Programming Techniques for Implementing Inference Software Efficiently

codeplay

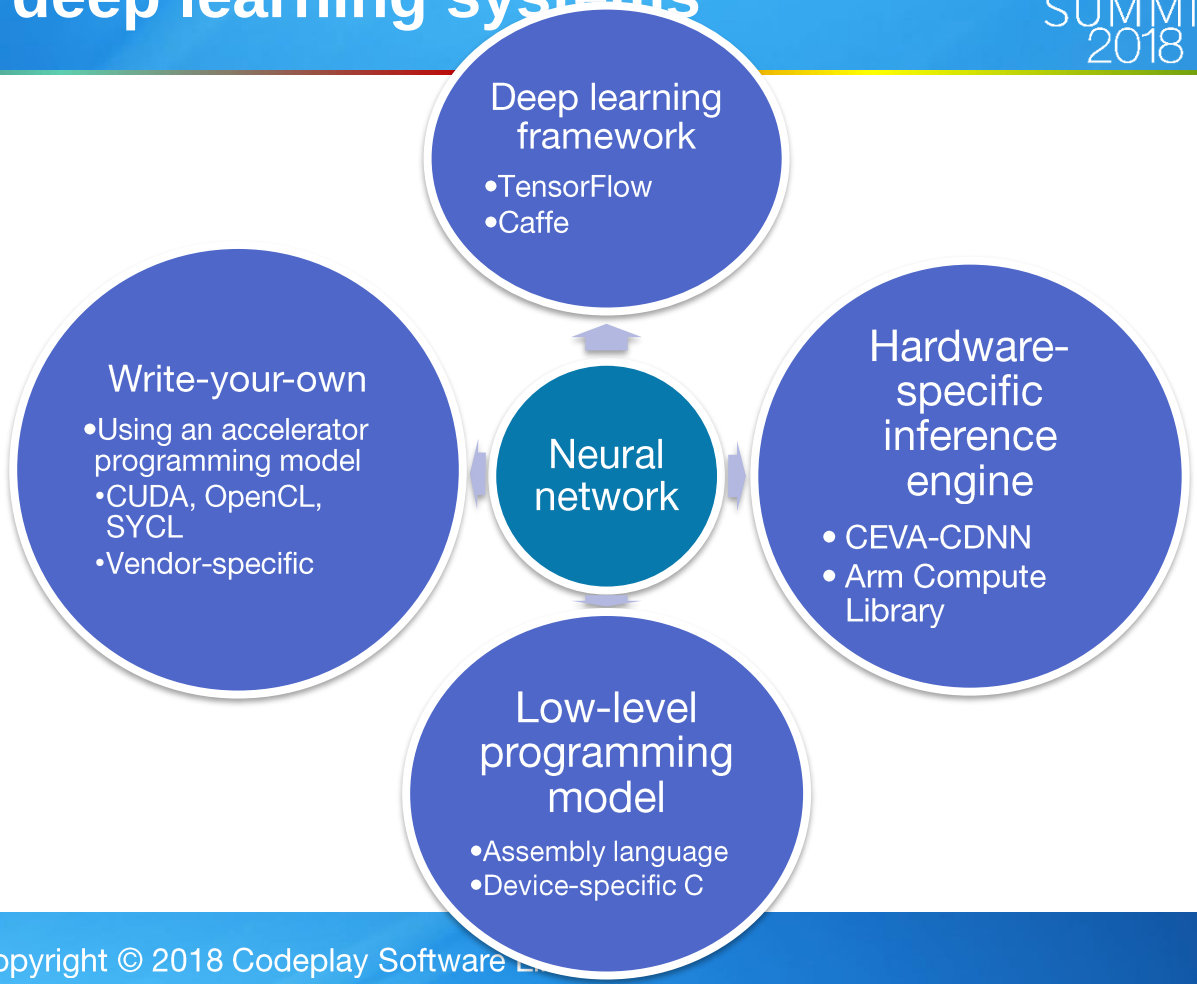
Andrew Richards, CEO, Codeplay

2018

Building embedded deep learning systems

How do you choose?

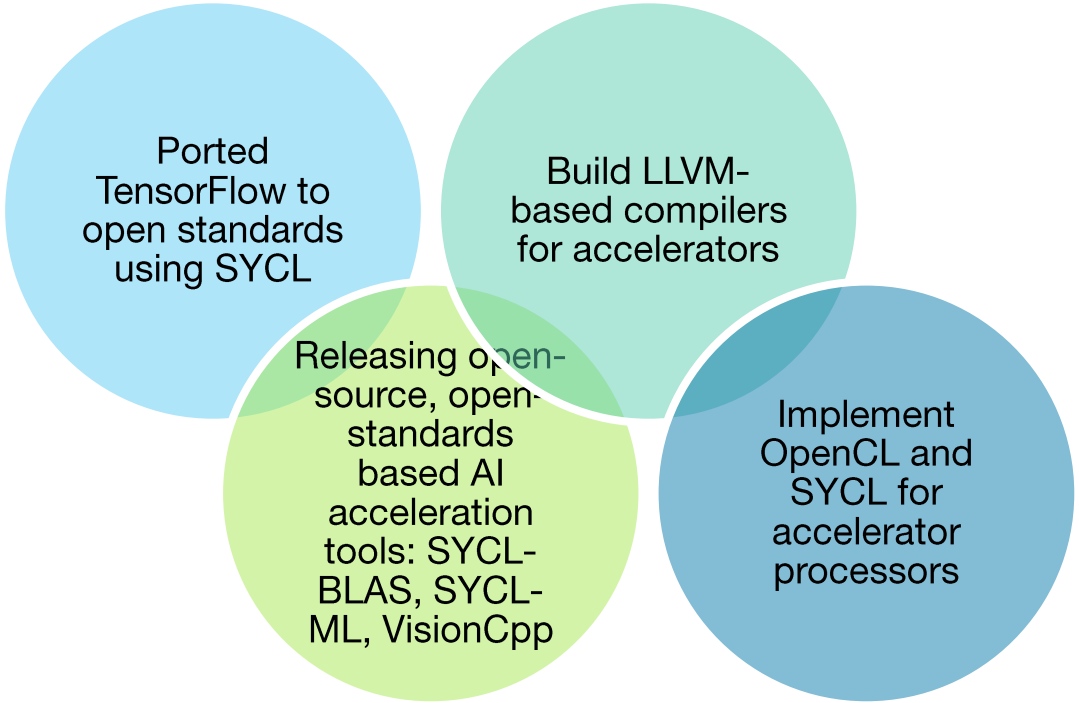
- Hardware acceleration is usually needed to get performance
- It's a fast-moving field so you may want rapid time-to-market



Who am I?

