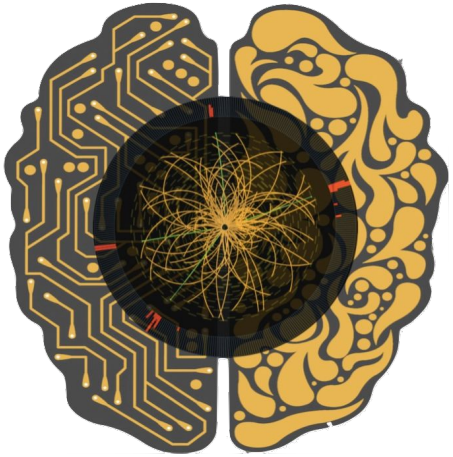


hls4ml

Community Vision, Needs, and Progress

Vladimir Loncar (MIT)

Fast ML for Science @ ICCAD 2023

November 2, 2023

OAC-2117997

a3d3.ai

Introduction

Machine learning is leading to rapid advancements across many scientific domains

Deep learning (DL) architectures based on neural networks (NNs) are demonstrated to be capable of solving a broad range of complex problems

Integration of ML into the experimental data processing infrastructure is enabling us to accelerate scientific discovery, from embedding real-time feature extraction close to the sensor, up to large-scale ML acceleration across distributed grid computing data centers

As scientific data sets become progressively larger, algorithms to process this data quickly become more complex

A computational challenge

Much of the advancements within ML can be attributed to the use of heterogeneous computing hardware

- Graphics processing units (GPUs), field-programmable gate arrays (FPGAs)...

The combination of DL and these processors is leading to a revolution in the way we analyze data

There's a growing need for Fast ML with real-time data processing capabilities, with generalized solutions for various scientific domains

