2020 DESIGN AND VERIFICATION™ DVCON CONFERENCE AND EXHIBITION UNITED STATES

Application Optimized HW/SW Design & Verification of a Machine Learning SoC

Lauro Rizzatti – Rizzatti LLC
Russell Klein – Mentor, A Siemens Business
Stephen Bailey – Mentor, A Siemens Business
Andrew Meier – Mentor, A Siemens Business

Mentor® A Siemens Business

accellera SYSTEMS INITIATIVE

Agenda

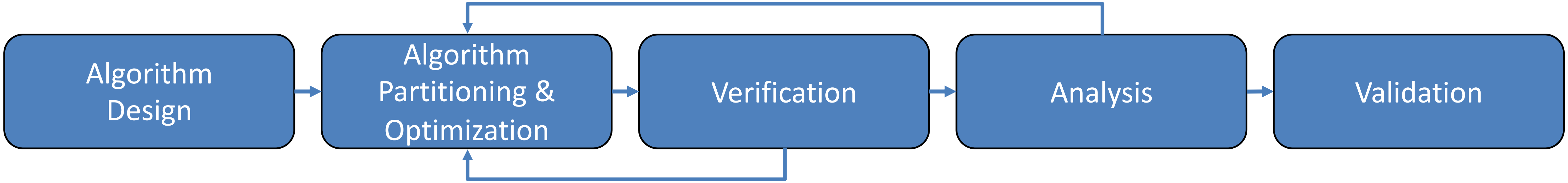
- Software to Systems - Lauro Rizzatti
- High-Level Synthesis (HLS) – Russell Klein
- Verification:
    - Hybrid Verification – Andrew Meier
    - Accelerated Verification – Stephen Bailey, John Stickley
- Conclusion and Q & A

Lauro Rizzatti

SOFTWARE TO SYSTEMS

Tutorial Objective & Contents

- This tutorial details the process of migrating an ML algorithm from generic software to a hardware implementation customized to the specific requirements of a system



- The migration advances through 5 steps:
  - #1: Design and verify an ML algorithm to be embedded in an application specific SoC
  - #2: Partition the algorithm in HW/SW and optimize it for performance/power/area in the context of the SoC and the accompanying software stack
  - #3: Verify the SoC at different levels of abstraction
  - #4: Analyze the SoC for power, performance, formal and coverage at the RT level
  - #5: Perform system validation via FPGA prototyping

What Problem Do We Address?

- Today, many embedded systems embody algorithms that were originally developed as software applications
  - Either on general purpose computers or on embedded systems
- Migrating these algorithms to demanding applications running on embedded systems is hitting a roadblock
  - Substantial increases in compute requirements cannot be met by slow performance enhancements of traditional embedded computing
  - Power constraints defeat conventional CPU-based architectures
- *<u>The algorithms must be accelerated in hardware</u>*
  - This tutorial will describe how to achieve this objective

Many Possible Architectures

- Algorithms are still evolving in leading edge technological domains, such as Machine Learning, 5G and state-of-the-art Video
  - What architecture is best?
  - No way to try very many alternatives in RTL
- Optimize for Power, Energy, Performance, Area
  - All need to be optimized
  - Finding the best trade-off is challenging
  - Having a SW-driven or application-driven methodology at the start in continued use in the flow is important
  - Data movement is key
    - Memory, bandwidth, and caching significantly impact all of these

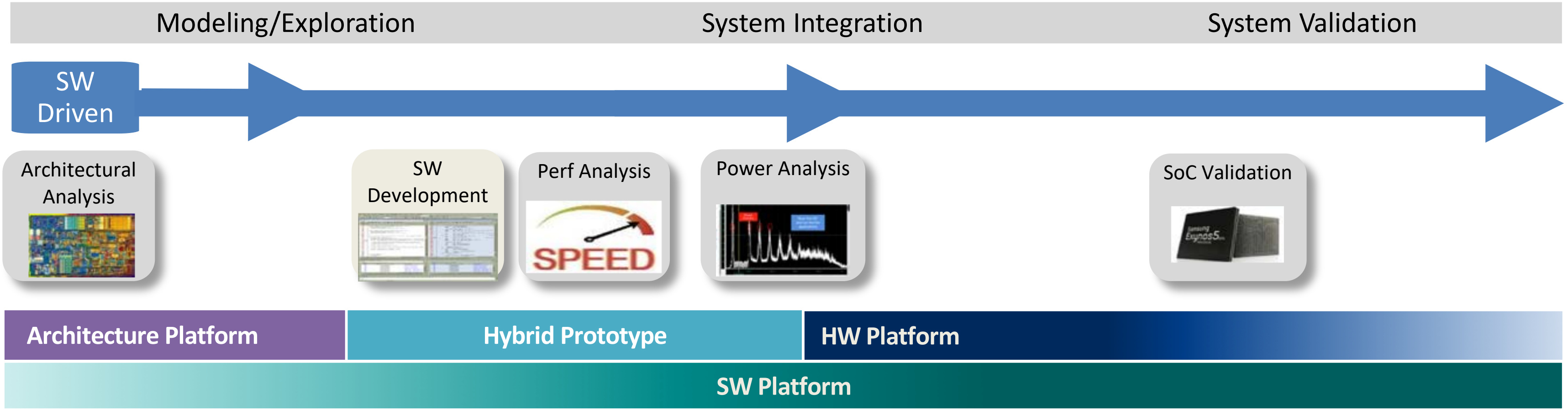
Software-driven system design

Existing Approach
SoC-Driven System Design

- Design objective defined by system architect
- HW/SW partitioning planned
- Virtual platform created and validated
- Power/performance optimization based on sub-system TB
- SW application optimized to run on HW platform

New Approach
Software-Driven System Design

- SW available at day one of project
- SW used to explore HW architecture
- Platforms evolve in parallel (HW/SW)
- SoC optimized in context of SW (power/performance)
- Pre-silicon SoC validated with SW
- Apps/benchmarks optimized for HW/SW platform



Modeling/Exploration | System Integration | System Validation

SW Driven

Architectural Analysis

SW Development

Perf Analysis

Power Analysis

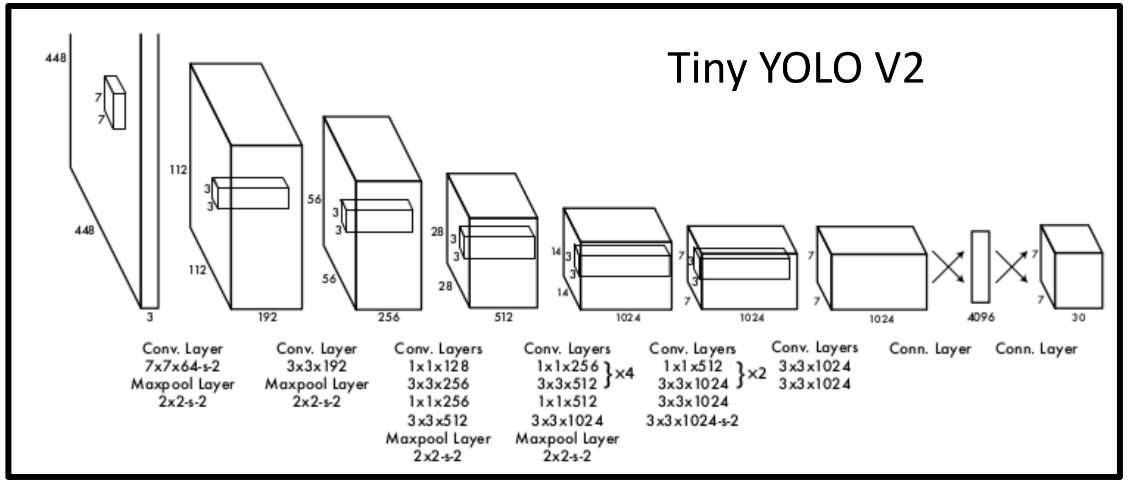
SoC Validation

Architecture Platform | Hybrid Prototype | HW Platform

SW Platform

Case-Study: Tiny YOLO V2 Algorithm

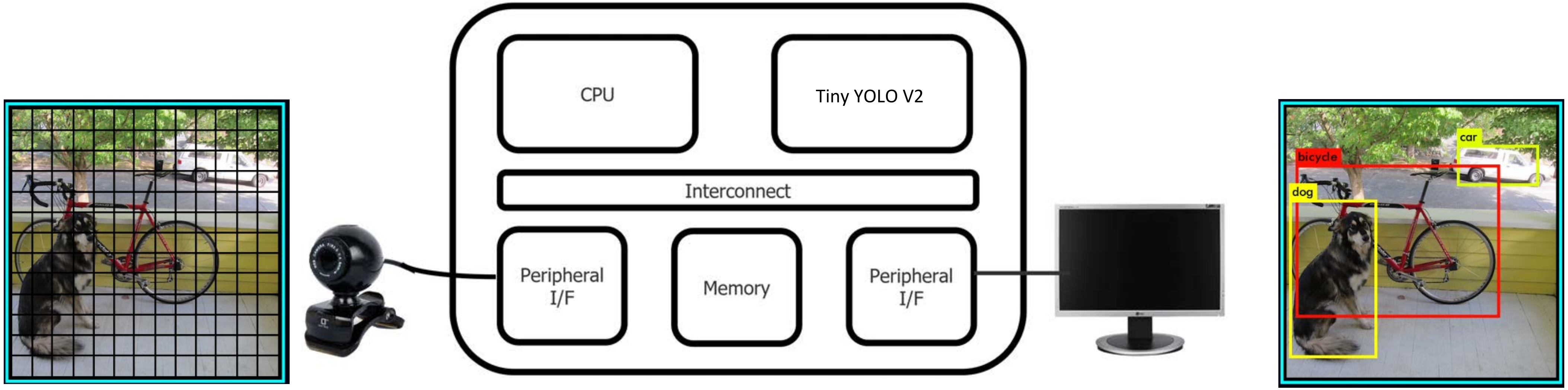
- Our tutorial is based on “Tiny YOLO V2*”, a low computational object recognition algorithm implemented in the TensorFlow framework
  - Tiny YOLO V2 is a 23-layer convolution neural network that reads a small format image and detects objects within the frame
  - It executes approximately 3.2 billion multiply accumulate (MAC) operations per inference
  - It can classify 20 objects, it is well studied, and has implementations in several machine learning frameworks
- Tiny YOLO is used in compute constrained or power constrained devices, such as cell phones or other devices where computational and battery power is concerned


Tiny YOLO V2
448
7
7
448
3
112
3
3
112
192
56
3
3
56
256
28
3
3
28
512
14
3
3
14
1024
7
3
3
7
1024
7
7
1024
4096
7
7
30
Conv. Layer 7x7x64-s-2 Maxpool Layer 2x2-s-2
Conv. Layer 3x3x192 Maxpool Layer 2x2-s-2
Conv. Layers 1x1x128 3x3x256 1x1x256 3x3x512 Maxpool Layer 2x2-s-2
Conv. Layers 1x1x256 3x3x512 }×4 1x1x512 3x3x1024 Maxpool Layer 2x2-s-2
Conv. Layers 1x1x512 3x3x1024 }×2 3x3x1024 3x3x1024-s-2
Conv. Layers 3x3x1024 3x3x1024
Conn. Layer
Conn. Layer

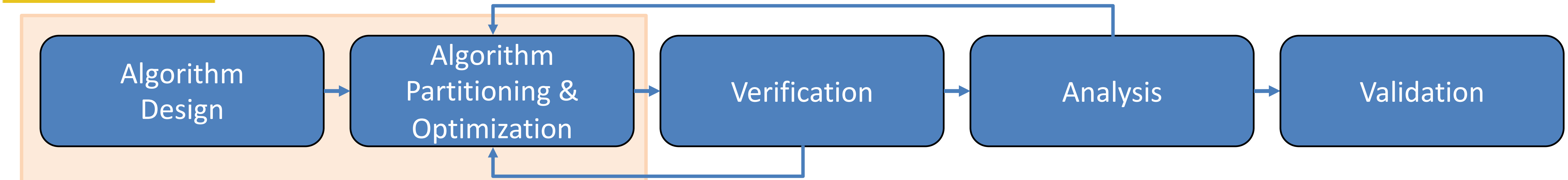
* See: https://pjreddie.com/darknet/yolov2/)

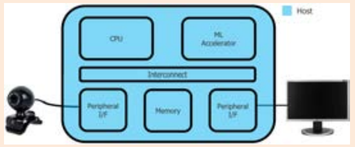
Our Embedded SoC

- Our over simplified SoC embeds the Tiny YOLO V2 algorithm, already trained, a CPU, memory, interconnect and two peripherals
- The SoC receives a feed from a video camera and outputs bounding boxes and labels of objects classified in the input feed

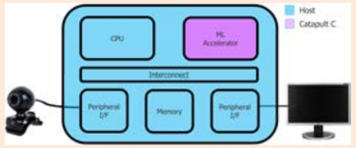


Our Story in Five Steps

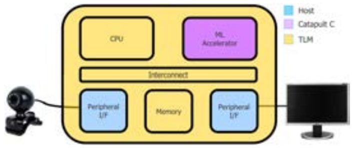




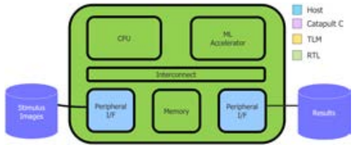
- Tiny YOLO algorithm, written in Python, executed in TensorFlow on a desktop or laptop as stand alone
- It inferences a camera input and it displays processed output on a screen
  - Verify algorithm works properly
- Speed ~ 0.4 sec/inference



- Manual conversion of Tiny YOLO to C for HLS
  - Target wide variety of implementation architectures without re-coding
  - Common testbench for different abstraction levels
  - Automated creation of bus interfaces to surrounding system
- Speed ~ 4 sec/inference



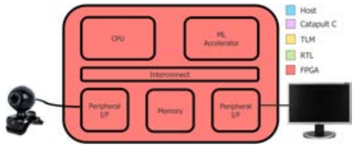
- Block-level verification at C and RT levels with a reusable verification environment
- Exploiting hybrid platform to maximize flexibility in verification
- And, enable earliest SW development and SW-driven verification
- Utilize HW-assisted verification for large dataset tests and full SoC verification



- Early & continuous power, performance analysis from algorithm through full SoC
- Utilize hybrid to focus analysis at block or broader levels
- Execute platform with same software stack from Hybrid platform
  - Realistic Performance
  - Accurate Power
  - Functional Coverage

Speed:

- 21,000 sec/inf RTL SW sim
- 10 sec/inf emulation
- 0.03 sec/inf prototype



- Block-level validation in SoC context with hybrid
- Prototype full SOC
  - Enable complete SW stack & system validation
  - Using real-world stimulus
- Pre-Si Validation
  - Connect to real interfaces, at speed
  - Prepare post-Si validation environment, tests and debug capabilities

Algorithm Partitioning

- A quick profile of Tiny Yolo shows 5.2 billion floating point operations are needed for an inference
- To produce multiple inferences per second requires greater throughput than software can deliver

Algorithm Partitioning

- Some aspects of the algorithm need to remain in software
- Some are appropriately targeted to hardware
- Hardware to be created by High-Level Synthesis (HLS) can be defined in C and linked into the larger algorithm
- Post HLS code (RTL) can be linked into the same algorithm for verification purposes

Yolo Tiny Example

- Image from camera is preprocessed to scale the pixel values and resize the image to meet the requirements of the algorithm
- Object recognition algorithm processes the image
  - Produces a table of results

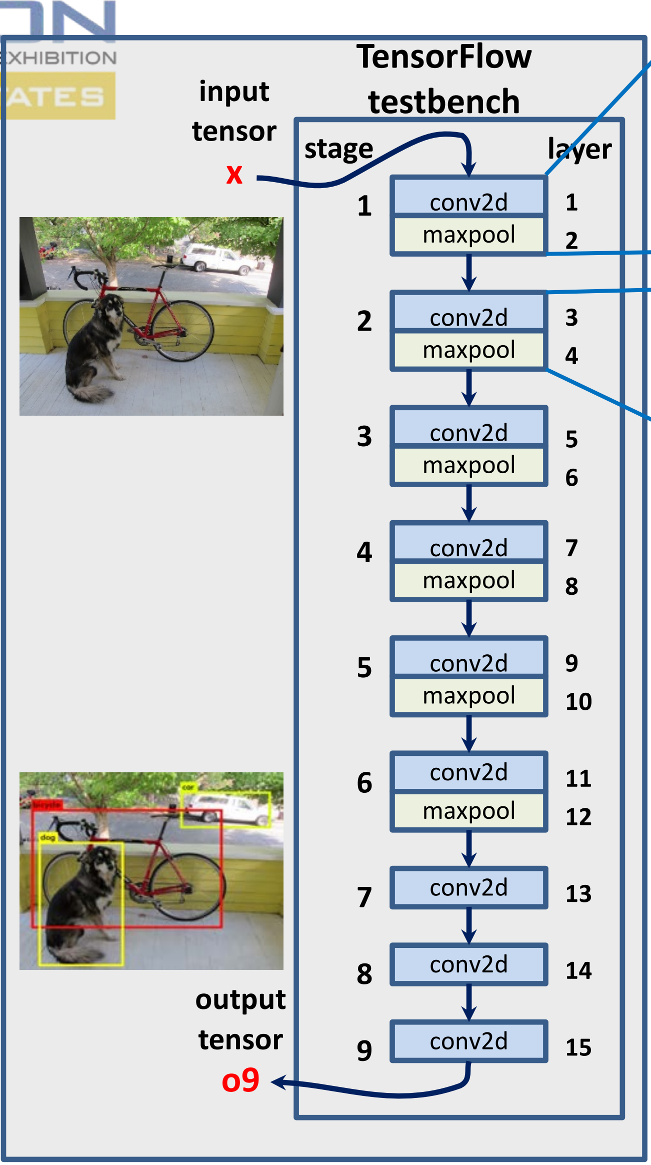
```
class : car, [x,y,w,h]=[571,133,231,142], Confidence = 0.92238536775112152
class : dog, [x,y,w,h]=[266,362,261,299], Confidence = 0.86217708349227905
```

