512KiB RAM Is Enough! Live Camera Face Recognition DNN on MCU

Maxim Zemlyanikin* Alexander Smorkalov*
Tatiana Khanova Anna Petrovicheva Grigory Serebryakov
Xperience AI
{maxim.zemlyanikin, alexander.smorkalov, tatiana, anna, grigory}@xperience.ai

Abstract

Small factor and ultra-low power devices are becoming more and more smart and capable even for deep learning network inference. And as the devices are "small", the challenge is becoming tougher. This paper covers full development and deployment pipeline of Face Recognition with a live camera — from model training and quantization to porting to RISC-V MCU with 512 kilobytes of internal RAM. Authors provide GreenWaves GAP8 SoC overview and the approaches for DNN model optimization and inference in the extreme environment. As the project outcome, authors were able to run Face Detection and Recognition with live QVGA camera and display preview on a battery-powered board. We will release the module for GAP8-compatible quantization[1].

1. Introduction

Nowadays most of the computer vision algorithms are based on deep learning. Deep neural networks have a great progress in terms of quality and range of tasks they are able to solve since the time they were proposed. However, having the winning solution for some task does not mean having the solution ready to be used on a real device. Edge cameras, robots, drones or any other battery-powered devices have very strict requirements on power consumption. In the past several years, the research society brought its attention to this problem, and papers devoted to energy-efficient networks started to emerge. Challenges on the energy-efficient inference and mobile-friendly DNNs (like LPIRC) also provide great growth of solutions able to cope with the production requirements in these areas. Last, but not least, the manufacturers of hardware devices devoted to the low-powered DNN inference have shipped their modules to the market, so now developers are equipped both with the scientific stack and actual hardware support for their applications.

In this paper, authors have covered the full development and deployment pipeline:

- the choice of the development platform and application deployment;
- an overview of the training procedure for extremely small, but powerful face recognition neural network;
- a novel quantization scheme applicable for the GAP8 chip;
- details of algorithm porting to the target device, including memory footprint optimization.

The key result of this project is the face recognition pipeline that meets the tough memory restrictions of 512 KiB RAM and 1.5MiB weights size and which is able to run on battery-powered device with production-ready accuracy and performance.

2. Related work

Choosing the right approach for the low-power DNN solution starts with the mobile-friendly architecture selection. There is a track of accurate yet small architectures in the literature: MobileNets family ([19], [33], [40]), SqueezeNets family ([22], [9]), ShuffleNets family ([38], [27]), ESPNets family([28], [29]), etc. Some of these architectures target the small number of computations, and some focus on reducing the memory footprint.

However, using the slim architecture typically is not enough to be ready for the inference on the low-power edge device. Further model adaptation techniques are required, such as model quantization ([23]), pruning ([17], [16]), distillation ([18]), hashing ([4]), vector quantization and Huffman coding [15]. Additionally various factorizations have been proposed to speed up pretrained networks [24], [25] and another approach is low-bit networks [6], [31], [21].

Even with originally small architecture and various techniques to compress it further, we still have business task to solve that typically differs from just image classification (the task that most architecture-introducing papers de-

*Authors equally contributed to this work

[1] https://github.com/xperience-ai/gap_quantization

scribe). The business application of this paper is smart doorbell that requires to be able to recognize people in the camera's field of view. Face recognition is a quickly evolving area with a number of public large-scale datasets available ([3], [13]). The great effort on recognition improvement is related to sophisticated loss functions ([14], [34], [30], [37], [26], [7], [35]).

3. Platform Overview

The project is aimed to create and deploy a smart doorbell application that would be highly energy-efficient. Very low energy consumption target does not leave much space for hardware selection as there are not so many hardware DNN accelerators available on the market. The authors selected GreenWaves Technologies GAP8 SoC [8], [10] as it provides general-purpose computing unit on the same SoC with DNN/CV-related hardware accelerator. According to documentation, it offers an impressive balance between the energy consumption and the number of operations per second. However, great power efficiency has its price — special modifications in the training procedures of the face detection and recognition algorithms, as well as porting of these algorithms to the device, were required to achieve the goal.

