Using High-level Synthesis and Emulation to Rapidly Develop AI Algorithms in Hardware

John Stickley – Emulation Technologist

Petri Solanti – Field Application Engineer, HLS

accellera SYSTEMS INITIATIVE

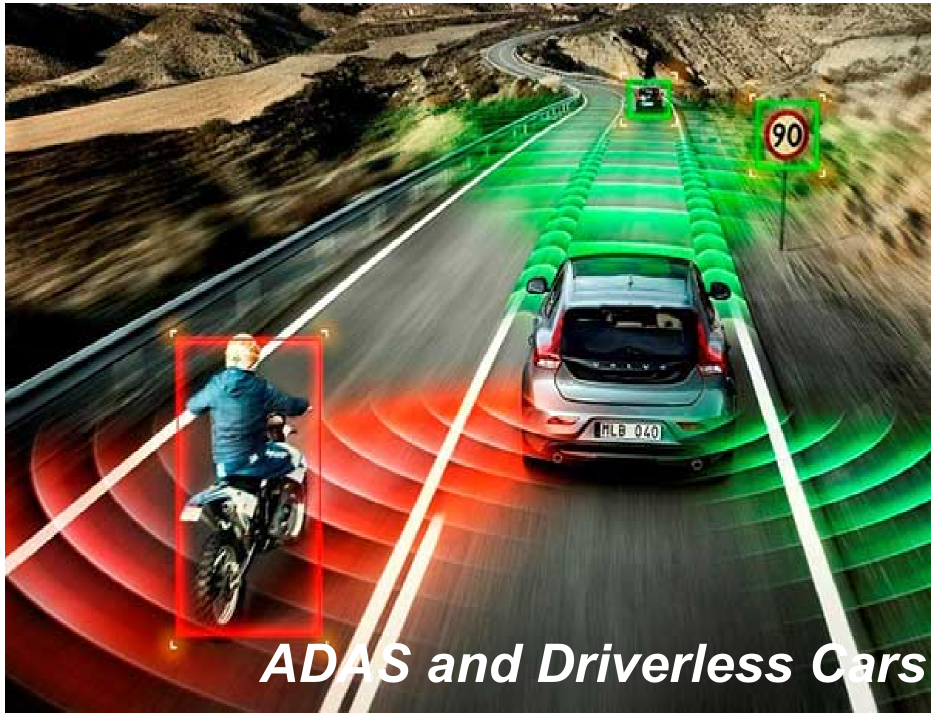
Mentor® A Siemens Business

2018 DESIGN AND VERIFICATION™ DVCON CONFERENCE AND EXHIBITION EUROPE

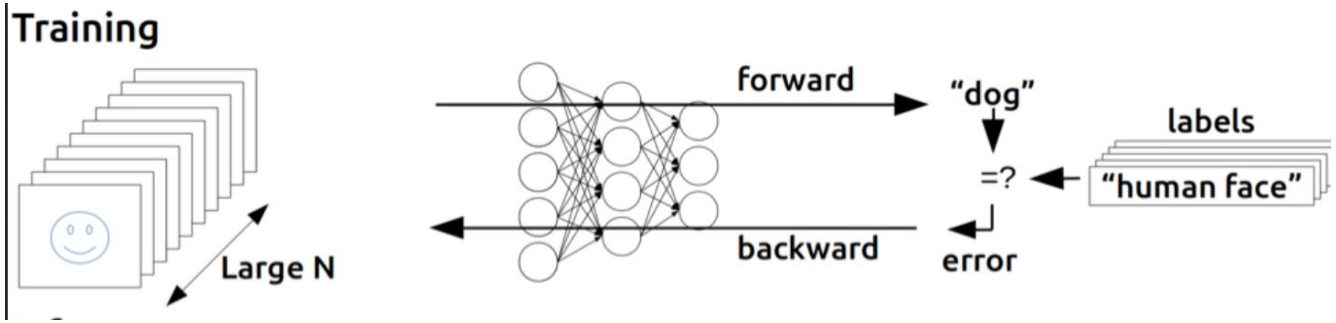
Computer Vision/AI Application Challenges

Automotive and other "real-time" applications especially challenging

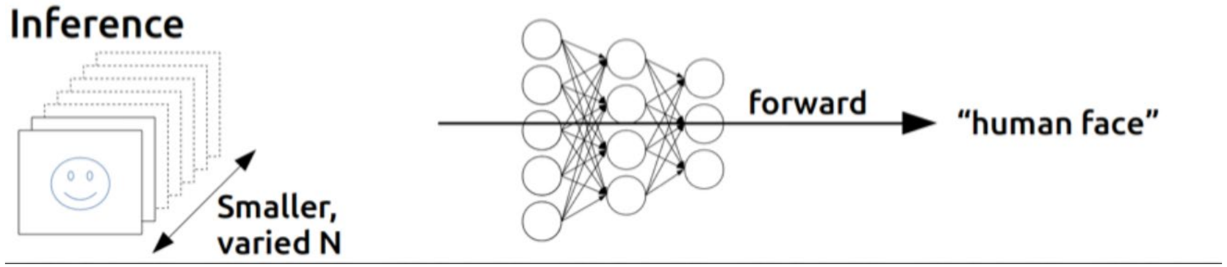
- Continually changing algorithms and sensors
- Computationally very expensive
  - Billions of operations/second
- High responsiveness required
  - High-bandwidth and low-latency
  - Real-time processing of data required
- Autonomous drive - solution required to be < 100w
- Each provider wants to add their "secret sauce"



Convolutional Neural Networks: Training vs Inferencing (Embedded AI)



- Very large datasets and memory, CPU/GPU farms, floating point required



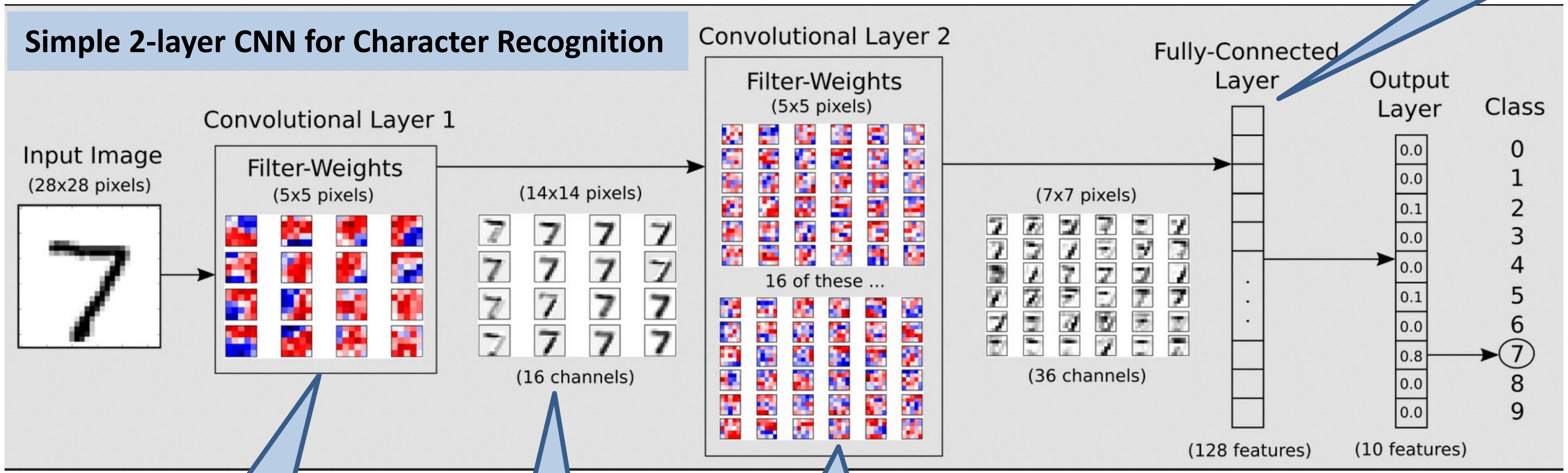
Catapult High-level Synthesis (HLS) fits here

- Uses data from trained network, end system often has real-time requirements, can go to FPGA/ASIC and dedicated HW, can be reduced to fixed point, can implement low-power

Next-generation Computer Vision Designs

Require Much More Parallelism

- Convolutional Neural Networks use lots of 2-d convolutional filters
  - Lots of multiply-accumulate math
- Multiple convolutional layers
- Networks are constant evolving
  - Data rates, number of layers, image size, etc..



Fully connected layer uses matrix multiplication

16 2-d convolutional filters

Feature maps

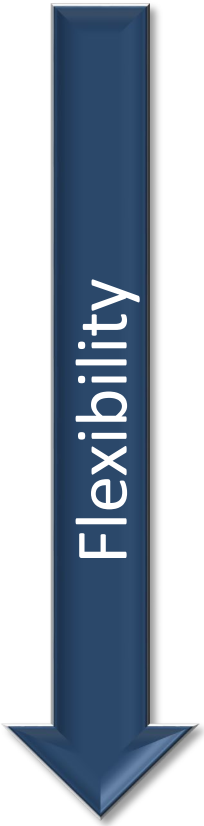
36x16 = 576 2-d convolutional filters

Acceleration of the *Architectural Exploration* Design Phase

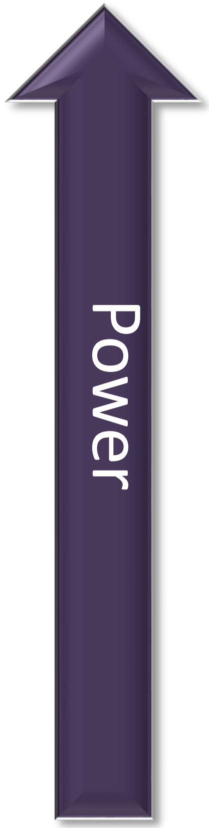
- HLS-synthesized CNN-based machine learning algorithms lend themselves favorably to take advantage of a highly parallel simulation engine such as an emulator
  - The speed of execution of simulating the design on an emulator does not change with the addition of more layers to the CNN algorithm
  - Whereas with software simulation, performance goes down roughly linearly with migration of each layer to RTL

What are the Choices for Hardware Platform?