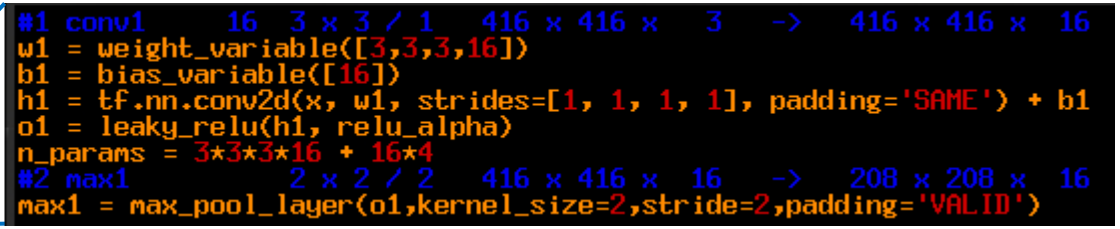
- High confidence recognitions are annotated on the image and the image is displayed

Yolo Tiny Python Implementation

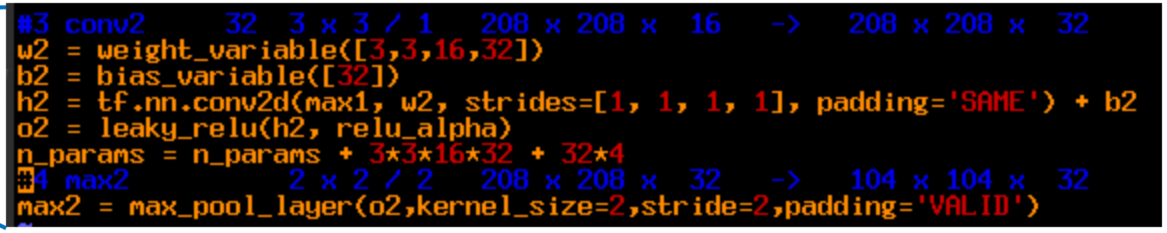
```
def detect_from_cvmat(self,img):
        s = time.time()
        self.h_img,self.w_img,_ = img.shape
        img_resized = cv2.resize(img, (416, 416))
        img_RGB = cv2.cvtColor(img_resized,cv2.COLOR_BGR2RGB)
        img_resized_np = np.asarray( img_RGB )
        inputs = np.zeros((1,416,416,3),dtype='float32')
        inputs[0] = (img_resized_np/255.0)*2.0-1.0
        in_dict = {self.x: inputs}
        net_output = self.sess.run(self.fc_19,feed_dict=in_dict)
        self.result = self.interpret_output(net_output[0])
        self.show_results(img,self.result)
        strtime = str(time.time()-s)
        if self.disp_console : print('Elapsed time : ' + strtime + ' secs' + '\n')
```

Verify at a Higher Level with Reusable Environment





```
#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')
```



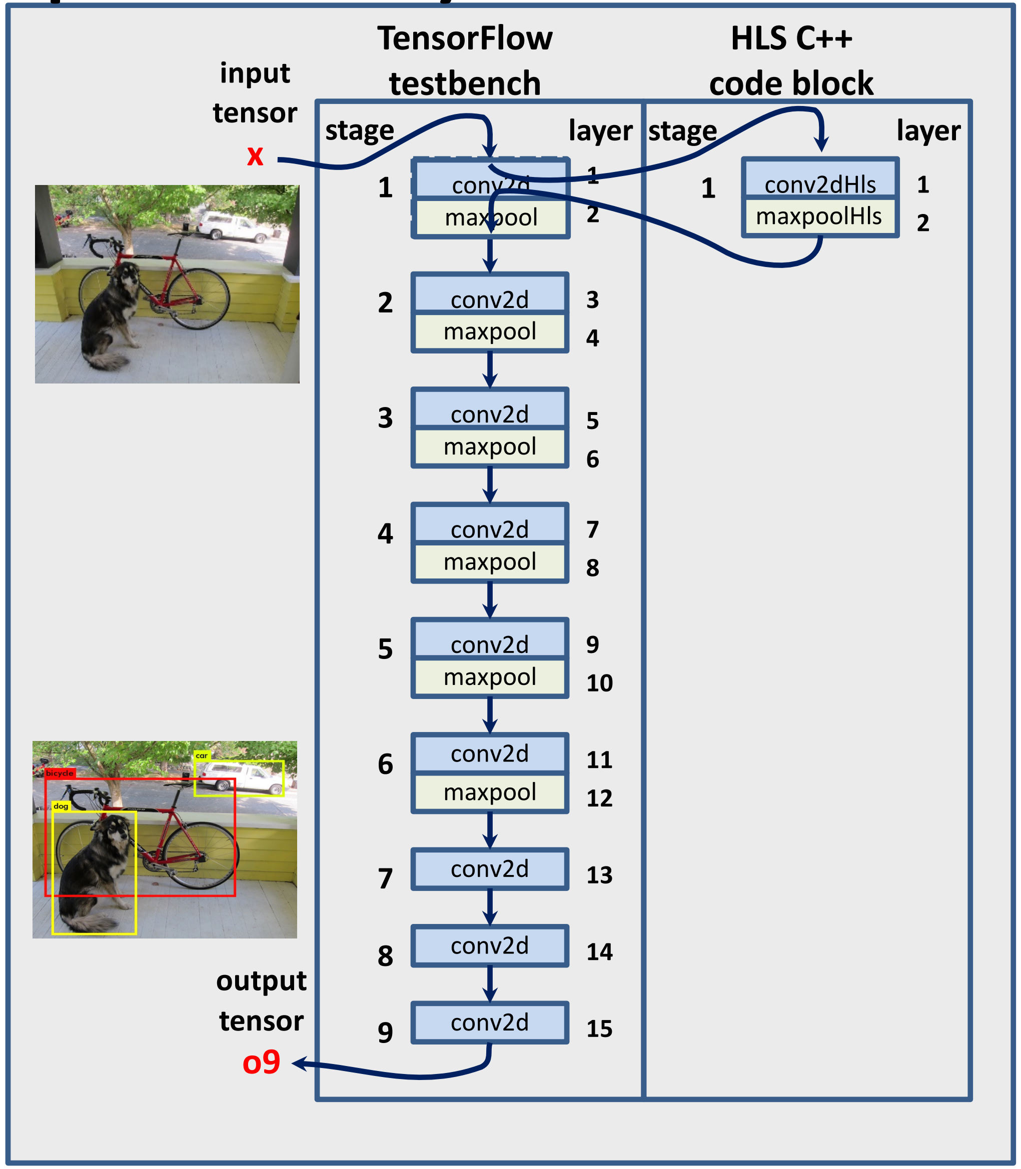
```
#3 conv2       32  3 x 3 / 1   208 x 208 x  16   ->   208 x 208 x  32
w2 = weight_variable([3,3,16,32])
b2 = bias_variable([32])
h2 = tf.nn.conv2d(max1, w2, strides=[1, 1, 1, 1], padding='SAME') + b2
o2 = leaky_relu(h2, relu_alpha)
n_params = n_params + 3*3*16*32 + 32*4
#4 max2           2 x 2 /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

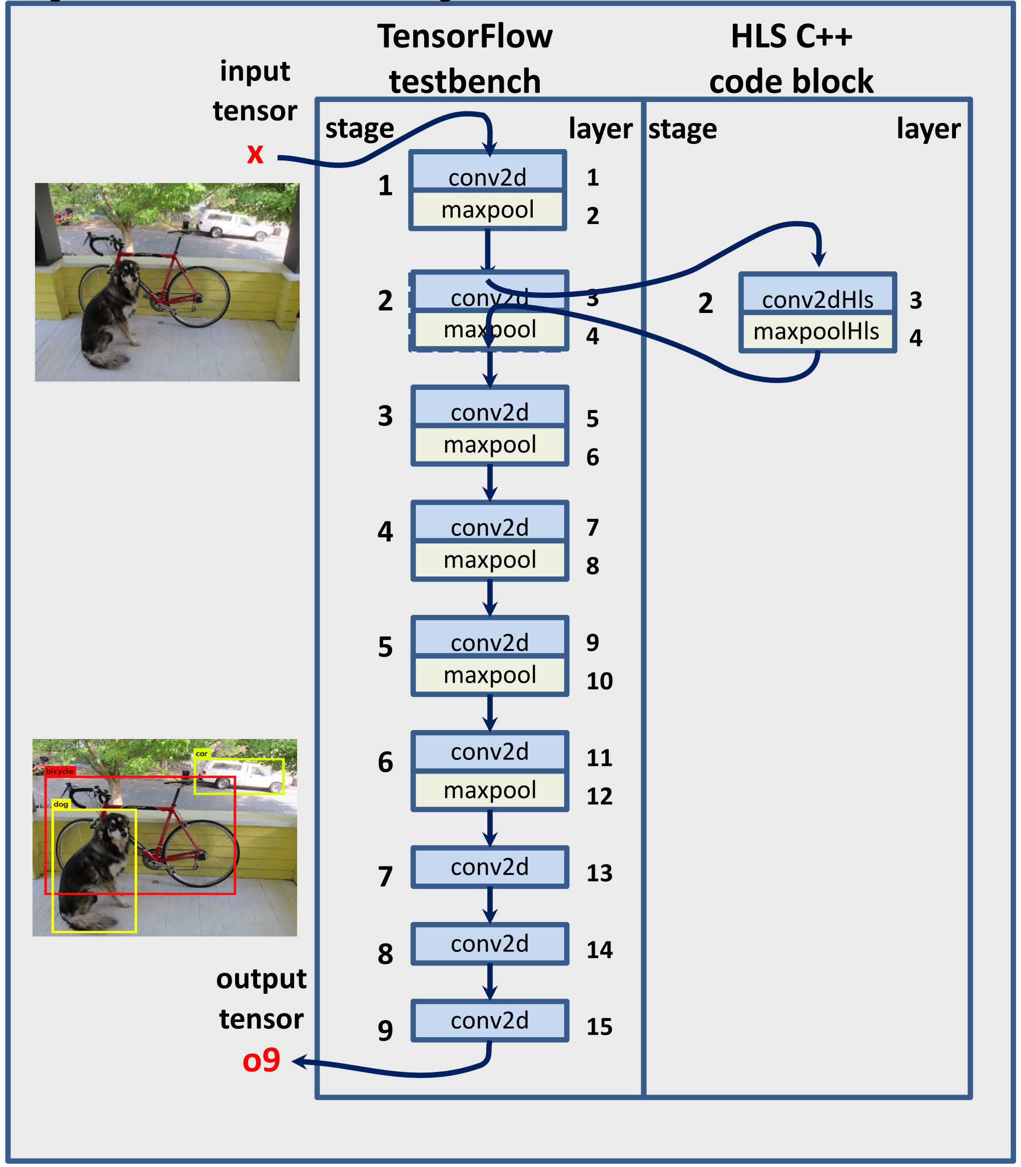
Yolo Tiny implementation with HLS inference

```
def detect_from_cvmat(self,img):
        s = time.time()
        self.h_img,self.w_img,_ = img.shape
        img_resized = cv2.resize(img, (416, 416))
        img_RGB = cv2.cvtColor(img_resized,cv2.COLOR_BGR2RGB)
        img_resized_np = np.asarray( img_RGB )
        inputs = np.zeros((1,416,416,3),dtype='float32')
        inputs[0] = (img_resized_np/255.0)*2.0-1.0
        in_dict = {self.x: inputs}

        net_output = self.sess.run(self.fc_19,feed_dict=in_dict)
        catapult_net_output = C_library.catapult_yolo_tiny(inputs)
        self.diff(net_output, catapult_net_output)