GreenWaves GAP8 is a RISC-V and PULP (Parallel Ultra-Low-Power Processing Platform) open-source platform-based IoT application processor targeted for intelligent applications and machine learning [10]. The compute part of the SoC consists of several RISC-V cores: a high-performance micro-controller core (FC) and 8 compute cores organized as a cluster for parallel execution of computationally-intensive tasks. In addition to the general-purpose cores, the SoC includes a Hardware Convolution Engine (HWCE) designed for CNN applications. The chip includes 512 KiB L2 memory accessible for all cores, 64 KiB L1 fast-access memory for compute cluster and 8 KiB of L1 memory owned by FC. GAP8 is capable to use external L3 memory and flash memory but it's not directly accessible — all operations require to copy data to L2 memory first. Authors use GAPuino board [12] with HIGHMAX HM01B0 camera and 2.8' ILI9341 SPI display.

GAP8 SDK provides support for PULP OS (native OS for PULP SoCs family), ARM®Embed^{TM}OS, FreeRTOS^{TM}, a set of drivers for recommended peripheral devices, sample applications, and an Autotiler tool. The Autotiler is a mechanism included into SDK for automated code generation for compute cluster by using a set of simple compute kernels and flexible memory access model definition with C code. The Autotiler library provides a set of highly optimized kernels for convolutional network layers and basic image processing.

4. Application Pipeline

The main application pipeline consists of 4 stages: frame capture, face detection, face recognition and user interaction (e.g. displaying message) with external event activation. In addition to regular face detection and recognition loop, application provides functionality to add known people to the trusted list using already calculated face descriptors.

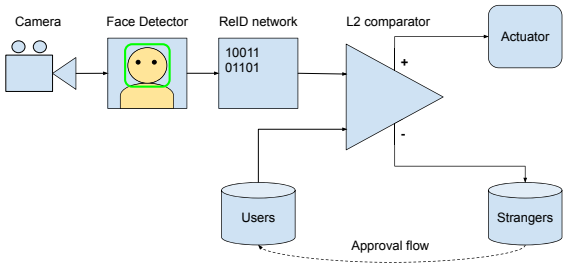

Figure 1: ReID pipeline

GAP8 SDK already includes a camera driver and an API for still image and video stream capture to L2 memory. The SDK also includes Haar cascade-based face detection demo derived from OpenCV implementation. The authors re-used it in the solution. Existing face detector provides very good trade-off between performance and quality for the doorbell scenario, since people are standing close to the camera and there is no need in perfect detection of small or distant faces.

There is one more popular part of the face recognition system — the face alignment module. Due to the resource constraints, it was not used in the application.

Face recognition quality is crucial in the application, so modern deep learning-based approaches were considered.

5. Face Recognition Model

There are three main degrees of freedom for deep learning-based face recognition solution design: data, network architecture and the loss function. We started from the resource-affected part of the solution: network architecture.

5.1. Face recognition network training

5.1.1 Network architecture

Recognition network should be integrated into an existing framework with hardware and face detection. Therefore, the network memory footprint should be as small as possible to fit L2 memory size. Several small-sized architectures were considered: MobileNet v1/v2, ShuffleNet v1/v2, SqueezeNet 1.0 and 1.1.

Table 1 shows that SqueezeNet 1.1 had the minimal number of parameters (excluding final fully connected layer) among the examined backbones. MFLOPs were calculated with 128x128 RGB input.

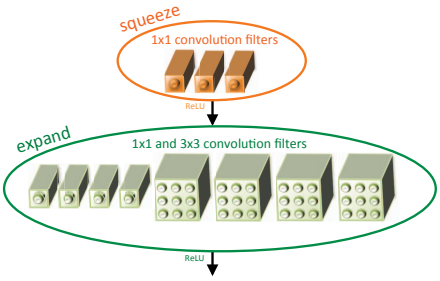

Figure 2: Fire module (image from original paper).

Model name	Number of parameters	MFLOPs
SqueezeNet 1.1	0.722	85
SqueezeNet 1.0	0.735	235
ShuffleNet	0.905	45
ShuffleNet v2	1.254	45
MobileNet v2	2.224	100
MobileNet	3.207	185

Table 1: Number of parameters and MFLOPs (excluding fully connected layer) for different models

Another advantage of SqueezeNet architecture is the absence of residual connections. Residual connections provide deep models with better training abilities, but have a major drawback for small memory footprint applications. The usage of residual connection increases (most of the time doubles) the memory needed for storing the layer activations and thus increases the memory required for the network inference.

One more constraint was the set of supported operations on the target device. For example, depthwise convolutions weren't supported by the GAP8 SDK at the moment of implementation.

As a result, SqueezeNet 1.1 architecture was chosen: it is the smallest and simplest one as it contains only 3x3, 1x1 convolutions, poolings, and a fully connected layer. For the feature extraction, we don't need a fully connected layer and it is removed in the inference time.