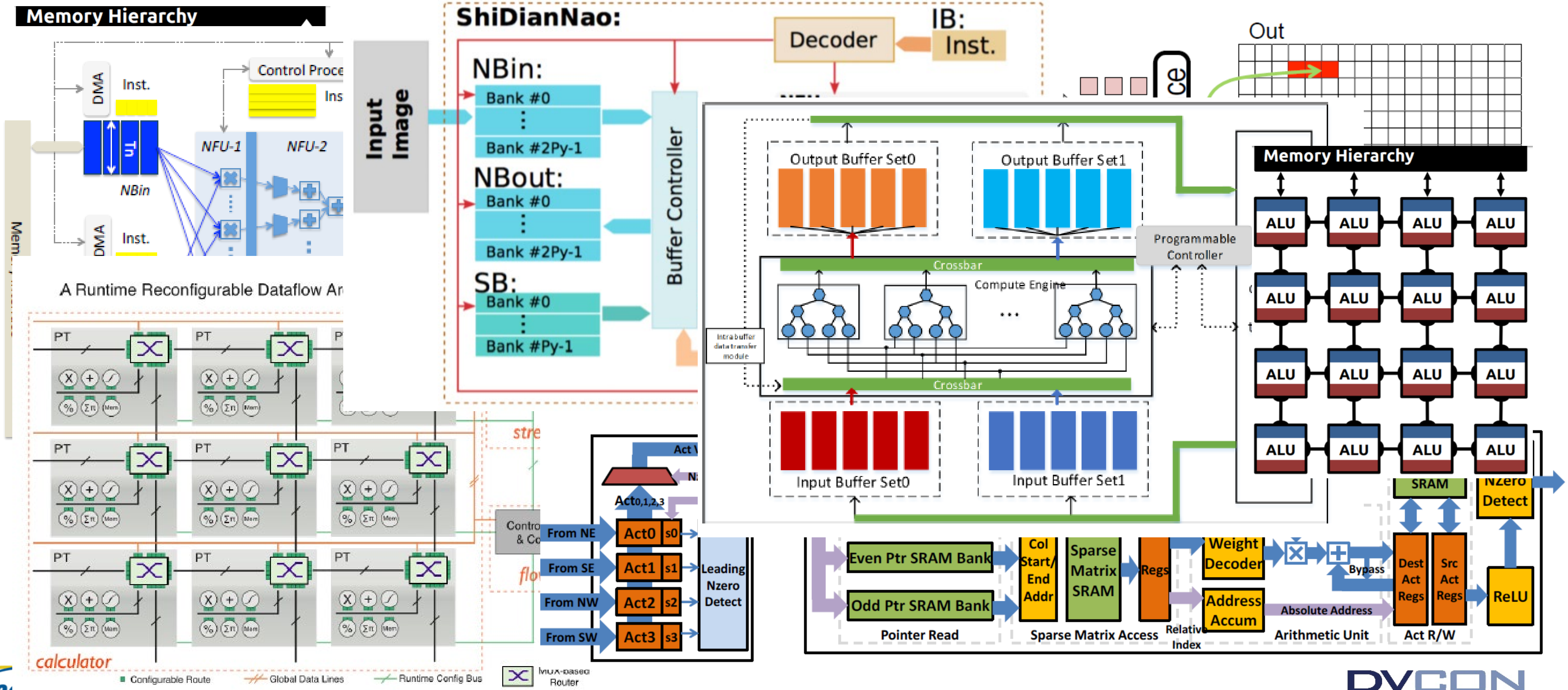
There is no clear winner today as this market is emerging



- CPU
  - Not fast or efficient enough
- DSP
  - Good at image processing but not enough performance for Deep AI
- GPU
  - Good at training but too power hungry for long term inferencing solution
- FPGA
  - Low-power, mostly meets performance/latency, RTL flow not practical, not the lowest power, eventually cost for volume a problem
- ASIC
  - Lowest power, meets performance/latency, high NRE and no field modifications/upgrades, Algorithms still changing, RTL flow not practical, lowest volume cost
- Dedicated AI and CV processors or accelerators in IP and ASIC
  - Popping up like weeds – high performance, locks customer in, many server target
- Some scalable combination of the above

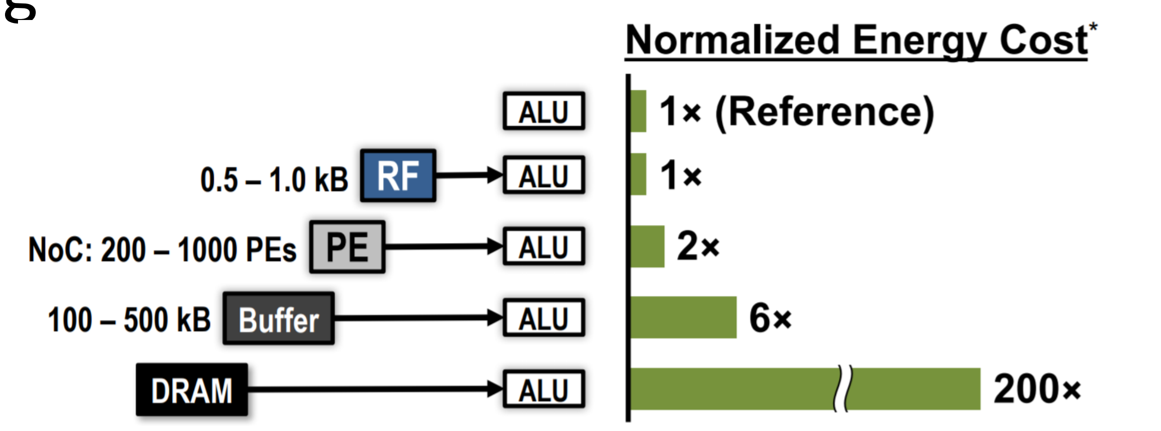


Numerous Possible Hardware/Memory CNN Architectures



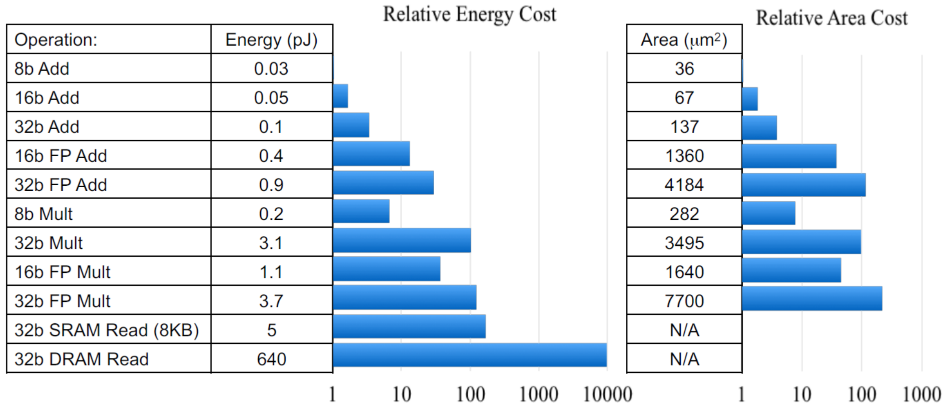
Memory Architecture and Power Considerations

- Keeping data local is key to minimizing power consumption
  - Very important for ASIC
- Floating-point is costly
  - Used in training of networks
  - Not needed in network inference engine
- Fixed-point doesn't need to be power-of-two



*NVIDIA 2017

Cost of Operations

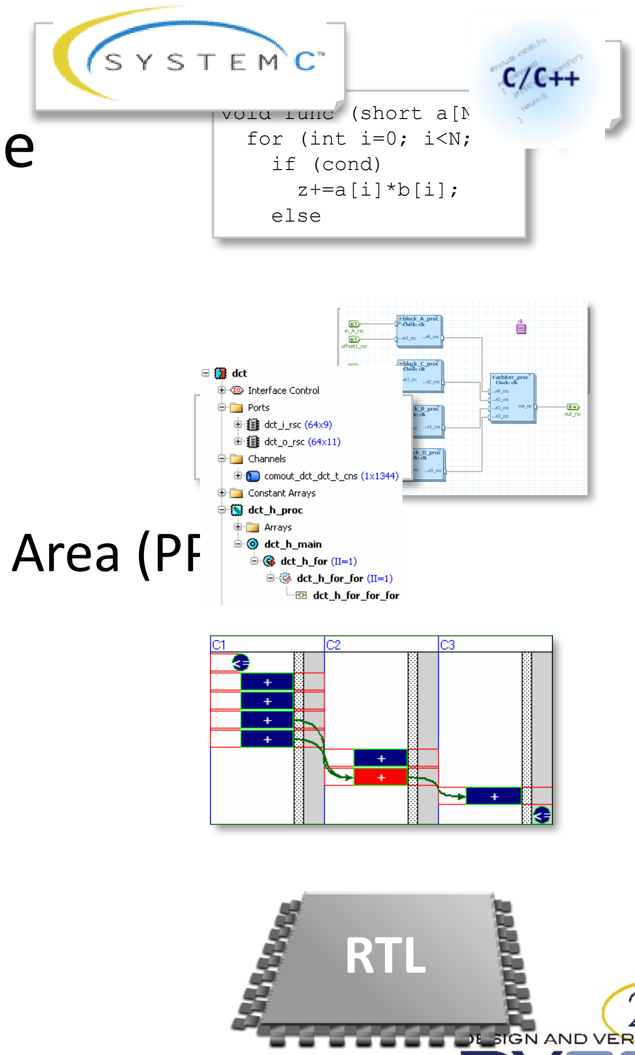


Operation:	Energy (pJ)	Area (μm²)
8b Add	0.03	36
16b Add	0.05	67
32b Add	0.1	137
16b FP Add	0.4	1360
32b FP Add	0.9	4184
8b Mult	0.2	282
32b Mult	3.1	3495
16b FP Mult	1.1	1640
32b FP Mult	3.7	7700
32b SRAM Read (8KB)	5	N/A
32b DRAM Read	640	N/A

Energy numbers are from Mark Horowitz "Computing's Energy Problem (and what we can do about it)", ISSCC 2014

Catapult HLS is the Only Solution for Rapid Algorithm to RTL

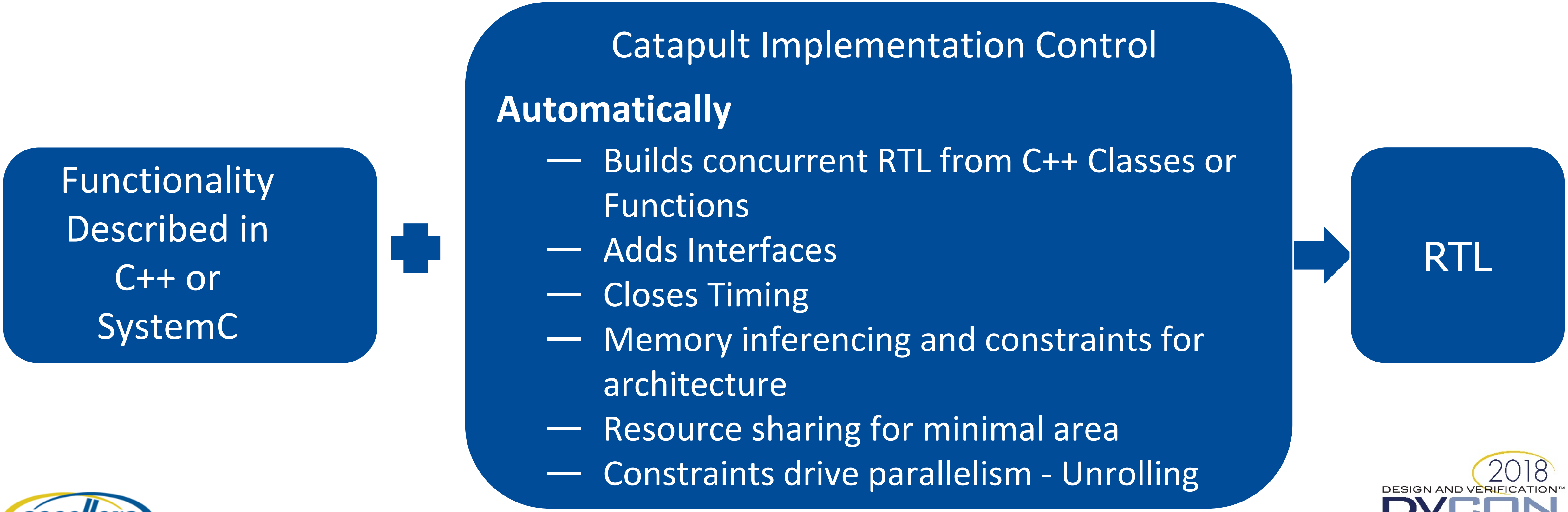
- Enable late functional changes without impacting schedule
  - Algorithms can be easily modified and regenerated
  - New technology nodes are easy (or FPGA to ASIC)
- Quickly evaluate power and performance of algorithms
  - Rapidly explore multiple options for optimal Power Performance Area (PF
- Accelerate design time with higher level of abstraction
  - 1 Year reduced to a few months
  - New features added in days not weeks
  - 5X less code than RTL



```
void func (short a[N
  for (int i=0; i<N;
    if (cond)
      z+=a[i]*b[i];
    else
```