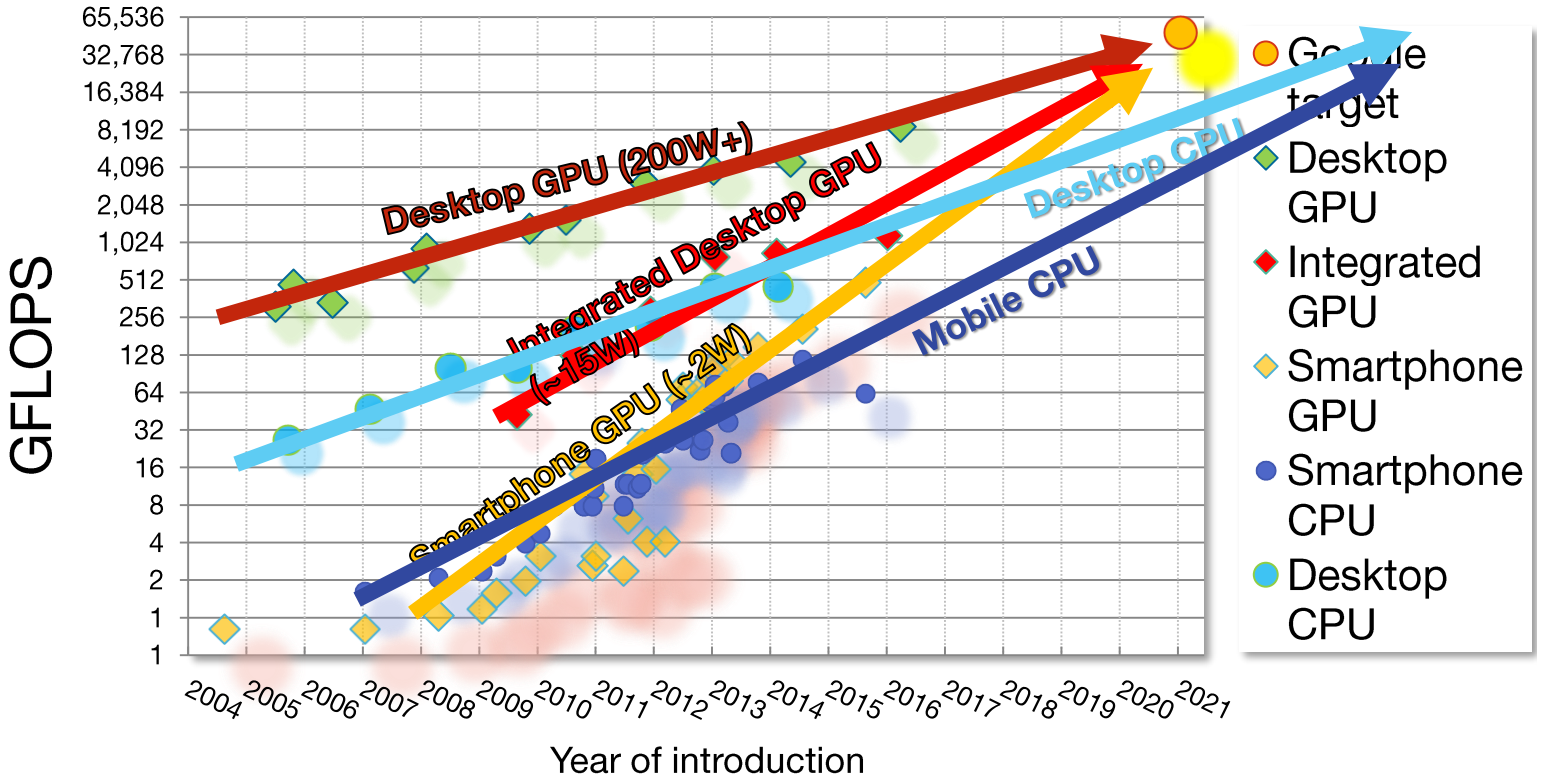
CEO of Codeplay

- Edinburgh, Scotland-based pioneer in GPU acceleration
- Chaired the SYCL and HSA Software specifications
- We build GPU compilers for semiconductor companies
- Now working to make AI acceleration safe for automotive



Understanding the problem

Performance trends



Questions to ask

Safety

- Automotive and medical applications need safety qualifications
- Networks are hard to qualify
- The software itself has safety issues: e.g., AI processors often “throttle” when hot

Tools support

- Debugging
- Profiling
- Optimization analysis
- Power analysis
- Conversion among different data types

Unusual AI operations

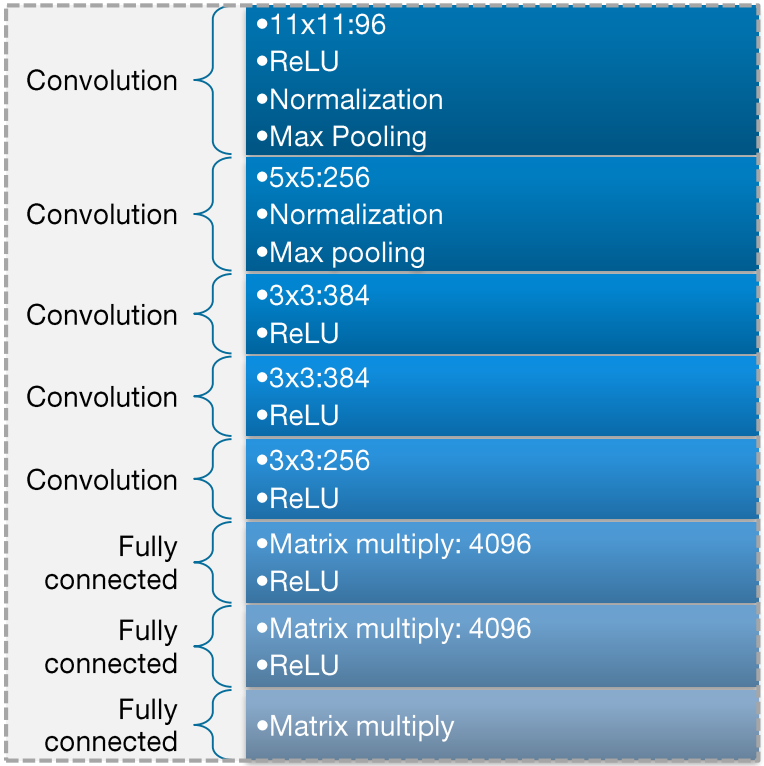
- How many operations are supported?
- Are the operations you need supported?
- Do you need to train on-device?