The application uses a grayscale camera, so the network was trained on grayscale images. Since vanilla SqueezeNet uses colored input ([22]), we added 1x1 convolution with one input and 3 output channels at the beginning of the network to employ transfer learning from the ImageNet dataset [32].

5.1.2 Data

VGGFace2 dataset [3] was used for network training. Its training part contains more than 3 million images of 8631 people. People in the dataset vary in age and ethnicity and have different poses. The standard benchmark for face recognition task is LFW [20] that was used for validation of the algorithms.

Loss function	$L2$	$L2_{norm}$	$L1$	$Cosine$
XE	0.9137	0.9303	0.9207	0.9302
XE + LS	0.9638	0.9628	0.9577	0.9643
CosFace [35]	0.8795	0.967	0.9045	0.9675

Table 2: LFW accuracy for different loss functions and distance metrics

5.1.3 Loss functions

There are several popular loss functions for face recognition training. The overview can be found in [36]. In brief, there are two big classes of the loss functions: euclidean-distance-based losses ([14], [34], [30], [37]) and softmax-based losses (cross entropy loss or its modifications: [26], [7], [35]).

The former ones embed images into the Euclidean space and minimize intra-class L2 distance and maximize inter-class L2 distance. The latter ones are trying to minimize and maximize angular or cosine distance between features.

When the networks are trained, test images are passed through the network to obtain the features. There are two standard options to calculate the distance when the features are extracted:

- to use cosine distance $1 - \frac{\mathbf{a} \cdot \mathbf{b}}{\|\mathbf{a}\| \|\mathbf{b}\|}$
- to use L2 distance $\|\mathbf{a} - \mathbf{b}\|_2$

We have tried standard Cross Entropy loss (XE), CosFace [35], Lifted Structured loss (LS) [30]. For validation, several distance metrics were tried: $L1$, $L2$, cosine distances and $L2$ along with embedding normalization $L2_{norm}$.

The network trained with CosFace [35] loss showed the best quality with cosine distance and L2 with embedding normalization. The network trained with $XE + LS$ loss functions and with L2 distance for evaluation showed slightly worse accuracy, but L2 distance is much simpler in implementation, especially if the fixed point computations are used. Thus we used the network trained with $XE + LS$ loss functions for further quantization and porting process.

5.2. Quantization

The convolution operation provided in the GAP8 SDK uses 16 or 8-bit integer values. At the same time, deep learning frameworks use floating-point numbers to represent the weights and activations in the neural network. Floating-point numbers cover a much wider dynamic range, but op-

erations with them are more resource consuming in comparison with the fixed-point numbers. Thus we need to use quantization — the way to convert floating-point numbers to fixed-point numbers while preserving the network quality.

A usual drawback of quantization is quality drop. To avoid the quality decrease, we decided to use conservative 16-bit quantization.

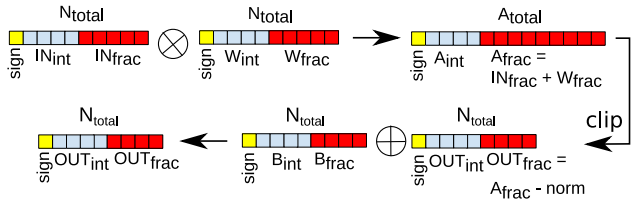

Figure 3: Convolution from GAP8 SDK

Convolution in the GAP8 SDK (Figure 3) convolves 16-bit signed integer input with weights, stores an intermediate result in 32-bit accumulator, then shifts it to the right by $norm$ bits, the least 16 bits are saved and, finally, 16-bit integer convolution bias is added to the result.

Common quantization tools, like the one from TF Lite [2] or Nervana Distiller [39], use scale parameter to represent tensors. This way we need to multiply the result of the fixed-point convolution by the scales of weight and input tensors. Since multiplication operation is redundant for the convolution implementation on the device, we are proposing an alternative quantization scheme.