Why Catapult HLS is So Much More Productive than RTL

- Catapult HLS separates functionality from implementation with powerful tool capabilities for controlling implementation



Functionality Described in C++ or SystemC

Catapult Implementation Control

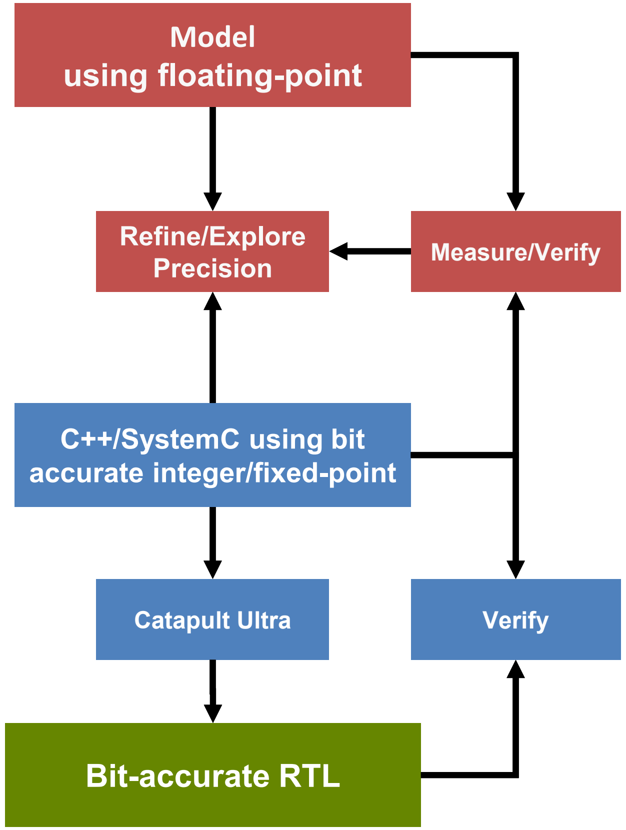
Automatically

- Builds concurrent RTL from C++ Classes or Functions
- Adds Interfaces
- Closes Timing
- Memory inferencing and constraints for architecture
- Resource sharing for minimal area
- Constraints drive parallelism - Unrolling

RTL

Precise Modeling of Bit-accuracy

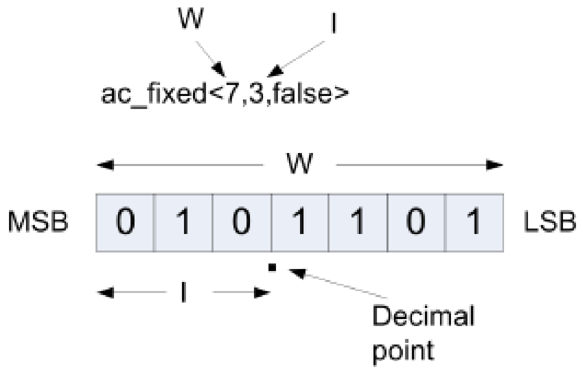
- HLS uses exact bit-widths to meet specification and save power/area
  - bit-widths are not always pow2 (1, 8, 16, 32, 64 bits)
- Rapid simulation of true hardware behavior
- RTL is correct by construction
  - Precise consistency of representation and simulation results between C++ algorithm and synthesized RTL


Model using floating-point
Refine/Explore Precision
Measure/Verify
C++/SystemC using bit accurate integer/fixed-point
Catapult Ultra
Verify
Bit-accurate RTL

The Algorithmic C fixed point data types are declared as:

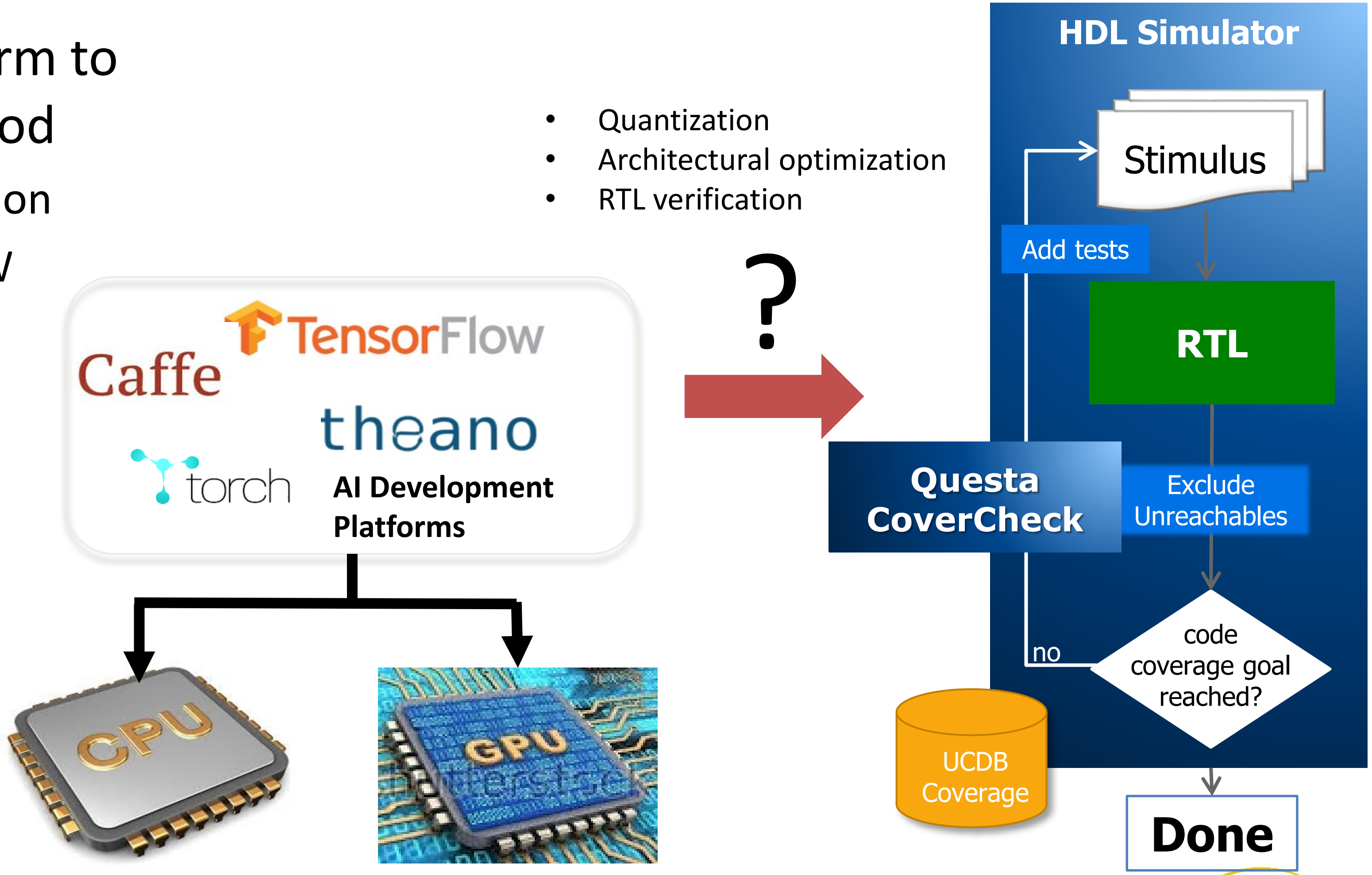
```
ac_fixed<W,I,S> x;
```

width #integer bits



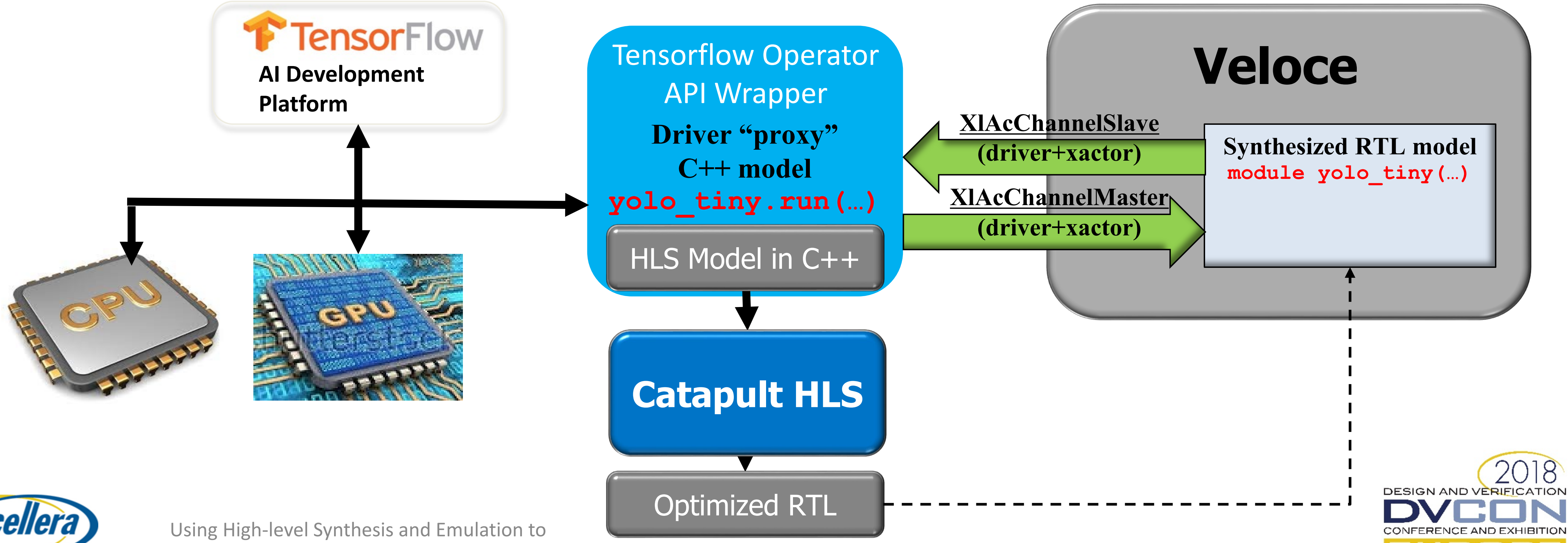
RTL Creation and Verification is Still a Bottleneck

- Going from AI development platform to optimized RTL is not well understood
  - How to verify hardware implementation
  - How to quantize and optimize the HW
- HLS delivers optimized RTL quickly but…
  - RTL verification is slow
  - Hours/days/weeks of simulation on complex CNN designs



