2020 embedded VISION summit®

Deploying AI Software to Embedded Devices Using Open Standards

Andrew Richards
Codeplay Software

September 2020

codeplay®

Understanding the Challenge

codeplay

What Kinds of “AI” are There and How Does That Affect Deployment?

If you want to get good performance-per-Watt, then each different type of AI algorithm should map to a different type of processor with different characteristics

Many applications involve multiple types of AI algorithm

Deep learning inference	• Low-precision inference, fixed sizes • .... (varying complexity of models) ... • Floating-point inference
Machine vision	• Classical machine vision • SLAM
Sensor fusion	• Kalman filters • Mapping from multiple sources • Occupancy grids
Decision making and planning	• Path-planning
Deep learning training	• Sometimes deployed: e.g. federated learning, or on-device learning for privacy

Types of Deep Learning Inference

Deep learning “inference” applies a trained neural network to real-world data

Some kinds of deep learning algorithm make much tougher demands on the software tools and processor hardware than other kinds

Image Classification

- Given an image of an object, recognizes that object
- Fixed-sizes: only one image and one object
- Good fit for very fixed-function hardware
- Low precision (e.g. 8-bit) is enough
- Widely supported by software tools & hardware

Object detection

- Given an image, find and classify all the objects in the image
- Variable numbers of objects in images, so this complicates the software
- Needs a mix of fixed-function hardware and some control software
- A surprising number of software tools & deep learning processor don’t handle this well

Semantic Segmentation

- Given an image, find objects, *but classify all pixels* by what object they are part of
- Variable numbers of objects in images
- Deconvolution operations to map classifications back to pixels
- This is much tougher for hardware and software to support: much rarer for current deep learning processors to support this

Behavior Prediction

- Autonomous systems need to predict where a vehicle or person will be in the future, not just where it is now
- These networks usually require higher-precision arithmetic than vision networks

Types of Machine Vision Algorithm

Not everything in an AI system is deep learning

There are lots of good reasons to use non-deep-learning algorithms with deep-learning

- Safety (2 approaches for redundancy); Extracting different information (e.g. spatial info); Explainability

SLAM

- Simultaneous Localization And Mapping
- Enables you to create a 3D map of the world
- Very important for mapping obstacles that deep learning won't recognize as objects

Object detection

- There are both deep learning and classical machine vision approaches to object detection
- Safety may require comparing a deep-learning with a non-deep-learning object detection

Pre-processing

- It is very common to have to pre-process data before putting it into deep learning inference
- e.g. lens distortion correction

Post-processing

- Many neural networks need to post-process the data to get useful data out
- e.g. semantic segmentation marks out where a road is visible, but you need to convert that into a 3D road representation

Sensor Fusion and Path Planning

The previous algorithms were all about sensing: detecting what is going on in the world

An autonomous system will then need to fuse the different sources of perception data together to try to create an accurate picture of the world

Finally, an autonomous system needs to plan how to act in its environment

Object Tracking

- Object recognition from different sensors and perception algorithms needs to be combined to understand where objects are and where they are going

3D Maps

- Different objects, terrains and roads need to be mapped
- This could combine existing mapping data with new perceived data

Path Planning

- An autonomous system needs to work out a plan of what it can do next, taking into account what other objects may do

Types of AI Processor Core

CPUs

- Easy to program
- Best for low-latency operations
- Best for small workloads (very small neural networks, or many sensor fusion algorithms)

GPUs

- Very high processing performance on large workloads
- Sometimes good, sometimes great software tools
- Terrible at latency-sensitive tasks (e.g. small Kalman filters)
- Not so great at sparse operations

FPGAs

- Very efficient at custom tasks
- Not as powerful at general compute operations as CPUs or GPUs
- Very good at latency-sensitive tasks
- Tools are tough, but getting better

To get best performance per Watt, you want to map algorithms to the right cores

DSPs

- Excellent at signal processing
- Being brought into a range of vision tasks
- Harder to program than GPUs, but may give greater efficiency

Deep learning accelerators

- Very high performance for the deep learning operations they support
- Not, usually, suitable for general programmability

Often, a system-on-chip has multiple types of processor core on the chip

Why Use Open Standards?

If you use proprietary tools, it's very hard to move your algorithms to the processor that will best perform for your application