The quantization steps are:

- Estimate the number of bits to store the integer part of the input, output feature maps, and weight tensors of every convolution.
  Run inference with some batches of data through the network and estimate the maximum absolute value for every tensor.
  If we have X tensor and the maximum absolute value in the X tensor is X_{max}, then we need $\lceil \log_2 X_{max} \rceil$ bits to store integer part of numbers in the X tensor.
  Let's denote number of bits to store integer parts of the input, output feature maps, and weight tensors of the convolution layer as IN_{int}, OUT_{int}, W_{int}, correspondingly.
- If we have N_{total} bits to store the activation and weight values and N_{int} bits are used for the integer part, then we have the following number of bits for fraction part:
  $$N_{frac} = N_{total} - N_{int} - 1 \quad (1)$$
  for signed numbers, or
  $$N_{frac} = N_{total} - N_{int} \quad (2)$$
  for unsigned numbers.
  Let's assume that we use signed numbers for simplicity and let's denote fraction bits for input, output feature maps and weight tensors as IN_{frac}, OUT_{frac}, W_{frac}.
- The next step is to quantize convolution weights and biases into N_{total} fixed-point values. The quantization formula is:
  $$Quantized\ value = \lfloor Value \times 2^{N_{frac}} \rceil. \quad (3)$$
  For example, for convolution weights we do the following:
  $$Quantized\ weight = \lfloor Weight \times 2^{W_{\text{frac}}} \rceil. \quad (4)$$
  Convolution bias is added to the output tensor after the convolution of the input tensor with weights. Because of that, the convolution bias should have the same number of fraction bits as output has: $B_{frac} = OUT_{frac}$.
- The next step is to choose the $norm$ parameter for convolution — the number of least significant bits in accumulator to be deleted. It is essential to keep in mind 2 things: we can't lose high orders of a number and should keep as many least significant bits as possible. The first step is to estimate the amount of bits to store the output activation result:
  $$ACC_{int}+ACC_{frac} = OUT_{int}+IN_{frac}+W_{frac} \quad (5)$$
  Then,
  $$norm = OUT_{int} + IN_{frac} + W_{frac} - N_{total} \quad (6)$$

The problem is that we also had to solve was the overflow of 32-bit accumulator. The theoretical maximum value of the output activation tensor if we do $k \times k$ convolution of C channels 16-bit signed input with 16-bit signed weight is $k^2 \times C \times (2^{15} - 1)^2$. For example, for $k = 3$ and $C = 512$ this value is much higher than maximum accumulator value $(2^{31} - 1)$, and thus an overflow may occur.

The result of a convolution of input and weight is stored in the intermediate 32-bit (or, in general, ACC_{total}) accumulator. The number of fraction bits for accumulator is
$$ACC_{frac} = IN_{frac} + W_{frac}. \quad (7)$$
The integer bits for accumulator ACC_{int} were estimated from the data
$$ACC_{int} = OUT_{int}. \quad (8)$$
It is necessary to check that
$$ACC_{int} + ACC_{frac} + 1 \leq ACC_{total}, \quad (9)$$
otherwise, the overflow will occur. If this condition is not met, then the fraction part should be reduced. The easiest way to do this is to reduce the number of the fraction bits of convolution weights:
$$\begin{aligned} ACC_{overflow} &= ACC_{int} + ACC_{frac} + 1 - ACC_{total} \\ W_{frac} &= W_{frac} - max(0, ACC_{overflow}) \end{aligned} \quad (10)$$

5.3. Ablation study

It is usually convenient for debugging to compare the network outputs between a PC and the target device. We

Model	LFW accuracy
Model with input normalization	0.9638
Model without input normalization	0.9638

Table 3: LFW accuracy with and without input normalization

Model	LFW accuracy
PyTorch model	0.9638
PyTorch quantized model	0.9633

Table 4: LFW accuracy with and without emulation

compared intermediate feature maps from PyTorch inference with the feature maps on the target device. They were significantly different from each other. We emulated the convolution kernel in PyTorch. We have processed several batches of the data and ensured that emulated result differs from board result not more than in last bit. It is caused by PyTorch roundings and we found it acceptable.

Another advantage of emulation: it allows us to measure the quality on the target device without using the device itself. One more step that is required to do it is to quantize the network input. It is possible to add quantization step into the preprocessing, but we propose a different idea. Usually, data normalization is used for preprocessing images: mean subtraction and division by the standard deviation. We propose to get rid of these operations. As a result, we would have a quantized network input and reduce the amount of image preprocessing operations. Convolution operation consists of two operations: convolution of input with weights and bias addition. As convolution is a linear transformation, then the following chain of operations is valid:

$$\begin{aligned} &Conv\left(\frac{input - mean}{std}, W\right) + b = \\ &Conv\left(\frac{input}{std}, W\right) - Conv\left(\frac{mean}{std}, W\right) + b = \\ &Conv\left(input, \frac{W}{std}\right) - Conv\left(\frac{mean}{std}, W\right) + b, \end{aligned} \quad (11)$$

where W denotes weights of the convolution and b is a bias. The last formula is equal to convolution of non-normalized input with $W' = \frac{W}{std}$ and adding $b' = b - Conv\left(\frac{mean}{std}, W\right)$. So, we should change weight and bias of the first convolution with W' and b', respectively. Table 3 shows that this operation doesn't affect the quality.