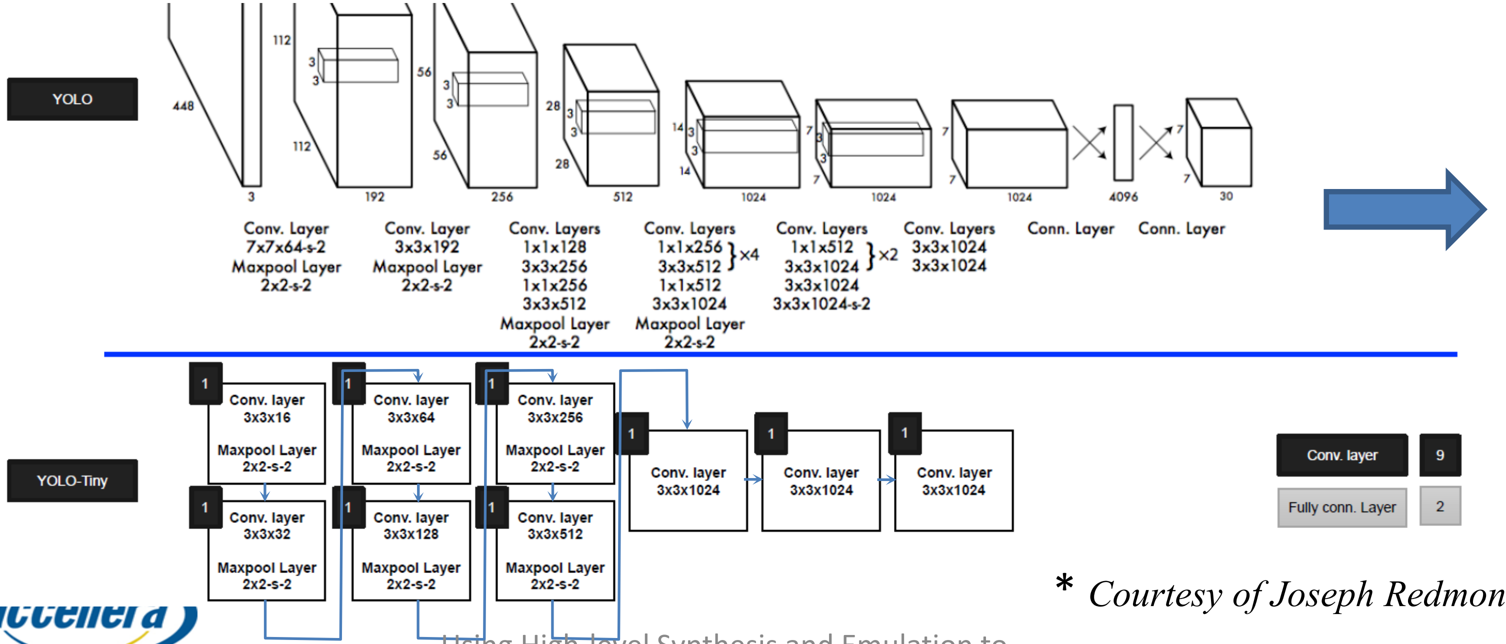
Catapult and Veloce Solve the Verification Bottleneck

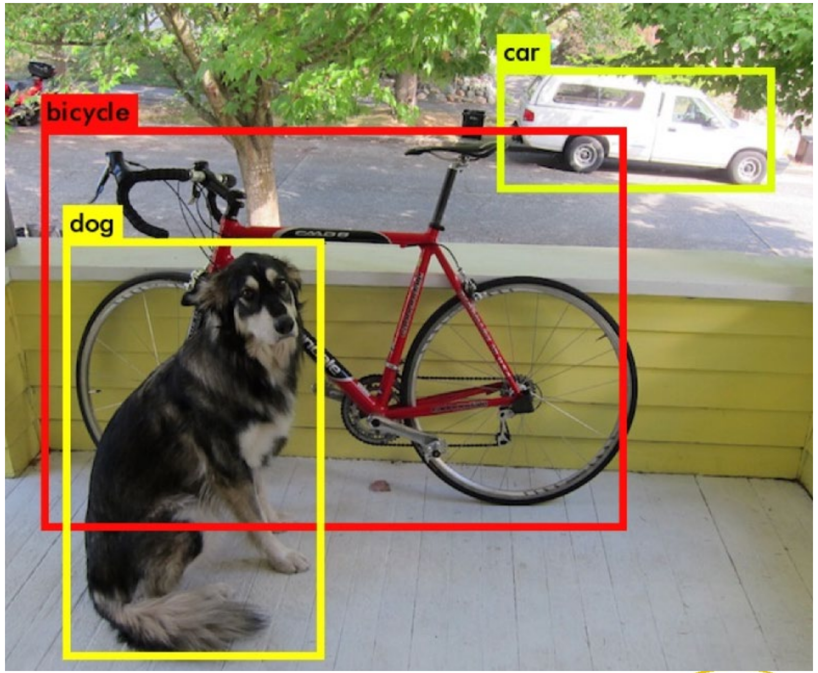
- Quickly verify synthesizable HLS C++ and RTL in the Tensorflow environment
  - Test the quantized HLS against the floating point model in tensorflow
- Reduce RTL verification from hours to minutes



Even “Small” CNNs are Computationally Intensive

- **Yolo Tiny***
  - used in object detection and classification for cell phones
  - Over 70 Billion MAC/Sec
  - Over 25 million weights
- Made up of mostly 2-d convolution and pooling layers





* *Courtesy of Joseph Redmon, https://pjreddie.com/darknet/yolo*

Yolo Tiny* *progressive refinement*

- This "Yolo Tiny" demo is based on the Google *TensorFlow* open-source machine learning technology based in Python
- The intent of the demo is to show techniques for progressive refinement from high-level abstracted TensorFlow CNN layer models written in Python3 down to HLS-synthesized RTL, i.e.,

  Original TensorFlow code ⇨ HLS "synthesis-friendly" C++ blocks ⇨ Synthesized RTL blocks

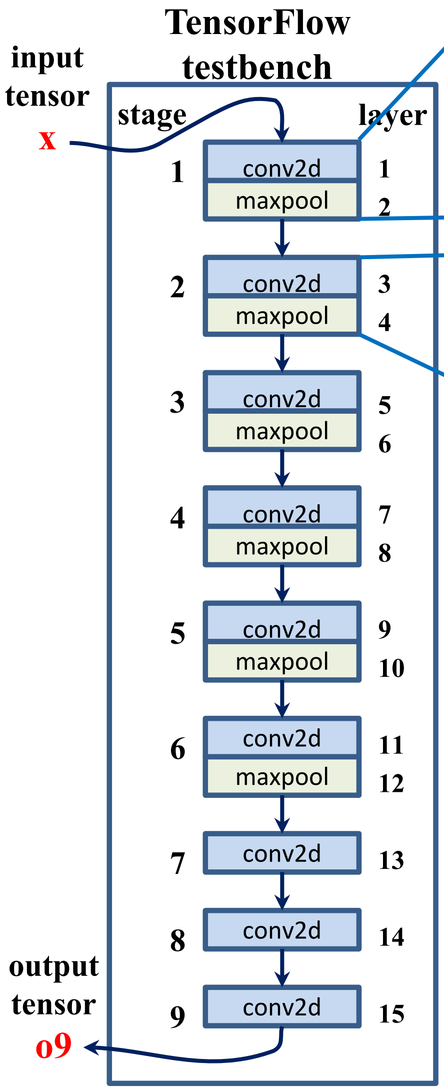
  - Then re-validation after each refinement, ultimately deploying an emulator for validation of synthesized RTL blocks

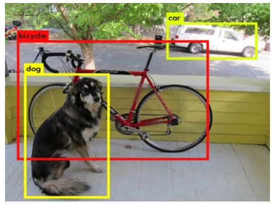
* *Courtesy of Joseph Redmon, https://pjreddie.com/darknet/yolo*

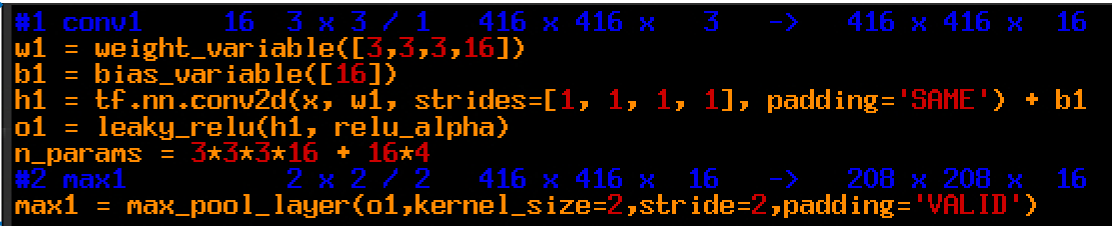
Architectural exploration acceleration and early tradeoff analysis

- Specifically we're exploring an approach that still keeps the whole flow in the architectural exploration phase of a machine learning design project, even while playing with the software/hardware representational tradeoffs
  - And, because emulation is deployed, being able to accelerate the RTL verification as well as starting to look at early power analysis, performance, etc.
- All of this can potentially be done before even thinking about bringing interface synthesis into the picture and targeting real hardware bus interfaces
  - I.e. doing all architectural exploration - even at RTL abstraction while using strictly abstract bit accurate native HLS data types such as ac_fixed<>, ac_channel<>'s for communication, etc.

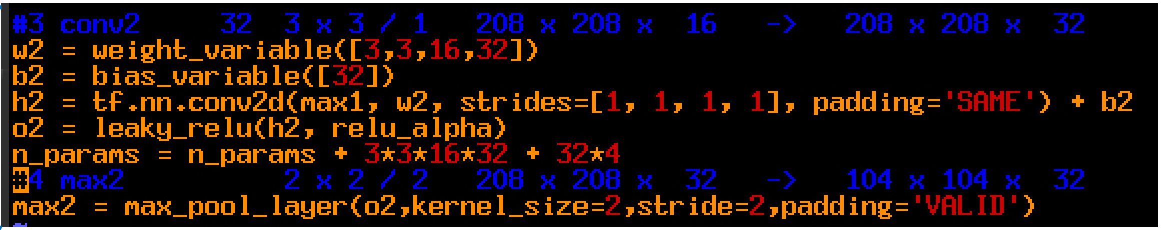
YoloTiny: Original python3/TensorFlow testbench







```
#1 conv1        16  3 x 3 / 1   416 x 416 x   3   ->   416 x 416 x  16
w1 = weight_variable([3,3,3,16])
b1 = bias_variable([16])
h1 = tf.nn.conv2d(x, w1, strides=[1, 1, 1, 1], padding='SAME') + b1
o1 = leaky_relu(h1, relu_alpha)
n_params = 3*3*3*16 + 16*4
#2 max1             2 x 2 / 2   416 x 416 x  16   ->   208 x 208 x  16
max1 = max_pool_layer(o1,kernel_size=2,stride=2,padding='VALID')
```

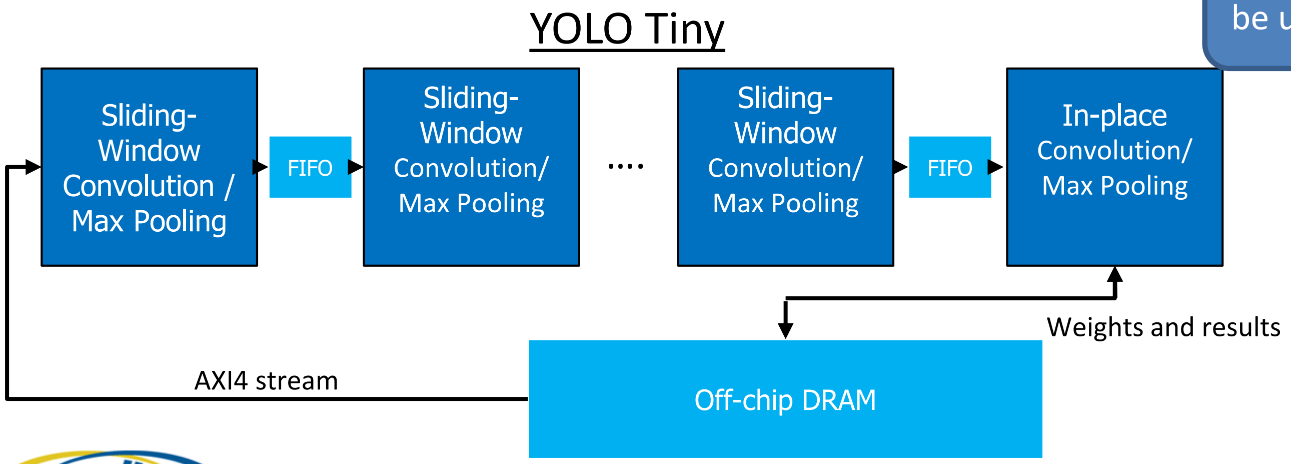


```
#3 conv2        32  3 x 3 / 1   208 x 208 x  16   ->   208 x 208 x  32
w2 = weight_variable([3,3,16,32])
b2 = bias_variable([32])
h2 = tf.nn.conv2d(max1, w2, strides=[1, 1, 1, 1], padding='SAME') + b2
o2 = leaky_relu(h2, relu_alpha)
n_params = n_params + 3*3*16*32 + 32*4
#4 max2             2 x 2 / 2   208 x 208 x  32   ->   104 x 104 x  32
max2 = max_pool_layer(o2,kernel_size=2,stride=2,padding='VALID')
```

