SOLVING SMART PARKING REQUIREMENTS WITH ALPR

Dr. Andrew Cavanaugh
XMOS

Professor Shaoming Zhang
Tongji University, Shanghai, China

To find out more, please visit the XMOS stand in the exhibition, or navigate to xmos.ai

ABSTRACT

Today's automatic license plate recognition (ALPR) systems are designed for fast moving, multilane recognition and come with a significant price tag. Requiring cloud connectivity and large amounts of processing, these complex systems are overkill for smart parking, and come rife with privacy concerns.

In partnership with the team from Cloudtop, led by prof. Shaoming Zhang at Tongji University, XMOS has developed a cost-effective, low power ALPR system that can read slow moving license plates from a distance of 3 to 5 meters, with high accuracy.

Running on two tiles of xcore.ai silicon (XMOS' 3rd generation microprocessor), each containing 8 hardware threads and 512kB of SRAM, the solution can deliver up to a 98% recognition accuracy at an unprecedented price-point. This is enabled by a single shot detector (SSD) network to find the license plate bounding box and a lightweight character recognition network to decode the province code and alphanumeric string that follows. These networks were scaled for the edge from large models designed for moving vehicles at longer ranges.

In this poster, we highlight our two-network solution, and the innovative optimizations XMOS uses to deliver a super light neural network 8-bit model with plate detection and OCR.

THE ALPR PROBLEM

ALPR is becoming ubiquitous in parking and tolling applications for space reservation, fee collection and violation enforcement. All ALPR systems photograph one or more cars to determine what license plate ID(s) are at that location at a given point in time. Figure 1 shows an example cloud-based solution with nominal energy and data requirements for each stage.

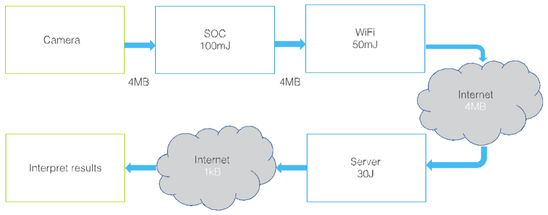


Figure 1

If the parameters of the problem allow for a Tiny ALPR solution, we can simplify this diagram to that of Figure 2 where we still have network connectivity, but only need to send entire images in the exceptional case where a violation is reported, and photographic evidence is needed.

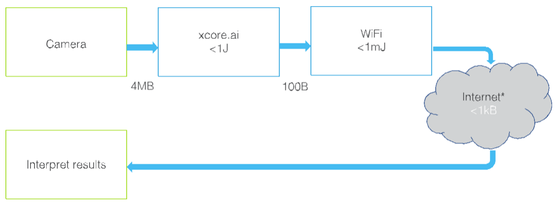


Figure 2

* Local WiFi, Zigbee , or Bluetooth would also be options with potentially lower power.

TINY ALPR WITH XMOS ON XCORE.AI

Cloudtop adapted an existing solution that used two large networks to perform ALPR on fast moving vehicles at traffic intersections to fit on xcore.ai by focusing on stationary vehicles in tightly constrained locations with well controlled lighting inside parking garages. Figure 3 shows the locations of different processing and storage blocks in the xcore.ai chip and external Flash memory.

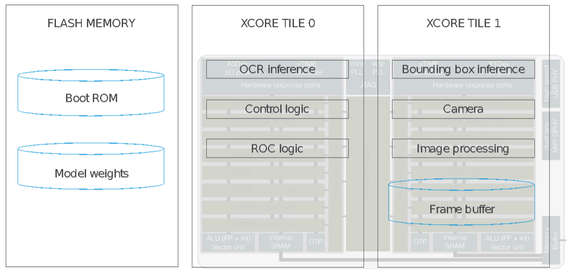


Figure 3

NETWORKS

The first network is a single shot detector (SSD) that locates the most likely bounding box of the license plate from a picture of a car. The input tensor for this network is only a subset of what the camera sensor captures, but this data is cropped to a region of interest as the data are streamed in. This decreases the size of the input tensor and of the frame buffer. If the problem were even more constrained a smaller sensor with a narrow FOV lens could further reduce compute and memory requirements.

NETWORK	DETECTION (SSD)	RECOGNITION
INPUT TENSOR SIZE	160 x 160 x 3	32 x 128 x 3
WEIGHTS	~350kB	~100kB
OUTPUT TENSOR SIZE(S)	1 x 460 x 4 and 1 x 460 x 2	1 x 16 x 66

The second network performs character recognition across the bounding box returned by the first network. In order to minimize memory usage a second image is taken at this point with the region of interest updated based on the bounding box. This network returns the most likely symbols in the license plate in 16 overlapping positions, these raw estimates are then filtered to generate a province code followed by 7 alphanumeric values.