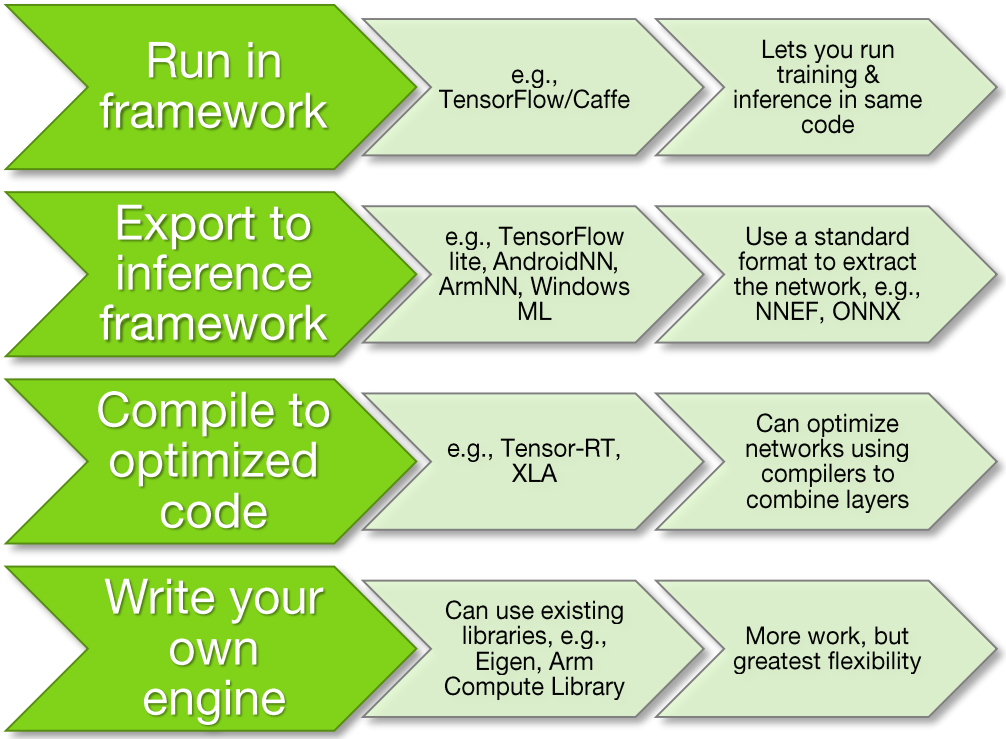
Range of processor choice

- Some inference engines are targeted at just one AI processor

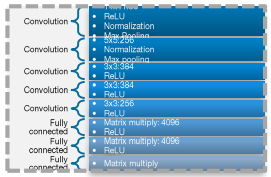
These are the questions that impact your choices

Running a network on an accelerator





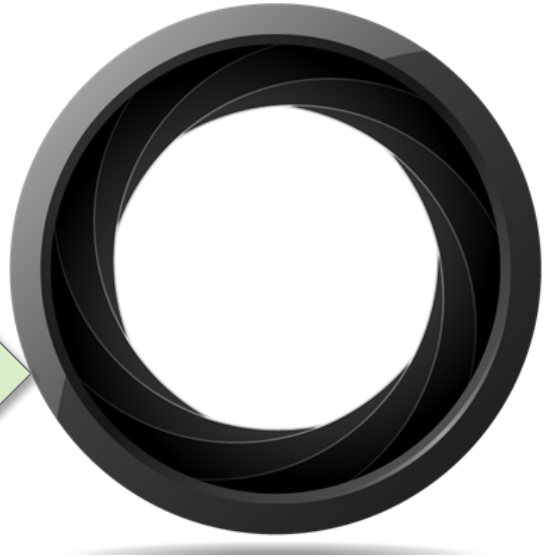
Running existing full AI frameworks on embedded systems



Run in framework

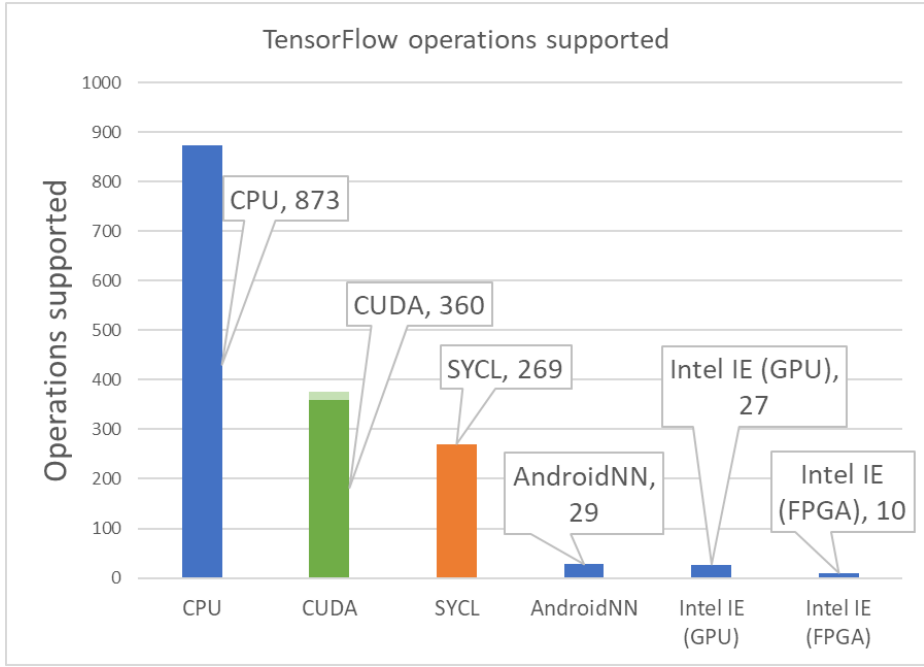
e.g., TensorFlow/Caffe

Lets you run training & inference in same code

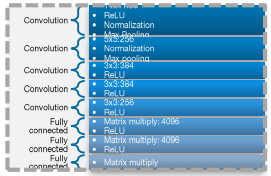


Framework support on embedded systems

- CPU: all frameworks work
- GPU and embedded accelerators
  - very limited support from frameworks
  - varies by processor
- You get great tools support
- High memory usage
- Widest range of operations support
- Easiest route is to port or recompile CUDA-based frameworks to your accelerator with HIP or SYCL