- 9 stage CNN with 9 *conv2d* layers the first 6 of which are separated by *maxpool* layers which then feed densely connected *conv2d* layers
- First *conv2d* layer is fed an input tensor *'x'* which is the 2-dimensional `preprocessed_image` from the top level python3 *test.py* testbench
- 9th stage provides recognized images in the output tensor *'o9'* which is fed back up to top the level *test.py* for post processing of the output image, with classification and bounding box info included
- Each *conv2d* image is fed learned weights and biases for that stage
- Where preceded by a *maxpool* layer, it is fed by the output of that layer, otherwise simply the output of the preceding *conv2d* layer

Quickly Implement CNN Architectures Using HLS

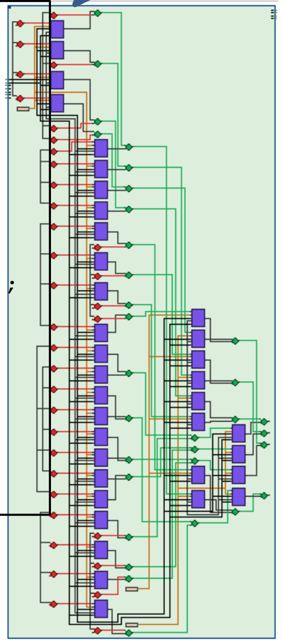
- Easily code multiple architectures in C++
  - Sliding-window architecture processes fmap data in raster order
  - In-place architecture reads weights once
- HLS constraints allow architectural exploration
  - Massive parallelism is possible
  - Evaluate power, performance, and area PPA across multiple architectures and microarchitectures


YOLO Tiny
Sliding-Window Convolution / Max Pooling
FIFO
Sliding-Window Convolution/ Max Pooling
....
Sliding-Window Convolution/ Max Pooling
FIFO
In-place Convolution/ Max Pooling
Weights and results
AXI4 stream
Off-chip DRAM

Loops can be unrolled

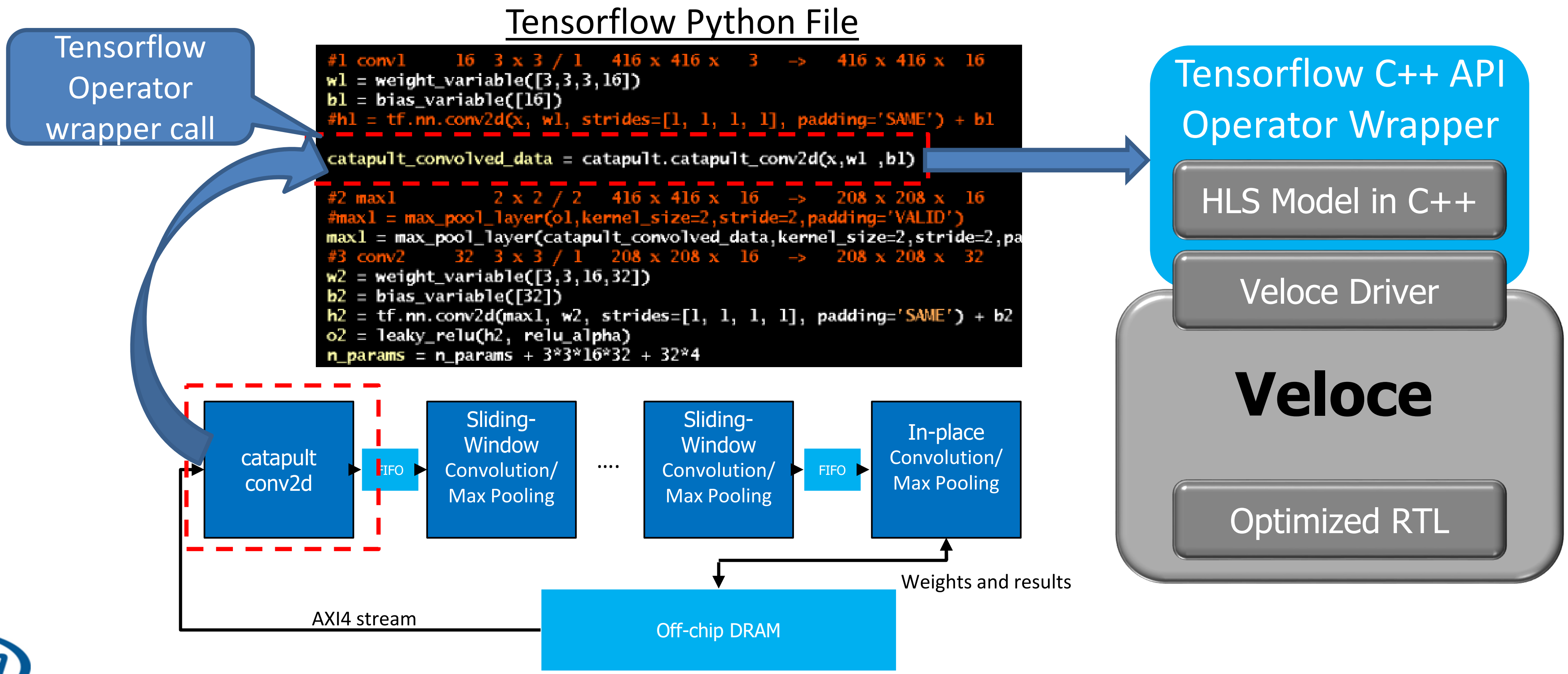
```
FMAP_HEIGHT:for(int r=0;r<IN_HEIGHT;r++){
  IN_CHAN:for(int ic=0;ic<IN_CHANNELS;ic++){
    FMAP_WIDTH:for(int c=0;c<IN_WIDTH+1;c++){
      < Read feature map data stream >
      < Sliding window of feature map data >
      OUT_CHAN:for(int oc=0;oc<OUT_CHANNELS;oc++){
        < Read kernel weights from SRAM >
        KERNEL_Y:for(int i=0;i<3;i++){
          KERNEL_X:for(int j=0;j<3;j++){
            acc += fmap_window[r+i][c+j] * kernel[i*3+j];
          }
        }
        < Write out partial output channel sums >
      }
    }
  }
}
```

36-parallel multipliers



Easily Test HW Models and RTL Quickly

- Swap any layer or the entire design
  - HLS C++ executable or RTL running on Veloce is a python function call in tensorflow

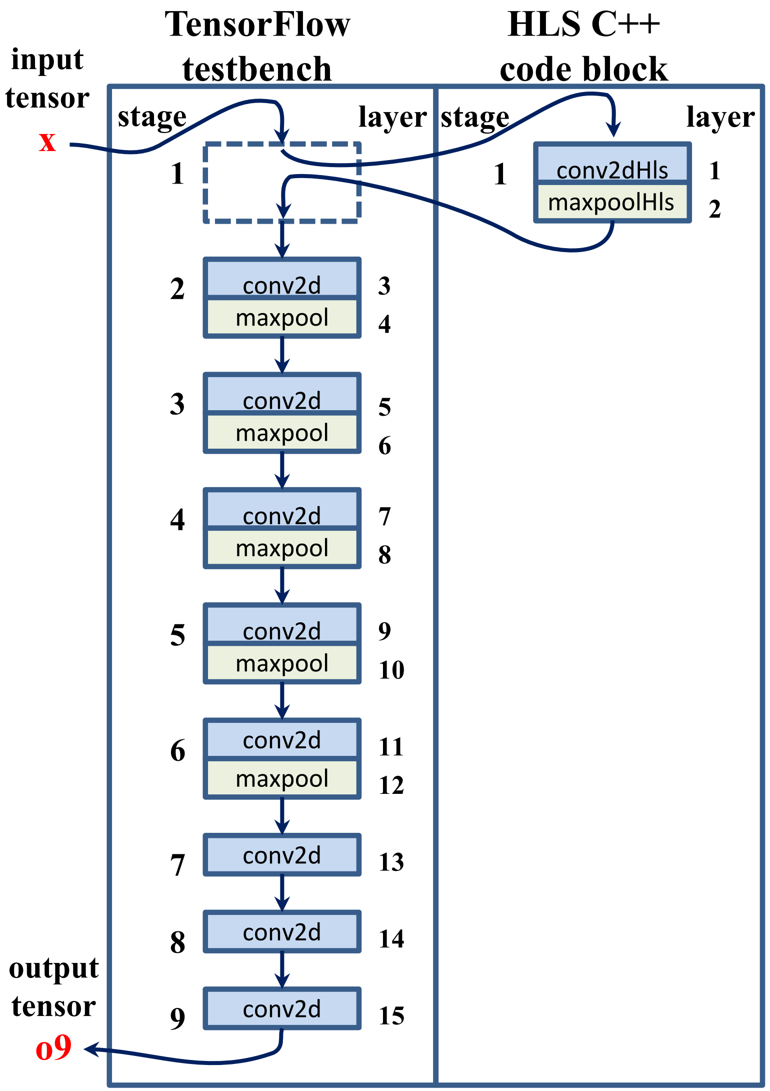


Tensorflow Python File

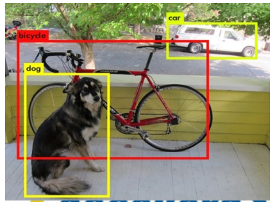
```
#1 conv1      16  3 x 3 / 1   416 x 416 x   3   ->   416 x 416 x  16
w1 = weight_variable([3,3,3,16])
b1 = bias_variable([16])
#h1 = tf.nn.conv2d(x, w1, strides=[1, 1, 1, 1], padding='SAME') + b1

catapult_convolved_data = catapult.catapult_conv2d(x,w1 ,b1)

#2 max1             2 x 2 / 2   416 x 416 x  16   ->   208 x 208 x  16
#max1 = max_pool_layer(o1,kernel_size=2,stride=2,padding='VALID')
max1 = max_pool_layer(catapult_convolved_data,kernel_size=2,stride=2,pa
#3 conv2      32  3 x 3 / 1   208 x 208 x  16   ->   208 x 208 x  32
w2 = weight_variable([3,3,16,32])
b2 = bias_variable([32])
h2 = tf.nn.conv2d(max1, w2, strides=[1, 1, 1, 1], padding='SAME') + b2
o2 = leaky_relu(h2, relu_alpha)
n_params = n_params + 3*3*16*32 + 32*4
```