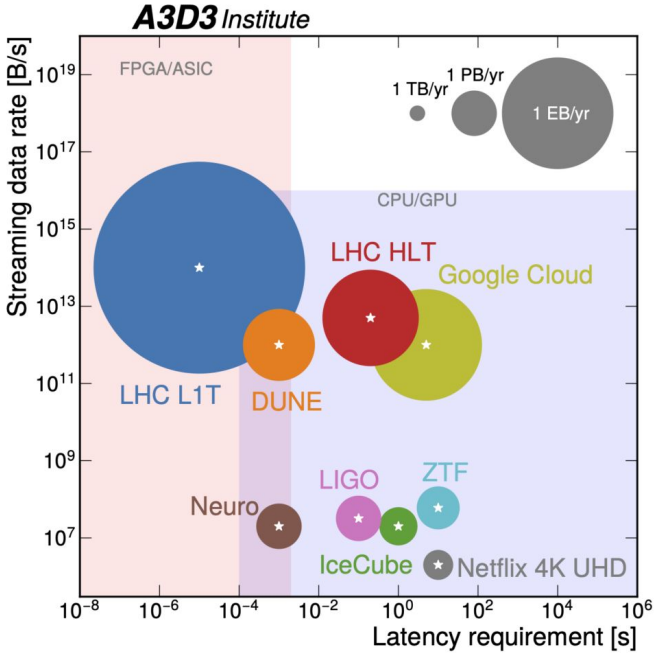


Image from A3D3 institute

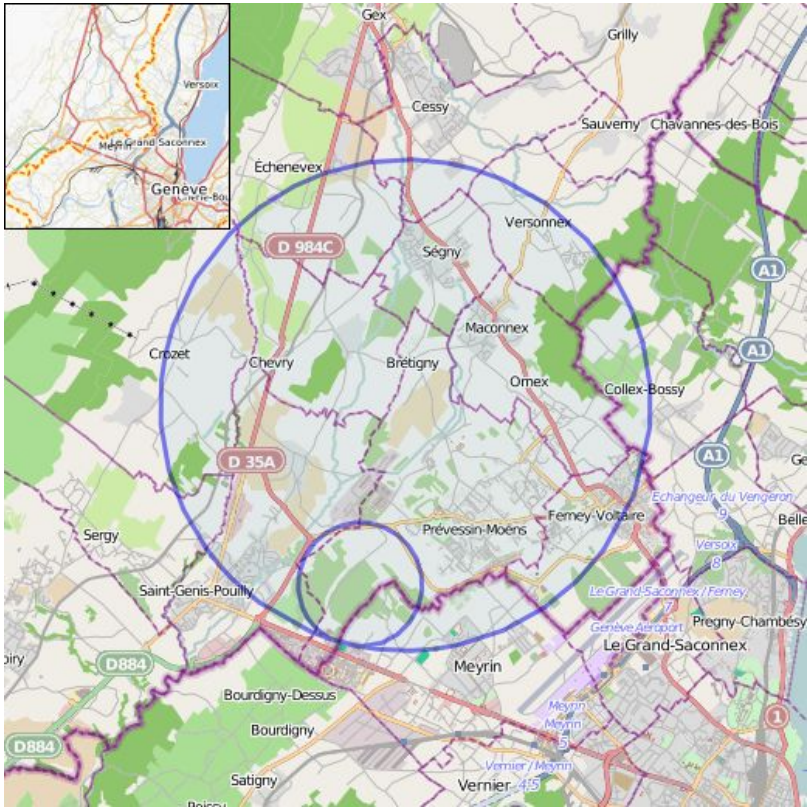
The Large Hadron Collider

27 km circumference accelerator at CERN on the border of France and Switzerland near Geneva

Accelerates protons close to the speed of light, and collides them at 14 TeV centre of mass energy

Searching for new fundamental physics of the universe!

Collisions happen at 4 points where there are detectors

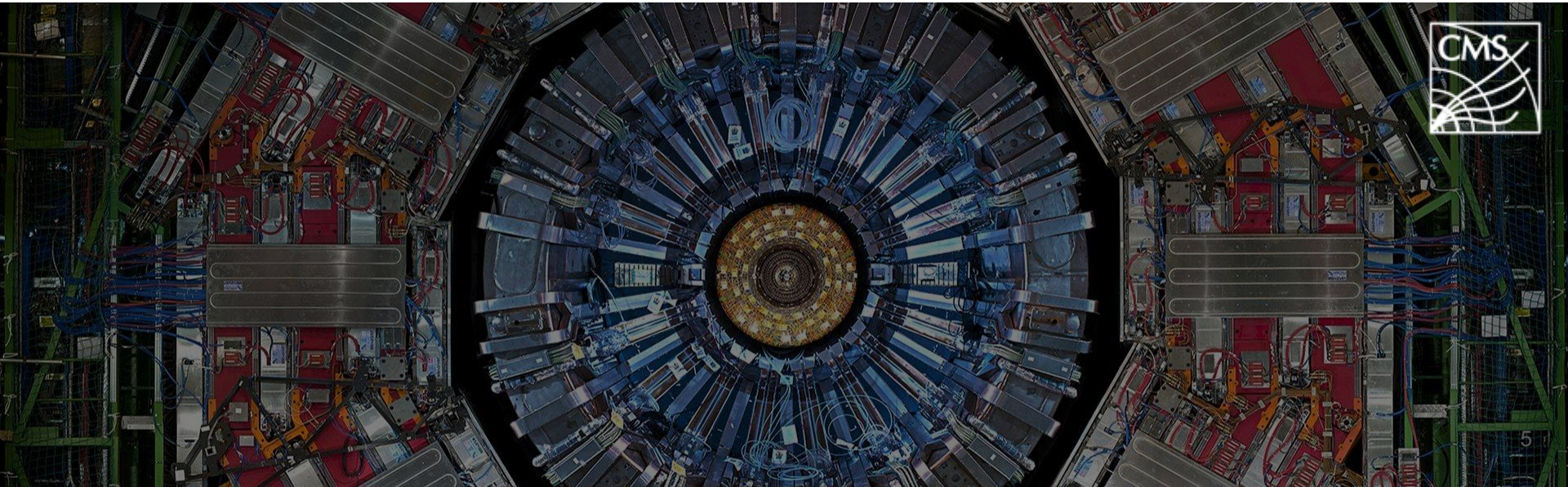


Challenges in LHC

LHC proton beams collide at a frequency of 40 MHz, producing O(100 TB/s) of data

“Triggering” - Filter events to reduce data rates to manageable levels

Strict latency constraints of O(1μs) - challenging to use ML in this environment



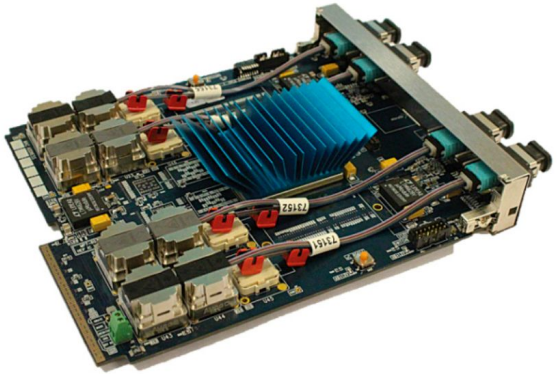
Trigger hardware

We need fast processing of raw data in *O*(µs)