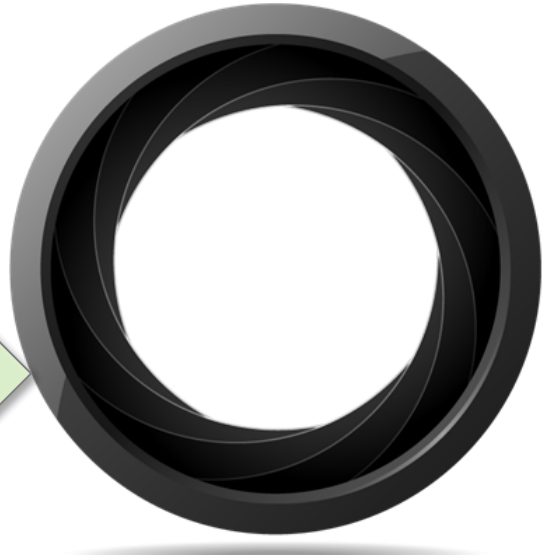
Exporting to an inference-only framework



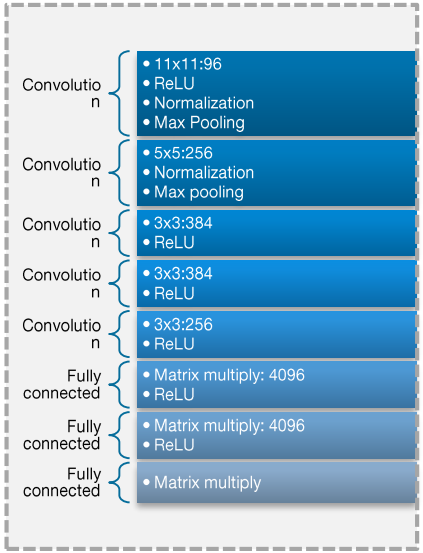
Export to inference framework

e.g., TensorFlow lite, AndroidNN, ArmNN, Windows ML

Use a standard format to extract the network, e.g., NNEF, ONNX



Neural network interchange formats



NNEF

- Industry-standard Neural Network Exchange Format from Khronos
- Tools available on github: https://github.com/KhronosGroup/NNEF-Tools

ONNX

- Standard exchange format from Amazon, Facebook, Microsoft
- Range of tools available at: https://onnx.ai/

Engine-specific format

- Some inference engines have their own format
- Means you need an exporter from the framework you use for the inference engine

- Graph of operations
- Coefficients

Embedded inference engine

- The inference engine you use must support all the operations you need

Embedded inference engines

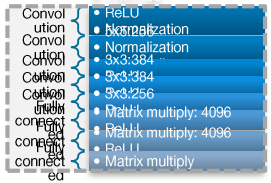
- TensorFlow-lite/Android NN API
- ArmNN: ARM CPU & Mali
- CEVA CDNN: CEVA XM and NeuPro AI family
- Qualcomm Neural Processing Engine: CPU, GPU, DSP
- Huawei HiAI: NPU
- NVIDIA TensorRT: GPU
- Intel Inference Engine: CPU, GPU, FPGA

- + many more from other vendors

Embedded inference engine

- The inference engine you use must support all the operations you need

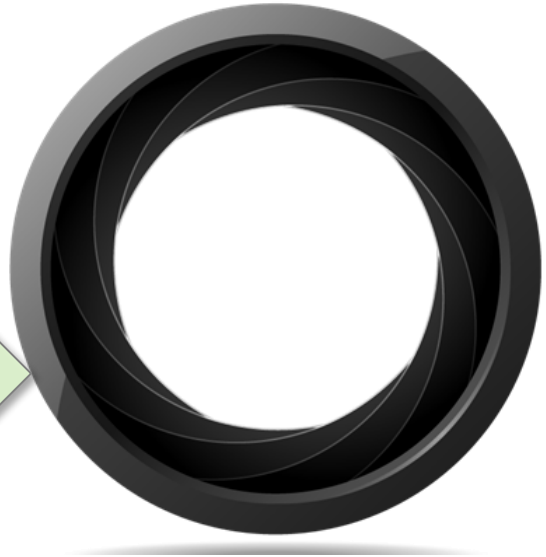
Compiling a network to optimized code



Compile to optimized code

e.g., Tensor-RT, XLA

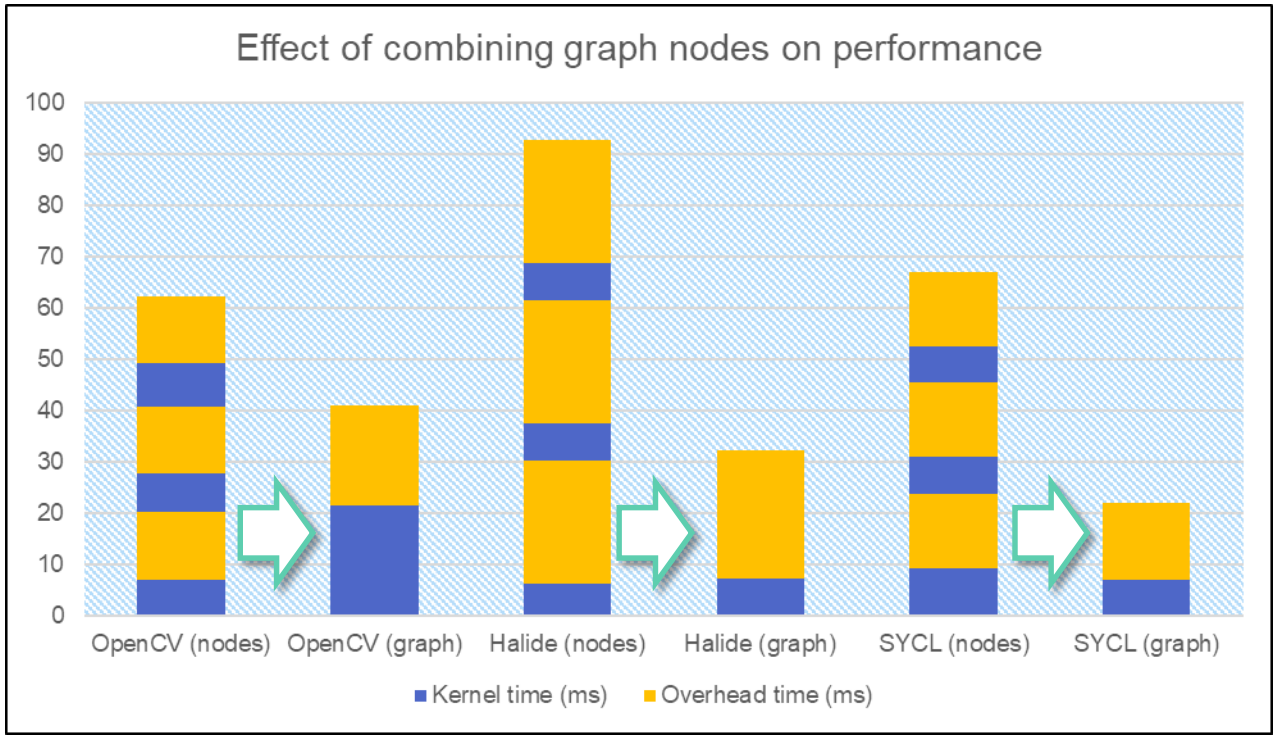
Can optimize networks using compilers to combine layers