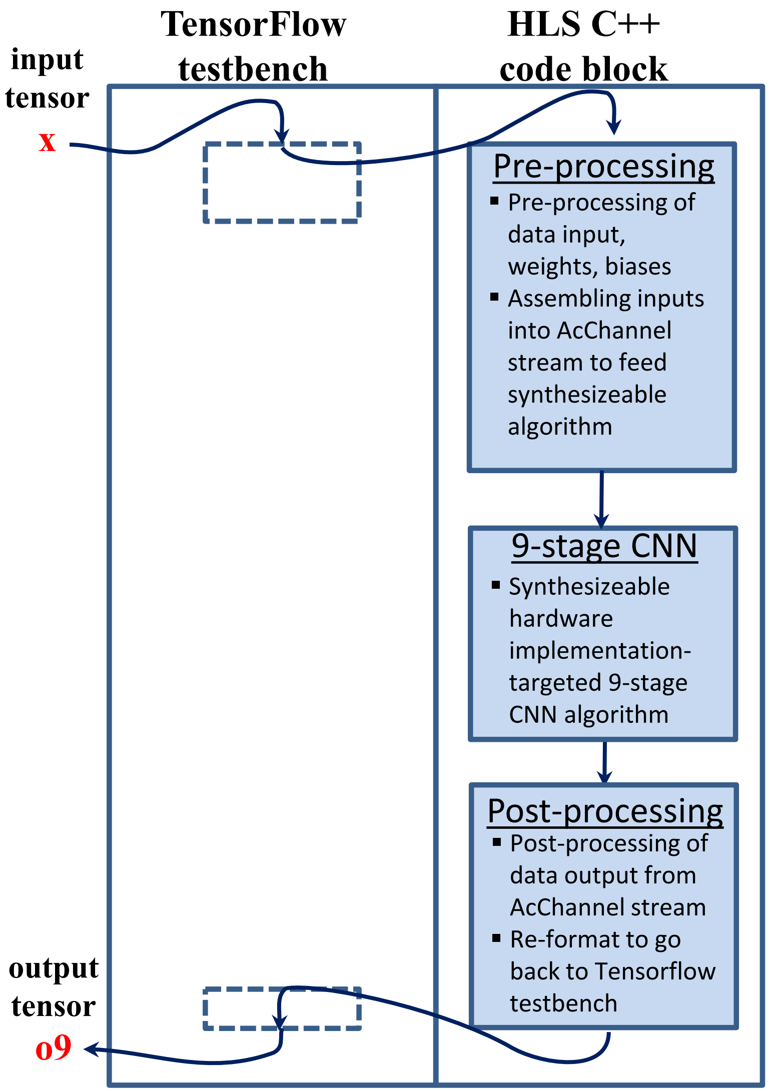
Tensorflow Operator wrapper call
Tensorflow C++ API Operator Wrapper
HLS Model in C++
Veloce Driver
Veloce
Optimized RTL
catapult conv2d
FIFO
Sliding-Window Convolution/ Max Pooling
....
Sliding-Window Convolution/ Max Pooling
FIFO
In-place Convolution/ Max Pooling
Weights and results
AXI4 stream
Off-chip DRAM

YoloTiny: Selected layers of TensorFlow testbench broken out to “HLS-friendly” C++ implementation-targeted algorithms



1-stage breakout





Pre-processing

- Pre-processing of data input, weights, biases
- Assembling inputs into AcChannel stream to feed synthesizeable algorithm

9-stage CNN

- Synthesizeable hardware implementation-targeted 9-stage CNN algorithm

Post-processing

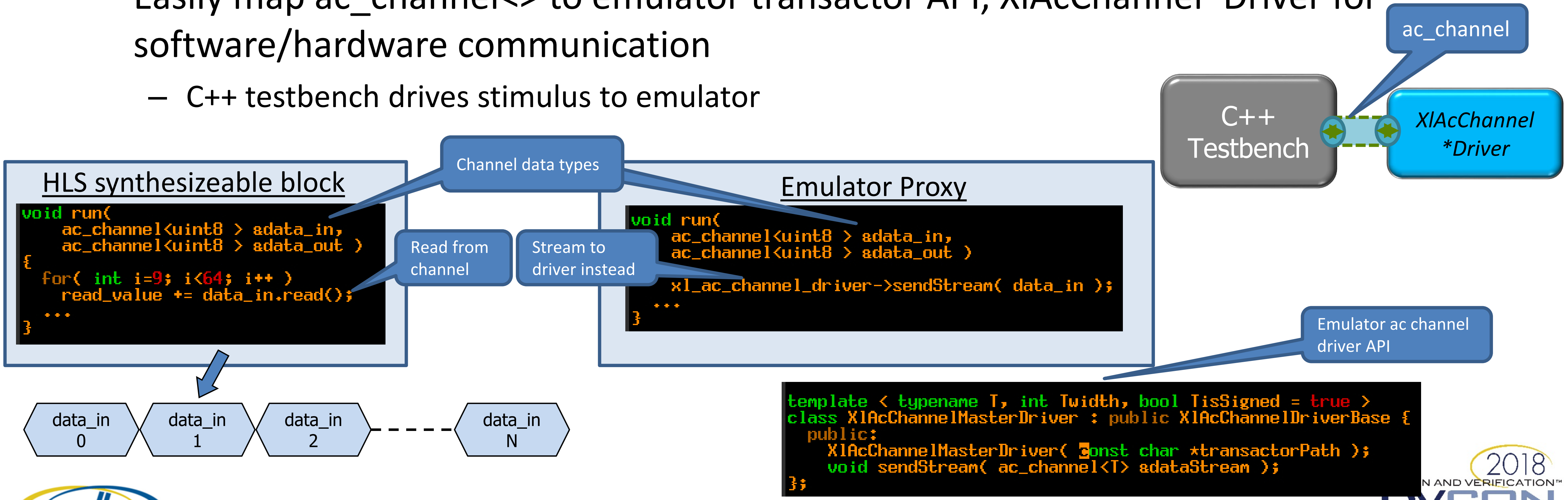
- Post-processing of data output from AcChannel stream
- Re-format to go back to Tensorflow testbench

9-stage breakout

- Here we break out 1 or more of the original TensorFlow layers to experiment with implementation synthesis
- We still run the new C++ code prototypes in the context of the original TensorFlow testbench
- We pre-verify the synthesizeable code even before we generate RTL from it

Easy Modeling, Synthesis, and Emulation of Streaming Interfaces

- AC channel parametrized class allows designers to model streaming data interfaces in untimed C++
- Easily map ac_channel<> to emulator transactor API, XlAcChannel*Driver for software/hardware communication
  - – C++ testbench drives stimulus to emulator



HLS synthesizeable block

```
void run(
    ac_channel<uint8 > &data_in,
    ac_channel<uint8 > &data_out )
{
  for( int i=9; i<64; i++ )
    read_value += data_in.read();
    ...
}
```

Emulator Proxy

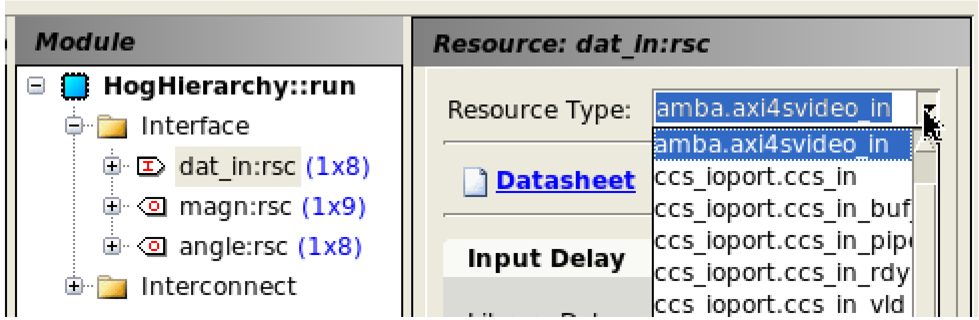
```
void run(
    ac_channel<uint8 > &data_in,
    ac_channel<uint8 > &data_out )

    xl_ac_channel_driver->sendStream( data_in );
   ...
}
```

```
template < typename T, int Twidth, bool TisSigned = true >
class XlAcChannelMasterDriver : public XlAcChannelDriverBase {
  public:
    XlAcChannelMasterDriver( const char *transactorPath );
    void sendStream( ac_channel<T> &dataStream );
};
```

Interface Synthesis Makes HW Communication with Veloce Easy

- Interface synthesis allows the interface protocol to be defined using the HLS tool
- ac_channel maps to data/ready/valid protocol in hardware (*_wait interface)
- *XlAcChannel*Transactor* and *_wait interfaces bolt together seamlessly

Catapult Architectural Constraints View



Module

- HogHierarchy::run
  - Interface
    - dat_in:rsc (1x8)
    - magn:rsc (1x9)
    - angle:rsc (1x8)
  - Interconnect

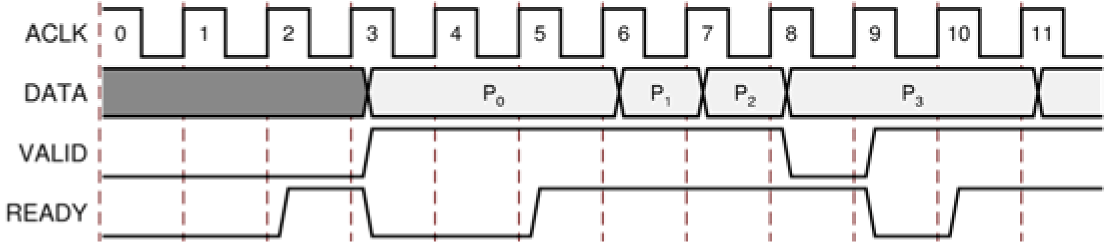
Resource: dat_in:rsc

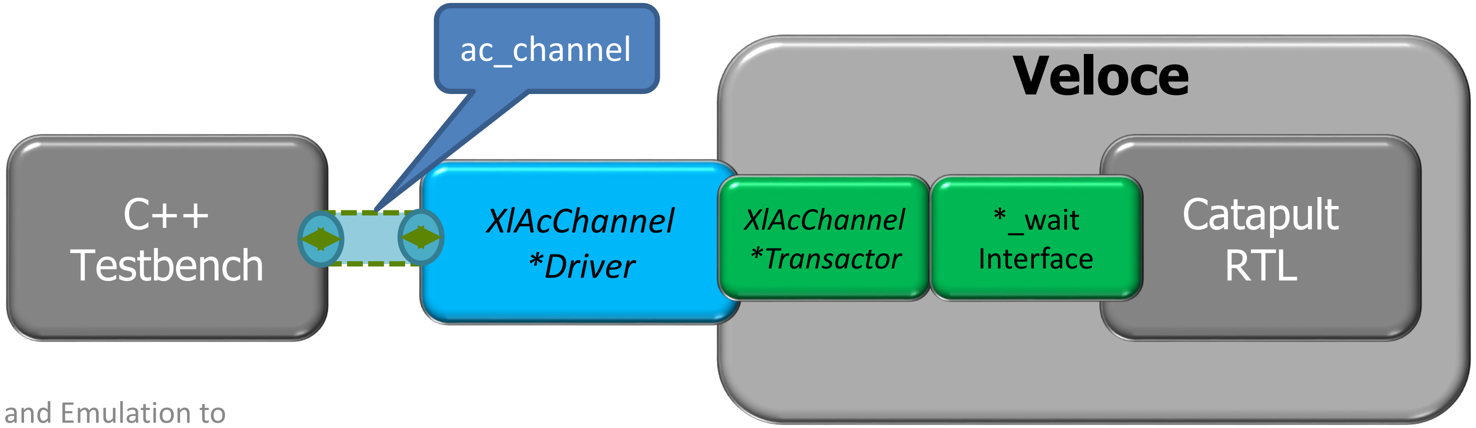
Resource Type: amba.axi4svideo_in

- amba.axi4svideo_in
- ccs_ioport.ccs_in
- ccs_ioport.ccs_in_buf
- ccs_ioport.ccs_in_pip
- ccs_ioport.ccs_in_rdy
- ccs_ioport.ccs_in_vld