- Not possible to use common hardware, such as CPUs, nor common operating systems

Must be flexible and modular to support reconfiguration and upgrade/maintenance of modules

- Field-programmable gate arrays (FPGAs)
- Perfect because:
  - Resource parallelism ➔ **low latency**
  - Pipeline parallelism ➔ **high throughput**



Designing low-latency ML processing pipelines

The design of low latency algorithms differs from other ML implementations

- We must tailor specific processing hardware to the task at hand to increase the overall algorithm performance
- Processor design + the design of algorithms = hardware ML co-design

However, designing hardware is challenging

- Designing efficient parallel algorithms for programmable hardware is even more challenging
- Usually done by domain experts using hardware description languages (HDLs)

  ➔ High-Level Synthesis (HLS)

High-Level Synthesis

An automated design process that takes functional description (usually in C/C++-like language) as input and produces register-transfer level (RTL) abstraction expressed in HDL

- No need to manually write HDL code
- C++ is not ideal language for hardware design, as it lacks definitions of concurrency and timing
- HLS tools extend the specification of the language with compiler directives aimed at guiding the conversion to RTL

HLS tools have advanced significantly in recent years (see Caroline's talk later today), making them a viable option for creating ML tools

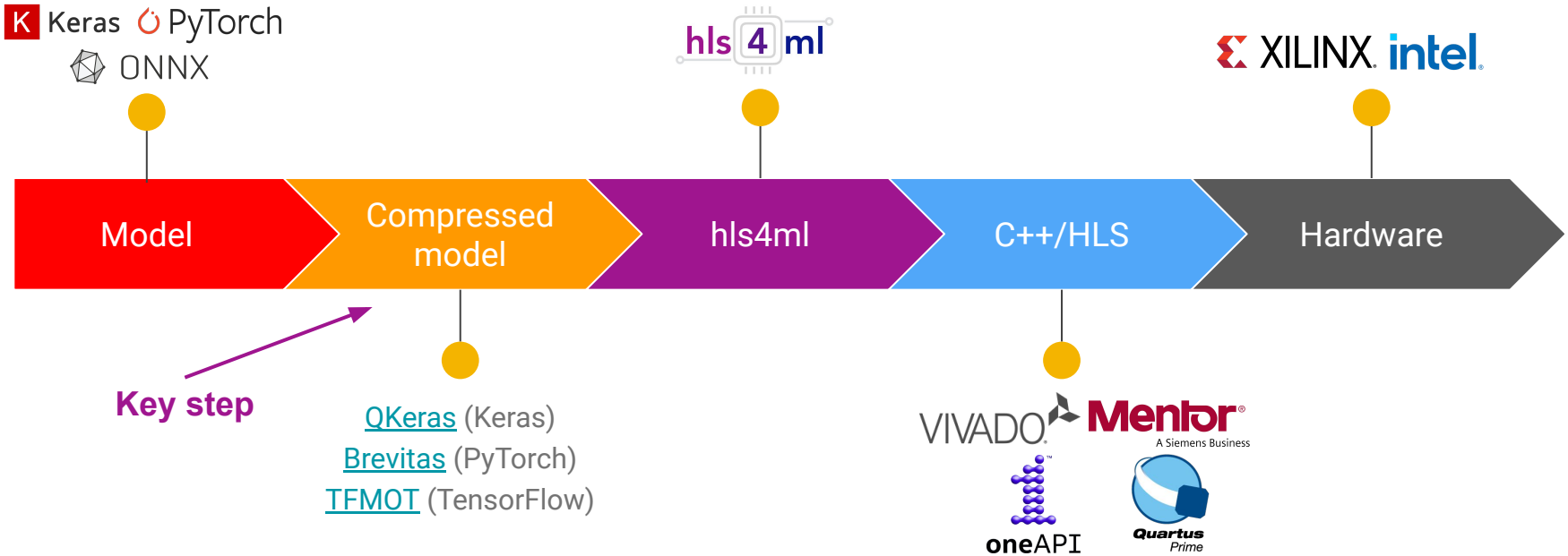
Using HLS we created a tool for converting DL models into high-performance hardware definition