        self.result = self.interpret_output(catapult_net_output[0])
        self.show_results(img,self.result)
        strtime = str(time.time()-s)
        if self.disp_console : print('Elapsed time : ' + strtime + ' secs' + '\n')
```

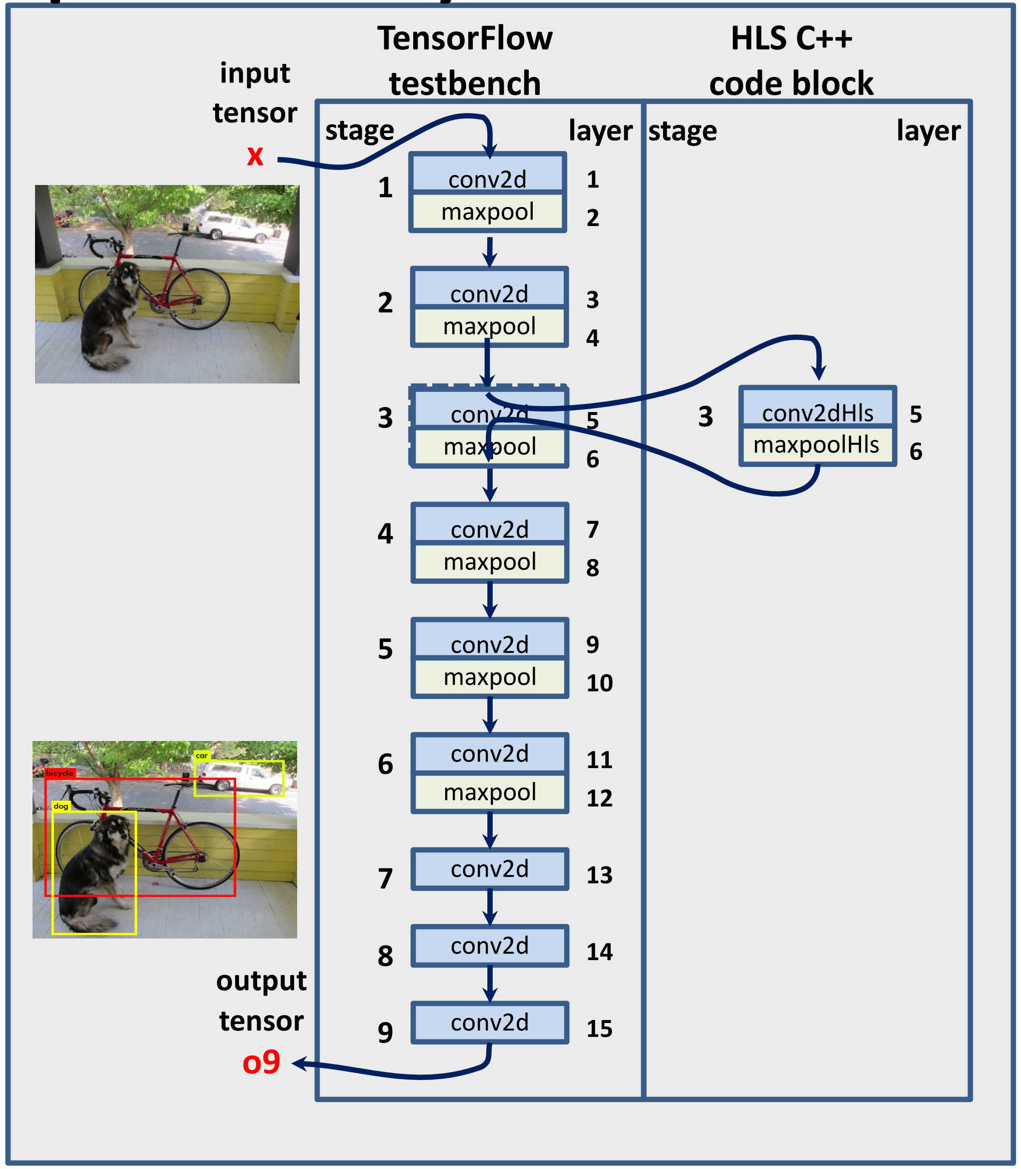
Replace Layers one at a Time



Replace Layers one at a Time



Replace Layers one at a Time



TensorFlow testbench

HLS C++ code block

input tensor

x

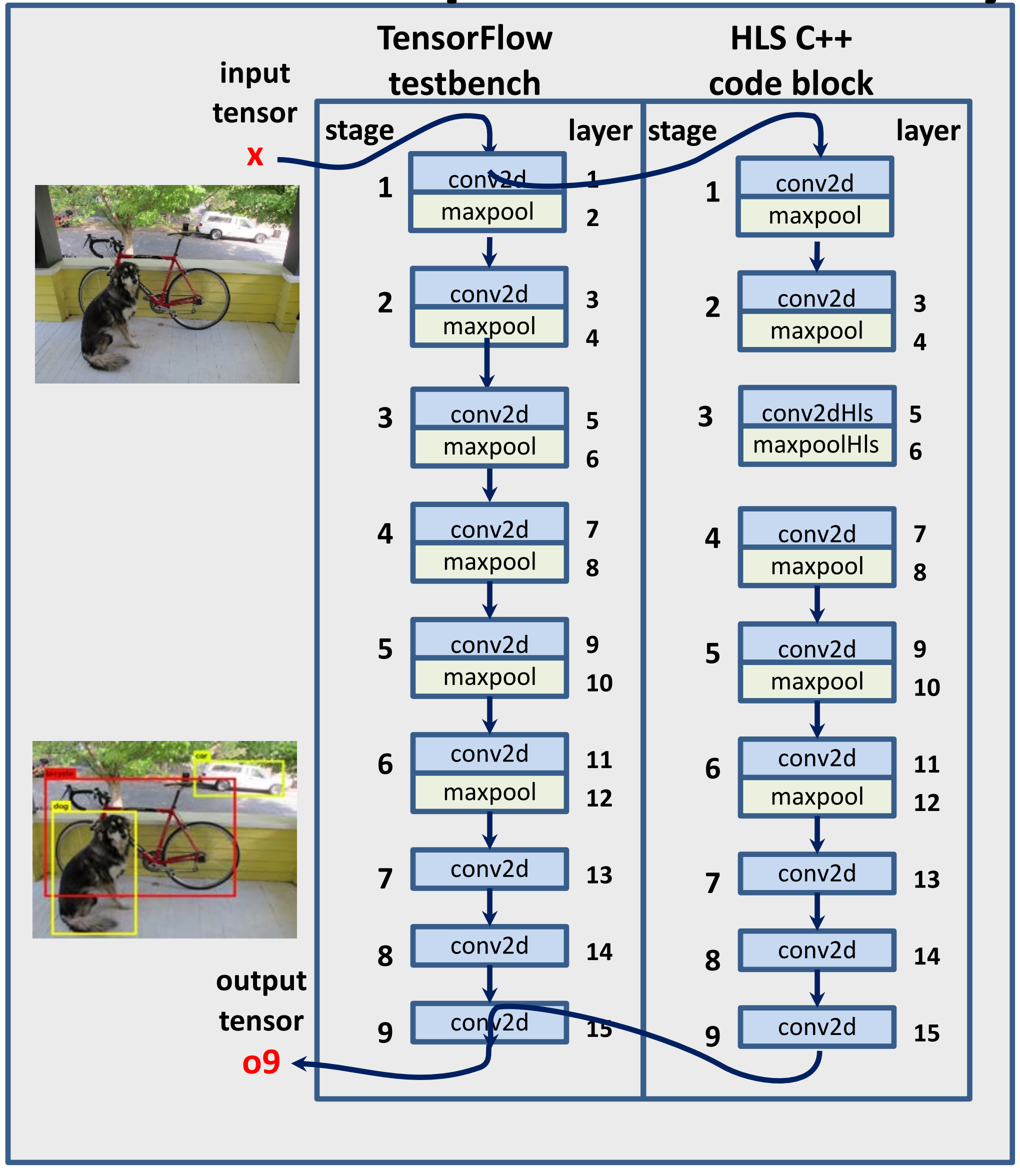
stage	TensorFlow testbench	layer
1	conv2d	1
	maxpool	2
2	conv2d	3
	maxpool	4
3	conv2d	5
	maxpool	6
4	conv2d	7
	maxpool	8
5	conv2d	9
	maxpool	10
6	conv2d	11
	maxpool	12
7	conv2d	13
8	conv2d	14
9	conv2d	15

stage	HLS C++ code block	layer
3	conv2dHls	5
	maxpoolHls	6

output tensor

o9

Then Replace All Layers



Path to Implementation

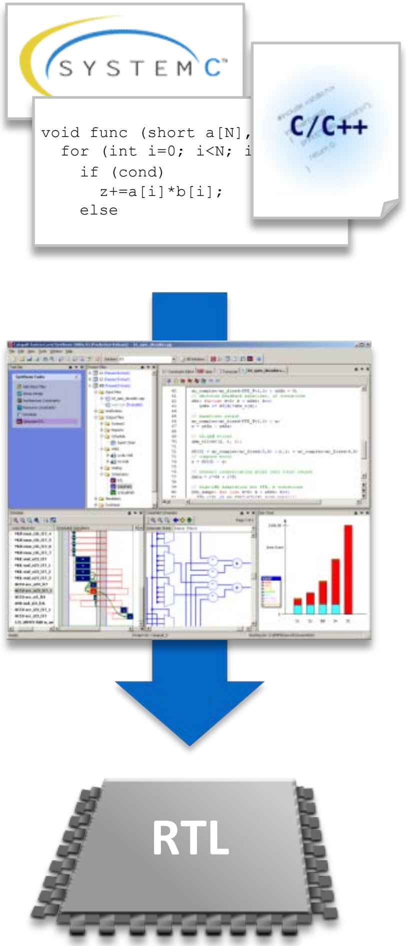
- Once the algorithmic C is shown to match the original python, then it can be used as a starting point for RTL development

Russell Klein

HIGH-LEVEL SYNTHESIS

What is High-Level Synthesis

- Transformation of algorithm to synthesizable RTL
  - Typically C, C++, or SystemC
  - Handles low-level details for designer
- Technology aware
  - Understands target silicon technology or FPGA device
  - Generates RTL based on technology library and target frequency



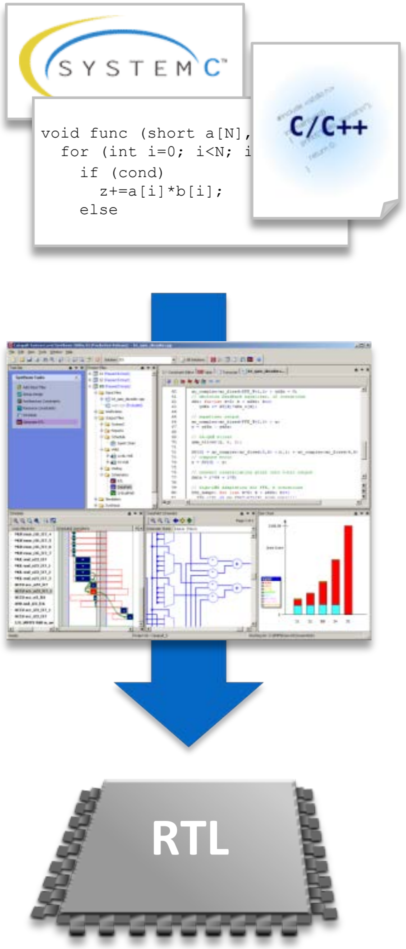
```
void func (short a[N],
  for (int i=0; i<N; i
    if (cond)
      z+=a[i]*b[i];
    else
```

Benefits of HLS

- Improved developer productivity
  - Design at a higher level of abstraction
  - Automate away a lot of the detailed work in creating RTL
- Reduced verification effort
  - Verifying an abstract algorithm is much faster and easier than verifying RTL
  - Prove that the resulting RTL is equivalent to the original algorithm
    - HLS tools enable this with dynamic simulation and formal proofs
- Exploration of design alternatives
  - Implementing different architectures in RTL is prohibitively expensive
  - At the algorithmic level it fast and easy

Design at a Higher Level

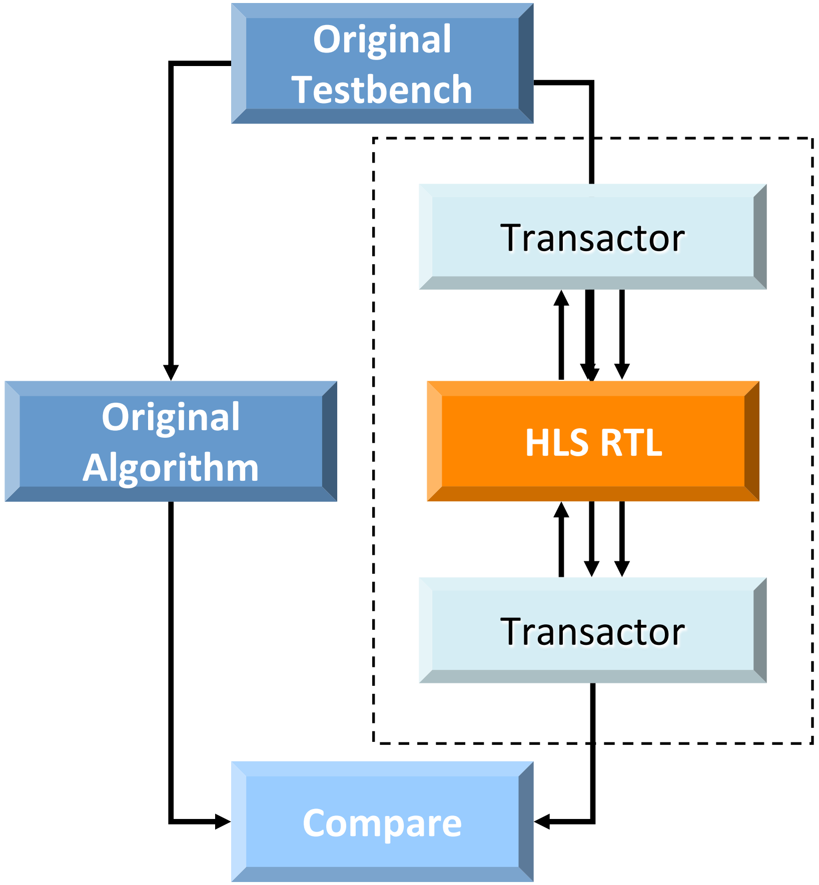
- Generate high quality RTL from higher level descriptions
  - Manual RTL coding errors and ECO’s are avoided
  - Designs are correct-by-construction
  - Time-consuming RTL design iterations are eliminated
  - Estimate and optimize power and performance before RTL synthesis
- Key applications designed with HLS
  - Video Compression/Decompression (H.265/HEVC, VP9)
  - Image processing (Mobile/4K/Ultra HD/3D)
  - Wireless/Wireline (Bluetooth, 5G, 802.11 Gb optical, DOCSIS)



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

Verification of RTL

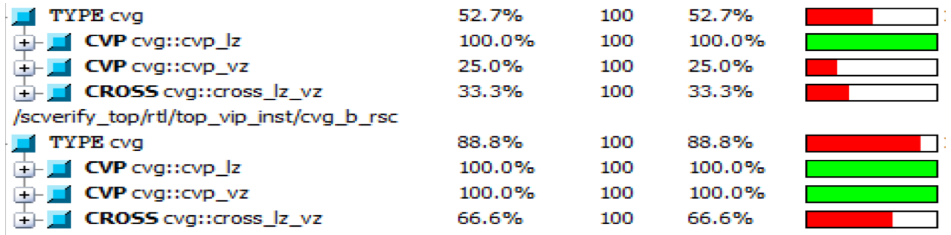
- Dynamic verification
  - Common input to algorithm and RTL
  - Compares output from RLT with output from original algorithm
  - Covered later
- Formal verification
  - Precise semantics and machine readable format for algorithm and RTL
  - Supports formal equivalency proof



Coverage and Assertions

- Assertions and Cover Points can be put in source C++ & SystemC
- Assertions and cover points propagate from source to RTL
- Enables verification at higher level

```
int alu(int a, int b, uint2 opcode) {
  cover(opcode==ADD);
  cover(opcode==SUB);
  cover(opcode==MUL);
  cover(opcode==DIV);

  short r;

  switch(opcode) {
    case ADD: r = a+b;
              break;
    case SUB: r = a-b;
              break;
    case MUL: r = a*b;
              break;
    case DIV: assert(b!=0);
              r = a/b;
              break;
  }
  return r;
}
```

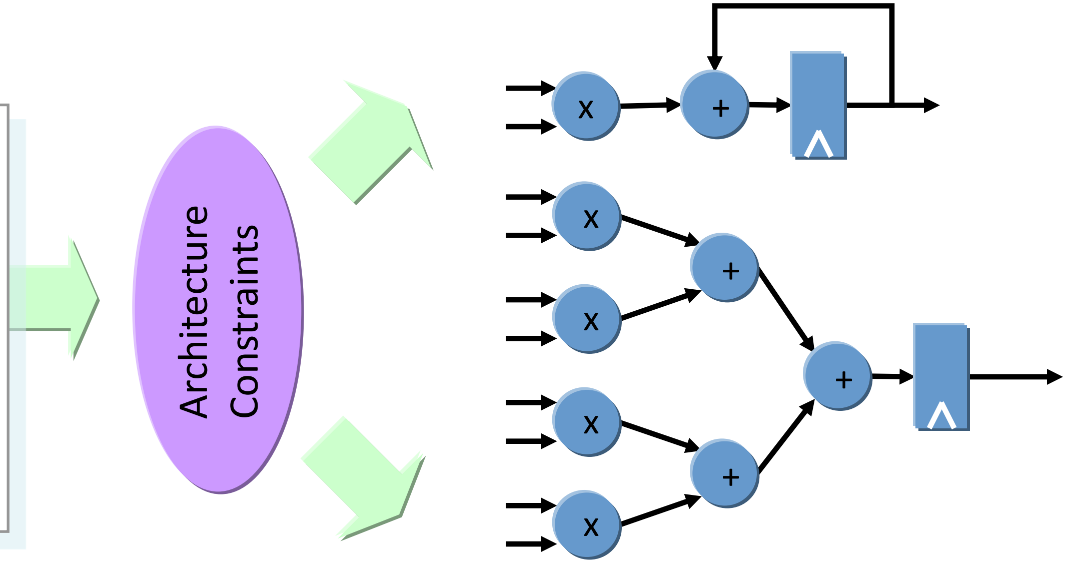


```
TYPE cvg                              52.7%    100   52.7%
  CVP cvg::cvp_lz                     100.0%   100   100.0%
  CVP cvg::cvp_vz                     25.0%    100   25.0%
  CROSS cvg::cross_lz_vz              33.3%    100   33.3%
/scverify_top/rtl/top_vip_inst/cvg_b_rsc
TYPE cvg                              88.8%    100   88.8%
  CVP cvg::cvp_lz                     100.0%   100   100.0%
  CVP cvg::cvp_vz                     100.0%   100   100.0%
  CROSS cvg::cross_lz_vz              66.6%    100   66.6%
```

Design Alternatives

- User control over the micro-architecture implementation
  - Parallelism, Throughput, Area, Latency (loop unrolling & pipelining)
  - Memories (DPRAM/SPRAM/split/bank) vs Registers (Resource allocation)
- Exploration is accomplished by applying constraints
  - Not by changing the source code

```
int mac(
  char data[N],
  char coef[N]
) {
  int accum=0;
  for (int i=0; i<N; i++)
    accum += data[i] * coef[i];
  return accum;
}
```

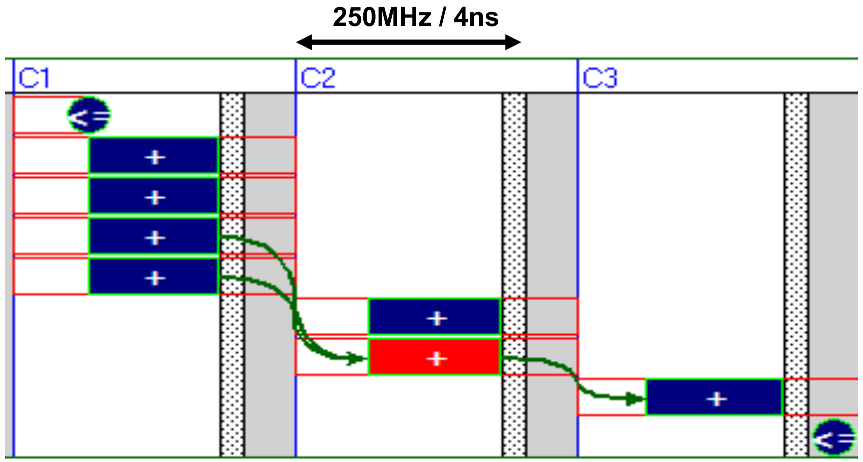


HLS Optimizations

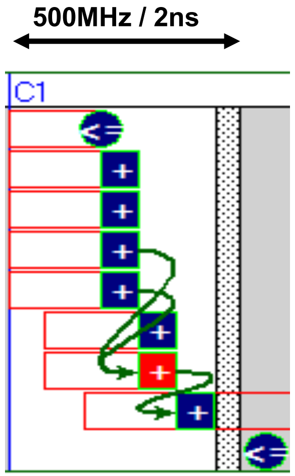
- Automatic Arithmetic optimizations and bit-width trimming
- Multi-objective scheduling
  - Area/Latency driven datapath scheduling
- Eliminates RTL technology penalty of I.P. reuse