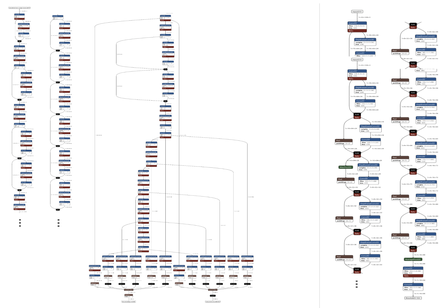


Figure 4: Detection and recognition network diagrams

RETHINKING THE TENSOR ARENA

Shrinking these networks was an iterative process and the first few iterations leveraged our LPDDR interface which allows very large models to be run efficiently by placing the model weights, arena, or both into an external memory. This increases energy consumption and cost, so we decided to limit ourselves to internal SRAM and our existing external Flash memory. A load_from_flash() operator allowed us to make our models orders of magnitude smaller (with the weights factored out into flash storage). This made the network small enough to put into the Tensor Arena along with the current layer's input, output, and weights.

This allowed us to handle networks that would not fit in available SRAM and leveraged the memory planner to remove redundant storage, but the Tensor Arena size did not necessarily shrink, and tended to be dominated by a few very wide layers in a given network. To solve this, we introduced a strided_slice() operator for splitting and a concatenate() operator for joining the network around these locations.

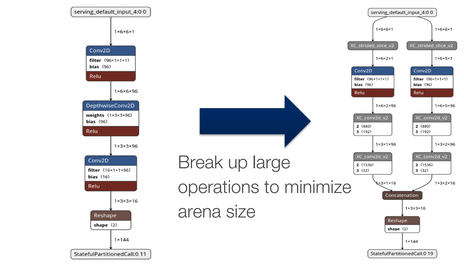


Figure 5: Example operator splitting substitution

Depending on threading and execution plans these splits could be used to boost performance or to shrink the Tensor Arena size, in order to create a Tiny ALPR we chose the latter.

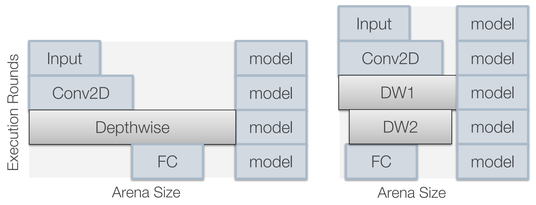


Figure 6: Operator splitting adds additional steps to the execution plan but reduces the required size of the Tensor Arena

HARDWARE PROTOTYPE

To test real world performance, we made a prototype for a long-term test complete with enclosure and the customer's preferred backhaul interface, which in this case was UART.

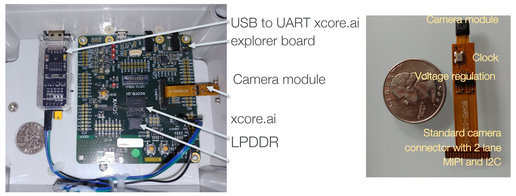
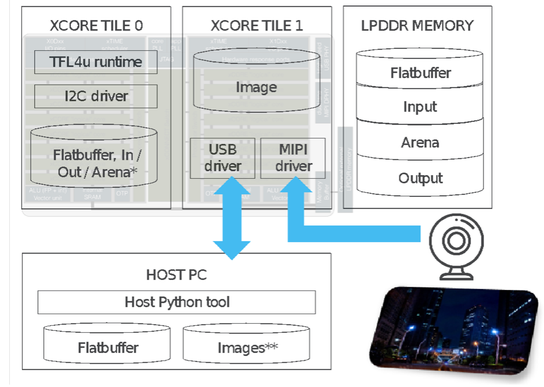


Figure 7: Prototype system for customer evaluation

XCORE.AI DEPLOYMENT FRAMEWORK

Evaluating the performance of networks and cameras on xcore.ai is made easier by two key software packages from XMOS: xcore-opt and AI Server. All that is required to use these tools is a model that can be converted to TensorFlow Lite and an xcore.ai explorer board.

A .tflite file can be converted using xcore-opt with a familiar python interface similar to that of TensorFlow, or by calling the xcore-opt executable in a similar manner to LLVM's opt. Converted models can also be executed on a host to check for any issues (including with the memory plan) before deploying the model on hardware.



* SRAM-only configuration available for small models
** Images can be live from camera or sent from host

Figure 8: Block diagram showing common AI Server configurations

To deploy the model on hardware run the AI Server executable on an xcore.ai explorer board, then, using the Python interface you can send models, images, and tell the server to do inference or to acquire data. Data can be acquired from the host or from a camera, and new cameras can be added to the AI server example application by defining three functions to send the appropriate settings / commands to the camera module over I2C:

- Init()
- Capture()
- stop() (optional)

Once these are defined for your camera most evaluation of the imaging and inferencing system can be done with Python via USB using either live or pre-captured images to test accuracy, throughput and memory usage. By removing the USB interface and substituting I2C or just writing the application logic on the xcore this prototype can turn into production ready firmware and tuned for performance and memory usage.

ACKNOWLEDGEMENTS

Tongji University

Cloudtop network renderings generated by netron.app