Kernel fusion in graph programming

In this example, we perform 3 image processing operations on an accelerator and compare 3 systems when executing individual nodes, or a whole graph

The system is an AMD APU and the operations are: RGB->HSV, channel masking, HSV->RGB

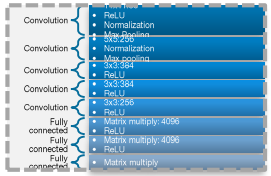


Halide and SYCL use kernel fusion, whereas OpenCV does not. For all 3 systems, the performance of the whole graph is significantly better than individual nodes executed on their own

Achieving kernel fusion optimization

- Compilation flow
  - e.g., XLA (TensorFlow), tvm (http://tvmlang.org/)
  - Take a graph, convert to a very high-level format and then compile it down
- Runtime graph generation
  - e.g., Android NN-API: this can (if supported) be fused & optimized
- Compile-time graphs
  - Domain-specific language: e.g., Halide
  - C++: e.g., Eigen (used in TensorFlow) – just needs a C++ compiler
  - Compile-time reduces runtime overhead and processing

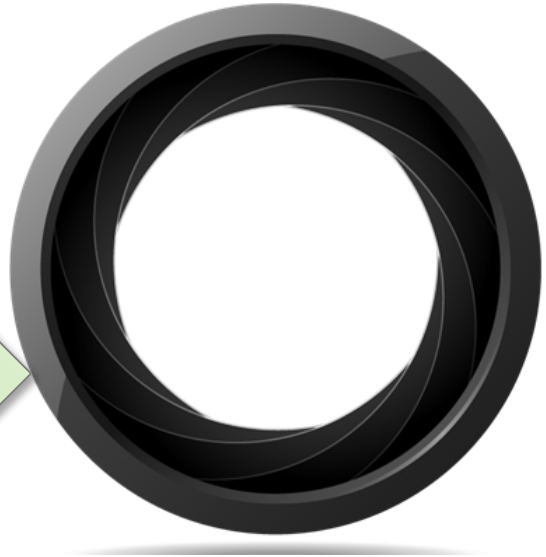
Writing your own inference engine



Write your own engine

Can use existing libraries, e.g., Eigen, Arm Compute Library

More work, but greatest flexibility



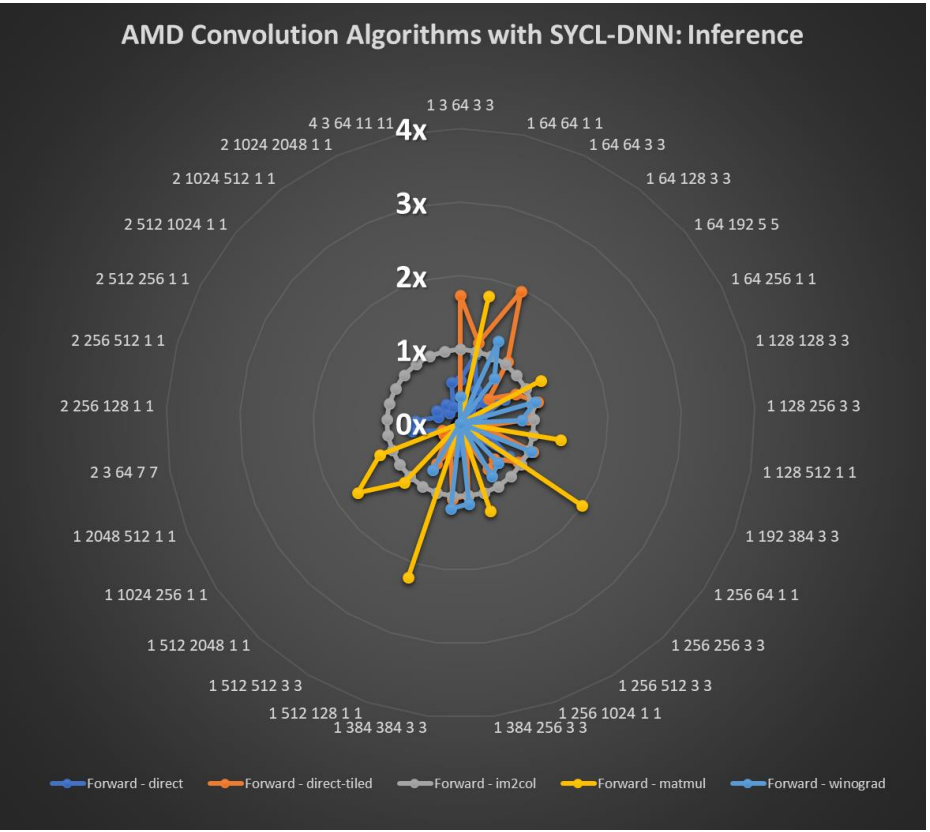
The challenges of machine learning performance