```
for (int i=0; i<8; i++){
  tmp+=a[i];
}
```

Technology Neutral Description



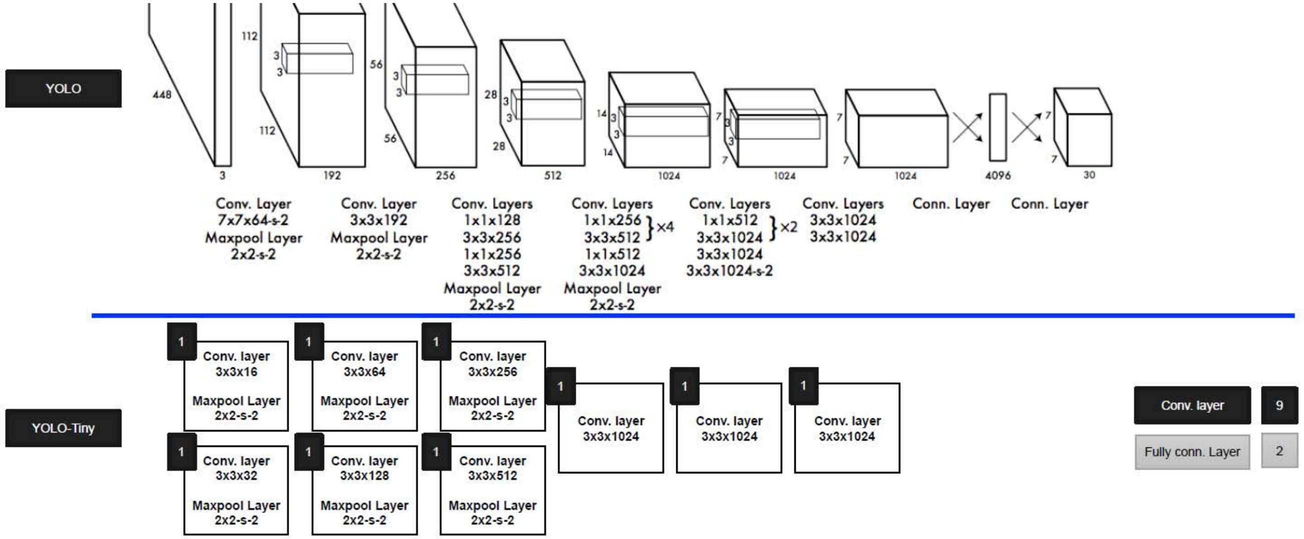
FPGA or SLOW ASIC
Delay of a 16bit add: 2.1 ns
Latency: 3 cycles



Faster Process
Delay of a 16bit add: 0.3 ns
Latency: 1 cycles

Yolo Tiny (v2)

- Algorithm for detecting and classifying objects in pictures
  - Used on cell phones and computationally limited systems
  - Over 5.2 billion floating point operations per inference
  - Over 25 million weight values
  - Neural network has 24 layers (full Yolo has 106)



YOLO

Conv. Layer 7x7x64-s-2 Maxpool Layer 2x2-s-2

Conv. Layer 3x3x192 Maxpool Layer 2x2-s-2

Conv. Layers 1x1x128 3x3x256 1x1x256 3x3x512 Maxpool Layer 2x2-s-2

Conv. Layers 1x1x256 3x3x512 }×4 1x1x512 3x3x1024 Maxpool Layer 2x2-s-2

Conv. Layers 1x1x512 3x3x1024 }×2 3x3x1024 3x3x1024-s-2

Conv. Layers 3x3x1024 3x3x1024

Conn. Layer

Conn. Layer

YOLO-Tiny

1 Conv. layer 3x3x16 Maxpool Layer 2x2-s-2

1 Conv. layer 3x3x32 Maxpool Layer 2x2-s-2

1 Conv. layer 3x3x64 Maxpool Layer 2x2-s-2

1 Conv. layer 3x3x128 Maxpool Layer 2x2-s-2

1 Conv. layer 3x3x256 Maxpool Layer 2x2-s-2

1 Conv. layer 3x3x512 Maxpool Layer 2x2-s-2

1 Conv. layer 3x3x1024

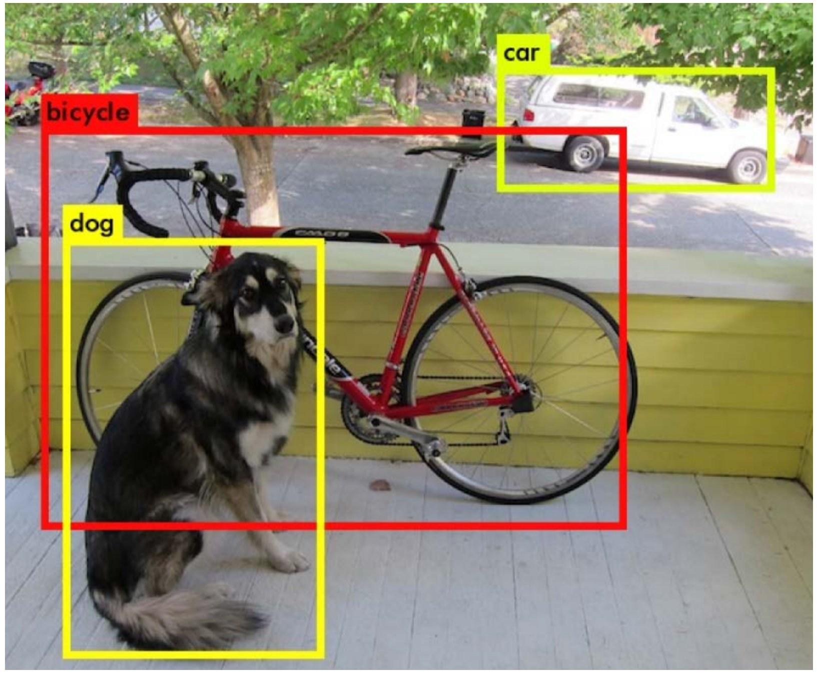
1 Conv. layer 3x3x1024

1 Conv. layer 3x3x1024

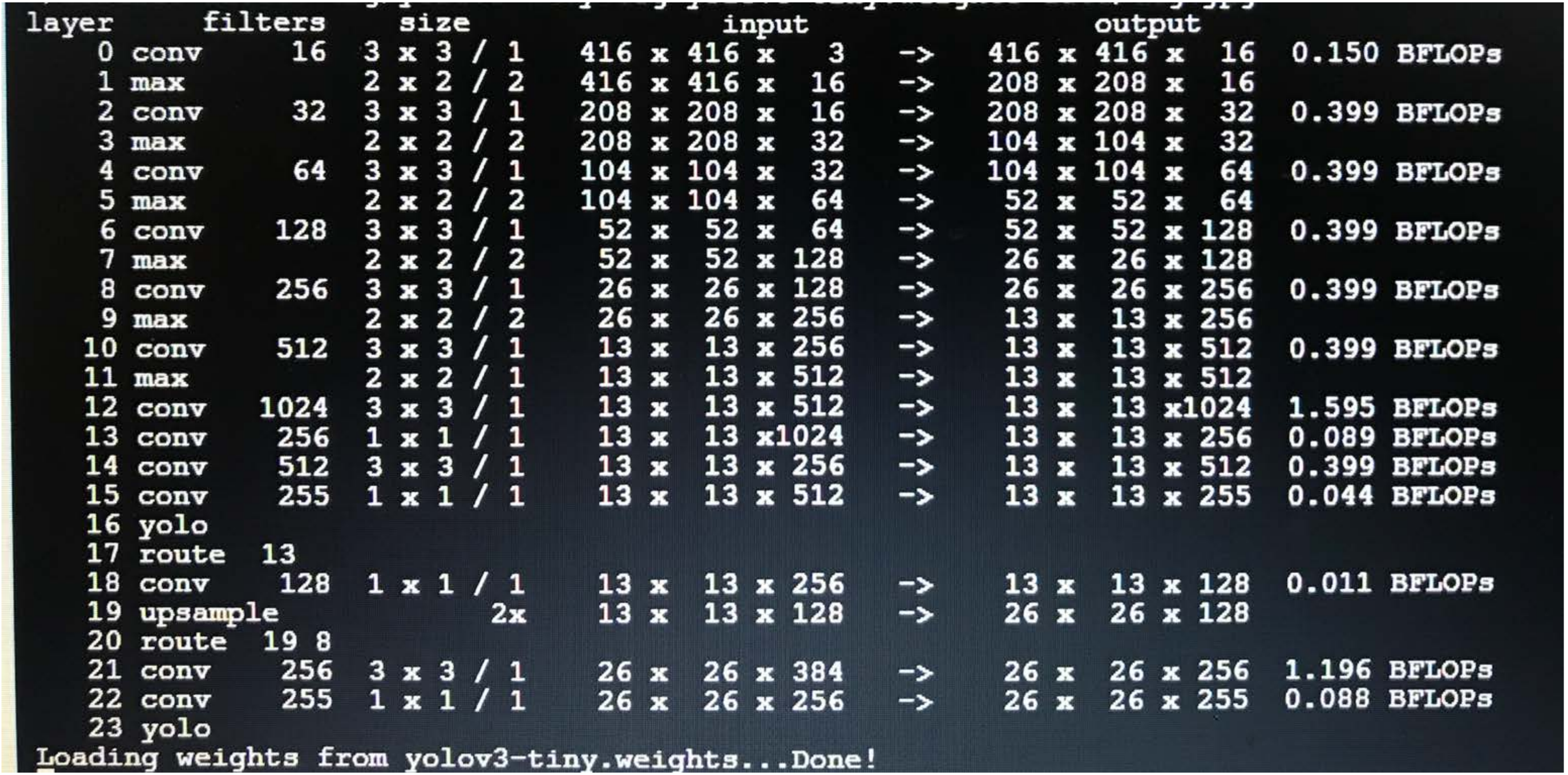
Conv. layer 9

Fully conn. Layer 2

*https://pjreddie.com/darknet/**yolo***

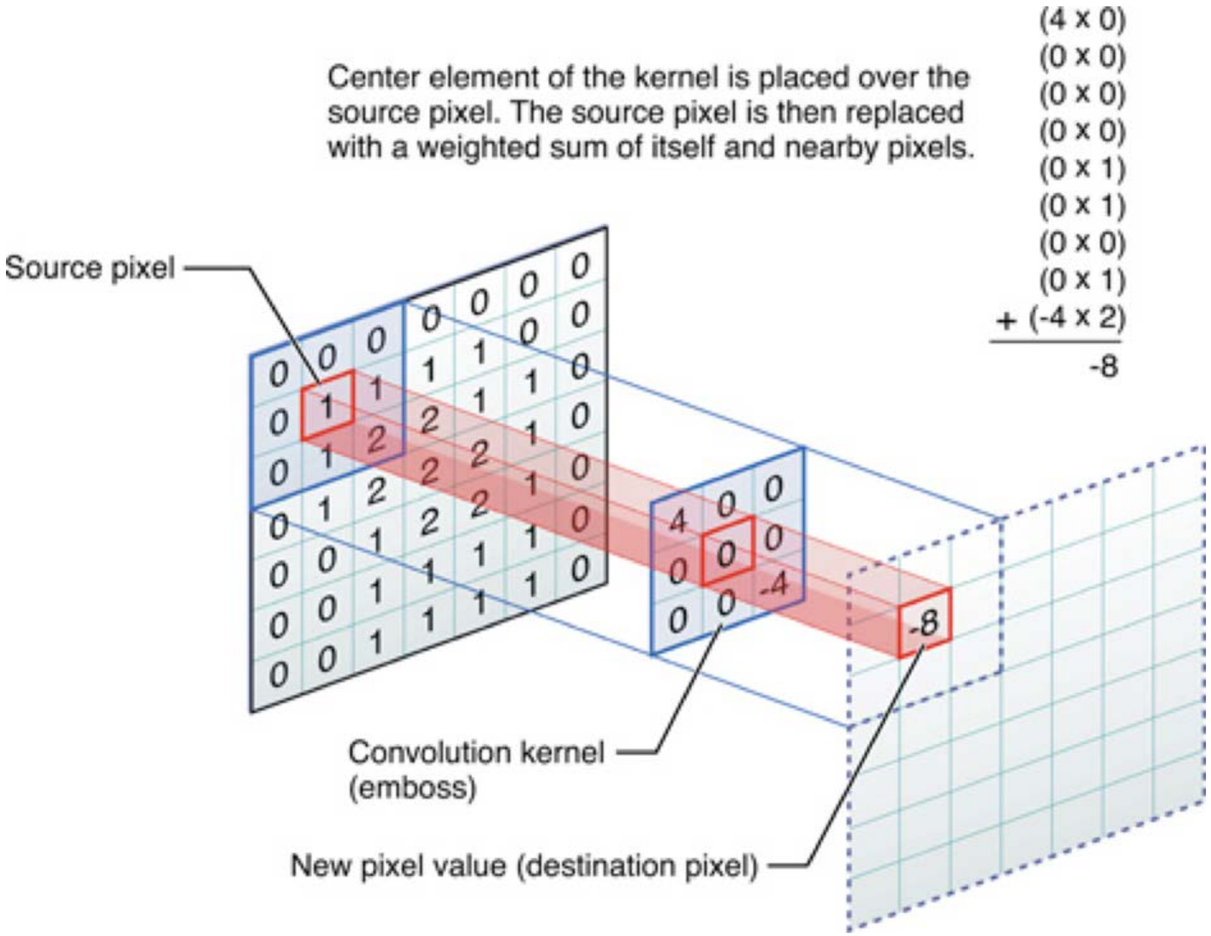


Yolo-Tiny Profile



```
layer     filters    size              input                output
    0 conv     16  3 x 3 / 1   416 x 416 x   3   ->   416 x 416 x  16  0.150 BFLOPs
    1 max          2 x 2 / 2   416 x 416 x  16   ->   208 x 208 x  16
    2 conv     32  3 x 3 / 1   208 x 208 x  16   ->   208 x 208 x  32  0.399 BFLOPs
    3 max          2 x 2 / 2   208 x 208 x  32   ->   104 x 104 x  32
    4 conv     64  3 x 3 / 1   104 x 104 x  32   ->   104 x 104 x  64  0.399 BFLOPs
    5 max          2 x 2 / 2   104 x 104 x  64   ->    52 x  52 x  64
    6 conv    128  3 x 3 / 1    52 x  52 x  64   ->    52 x  52 x 128  0.399 BFLOPs
    7 max          2 x 2 / 2    52 x  52 x 128   ->    26 x  26 x 128
    8 conv    256  3 x 3 / 1    26 x  26 x 128   ->    26 x  26 x 256  0.399 BFLOPs
    9 max          2 x 2 / 2    26 x  26 x 256   ->    13 x  13 x 256
   10 conv    512  3 x 3 / 1    13 x  13 x 256   ->    13 x  13 x 512  0.399 BFLOPs
   11 max          2 x 2 / 1    13 x  13 x 512   ->    13 x  13 x 512
   12 conv   1024  3 x 3 / 1    13 x  13 x 512   ->    13 x  13 x1024  1.595 BFLOPs
   13 conv    256  1 x 1 / 1    13 x  13 x1024   ->    13 x  13 x 256  0.089 BFLOPs
   14 conv    512  3 x 3 / 1    13 x  13 x 256   ->    13 x  13 x 512  0.399 BFLOPs
   15 conv    255  1 x 1 / 1    13 x  13 x 512   ->    13 x  13 x 255  0.044 BFLOPs
   16 yolo
   17 route  13
   18 conv    128  1 x 1 / 1    13 x  13 x 256   ->    13 x  13 x 128  0.011 BFLOPs
   19 upsample            2x    13 x  13 x 128   ->    26 x  26 x 128
   20 route  19 8
   21 conv    256  3 x 3 / 1    26 x  26 x 384   ->    26 x  26 x 256  1.196 BFLOPs
   22 conv    255  1 x 1 / 1    26 x  26 x 256   ->    26 x  26 x 255  0.088 BFLOPs
   23 yolo
Loading weights from yolov3-tiny.weights...Done!
```

What is convolution?

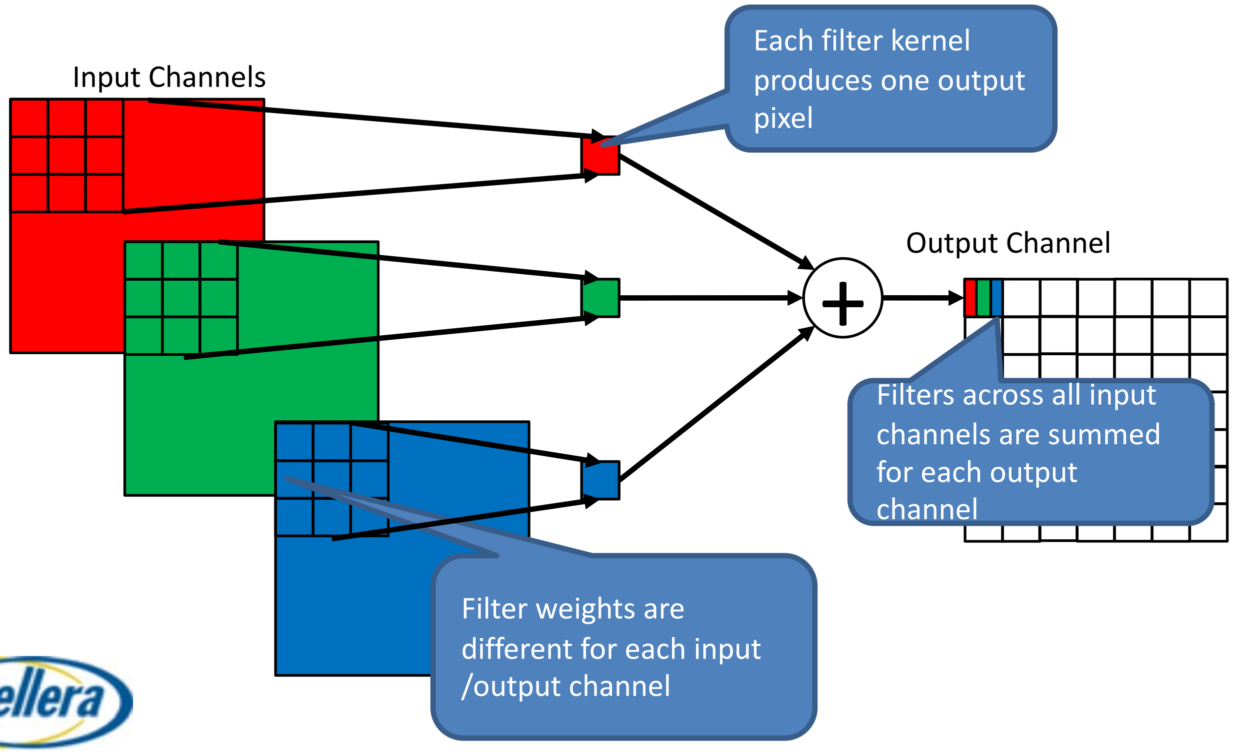

Center element of the kernel is placed over the source pixel. The source pixel is then replaced with a weighted sum of itself and nearby pixels.
(4 x 0)
(0 x 0)
(0 x 0)
(0 x 0)
(0 x 1)
(0 x 1)
(0 x 0)
(0 x 1)
+ (-4 x 2)
-8
Source pixel
Convolution kernel (emboss)
New pixel value (destination pixel)

Multiply one array by another, element by element, and sum the results

Source: Embedded-Vision.com

Convolution used in CNNs

- Each output channel uses 2-d convolutions across all input channels
  - Billions of Multiply/Accumulate operations
- Embarrassingly parallel



Each filter kernel produces one output pixel

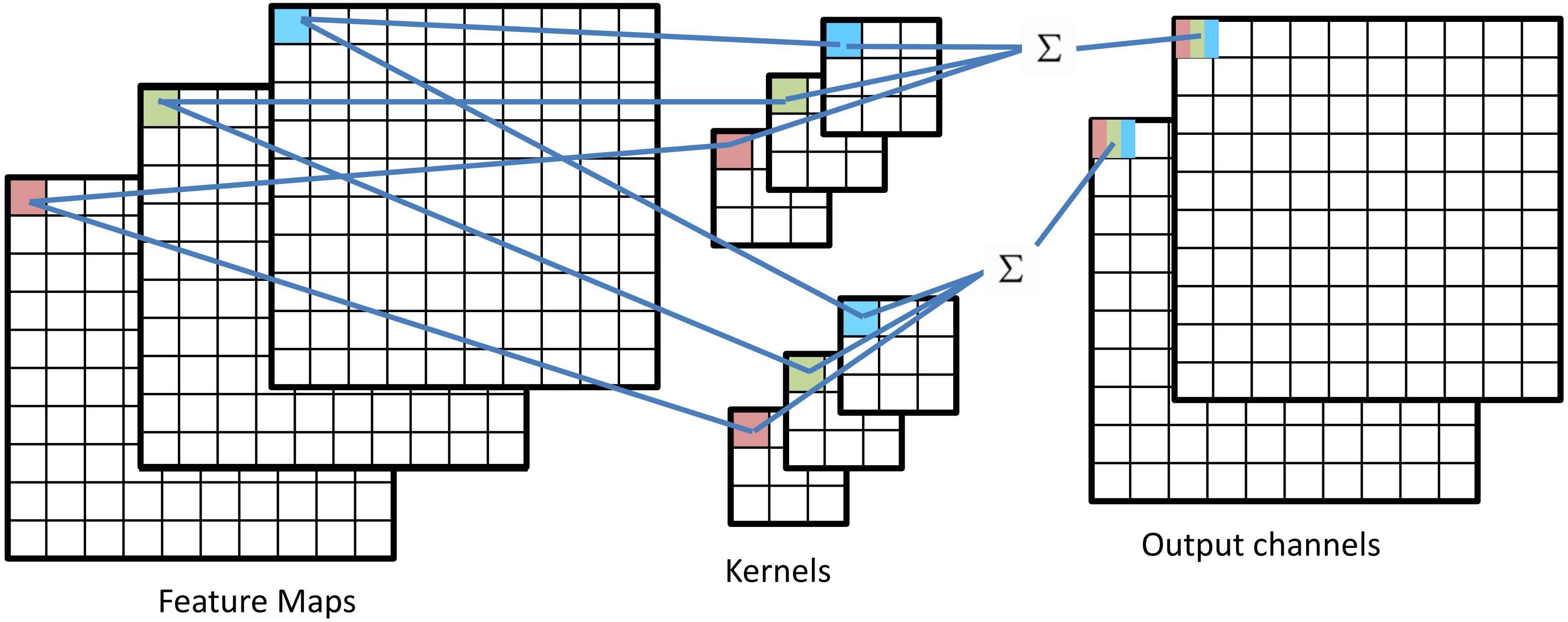
Filters across all input channels are summed for each output channel

Filter weights are different for each input /output channel

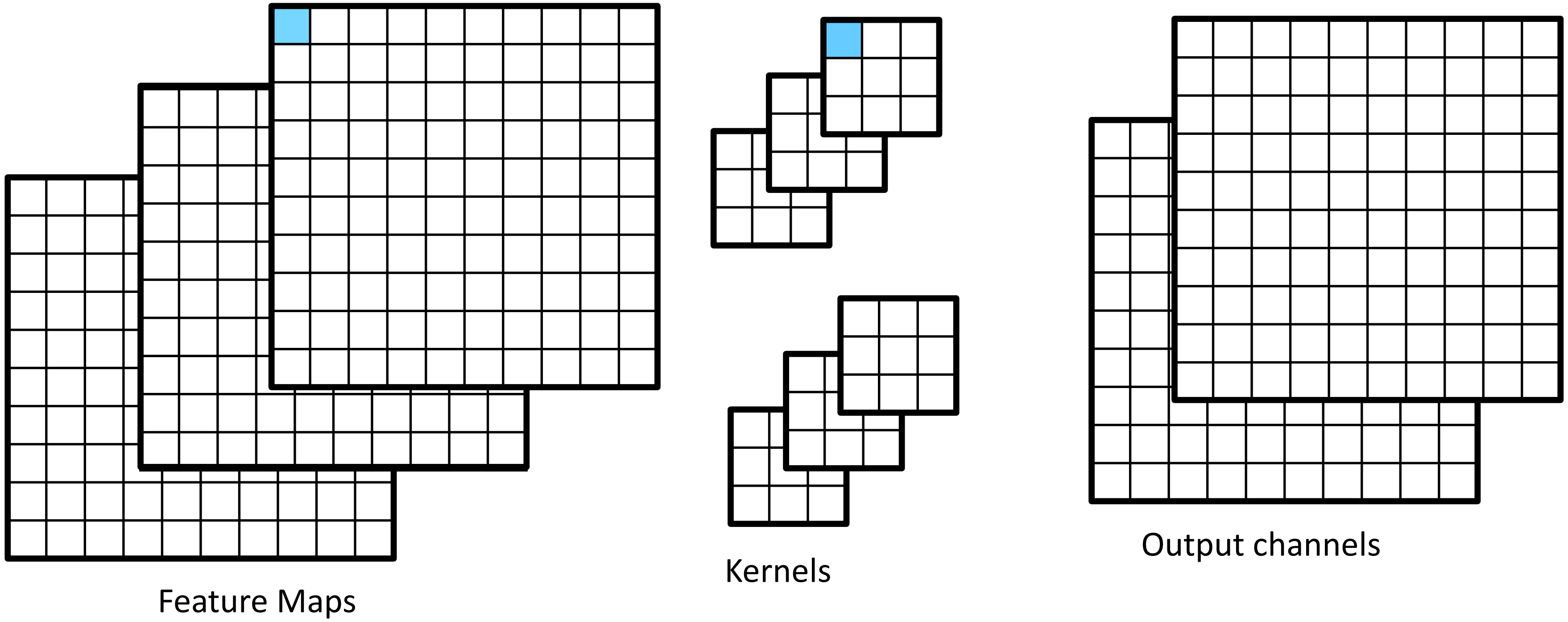
Pure 4-D Convolution Algorithm