➔ High-Level Synthesis for Machine Learning, **hls4ml**

hls4ml compiler

Open source tool for automated conversion of DL models into hardware

- Developed by the scientific community with numerous contributors



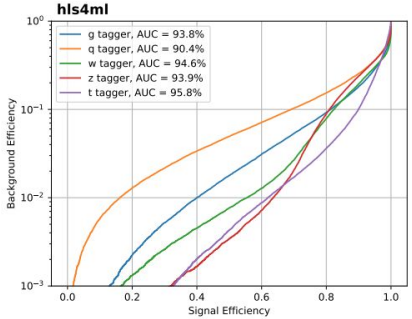
Supported ML architectures

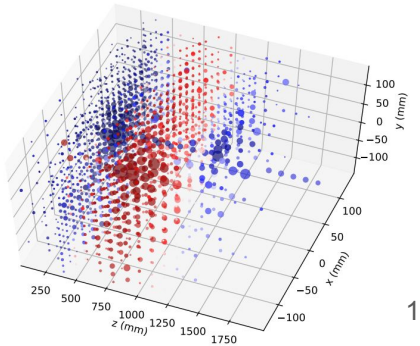
Common architectures:

- Fully-connected networks (**MLPs**)
- Convolutional networks (**CNNs**)
- Recurrent networks (**RNNs**)
- Autoencoders based on common architectures…

Custom implementations:

- Specific types of graph networks (**GNNs**)
  - Limited, not generalizable

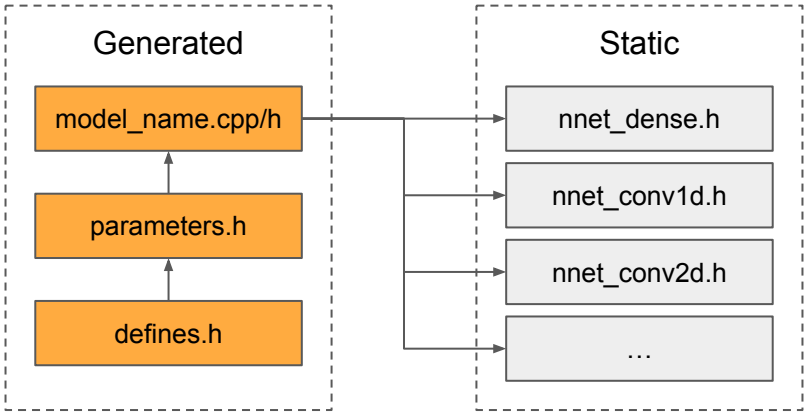




Current hls4ml code generation

hls4ml generates the model architecture function calls, while relying on hand-tuned algorithms implemented in HLS

- Implemented in `nnet_utils` package as reusable C++ templates
- High performance, low flexibility



Challenges with the current hls4ml

Handmade HLS implementations have a static interface that needs to be matched precisely

- Difficult to change data representation
- Few dataflow conventions adopted and the library must be built around that

➔ Limited possibility of generalization

Limited customization through C++ templates

- Maximum performance is only possible through compile-time constants
- HLS synthesis is guided by compiler directives (pragmas) that are difficult to template

➔ Duplication is the only solution

Challenges with the current hls4ml (contd.)

Difficult to compose operations

- Good implementations of higher-level concepts, like layers of a network
- Lower-level tensor operations are more difficult to combine to produce new “layers”
- Simple operations like fusing activations into the output of dot product has to be done manually

➔ Cumbersome to extend and maintain

Rigid flow of the top function

- With static implementations, the data flow hierarchy must be rigid
- Good performance out-of-the-box, but not option to optimize further for a specific target

➔ No design space exploration possible (DSE)

Deployment of emerging NN architectures

The aforementioned challenges are most evident when applied to novel NN architectures

- Usually expressed with lower-level tensor operations rather than traditional higher-level “layers”