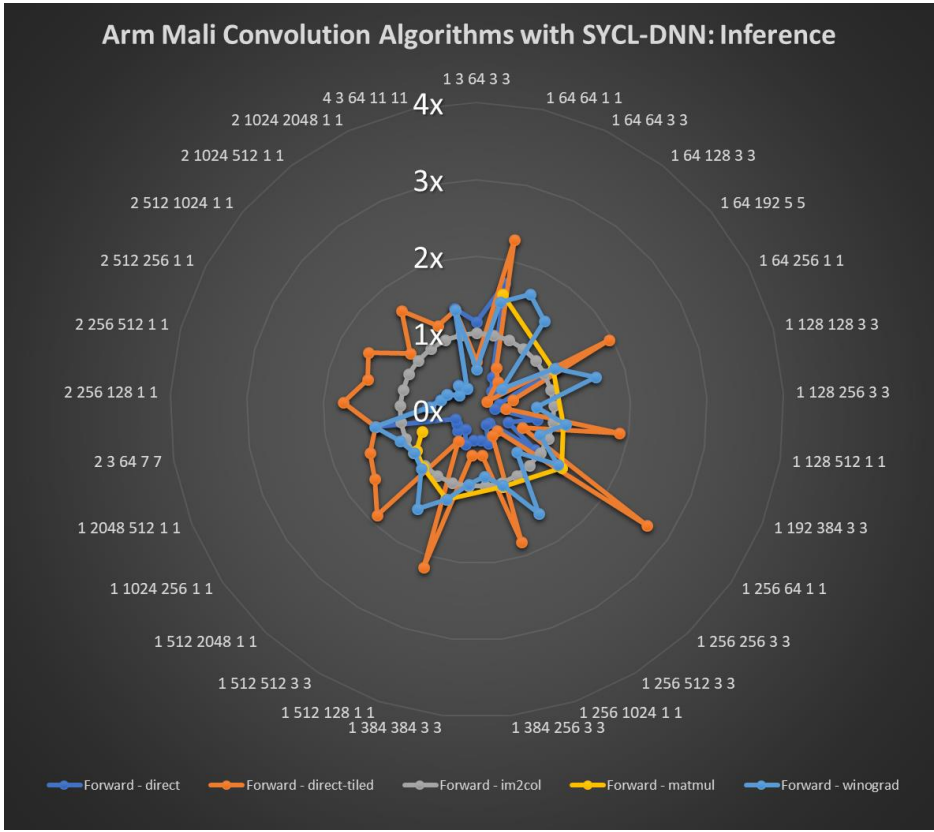
1. Acceleration
   - need to run most of the processing on accelerators
2. Data management
   - Need to keep data on accelerator in friendly data layout
3. Adapting the algorithm to the hardware
   - Different accelerators need different algorithms
4. Making use of fixed-function hardware
   - e.g., BLAS and convolutions
5. Maintainability
   - How can you keep your software running fast on new hardware?

Classes of Machine-Learning algorithm

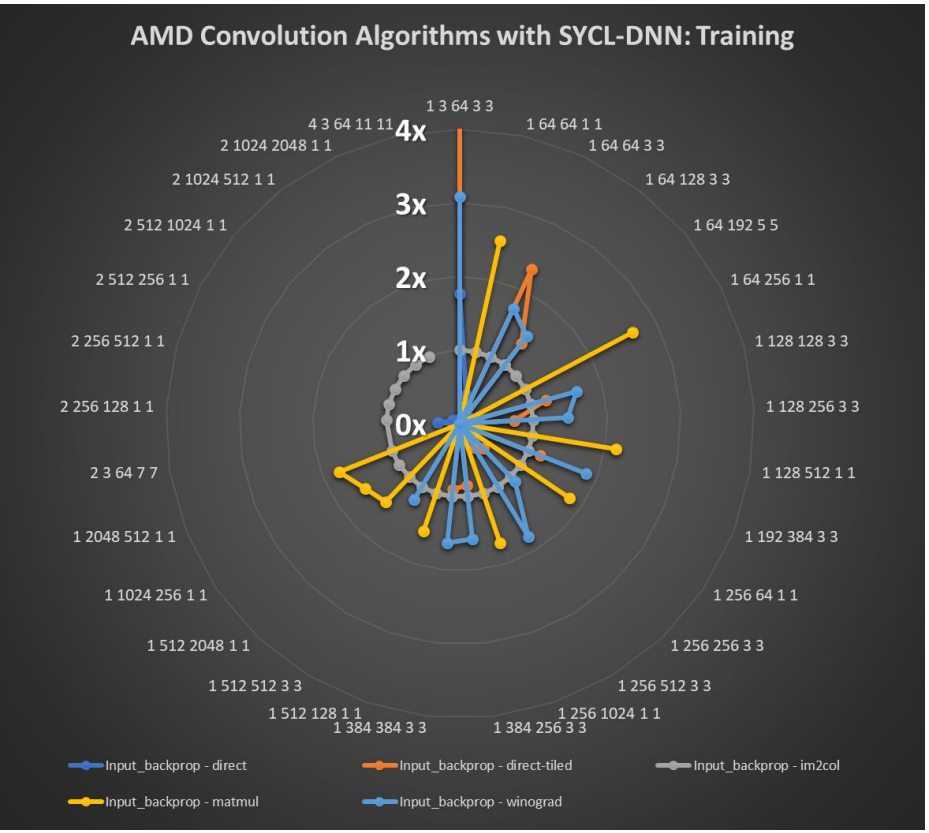
- Convolutions
  - Lots of variations, very algorithm-sensitive
- Matrix multiply
- Component-wise operations
  - Bandwidth-bound
  - Lots of them: need to develop quickly
- Reductions
  - Including partial-reductions, e.g., for max-pooling

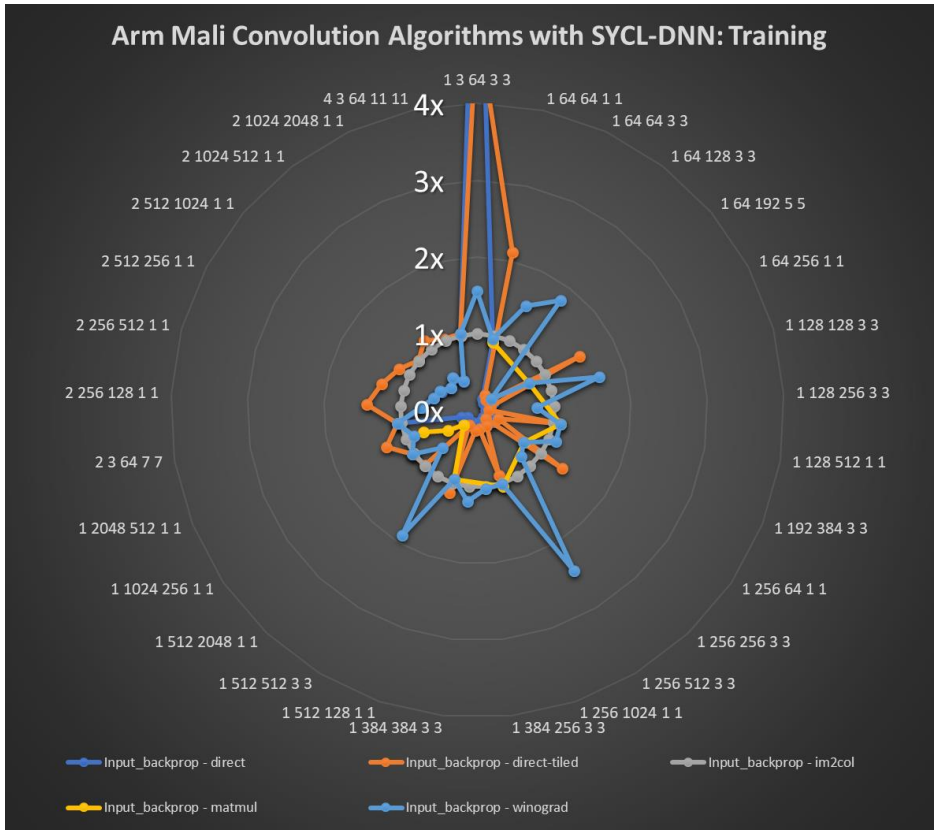
Impact of algorithms on performance (inference)