```
OUT_CHAN:for(int oc=0;oc<OUT_CHANNELS;oc++){
  IN_CHAN:for(int ic=0;ic<IN_CHANNELS;ic++){
    FMAP_HEIGHT:for(int r=0;r<IN_HEIGHT;r++){
      FMAP_WIDTH:for(int c=0;c<IN_WIDTH;c++){
        KERNEL_Y:for(int i=0;i<3;i++){
          KERNEL_X:for(int j=0;j<3;j++){
            acc[r][c] += fmap[ic][r-i/2][c-j/2]
                      * kernel[ic][oc][i][j];
          }
        }
      }
      fmap_out[oc][r][c] = acc[r][c];
    }
  }
}
```

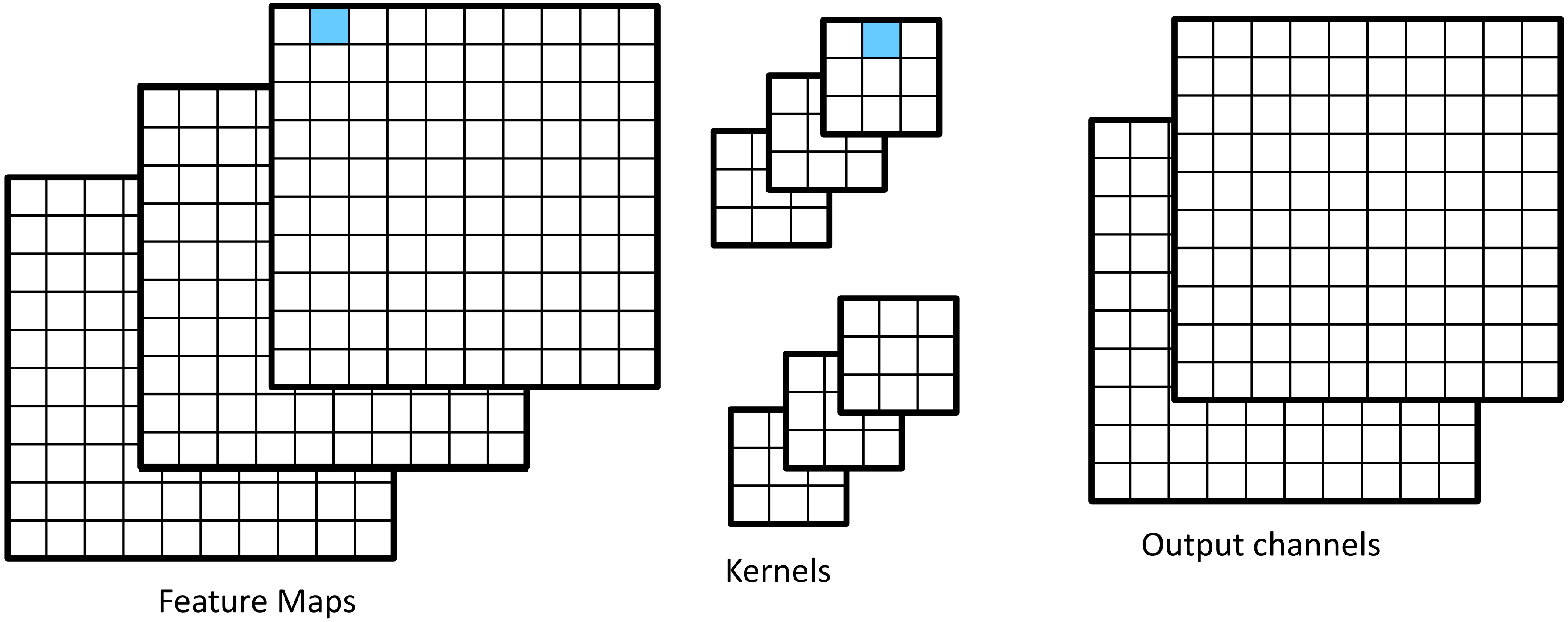
TensorFlow 2d convolution



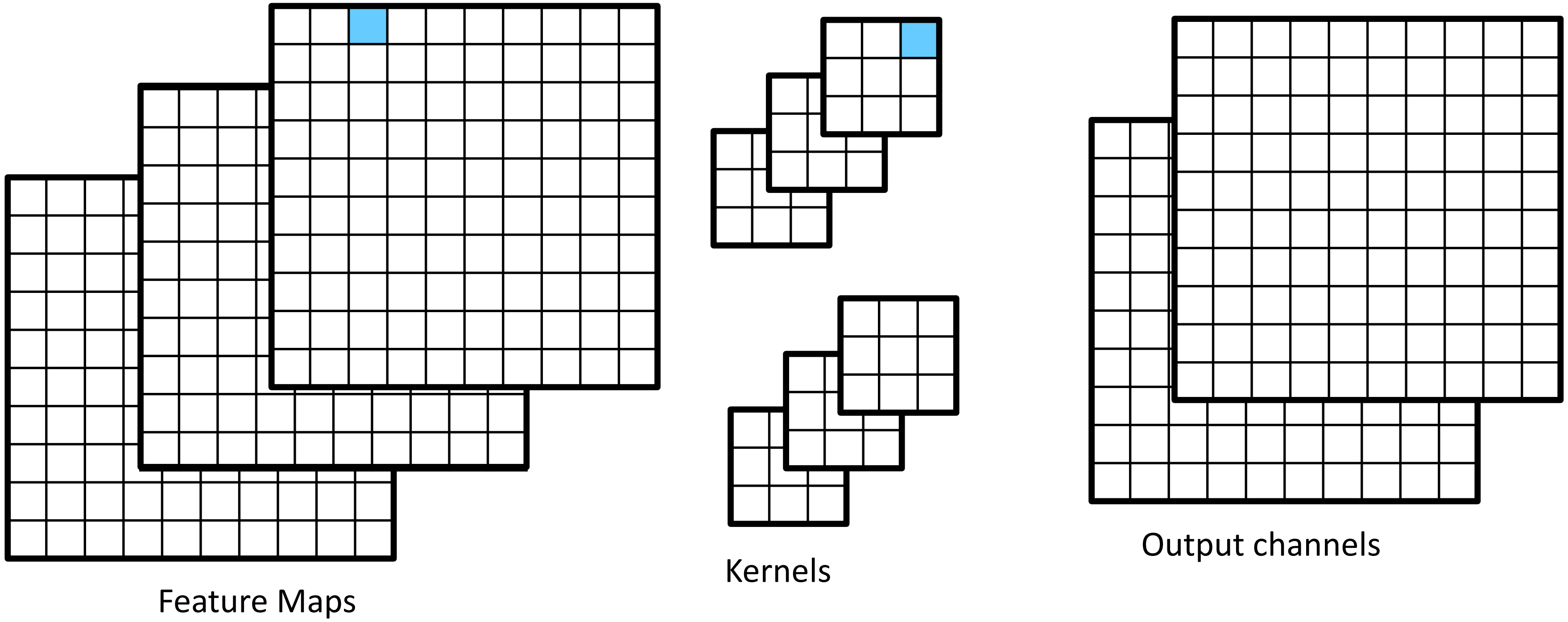
TensorFlow 2d convolution



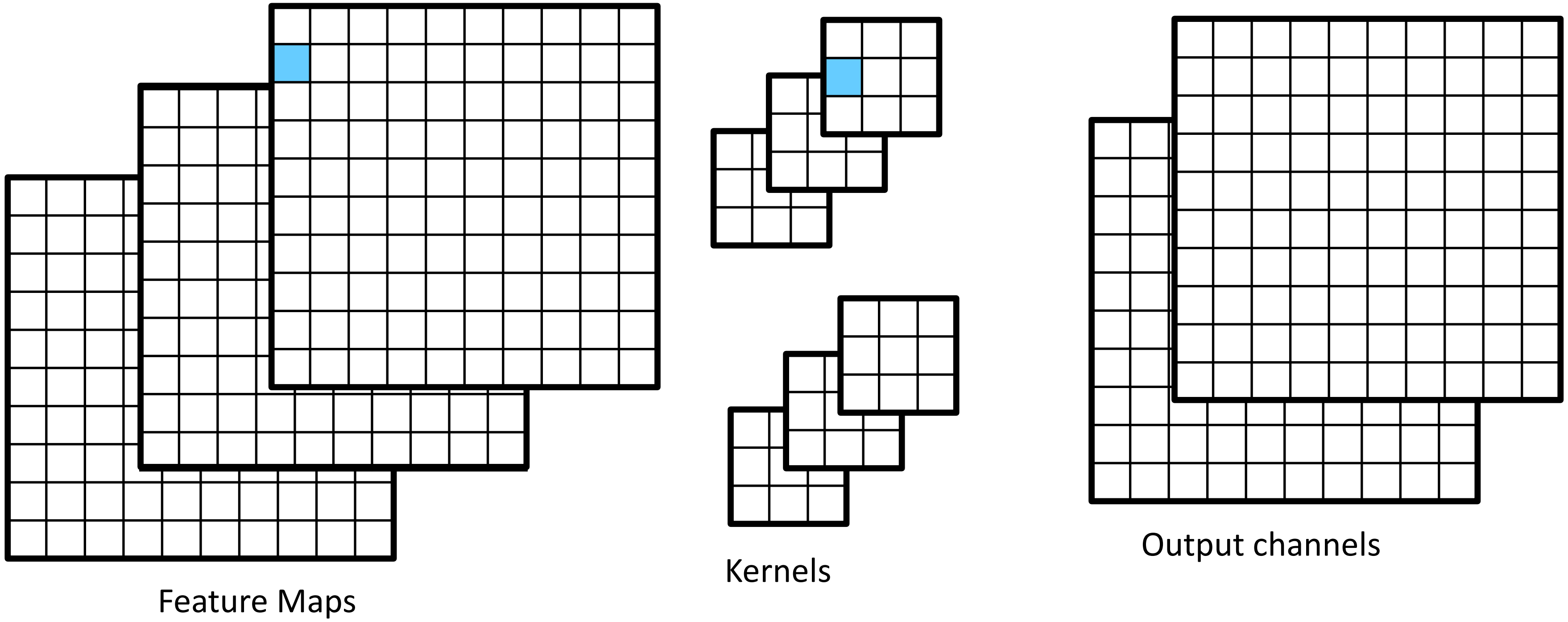
TensorFlow 2d convolution



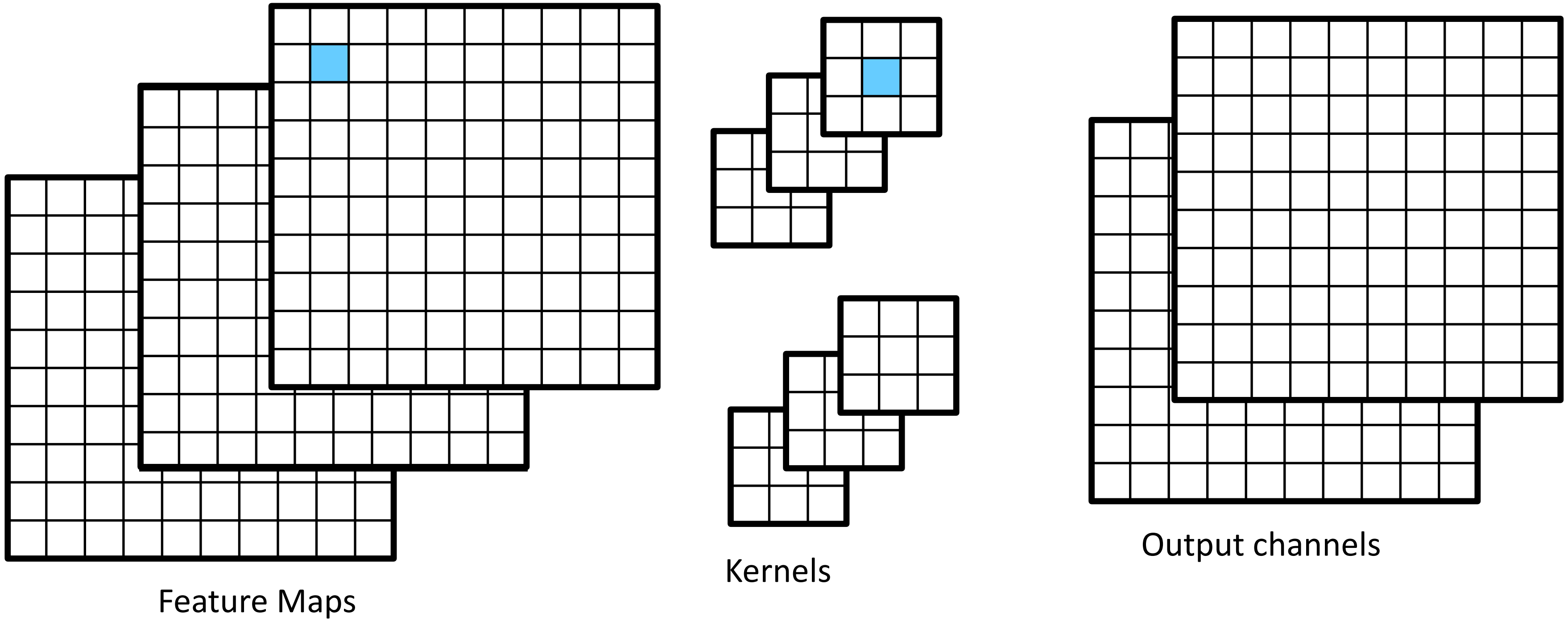
TensorFlow 2d convolution



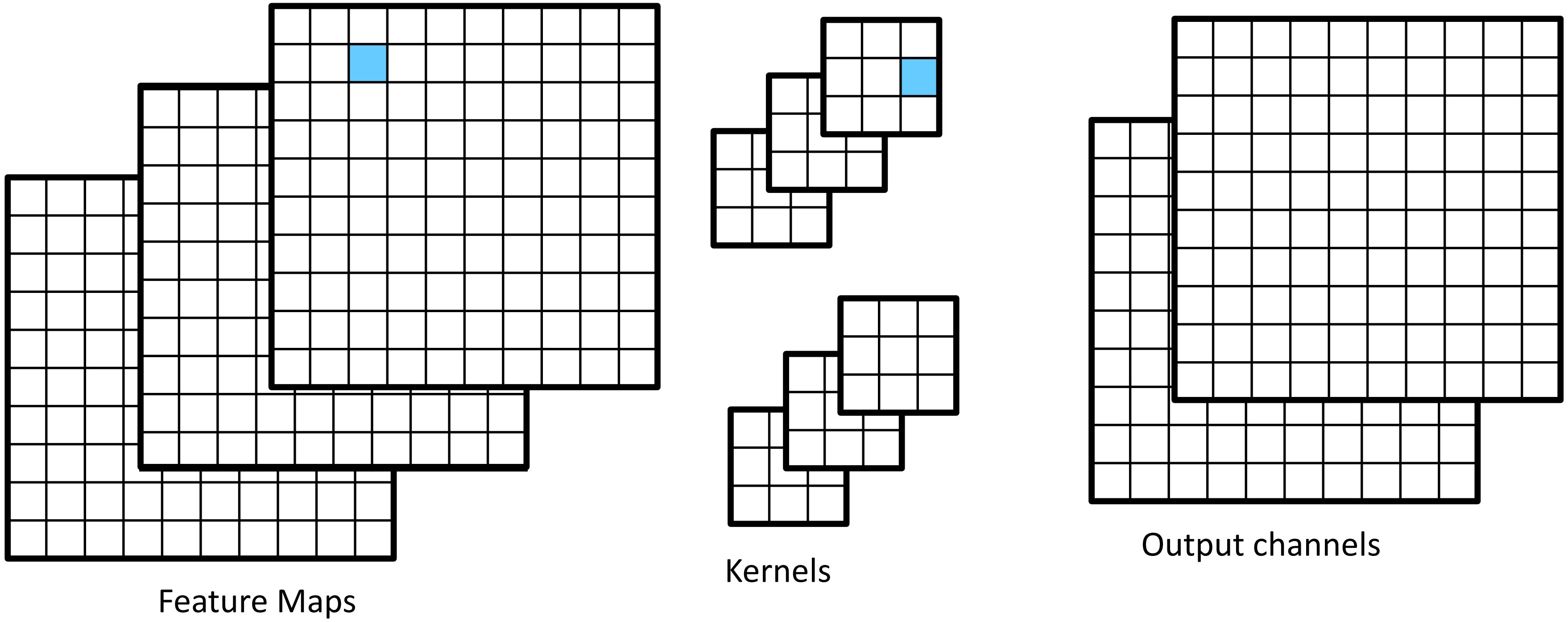
TensorFlow 2d convolution



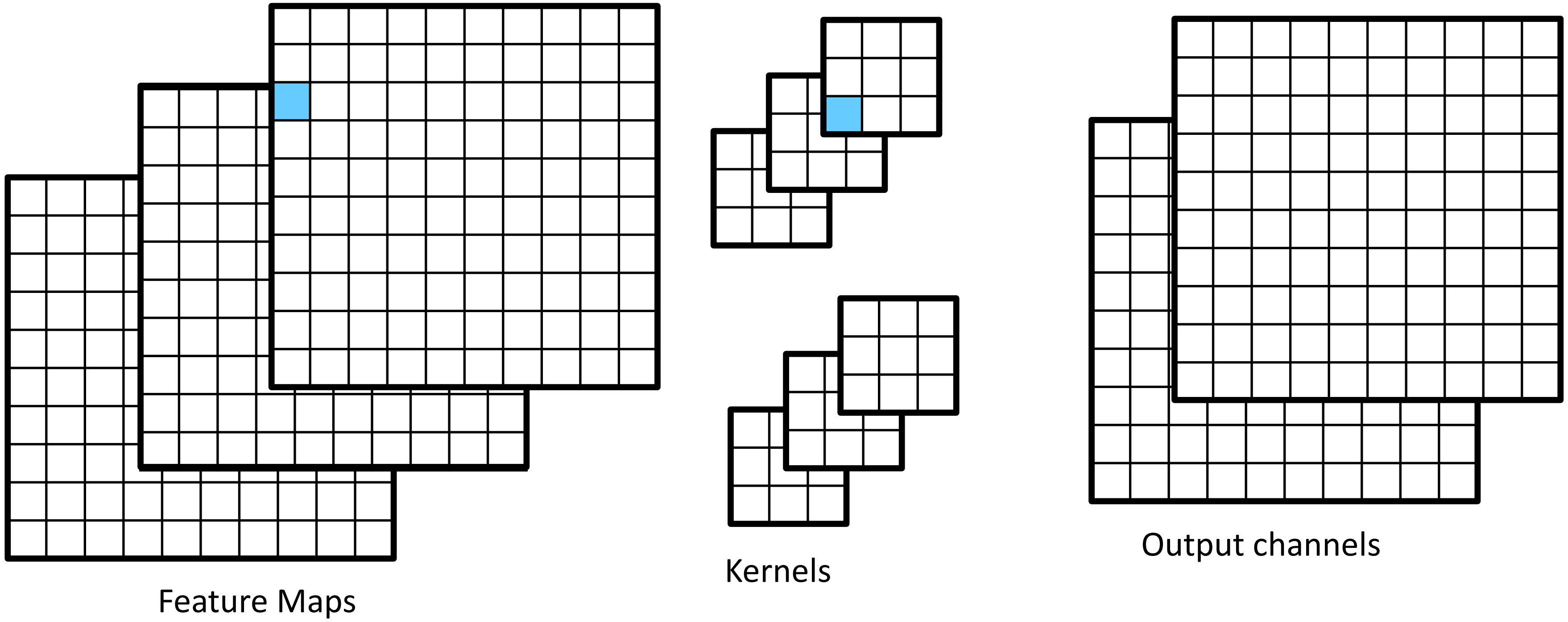
TensorFlow 2d convolution



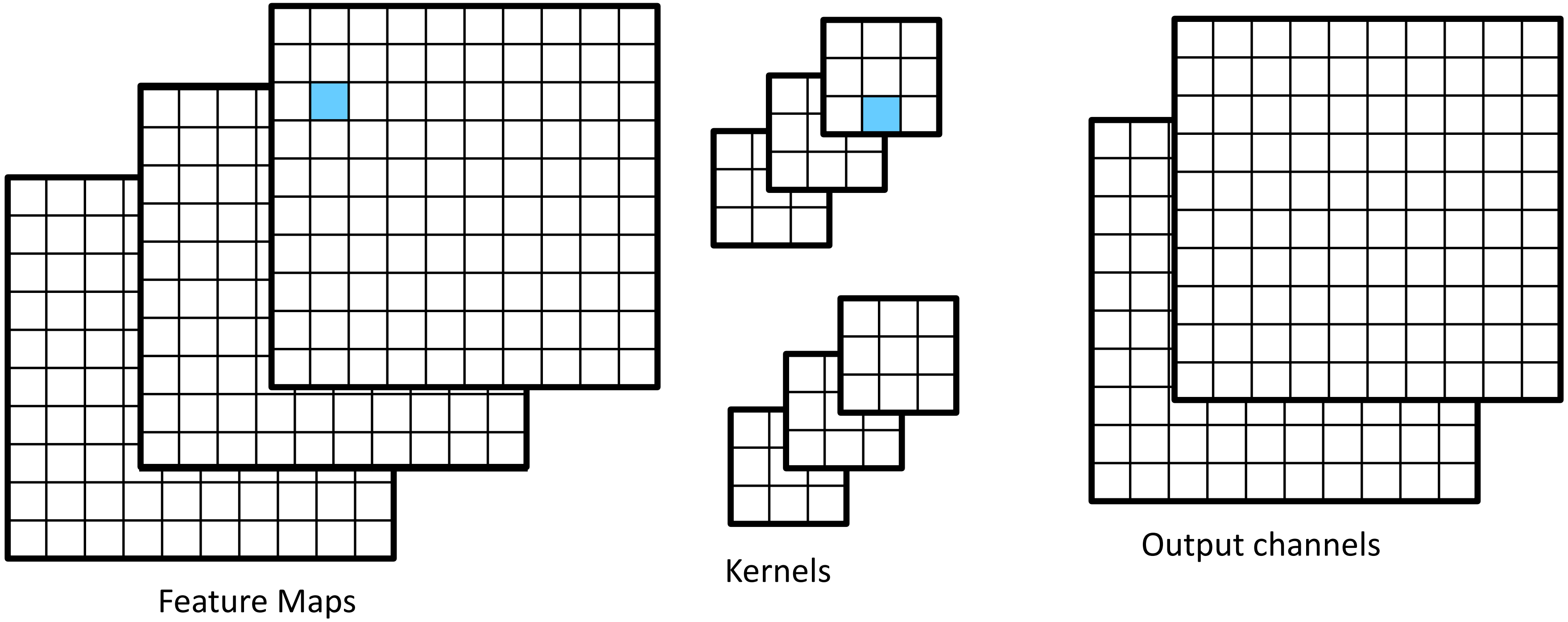
TensorFlow 2d convolution



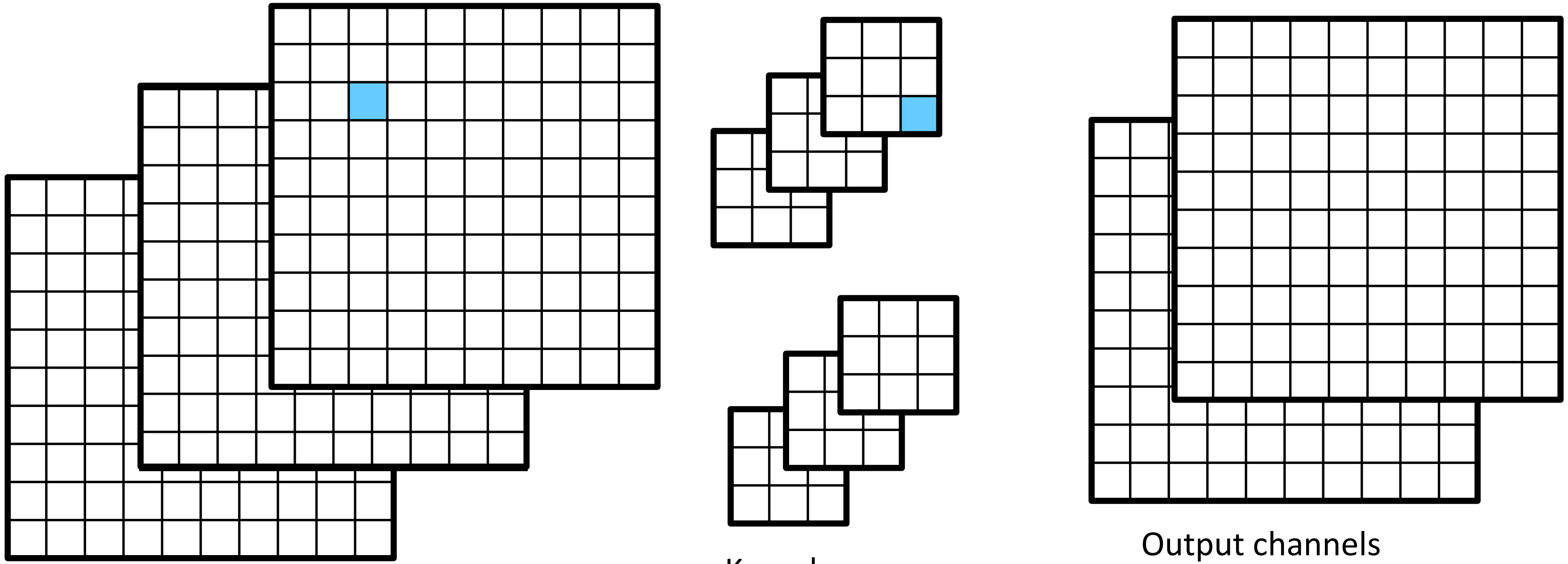
TensorFlow 2d convolution



TensorFlow 2d convolution



TensorFlow 2d convolution

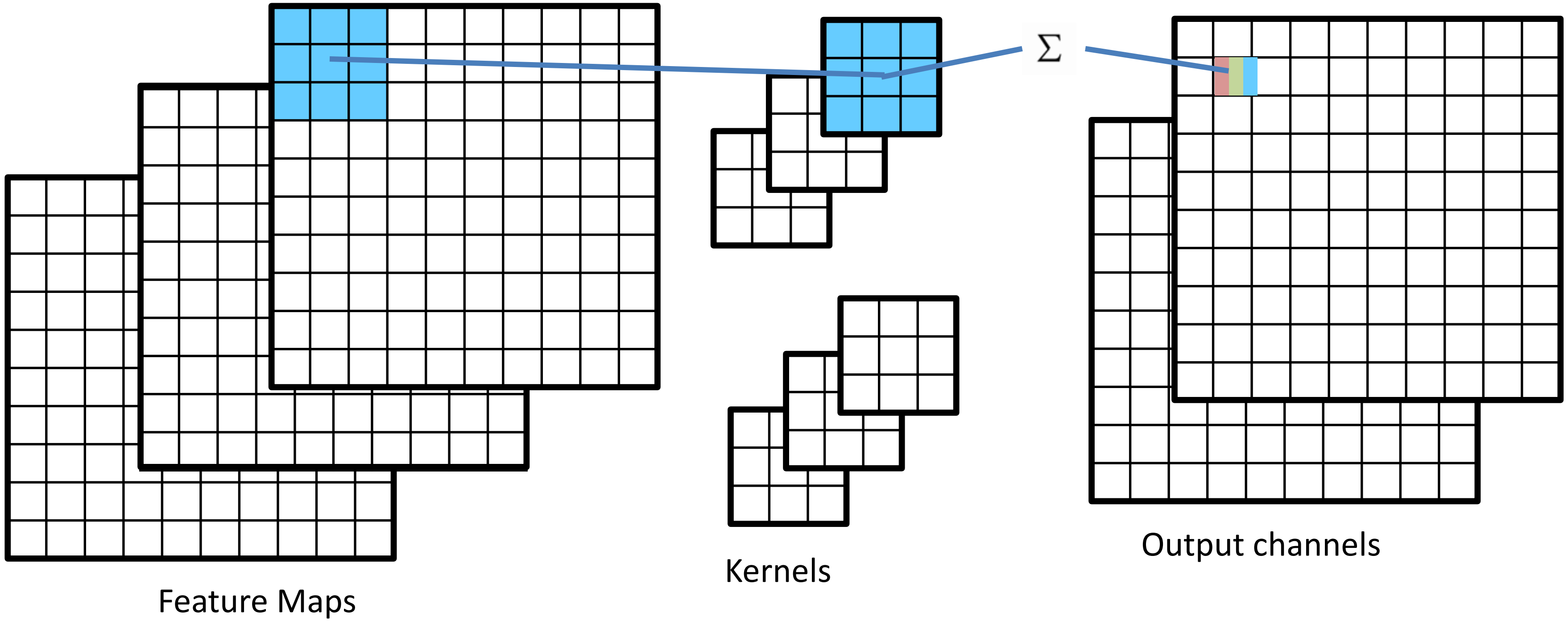


Feature Maps

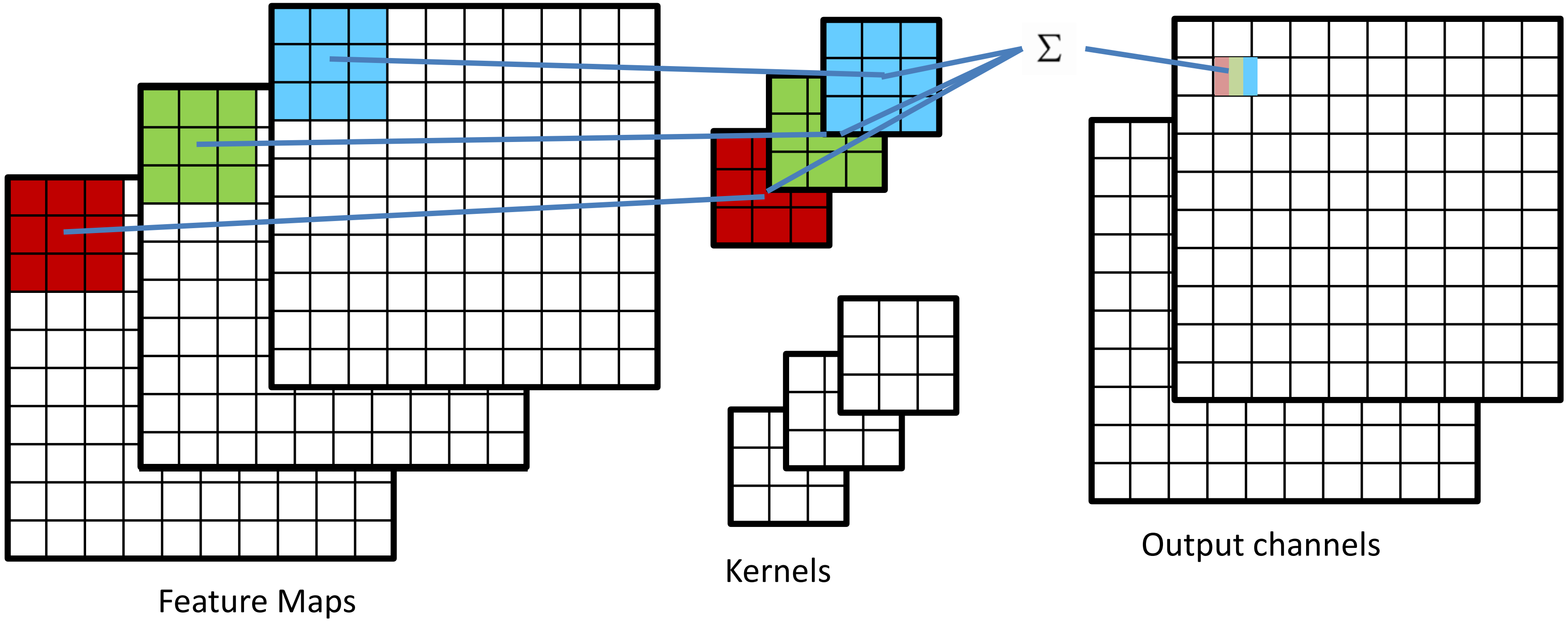
Kernels

Output channels

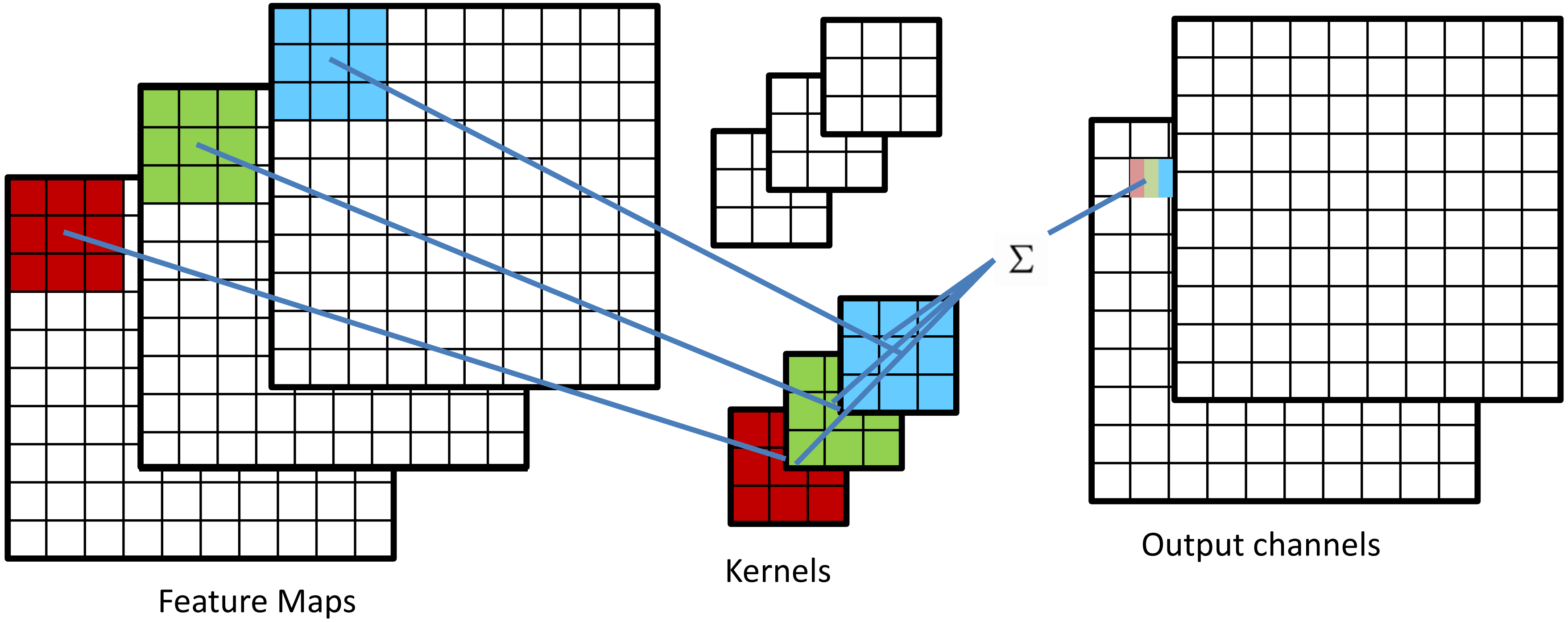
TensorFlow 2d convolution



TensorFlow 2d convolution



TensorFlow 2d convolution

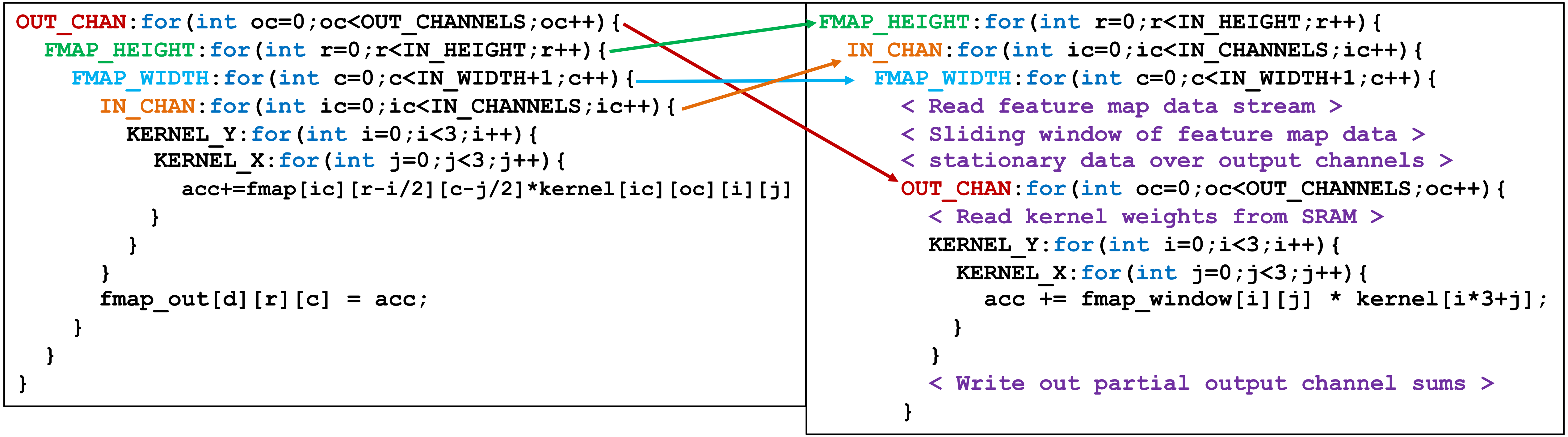


Architecture Alternatives

- Feature map constant
  - Read in each feature map, and apply all convolution kernels to it
  - Requires memory large enough to hold all output channels (partial sums)
- Output channel constant
  - Complete computation for each output channel in order
  - Requires memory large enough for only one output channel
  - Requires re-reading feature maps
- Tiled architecture
  - Compute outputs for a region of each input feature map
  - Requires even less memory, but more re-reads both feature maps and kernels

Different Architectures

- By simply reordering the loops different architectures can be created



Output Channel Constant