Table 4 shows accuracy values for the floating-point PyTorch model and for the quantized model with convolution emulation on PC. There is only a 0.0005 accuracy drop.

Also, we determined how different aspects affect the quality of our algorithm: bounding box origination (B), face alignment (A), quantization (Q) and the device where the face recognition network was run. We should remember that in production scenario we won't have the ground truth bounding boxes and face alignments provided along with LFW. We made an accuracy measurement on LFW with bounding boxes detected by the face detector from OpenCV dnn module [1] and without face alignment to estimate the quality decrease. It turns out that bounding box quality and the face alignment algorithm heavily affect the metric (Table 5). At the same time, quantization has only a small influence on the quality. Our quantized model on PC has 0.0007 higher quality in comparison with usual PyTorch model. Finally, we measured the quality using embeddings that were calculated on the target device. Table 5 shows that after the porting we had only 0.0023 worse quality than non-quantized model had.

Device	Q	B	A	LFW accuracy
PC	No	ground truth	Yes	0.9638
PC	No	detections	No	0.9323
PC	Yes	detections	No	0.9330
GAP8	Yes	detections	No	0.9300

Table 5: Influence of different aspects on the face recognition quality

6. Porting to Target Platform

The next step after CNN model training and quantization is porting to the target platform. The procedure includes the following steps: data conversion to a suitable layout, DNN model implementation with existing GAP8 SDK primitives with operations fusing, memory layout optimization, integration with existing components like face detection. As different stages of the pipeline like Face Detection and Face Recognition are not executed at the same time, the application can re-use the same memory buffers, thus reducing L2 footprint. Porting procedure shows that L2 utilization is the most important bottleneck for the application.

6.1. Autotiler Model

GAP8 SDK provides a set of optimized CNN kernels for popular layer implementations that allows to work with CNN application on the per-layer level. All kernels and primitives included in Autotiler library are implemented with NHWC memory layout in mind ([11]). It means that weights values and the test data exported from PyTorch should be transposed from NCHW format before the model execution on GAP8 . SqueezeNet architecture includes the following types of building blocks: 3x3 and 1x1 convolutions, average and max pooling, ReLU and concatenation.

Autotiler library provides all the compute operations in this list and also gives an opportunity to fuse blocks of convolution, ReLU, and pooling for better performance and smaller memory footprint. The last operation of activation concatenation does not require channel shuffling and can be done with proper buffer organization in the device memory.

6.2. Memory Footprint Optimization

Initial analysis shows that 512 KiB of L2 memory is not enough to store all the weights and activations together with intermediate data. The only way to achieve the goal is to store the weights in external L3 memory and download them to L2 memory before each layer inference. Also, per-layer inference requires layer input, activations, weights and biases stored in L2 during inference. The step to the next layer in the network should produce minimal extra overhead for efficient compute cluster utilization. Input, output and weight buffers should be continuous buffers, and dynamic memory allocation cannot be applied as very limited memory resources are available. The only solution found by authors is static compile-time buffer allocation on top of the continuous memory pool for each layer individually, taking into account the layer output usage on the next steps. This solution requires inference graph analysis and different techniques for different network subgraphs.

The first subgraph type in SqueezeNet architecture is a linear chain of layers, where the output of i-th layer is used as an input of i+1-th layer. For this case, chain of two buffers with a floating border is applied. Input and output are allocated starting from the beginning of the memory pool and starting from the floating border. Floating border address is selected to fit the current input of the layer and output of the next layer, to not reallocate already calculated data and achieve continuous memory buffers for free. The piece of memory after the second buffer is used as an area for weight and bias loading from external L3 memory. Border address is calculated by formula (12) and illustrated by Figure (4).