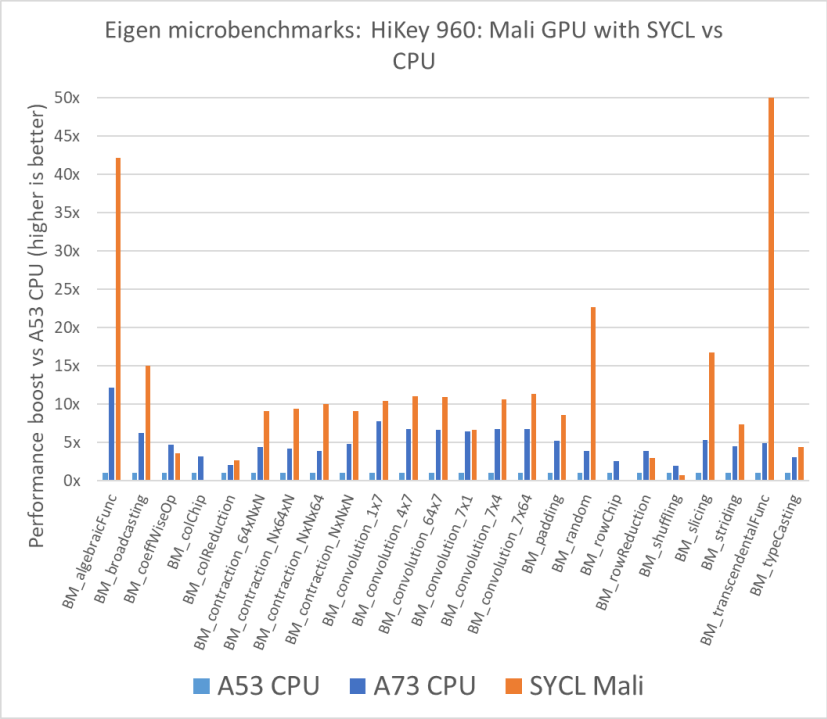


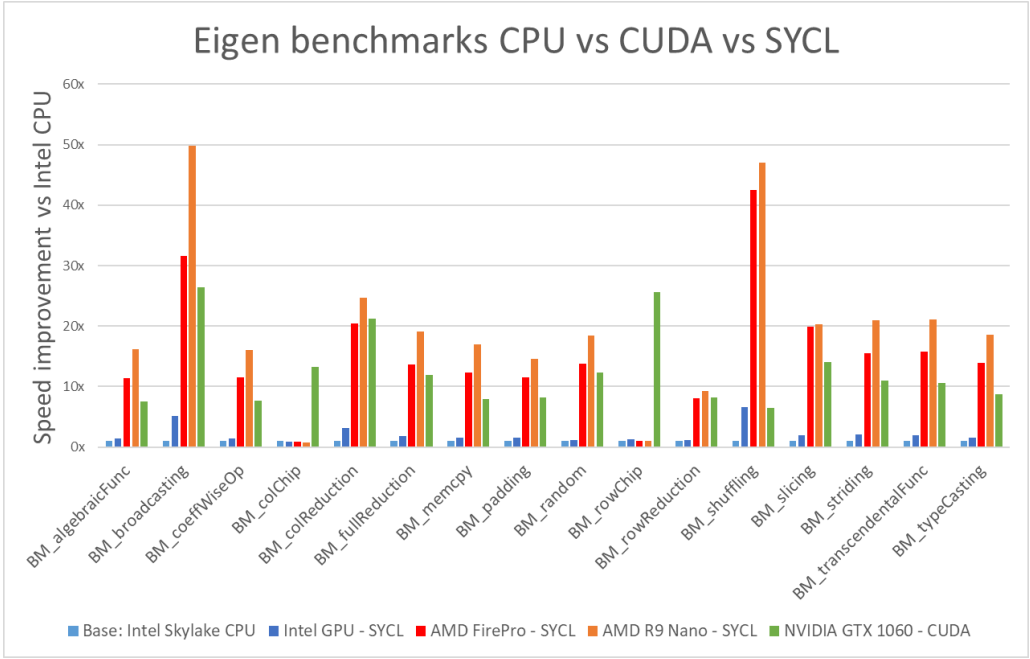
Impact of algorithms on performance (training)