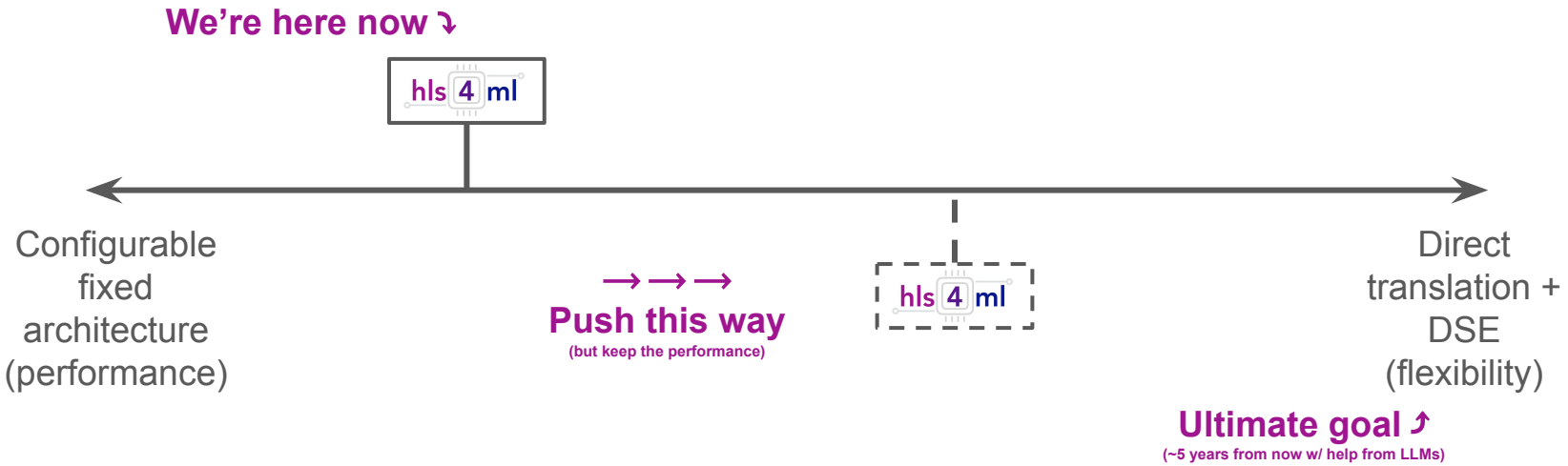
For example:

- Graph Neural Networks - explored for use in particle trajectory reconstruction
- Transformers - Demonstrated superior particle classification performance

Supporting these in **hls4ml** would require a ground-up implementation with careful tuning tailored for specific models

- Lack of flexibility would hurt adoption in different domains

Performance vs flexibility



We need the best of both ends

What could we do better?

Need to find balance between fixed architecture and full flexibility

Borrow ideas from the tools for direct translation coupled with DSE

- Find middle ground between full automation and hand-tuning of algorithms
- Map operations to highly optimized primitives, but leave the possibility to compose operations at a higher level
- But this won't address the issue of handmade HLS functions not being flexible enough

➔ Generate code with more flexibility via "Domain-specific language"

Prototype based on **PyLog** has been created (link)

- We need the code-generation to be internal, partial and geared towards our use cases

How to achieve this vision?

Go down one level from the concept of “layers” of a network and work directly with tensor operations

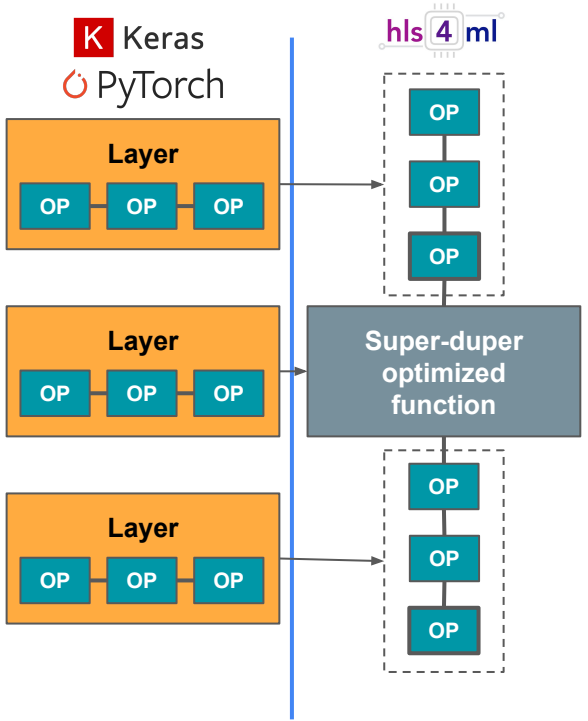
- Allow building “up” by composing these operations