$$border = max(ActivationSize_{i+1}, InputSize_i) \quad (12)$$

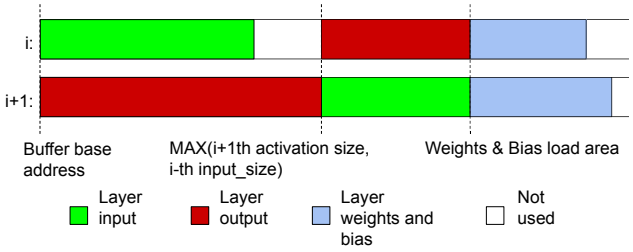

Figure 4: Simple convolution chain memory layout

The second subgraph type is SqueezeNet Fire module. The module consists of squeeze convolution and two expand convolutions that use the squeeze operation output as input. The results of two expand convolutions are concatenated and have to be stored in L2 in continuous buffer during both expand stages. The approach with a floating memory border can be adapted to the Fire module too, but it includes three layers instead of two in the previous case.

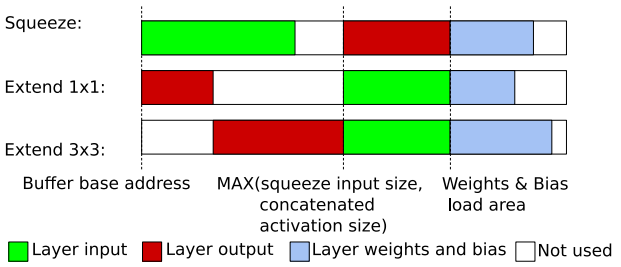

Figure 5: Fire module memory layout

The scheme (Figure 5) illustrates how to use the left part of the buffer as squeeze input and as a place for concatenation of the expand layers. The right side of the buffer plays the role of the output of squeeze layer and the input for expand layers and cannot be thrown away as intermediate data. The border address is calculated by the formula (13)

$$\begin{aligned} border = max(&SqueezeActivationSize, \\ &Expand1x1ActivationSize+ \\ &Expand3x3ActivationSize) \end{aligned} \quad (13)$$

It's important to mention that Fire module is compact in terms of memory, and therefore two sequential Fire modules use base memory pool address as input and don't affect layout of each other.

By using two memory management approaches, we can estimate minimal L2 buffer size needed for per-layer network inference without memory movements and intermediate data swapping with external memory. The maximum amount of memory is consumed by the first two convolutions in the architecture and all Fire modules. Memory consumption statistics are shown in the table (6).

6.3. Integration into Pipeline

CNN inference part introduces very significant memory footprint and requires to reuse the other parts and memory buffers as much as possible.

The first candidate for memory re-usage is the original camera frame buffer. The application stops the camera between the analysis of sequential frames. As soon as the face area is extracted for the CNN inference, the remaining part of the frame memory can be re-used. The application uses the same large memory pool for CNN part and disposes the frame data before the network inference.

The next memory allocation hot spot is the Face Detection block that consists of constant cascade data and inter-

Block Name	Memory Consumption, bytes
Intro convolutions	221324
Fire 1	295328
Fire 2	106912
Fire 3	203584
Fire 4	113472
Fire 5	42720
Fire 6	208608
Fire 7	351872
Fire 8	351872
Maximum	**351872**

Table 6: Per-block memory consumption in GAP8

mediate per-frame data like pyramid layers and integral images. The intermediate data used by the face cascade is disposable and can be thrown away before the CNN inference that makes memory pool re-usable again. Cascade data consists of a lot of small buffers, and its swap to external memory takes significant time for recovery after the CNN inference. Therefore, authors decided to store it in the L2 area constantly. The only important area that is needed for both face detection and CNN inference is the buffer for the network input data. Its location in the pool is pre-defined and re-used on all CNN inference calls to exclude extra memory operations. Rough memory map is presented on Figure (6).

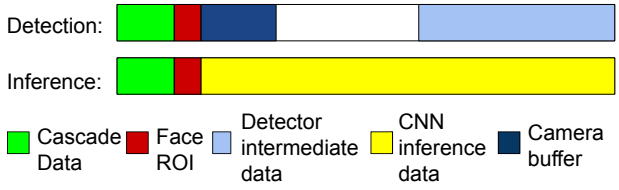

Figure 6: Fire module memory layout

7. Conclusion

The authors implemented a face detection and recognition pipeline that runs on a battery-powered IoT device. Using a small network architecture and a novel quantization pipeline, the authors created a face recognition network with only 1.5 Mb of weights. The quantization also allowed to switch to 16-bit fixed point integer computations without accuracy loss.

On the production side, the authors engineered the pipeline that squeezes into 512 KiB L2 memory and works under 1 second on the target device. The solution fits the production requirements on the recognition quality and inference speed.