Open standards let you find the best cost/performance/power for your application

Open standards let you support your application over a longer period of time

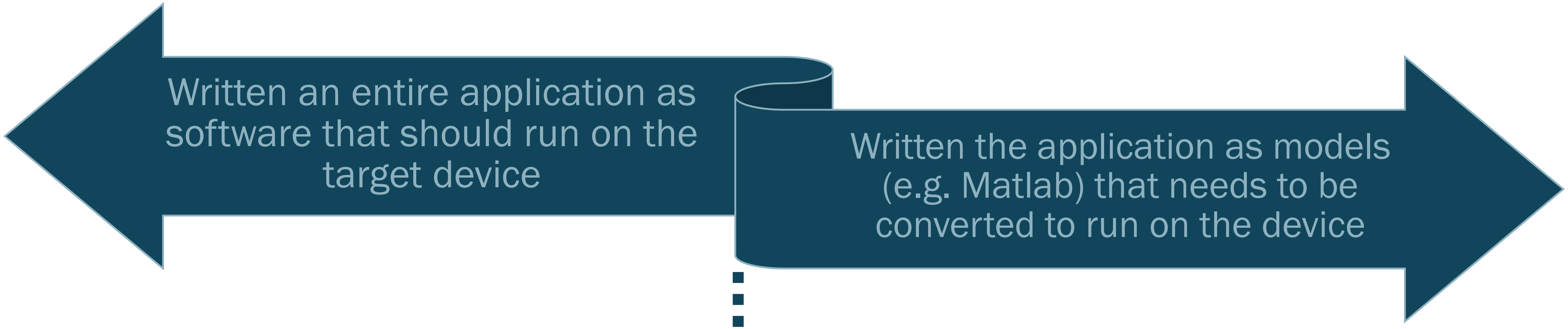
Open standards are well-specified and so integrate well and can be maintained

Open standards are a safe investment: the standards you are building your application on will stay supported in the future

Analysing Your Application

codeplay®

Step 1: How Did you Design your Algorithms?



Written an entire application as software that should run on the target device

Written the application as models (e.g. Matlab) that needs to be converted to run on the device

If your application is software that could run on the device, what did you write it with?

- □ CUDA
- □ OpenCV
- □ TensorFlow
- □ PyTorch

How do you want to proceed?

- □ Do you want to write the application as embedded Software?
- □ Or, do you want to generate the software from the models?

Often we see applications that are a mix of these two

From Proprietary to Standard

Original Software	Common Examples	Challenges	Standard Deployment Options
CUDA	TensorFlow, PyTorch, Eigen, + lots of in-house code	NVIDIA GPU-only	SYCL, oneAPI (you can run both on NVIDIA GPUs as well as many other accelerators)
OpenCL	OpenCV, Glow, TVM	Portable, but not *performance-portable:* Will run on new OpenCL device, but will need optimizing per-device	OpenCL
AI Graph Compilers	XLA, Glow, TVM	Each graph compiler supports a different set of options, datatypes and runtime APIs	OpenVX, or use the same graph compiler on top of OpenCL (Glow & TVM allow this), or use ONNX to transform to device-specific AI compiler

Step 2: How Does your Application Perform?

Always profile your application before optimizing!

With a system-on-chip, you will get the best performance by running different algorithms on different processor cores at the same time

⇒ so, you can run deep learning & machine vision on different cores

Some operations are *bandwidth-bound* (i.e. the performance is limited by memory access) and some are *compute-bound* (i.e. performance is limited by FLOPS/TOPS)

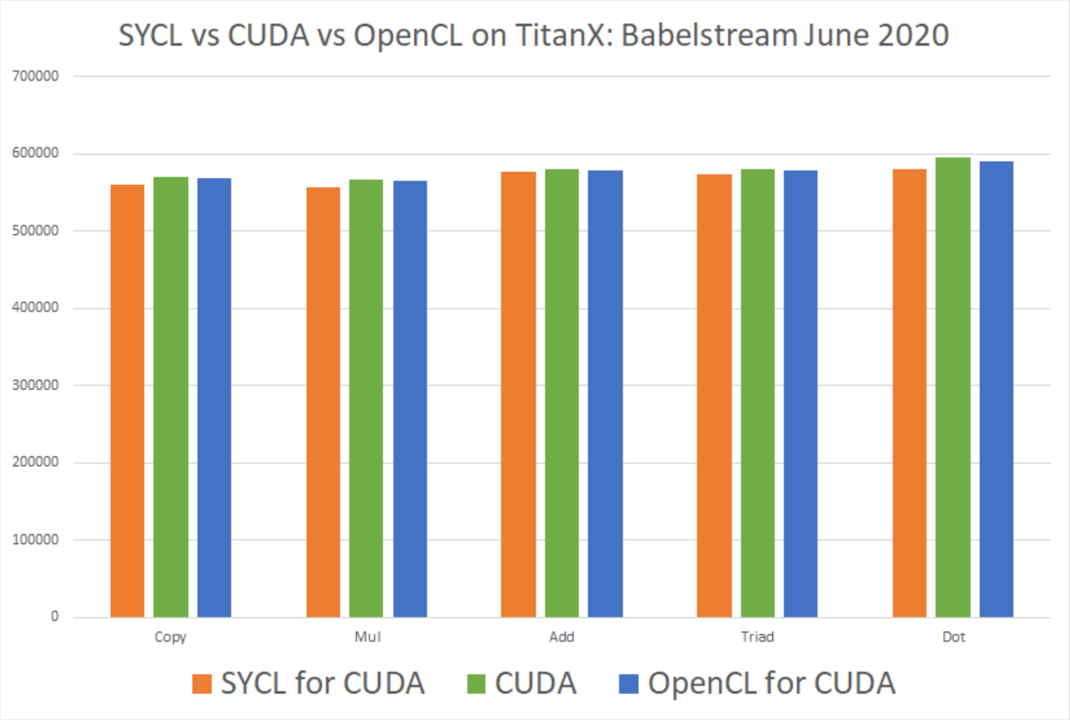
⇒ No point running a bandwidth-bound operation on a faster-computation processor