Support both the current way of defining functions for complex collections of tensor operations (aka “layers”)

- E.g., mark a function in PyTorch/Keras that won’t be traced through to lower ops

But this won’t address the issue of handmade HLS functions not being flexible enough

- We need a “DSL” of sorts



“Domain-specific language”

No real need for (yet another) language, we can use metaprogramming in Python

- But extended with “hints” that translate into HLS pragmas
- A mini library
- Inspired from PyLog

For example:

```
import hls4ml.name_of_new_code_gen_library as cg

def dense(data, res, weights, biases):
    mult = cg.array(some_size, shape, pragmas...)
    acc = cg.array(some_size, shape, pragmas...)

    cg.pipeline(ii=rf) # equivalent to #pragma HLS PIPELINE II=...
    cg.limit(multiplier_limit) # another pragma

    # Do the matrix-vector multiply
    for i, cache in enumerate(data): # Mixture of loop styles
        for j in range(n_out):
            mult[i,j] = cache * weights[i,j]
```

1

```
    # Accumulate multiplication result
    for i in range(n_in):
        for j in range(n_out):
            acc[j] += mult[i,j]

    # Cast to "res_t" type
    for i, a in enumerate(acc):
        res[i] = cg.cast(a, res_type)
```

2

How would this be implemented in hls4ml?

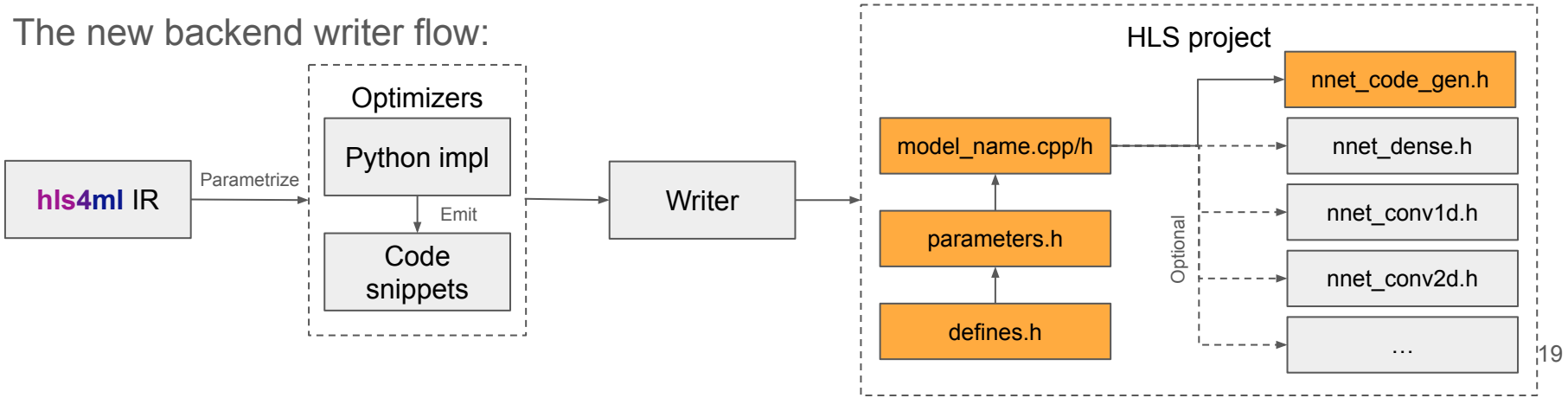
A new (Vitis-only, no Vivado) backend, separate from existing backends

Extend the IR with new (lower-level) operations

- The existing layers can remain, and all types are reusable

Aim for interoperability with existing HLS functions

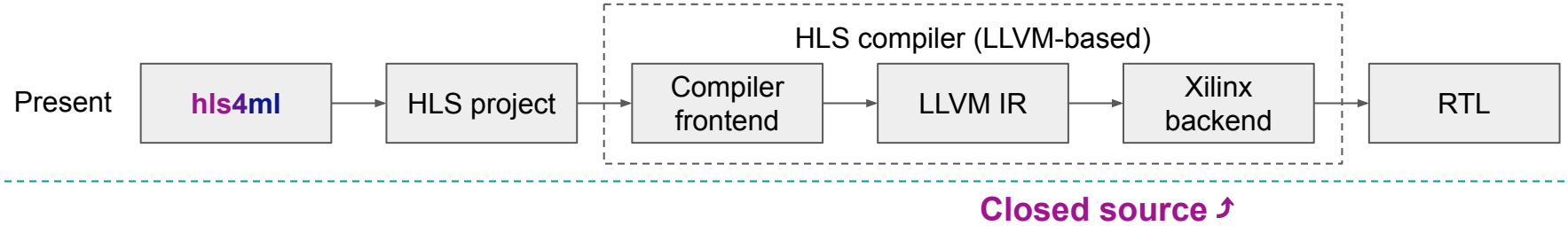
The new backend writer flow:



Integrating with the wider ecosystem

But all of this is still HLS, we still have that intermediate step of generating C++ source code

- C++ is ill-suited for expressing parallelization with scheduling/timing and relies on vendor-specific, closed-source toolchain



While starting to open up recently, key components of HLS synthesis tools are still closed source with restrictive licensing, leading to vendor lock-in

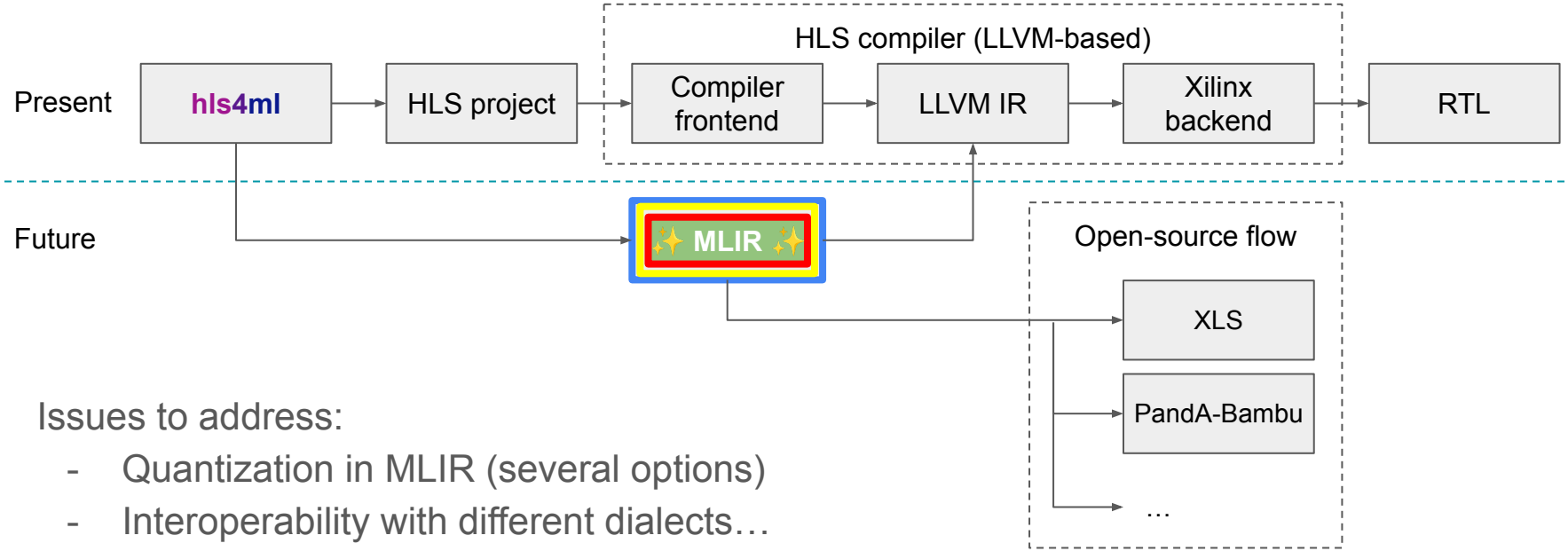
- Adopting the underlying compiler infrastructure ecosystem would give us more options

…

Fully open-source co-design flow

Leverage the emerging compiler technology like MLIR

- Enables us to target different compiler backends, including open source FPGA synthesis tools



Issues to address:

- Quantization in MLIR (several options)
- Interoperability with different dialects…

Summary

There's a growing demand for real-time data processing with ML

To address these challenges, we need tools for hardware ML co-design

Promising tool in this space is **hls4ml** - a software package for translation of trained neural networks into FPGA firmware

- Fast inference times, O(1μs) latency
- Actively extended to provide more flexibility and address future developments in ML

More information:

- **hls4ml** website
- FastML collaboration