Datasheet

Input Delay





ac_channel

Veloce

C++ Testbench

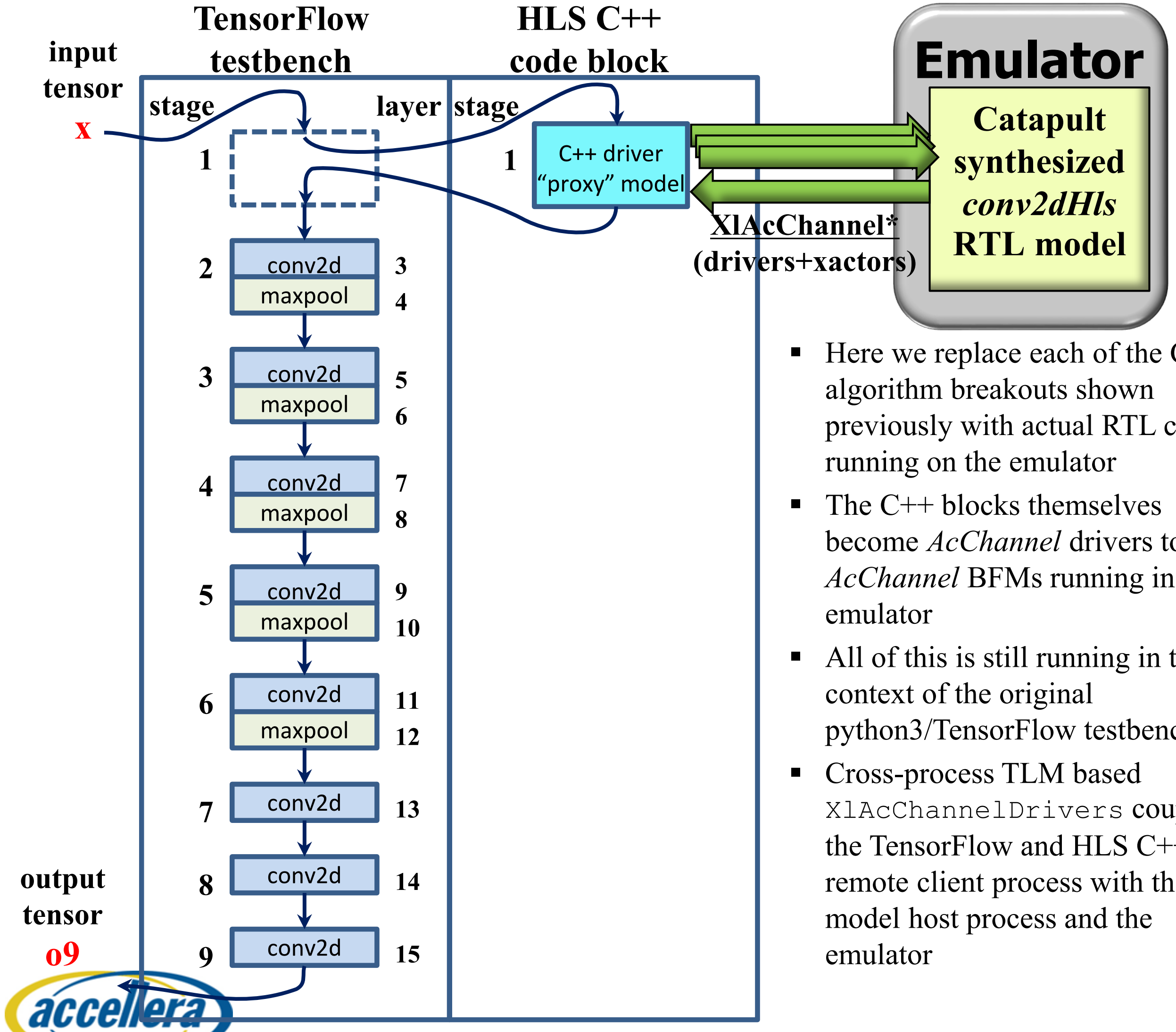
*XlAcChannel *Driver*

*XlAcChannel *Transactor*

*_wait Interface

Catapult RTL

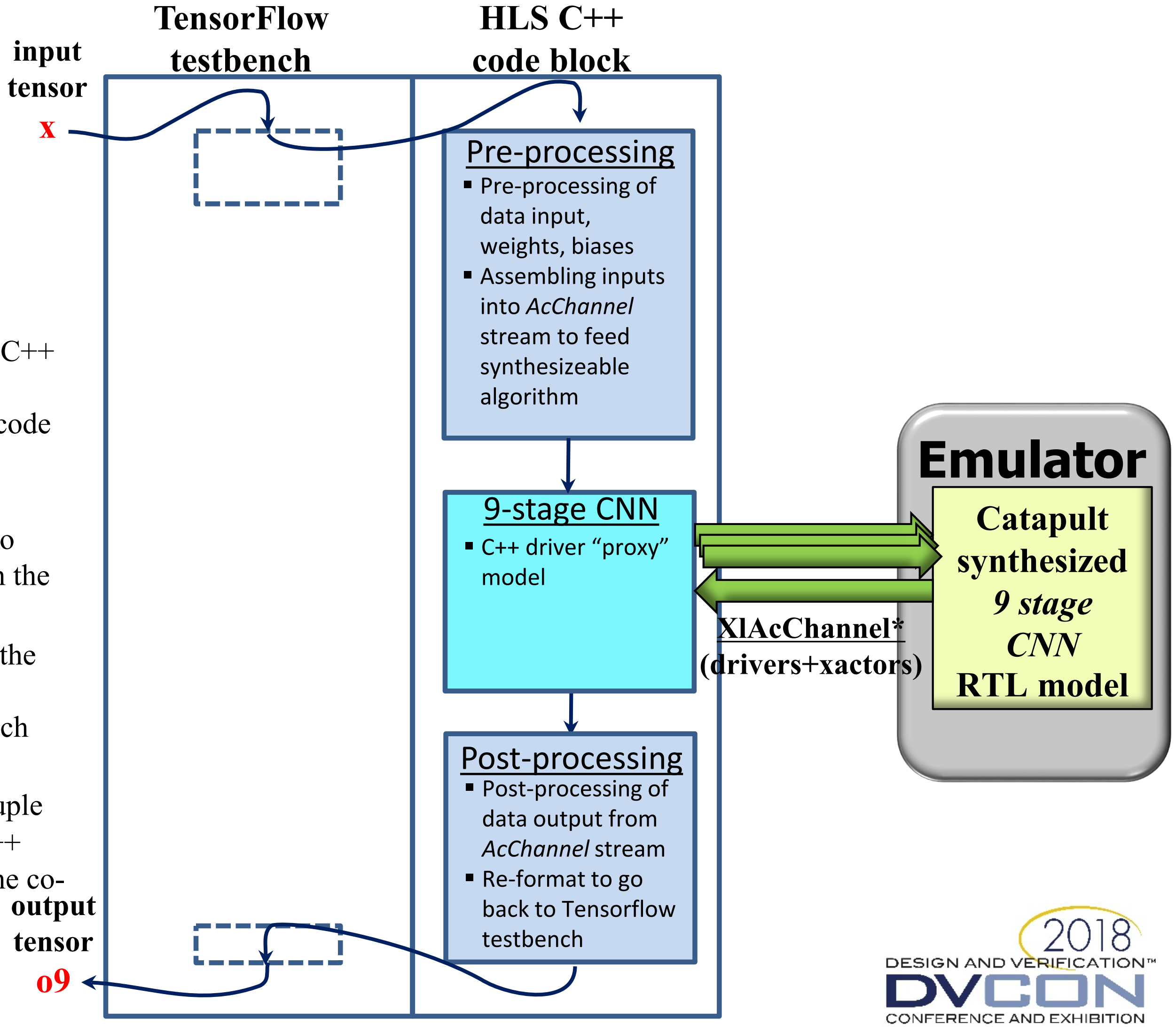
YoloTiny: C++ implementations of CNNs replaced with synthesized RTL blocks



input tensor
x
TensorFlow testbench
stage
layer
HLS C++ code block
stage
1
2
3
4
5
6
7
8
9
conv2d
maxpool
3
4
5
6
7
8
9
10
11
12
13
14
15
1
C++ driver "proxy" model
XlAcChannel*
(drivers+xactors)
Emulator
Catapult synthesized *conv2dHls* RTL model
output tensor
o9

1-stage breakout

- Here we replace each of the C++ algorithm breakouts shown previously with actual RTL code running on the emulator
- The C++ blocks themselves become *AcChannel* drivers to *AcChannel* BFMs running in the emulator
- All of this is still running in the context of the original python3/TensorFlow testbench
- Cross-process TLM based `XlAcChannelDrivers` couple the TensorFlow and HLS C++ remote client process with the co-model host process and the emulator



input tensor
x
TensorFlow testbench
HLS C++ code block
Pre-processing
- Pre-processing of data input, weights, biases
- Assembling inputs into *AcChannel* stream to feed synthesizeable algorithm

9-stage CNN
- C++ driver "proxy" model

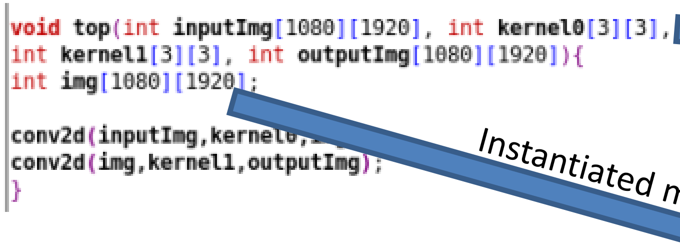
Post-processing
- Post-processing of data output from *AcChannel* stream
- Re-format to go back to Tensorflow testbench

XlAcChannel*
(drivers+xactors)
Emulator
Catapult synthesized *9 stage CNN* RTL model
output tensor
o9

9-stage breakout

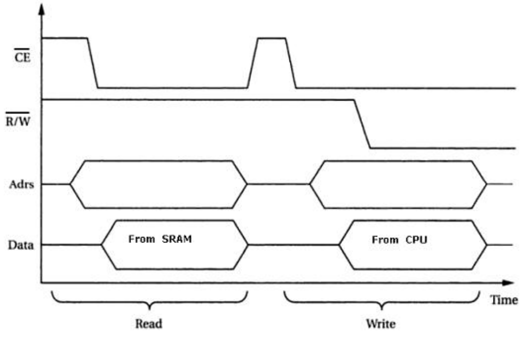
Memory Inference and 0-time Back-Door Memory Accesses

- Large C++ arrays automatically mapped to ASIC or FPGA memories
- Well supported in emulation using general purpose **`XlMemoryTransactor`** module
- Arrays on the design interface can be synthesized as memory interfaces
- Internal arrays synthesized to instantiated (black boxed) memories



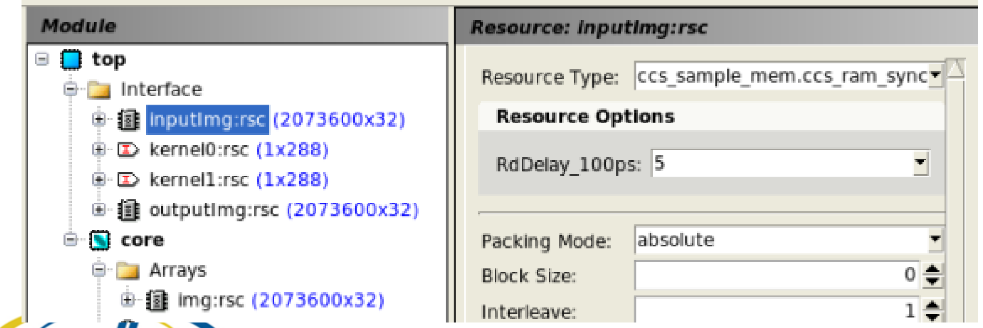
```
void top(int inputImg[1080][1920], int kernel0[3][3],
int kernel1[3][3], int outputImg[1080][1920]){
int img[1080][1920];

conv2d(inputImg,kernel0,
conv2d(img,kernel1,outputImg);
}
```