8. Future work

The authors now continue the performance optimizations. The next step is to to fuse the first two convolutions before the Fire modules to reduce the amount of computations.

GAP8 SoC provides HWCE [5] module for hardware acceleration of convolution networks that was not used in the project due to limited support of strides and paddings in the SDK API. This hardware unit is an option for application performance optimization, but some SDK API changes and network architecture modifications are required.

Besides the computational optimizations, L3 data transactions are found to be a time-consuming step. Authors are going to use preemptive micro-DMA calls to do it in parallel with layer computations in all cases where it's possible. The current Autotiler library version already has this feature and can be used in the next versions of recognition pipeline.

References

[1] Opencv dnn face detector. `https://github.com/opencv/opencv/tree/master/samples/dnn`.

[2] Tensorflow lite. `https://www.tensorflow.org/lite`.

[3] Q. Cao, L. Shen, W. Xie, O. M. Parkhi, and A. Zisserman. Vggface2: A dataset for recognising faces across pose and age. In *2018 13th IEEE International Conference on Automatic Face & Gesture Recognition (FG 2018)*, pages 67–74. IEEE, 2018.

[4] W. Chen, J. Wilson, S. Tyree, K. Weinberger, and Y. Chen. Compressing neural networks with the hashing trick. In *International Conference on Machine Learning*, pages 2285–2294, 2015.

[5] F. Conti and L. Benini. A ultra-low-energy convolution engine for fast brain-inspired vision in multicore clusters. In *Proceedings of the 2015 Design, Automation & Test in Europe Conference & Exhibition*, pages 683–688. EDA Consortium, 2015.

[6] M. Courbariaux, Y. Bengio, and J.-P. David. Training deep neural networks with low precision multiplications. *arXiv preprint arXiv:1412.7024*, 2014.

[7] J. Deng, J. Guo, N. Xue, and S. Zafeiriou. Arcface: Additive angular margin loss for deep face recognition. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 4690–4699, 2019.

[8] E. Flamand, D. Rossi, F. Conti, I. Loi, A. Pullini, F. Rotenberg, and L. Benini. Gap-8: A risc-v soc for ai at the edge of the iot. In *2018 IEEE 29th International Conference on Application-specific Systems, Architectures and Processors (ASAP)*, pages 1–4, July 2018.

[9] A. Gholami, K. Kwon, B. Wu, Z. Tai, X. Yue, P. Jin, S. Zhao, and K. Keutzer. Squeezenext: Hardware-aware neural network design. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition Workshops*, pages 1638–1647, 2018.

[10] GreenWaves Technologies. GAP8 Hardware Reference Manual.

[11] GreenWaves Technologies. GAP8 Software Development Kit Documentation.

[12] GreenWaves Technologies. GAPuino User's Manual.

[13] Y. Guo, L. Zhang, Y. Hu, X. He, and J. Gao. Ms-celeb-1m: A dataset and benchmark for large-scale face recognition. In *European Conference on Computer Vision*, pages 87–102. Springer, 2016.

[14] R. Hadsell, S. Chopra, and Y. LeCun. Dimensionality reduction by learning an invariant mapping. In *2006 IEEE Computer Society Conference on Computer Vision and Pattern Recognition (CVPR'06)*, volume 2, pages 1735–1742. IEEE, 2006.

[15] S. Han, H. Mao, and W. J. Dally. Deep compression: Compressing deep neural networks with pruning, trained quantization and huffman coding. *arXiv preprint arXiv:1510.00149*, 2015.

[16] S. Han, J. Pool, J. Tran, and W. Dally. Learning both weights and connections for efficient neural network. In *Advances in neural information processing systems*, pages 1135–1143, 2015.

[17] B. Hassibi and D. G. Stork. Second order derivatives for network pruning: Optimal brain surgeon. In *Advances in neural information processing systems*, pages 164–171, 1993.

[18] G. Hinton, O. Vinyals, and J. Dean. Distilling the knowledge in a neural network. *arXiv preprint arXiv:1503.02531*, 2015.

[19] A. G. Howard, M. Zhu, B. Chen, D. Kalenichenko, W. Wang, T. Weyand, M. Andreetto, and H. Adam. Mobilenets: Efficient convolutional neural networks for mobile vision applications. *arXiv preprint arXiv:1704.04861*, 2017.