```
OUT_CHAN:for(int oc=0;oc<OUT_CHANNELS;oc++){
  FMAP_HEIGHT:for(int r=0;r<IN_HEIGHT;r++){
    FMAP_WIDTH:for(int c=0;c<IN_WIDTH+1;c++){
      IN_CHAN:for(int ic=0;ic<IN_CHANNELS;ic++){
        KERNEL_Y:for(int i=0;i<3;i++){
          KERNEL_X:for(int j=0;j<3;j++){
            acc+=fmap[ic][r-i/2][c-j/2]*kernel[ic][oc][i][j]
          }
        }
      }
      fmap_out[d][r][c] = acc;
    }
  }
}
```

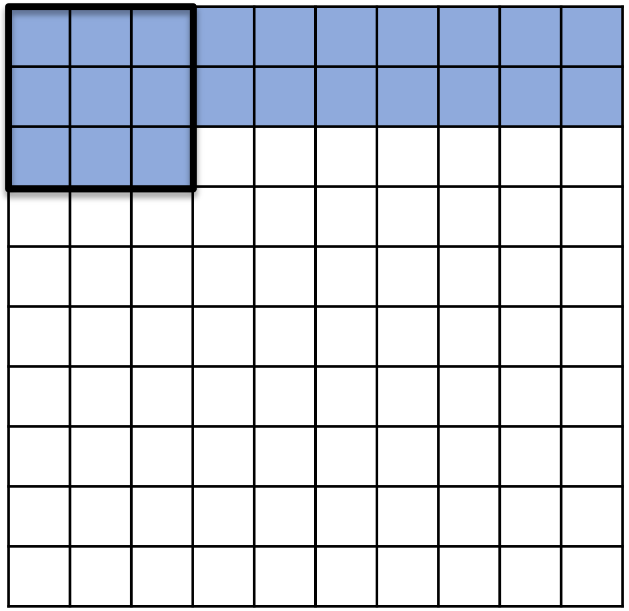
Feature Map Constant