Memory interface protocol



Instantiated memory transactor with 0-time back-door interface

Catapult Architectural Constraints View



Module
- top
  - Interface
    - inputImg:rsc (2073600x32)
    - kernel0:rsc (1x288)
    - kernel1:rsc (1x288)
    - outputImg:rsc (2073600x32)
  - core
    - Arrays
      - img:rsc (2073600x32)

Resource: inputImg:rsc

Resource Type: ccs_sample_mem.ccs_ram_sync

Resource Options

RdDelay_100ps: 5

Packing Mode: absolute

Block Size: 0

Interleave: 1

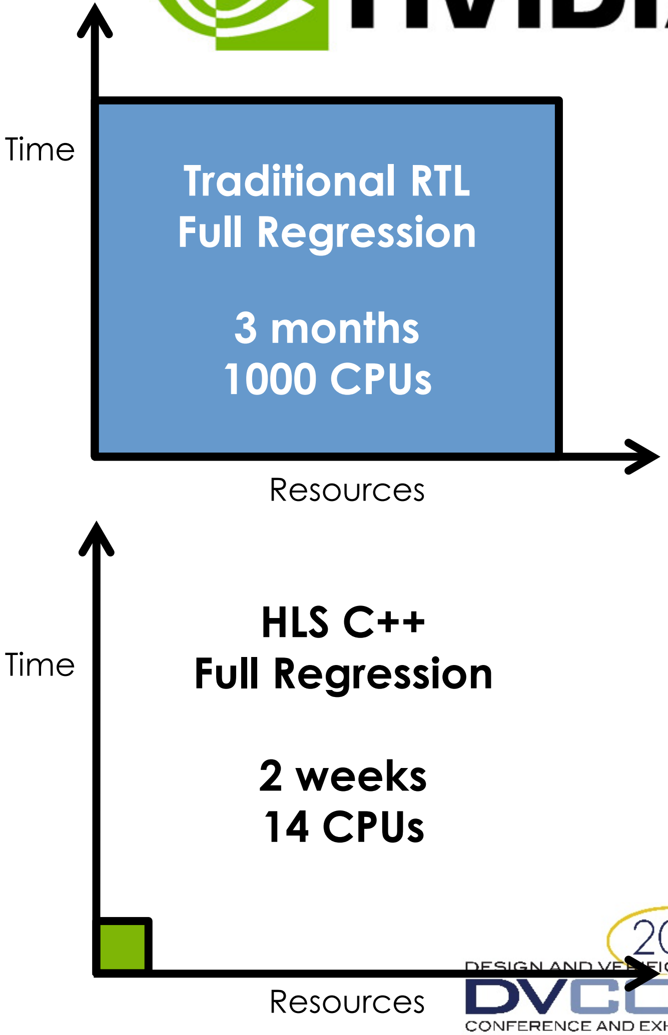
Sythesized RTL

`XlMemoryTransactor`
`(open-kit)`

SOME CUSTOMER TESTIMONIAL EXAMPLES

NVIDIA Cuts Verification Cost by 80%

- 10M gates video decoder for Tegra X1
- Schedule and goals couldn't be met without adding 20 engineers to a team of already 60
- Invested in Catapult instead
  - Improved design productivity by 50%
  - Cut verification cost by 80%
- "Saved their skin – Twice"
  - Converted VP9/H.265 from 8 to 10 bit color in weeks
  - Re-optimized IP from 20nm/500Mhz to 28nm/800Mhz in 3 days



Source: *NVIDIA white paper*
http://go.mentor.com/4N9cP

NVIDIA Research New Methodology with Catapult

Machine Learning Accelerator SoC using an Object-Oriented HLS flow

- NVIDIA Research with DARPA - New methodology for 10x faster chip design
- Developing libraries of HLS components to target 80% of future NVIDIA chips
- Used in NVDLA HW
- 2 DAC Papers; 2016,2018

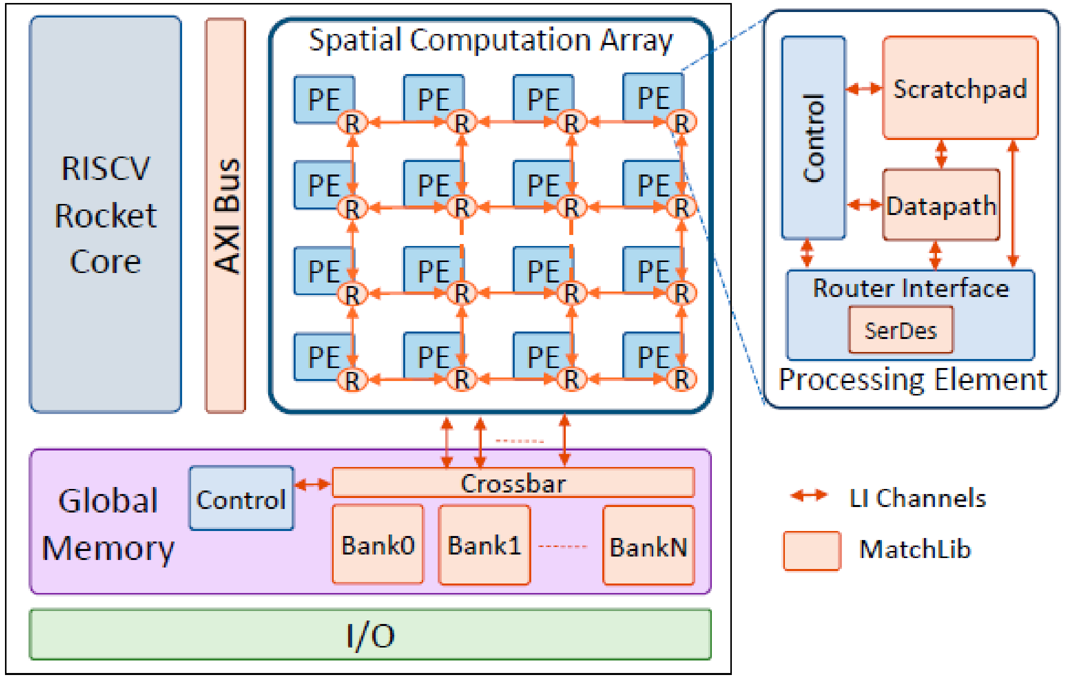


Figure 5: Prototype SoC

[Hardware Accelerator for Mobile Computer Vision Applications]

[Digital VLSI Flow for High-Productivity SoC Design]

Bosch Automotive Catapult Success

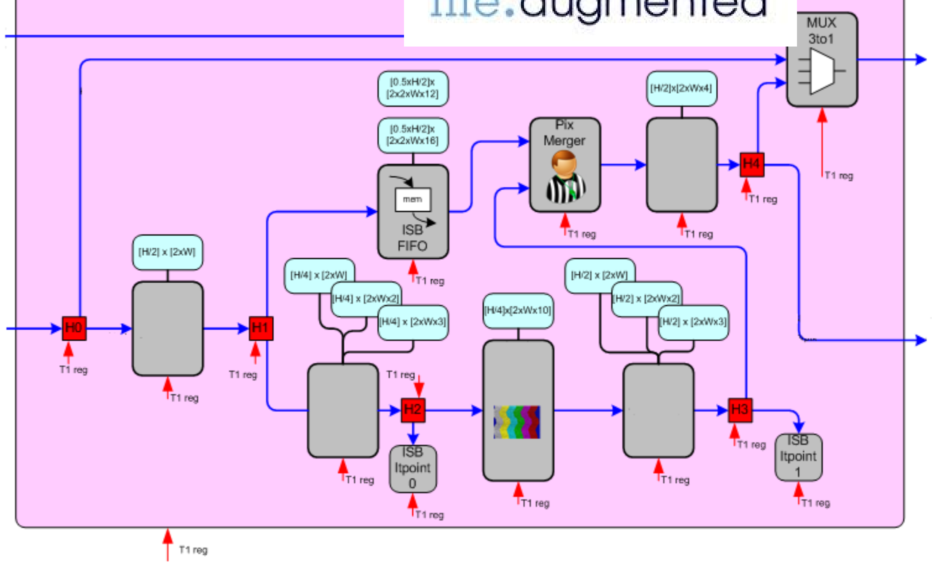
- Bosch needed big change to stay #1 in Automotive Safety
- Started new subsidiary for Autonomous Driving
- Decided HLS * lack of resources *quickly react to changes
- Mentor worked as key partner with a focus on success
- **Result - Catapult HLS success on time first IP deliverable**
- Delivered new designs ahead of schedule in 7 months with evolving specifications; improved quality over RTL

[BOSCH VISIONTEC Rapidly Brings New Automotive IP to Market using Catapult HLS Platform]()

ST Imaging HLS Success for ISP (Automotive)

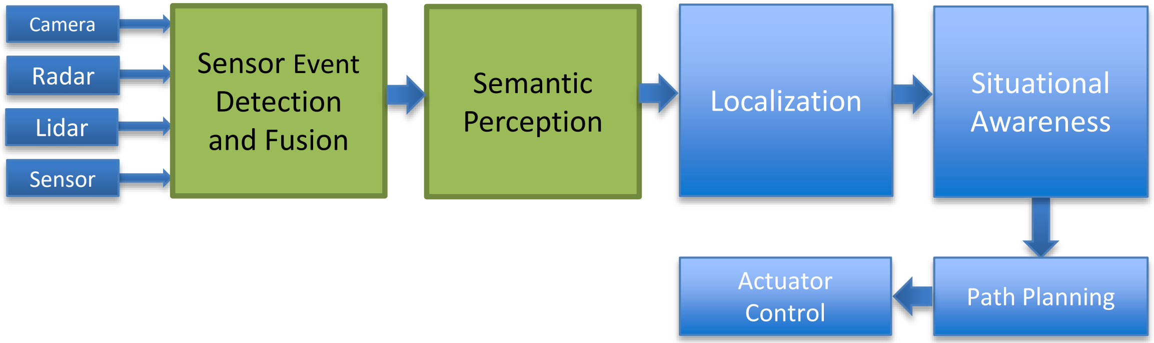
- To date created 50+ Image Processing IPs using HLS Imaging Template

- Why they use HLS and Catapult (their words)
  - Increase IP value
  - Improve IP performance versus power & area
  - Reduce project cost
- Experience with HLS
  - Less code to write and debug
  - Fast integration of new features
  - Algorithm and architecture exploration possible
  - Fast Verification using C++
- On-Demand Webinar and White Paper



STMicroelectronics Quickly Brings Automotive Image Signal Processing to Market with High-Level Synthesis

Mentor Automotive DRS360 Using Catapult for Both Computer Vision and Neural Networking Acceleration



	Classic Development	**With CatapultC**
Image Filtering	2 man/week (expert)	1 man/week (beginner)
Features Detection	2 man/week (expert)	1 man/week (intermediate)
Neural Network (FF)	2 man/week (expert)	*Few hours*

- CatapultC is *close* to SW development
  - It is C++ with more constraints
- 3x SW engineers on the market than HW engineers

➔Ramp up is reduced dramatically

➔Produce deployable PoC in short delays

➔Optimize for production when final requirements are set

Summary

- Next generation AI algorithms are massively complex
- Delivering optimized RTL with the best PPA on time is very difficult
  - Achieving the most optimal architecture is hard to do in hand-code RTL
  - Going from AI development platform to RTL is not well understood
  - Verify the RTL is too time consuming
    - Billions of computations
    - Massively parallel hardware
- Catapult and Veloce provide a push-button path from high-level model to rapidly verified RTL