[20] G. B. Huang, M. Ramesh, T. Berg, and E. Learned-Miller. Labeled faces in the wild: A database for studying face recognition in unconstrained environments. Technical Report 07-49, University of Massachusetts, Amherst, October 2007.

[21] I. Hubara, M. Courbariaux, D. Soudry, R. El-Yaniv, and Y. Bengio. Quantized neural networks: Training neural networks with low precision weights and activations. *The Journal of Machine Learning Research*, 18(1):6869–6898, 2017.

[22] F. N. Iandola, S. Han, M. W. Moskewicz, K. Ashraf, W. J. Dally, and K. Keutzer. Squeezenet: Alexnet-level accuracy with 50x fewer parameters and¡ 0.5 mb model size. *arXiv preprint arXiv:1602.07360*, 2016.

[23] B. Jacob, S. Kligys, B. Chen, M. Zhu, M. Tang, A. Howard, H. Adam, and D. Kalenichenko. Quantization and training of neural networks for efficient integer-arithmetic-only inference. In *The IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, June 2018.

[24] M. Jaderberg, A. Vedaldi, and A. Zisserman. Speeding up convolutional neural networks with low rank expansions. *arXiv preprint arXiv:1405.3866*, 2014.

[25] V. Lebedev, Y. Ganin, M. Rakhuba, I. Oseledets, and V. Lempitsky. Speeding-up convolutional neural networks using fine-tuned cp-decomposition. *arXiv preprint arXiv:1412.6553*, 2014.

[26] W. Liu, Y. Wen, Z. Yu, and M. Yang. Large-margin softmax loss for convolutional neural networks. In *ICML*, volume 2, page 7, 2016.

[27] N. Ma, X. Zhang, H.-T. Zheng, and J. Sun. Shufflenet v2: Practical guidelines for efficient cnn architecture design. In *Proceedings of the European Conference on Computer Vision (ECCV)*, pages 116–131, 2018.

[28] S. Mehta, M. Rastegari, A. Caspi, L. Shapiro, and H. Hajishirzi. Espnet: Efficient spatial pyramid of dilated convolutions for semantic segmentation. In *Proceedings of the European Conference on Computer Vision (ECCV)*, pages 552–568, 2018.

[29] S. Mehta, M. Rastegari, L. Shapiro, and H. Hajishirzi. Espnetv2: A light-weight, power efficient, and general purpose convolutional neural network. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 9190–9200, 2019.

[30] H. Oh Song, Y. Xiang, S. Jegelka, and S. Savarese. Deep metric learning via lifted structured feature embedding. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 4004–4012, 2016.

[31] M. Rastegari, V. Ordonez, J. Redmon, and A. Farhadi. Xnor-net: Imagenet classification using binary convolutional neural networks. In *European Conference on Computer Vision*, pages 525–542. Springer, 2016.

[32] O. Russakovsky, J. Deng, H. Su, J. Krause, S. Satheesh, S. Ma, Z. Huang, A. Karpathy, A. Khosla, M. Bernstein, et al. Imagenet large scale visual recognition challenge. *International journal of computer vision*, 115(3):211–252, 2015.

[33] M. Sandler, A. Howard, M. Zhu, A. Zhmoginov, and L.-C. Chen. Mobilenetv2: Inverted residuals and linear bottlenecks. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 4510–4520, 2018.

[34] F. Schroff, D. Kalenichenko, and J. Philbin. Facenet: A unified embedding for face recognition and clustering. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 815–823, 2015.

[35] H. Wang, Y. Wang, Z. Zhou, X. Ji, D. Gong, J. Zhou, Z. Li, and W. Liu. Cosface: Large margin cosine loss for deep face recognition. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 5265–5274, 2018.

[36] M. Wang and W. Deng. Deep face recognition: A survey. *arXiv preprint arXiv:1804.06655*, 2018.

[37] Y. Wen, K. Zhang, Z. Li, and Y. Qiao. A discriminative feature learning approach for deep face recognition. In *European conference on computer vision*, pages 499–515. Springer, 2016.

[38] X. Zhang, X. Zhou, M. Lin, and J. Sun. Shufflenet: An extremely efficient convolutional neural network for mobile devices. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 6848–6856, 2018.

[39] N. Zmora, G. Jacob, L. Zlotnik, B. Elharar, and G. Novik. Neural Network Distiller, June 2018.

[40] B. Zoph, V. Vasudevan, J. Shlens, and Q. V. Le. Learning transferable architectures for scalable image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 8697–8710, 2018.