```
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 >
      < stationary data over output channels >
      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[i][j] * kernel[i*3+j];
          }
        }
        < Write out partial output channel sums >
      }
```

Synthesis considerations

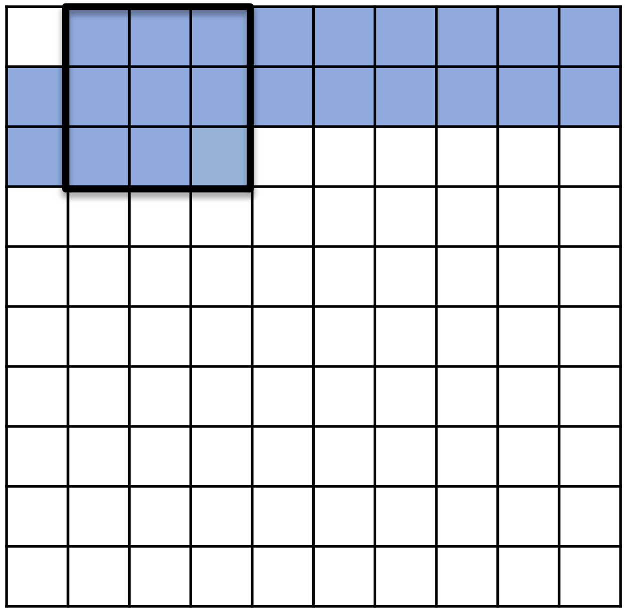
- Synthesizing “as-is” results in one multiplication per clock
  - Faster than software, but does not take advantage of parallelism in the algorithm
- The feature_map and kernels variables are mapped to memories
  - Each memory can perform one read per clock cycle
- Possible solutions
  - Multi-port the memories (expensive in area, routing resources)
  - Promote memories to registers (very expensive in area, power)
  - Partition memories
  - Map into a shift register

Shift Register



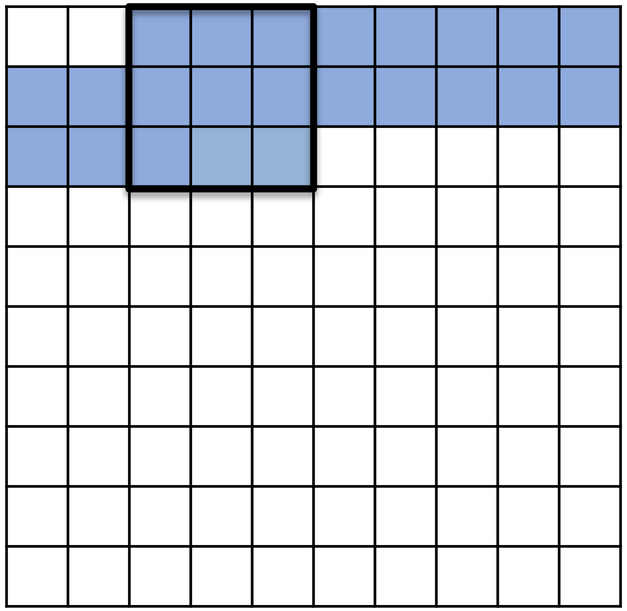
Create a shift register 2 lines + 3 pixels

Shift Register



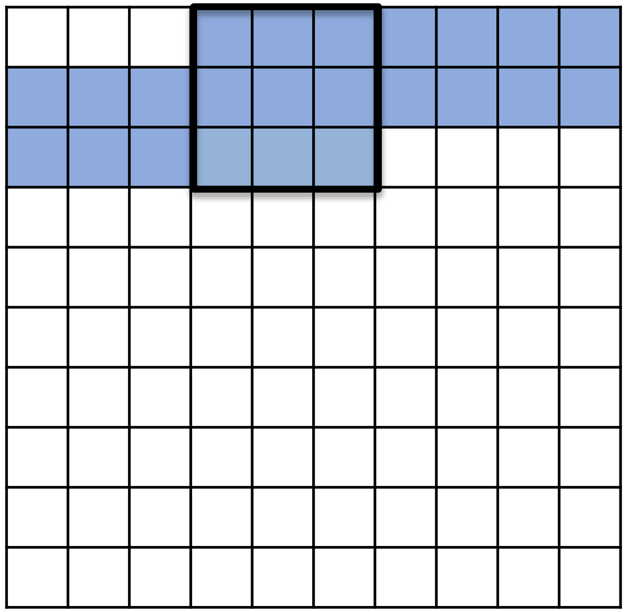
Create a shift register 2 lines + 3 pixels

Shift Register



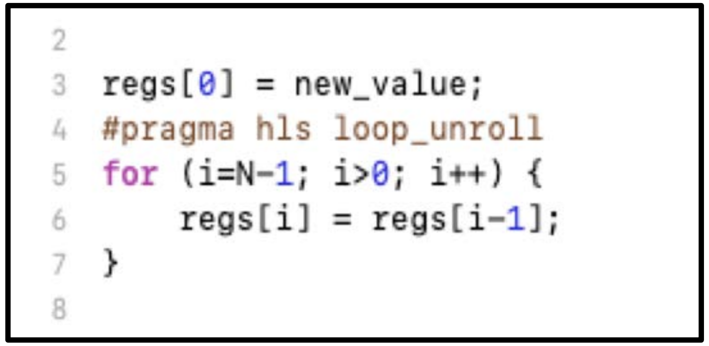
Create a shift register 2 lines + 3 pixels

Shift Register

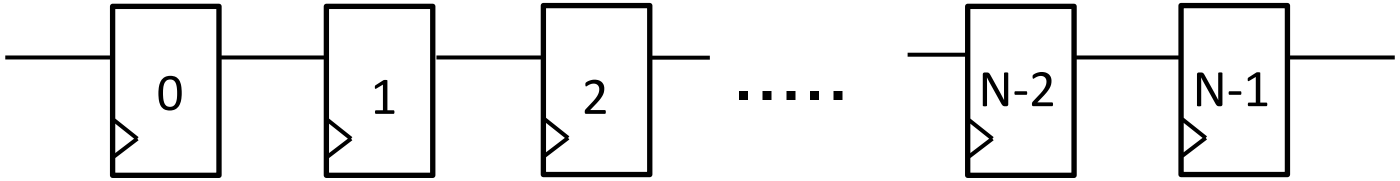


Create a shift register 2 lines + 3 pixels

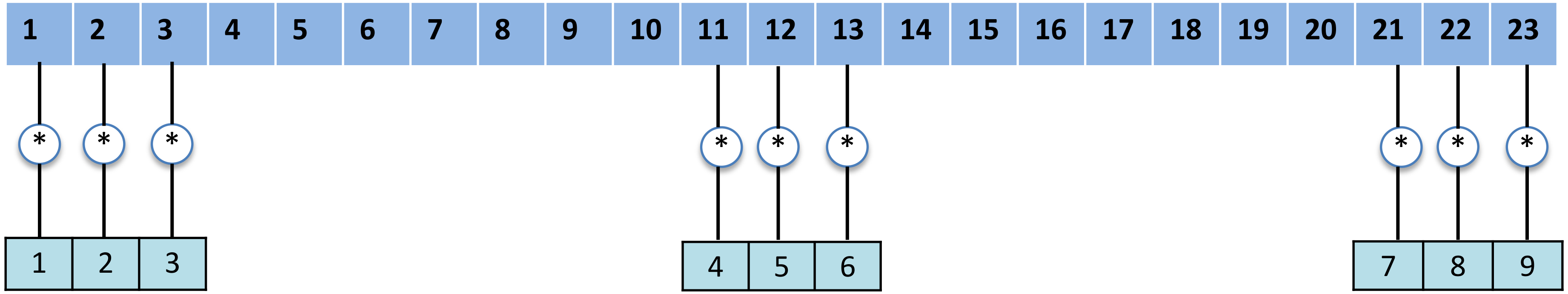
Shift Register