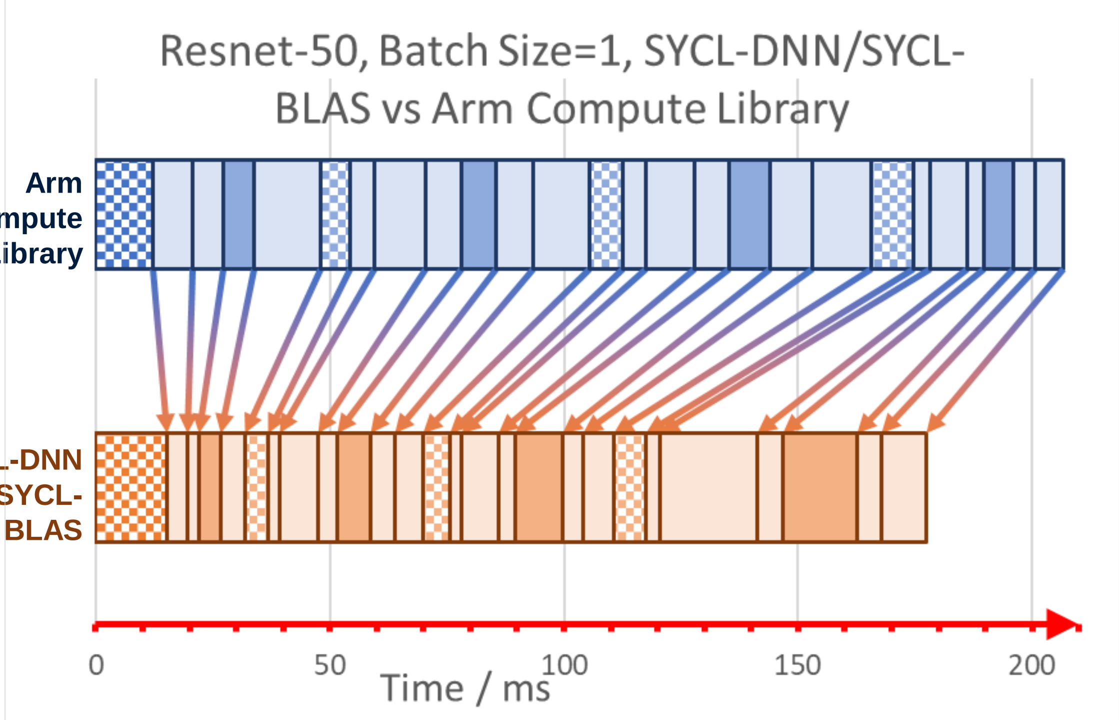
⇒ For many AI processors, there is *on-chip-memory*: this is much faster, but much smaller, than typical off-chip-memory. Can you fit your model in on-chip-memory?

Performance Optimization with Open Standards

For bandwidth-bound operations there is very little difference between programming models (memory dominates)

For compute-bound operations, you need to replace a device-specific library (e.g. cuBLAS) with a new device-specific library (e.g. MKL-BLAS) or with an autotuning library (e.g. TVM, clBlast or SYCL-DNN)





Step 3: Mapping Algorithms to Processors

Small algorithms (e.g. small Kalman filters) don’t offload well to a processor, unless you can group a lot of them together

Vision networks are usually int8 or int16, but behaviour-prediction networks are usually floating-point: the data types restrict the processors that will support the network

Adapting your network for embedded devices:

- Do you need high precision floating-point?
- Can you use sparse networks? Not so good for GPUs but can be better for some cores
- You need to adapt the training of your network for the restrictions of the device
- Can you fit it into on-chip memory to get the performance up?

Custom, hand-coded algorithms, need to go onto programmable cores

Doing the Deployment

Re-training Networks for Device Precision

Most efficient AI cores use reduced precision to achieve performance improvements

First, train your network with floating-point precision

Then, you need to slightly re-train your network for the precision of the device: this works by applying the device precision into the training process and the training then adapts the weights a little to match the impact of precision changes

Reduced precision and sparsity can achieve big improvements in performance in some cases, with often only a very small impact in accuracy

You then need to re-test your network to ensure it is still accurate enough

Optimization On-device

Different processors have *massively variable* performance across different algorithms

This means, you can’t expect good performance when deploying to a new device straight away: you will have to do some per-device optimization

Use profiling tools where available to understand the performance on-device

Memory access patterns vary widely by device

You want to achieve:

- Good use of on-chip memory (both shared and local)
- Good use of DMA: this can massively speed up memory access on non-GPU-cores
- Don’t optimize compute on bandwidth-bound operations: waste of time

Resource Slide

SYCL

Standard documentation

https://www.sycl.tech

OpenCL

https://www.khronos.org/opencl/

OpenVX

https://www.khronos.org/openvx/

Codeplay

https://www.codeplay.com

oneAPI

https://www.oneapi.com/

Compatibility Tool (auto-converts CUDA to DPC++)

https://software.intel.com/content/www/us/en/develop/tools/oneapi/components/dpc-compatibility-tool.html