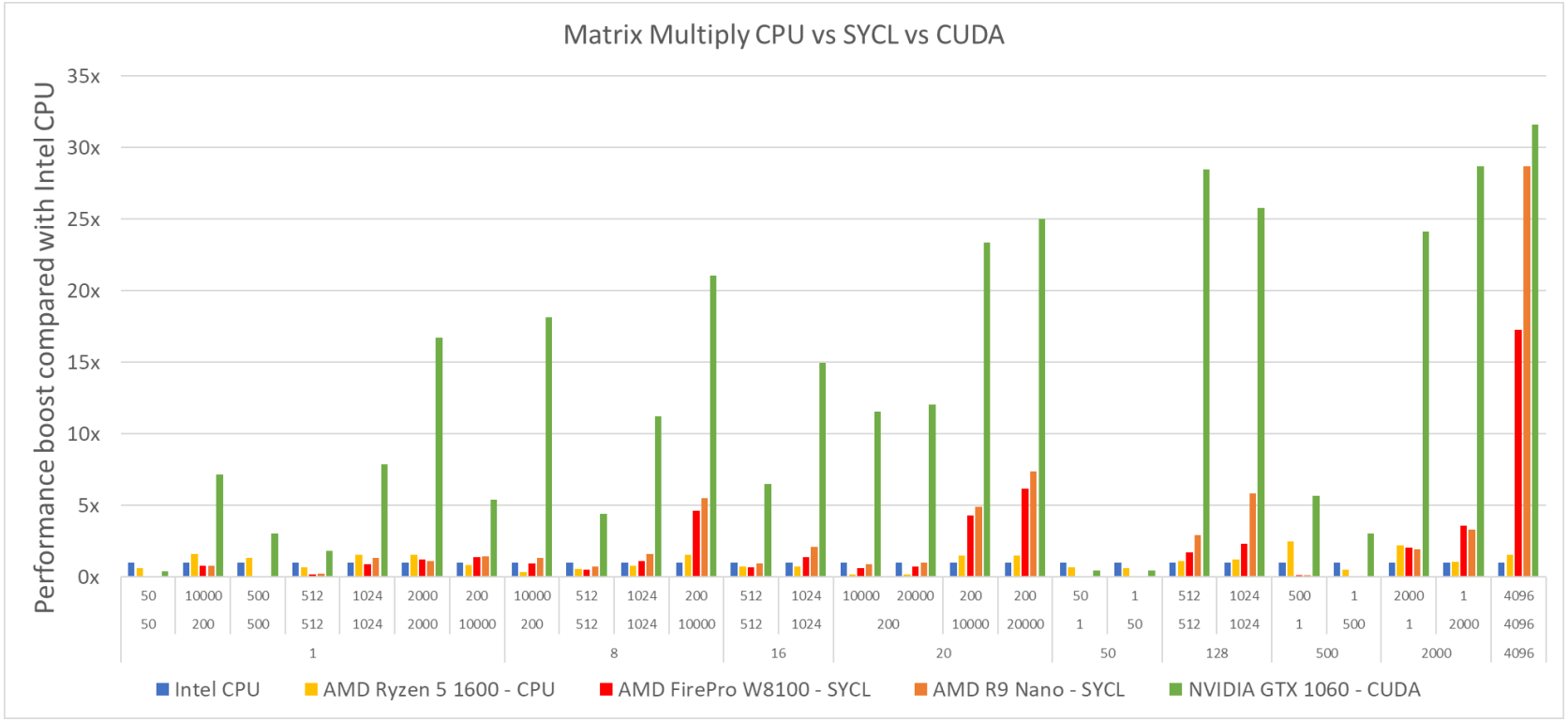


Component-wise & reduction operations performance





Matrix multiply performance



Performance conclusions

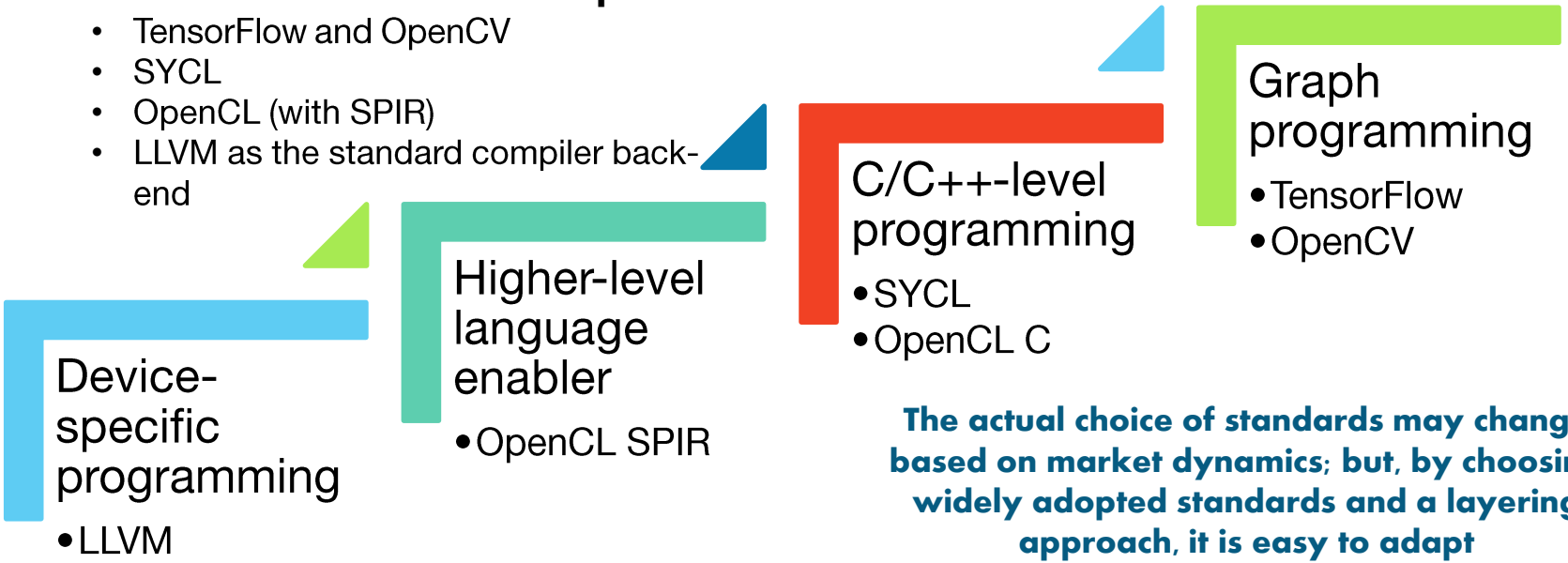
- Convolutions
  - Need to vary the algorithm by convolution shape
- Matrix multiply
  - Need to vary the algorithm by matrix shape
  - Hard to beat cuBLAS: Need a lot of highly optimized kernels
- Component-wise operations and reductions
  - Need to develop lots of them, with kernel-fusion support
  - C++ does this for you (e.g., Eigen, VisionCpp), or Halide can also do this
- CPU synchronization
  - Need to minimize this absolutely, especially on small data sizes. **Keep off CPU!**

Codeplay's approach

For Codeplay, these are our layer choices

We have chosen a layer of standards, based on current market adoption

- TensorFlow and OpenCV
- SYCL
- OpenCL (with SPIR)
- LLVM as the standard compiler back-end



The actual choice of standards may change based on market dynamics; but, by choosing widely adopted standards and a layering approach, it is easy to adapt

Resources

- OpenCL https://www.khronos.org/opencl/
- OpenVX https://www.khronos.org/openvx/
- HSA http://www.hsafoundation.com/
- NNEF https://www.khronos.org/nnef
- SYCL http://sycl.tech
- OpenCV http://opencv.org/
- Halide http://halide-lang.org/
- VisionCpp https://github.com/codeplaysoftware/visioncpp
- SYCL-BLAS https://github.com/codeplaysoftware/sycl-blas
- TensorFlow-SYCL https://github.com/codeplaysoftware/tensorflow
- Eigen http://eigen.tuxfamily.org