-op-split-bottom-op=6,10,14 \
          --xcore-op-split-num-splits=10,10,10 \
          --xcore-conv-channelwise-split-size=80000 \
          -xp model --xcore-conv-err-threshold=2.1 \
          mobilenetv2.tflite -o model.tflite -f weights.flash
```

These options have been changed as follows

- `--xcore-op-split-top-op=0,7,10 --xcore-op-split-bottom-op=6,9,14` This instructs the optimiser to split it from operator 0 to 6 inclusive, from operator 7 to 10 inclusive, and from 11 to 14 inclusive.
- `--xcore-op-split-num-splits=10,10,10` this instructs the optimizer to chop the operations into 10 chunks for for each of the three splits.

The output of the memory map now has improved to be 268,080 bytes; saving another 40 kBytes. In addition, the run-time has gone down by 25%.

We can try and vary the number of chunks and vary where we apply the splits:

Split top(s)	0	0,7	0,7,11	0,7	0,7,11
Split bottom(s)	14	6,14	6,10,14	6,14	6,10,14
Split count(s)	10	10,10	10,10,10	8,4	8,4,2
Config in run.sh	0	3	2	4	1
Tensor arena (bytes)	308,848	267,680	268,080	259,344	258,640
Residual (bytes)	171,656	134,928	137,192	101,712	98,760
Total RAM (bytes)	460,524	403,636	404,364	385,476	**383,364**
Weights (bytes)	3,583,952	3,583,952	3,583,952	3,583,952	3,583,952
Number of operators	489	356	411	312	307
Execution time (ms)	209	154	152	149	**148**

The first three columns show the effect of applying mulitple splits whilst keeping the number of chunks the same (10) in all three cases. Note that the number of operators

and amount of memory are minimal for the dual-split, and execution time is optimal for the triple-split.

The value `10` that the xformer suggested is optimal for the single split, but when applying multiple splits it is worthwhile to cut them into fewer chunks. That reduces the overhead of the splitting in both memory and time. This is shown in the final two columns where we apply fewer splits In this case, the triple split is the most efficient in terms of both memory and time.

Note that the execution time in this example includes the time taken to load coefficients from flash which accounts for 72 ms. THis is due to the large number of parameters. As many of the parameters are used only once during the final fully connected layer, there is little scope to hide these times amongst compute. In typical networks, the final fully connected layer will be smaller reducing the parameters count and total execution time.

An excerpt of the graph produced in the right-hand column (excluding loading from flash and only the first few operators) is shown in *The split graph; the first part is split eight times, the second part is split four times, the third part if split twice, below that there are no splits.*.

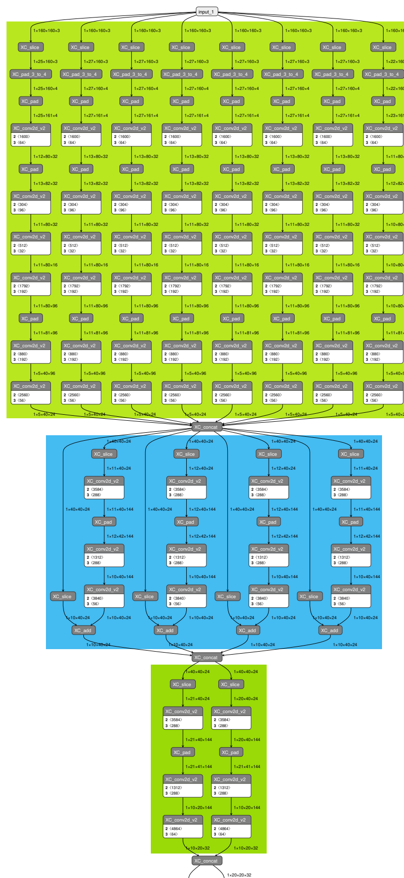

Fig. 3: The split graph; the first part is split eight times, the second part is split four times, the third part if split twice, below that there are no splits.

5 Running the Model on Hardware

This Application Note provides a source code to run the model on hardware. The required HW is the following:

- 1x *xcore.ai Explorer Development Kit'('XK-EVK-XU316)*.
- 2x Micro USB cable (for Power supply and xTag).

This application includes a CMake configuration that automates the build process. The CMake file sets up a Python virtual environment, installs all necessary Python packages and C libraries, and converts the TFLite model into a C++ header file for integration. Additional files required for linking and compiling the application with the XMOS AI tools are also generated automatically.

To compile and run the application on the XK-EVK-XU316, follow these steps:

```
# compile
cmake -G "Unix Makefiles" -B build
xmake -C build
# flash
xflash --target XK-EVK-XU316  --data app_an02008/src/weights.bin
# run
xrun --xscope app_an02008/bin/app_an02008.xe
```

After running the above commands, the model will run on the XK-EVK-XU316 board and a profile of each layer will be printed to the console. The output will include the time taken for each layer, the number of operations, and the memory used by each layer. Here is an example of the output:

```
Cumulative times for invoke()...
20    OP_XC_slice                       18471        0.18ms
10    OP_XC_pad_3_to_4                  155816       1.56ms
63    OP_XC_pad                         281739       2.82ms
86    OP_XC_ld_weights                  7120681      71.21ms
182   OP_XC_conv2d_v2                   9134347      91.34ms
19    OP_XC_add                         53282        0.53ms
5     OP_XC_concat                      6913         0.07ms
1     OP_XC_mean                        203804       2.04ms
1     OP_XC_no_op                       23           0.00ms
1     OP_XC_softmax                     11031        0.11ms

Total time for invoke() - 16986107   169.86ms
```

6 Summary

This app-note gives a brief example on how to use “operator splitting” to make larger networks fit on an xcore.

Copyright © 2026, All Rights Reserved.

XMOS Ltd. is the owner or licensee of this design, code, or Information (collectively, the ”Information”) and is providing it to you ”AS IS” with no warranty of any kind, express or implied and shall have no liability in relation to its use. XMOS Ltd makes no representation that the Information, or any particular implementation thereof, is or will be free from any claims of infringement and again, shall have no liability in relation to any such claims.

XMOS, XCORE, VocalFusion and the XMOS logo are registered trademarks of XMOS Ltd. in the United Kingdom and other countries and may not be used without written permission. Company and product names mentioned in this document are the trademarks or registered trademarks of their respective owners.