```

regs[0] = new_value;
#pragma hls loop_unroll
for (i=N-1; i>0; i++) {
    regs[i] = regs[i-1];
}

```

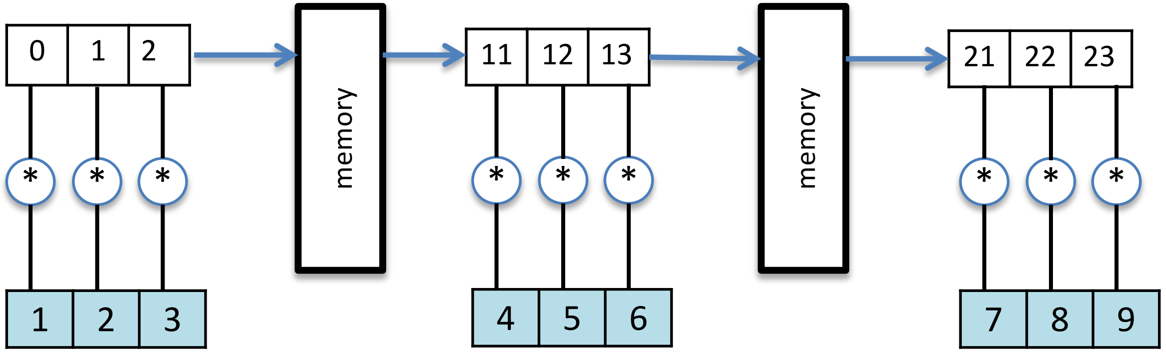


Parallel multipliers



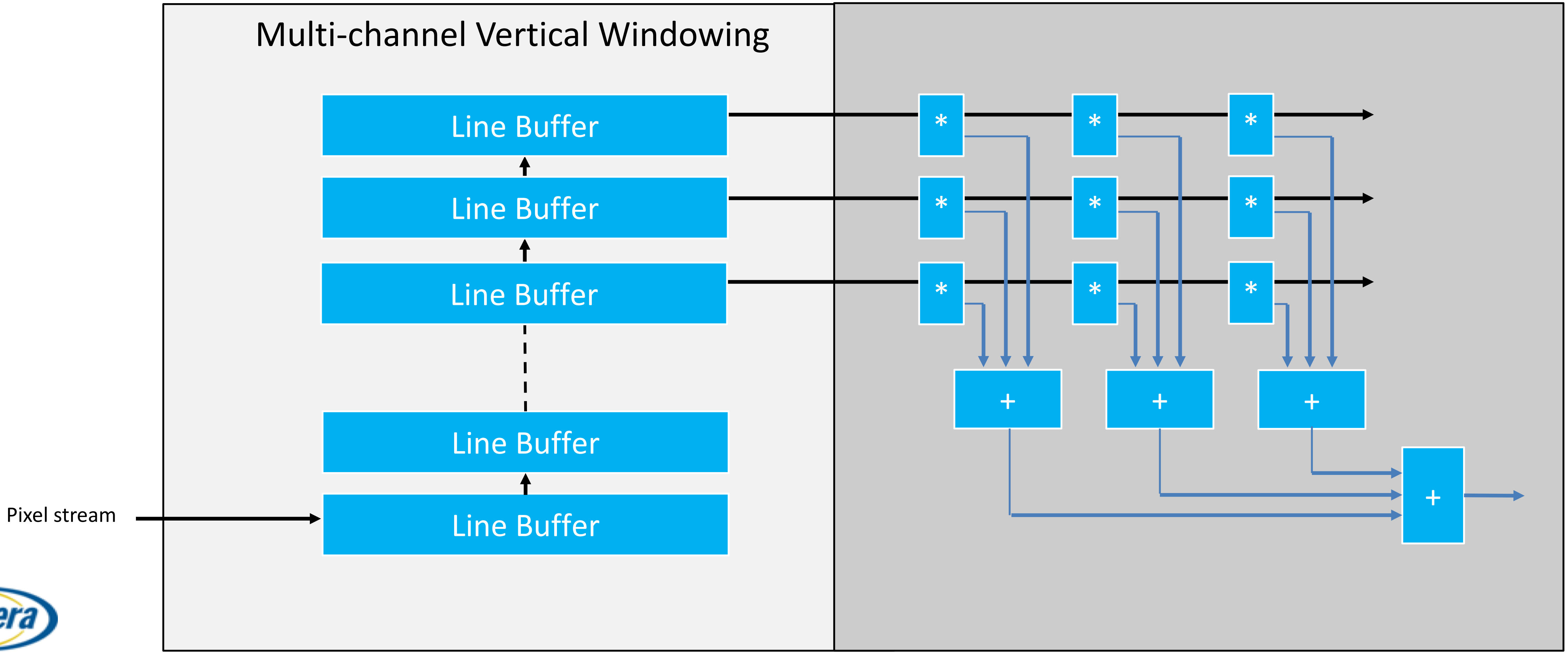
Parallel multipliers

For large feature maps this can be too many registers to be efficient. Add memories where there are no multiplier taps



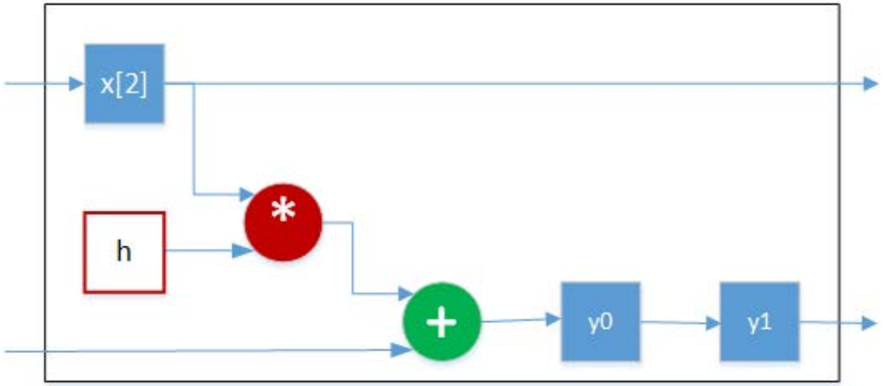
Multi-channel Sliding Window

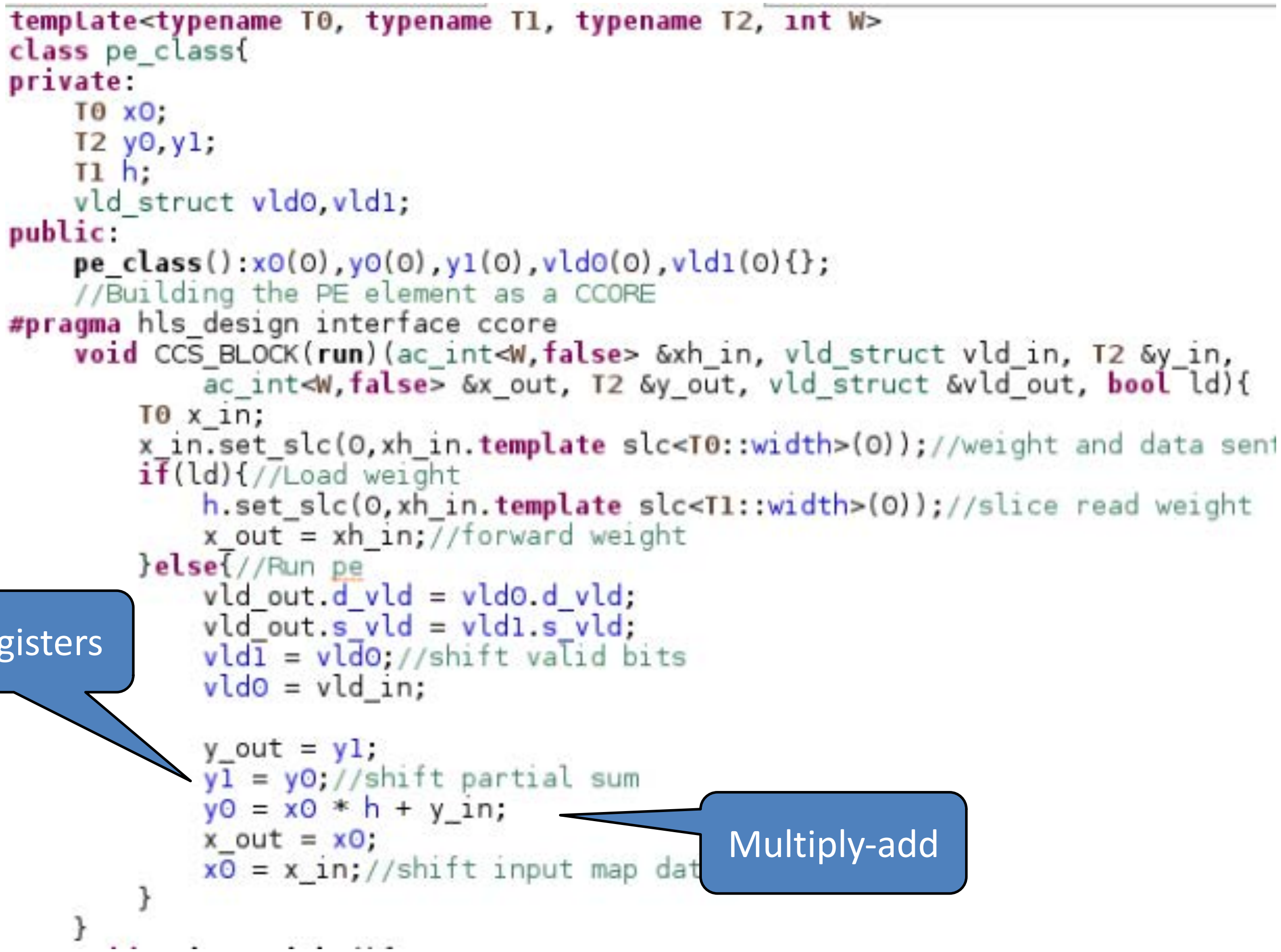
2-D convolution can be efficiently built by separating into vertical and horizontal sliding windowing plus accumulation buffers



C++ Class for Processing Element (PE)

- PE is explicitly described in a C++ class
  - Multiply-add
  - Shift registers
- Class can then be re-used in an array





```
template<typename T0, typename T1, typename T2, int W>
class pe_class{
private:
    T0 x0;
    T2 y0,y1;
    T1 h;
    vld_struct vld0,vld1;
public:
    pe_class():x0(0),y0(0),y1(0),vld0(0),vld1(0){};
    //Building the PE element as a CCORE
#pragma hls_design interface ccore
    void CCS_BLOCK(run)(ac_int<W,false> &xh_in, vld_struct vld_in, T2 &y_in,
            ac_int<W,false> &x_out, T2 &y_out, vld_struct &vld_out, bool ld){
        T0 x_in;
        x_in.set_slc(0,xh_in.template slc<T0::width>(0));//weight and data sent
        if(ld){//Load weight
            h.set_slc(0,xh_in.template slc<T1::width>(0));//slice read weight
            x_out = xh_in;//forward weight
        }else{//Run pe
            vld_out.d_vld = vld0.d_vld;
            vld_out.s_vld = vld1.s_vld;
            vld1 = vld0;//shift valid bits
            vld0 = vld_in;

            y_out = y1;
            y1 = y0;//shift partial sum
            y0 = x0 * h + y_in;
            x_out = x0;
            x0 = x_in;//shift input map dat
        }
    }
```

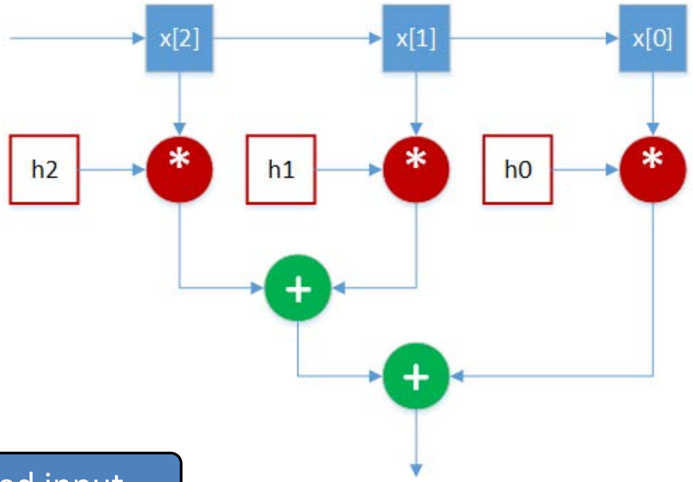
Shift registers

Multiply-add

Scalable PE Array Architecture

- Multiply-add tree convolution can be transformed into a chain of processing elements (PE)
  - FPGA routing friendly
- Systolic array is the simplest PE array
  - Simpler interconnect and easy to understand
  - There are better ones in use today (SCNN, Eyeriss, Chain)

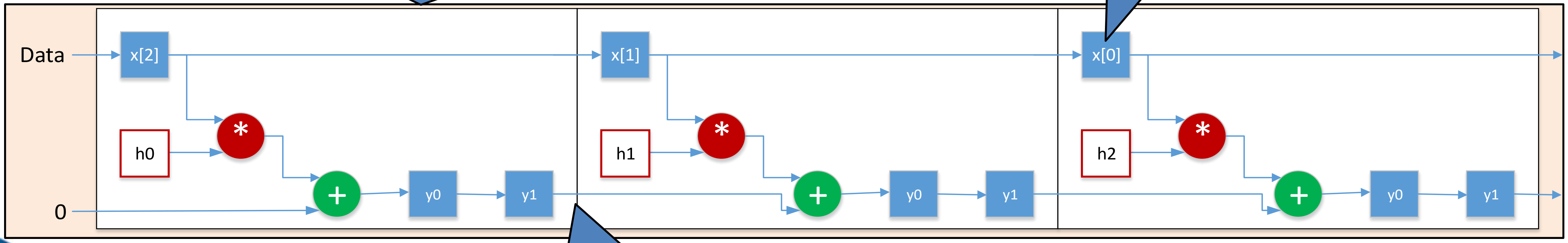
1x3 Traditional convolution



Processing Element (PE)

1x3 PE Array convolution

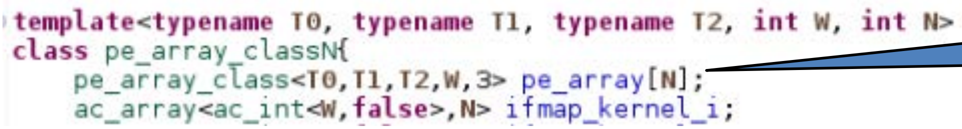
Shifted input



Partial sum input

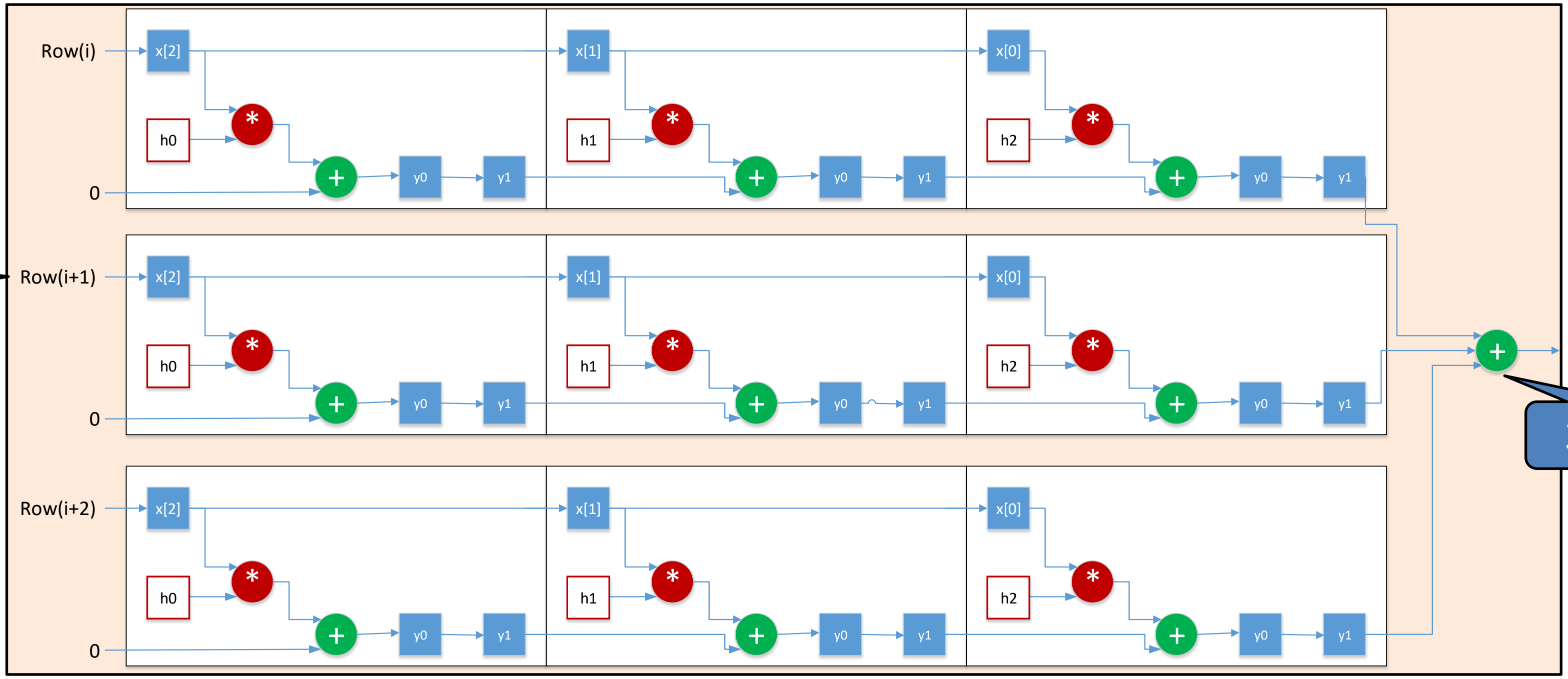
Matrix of PEs

- Multiple 1-d convolutions
  - Easy, just another array of classes



```
template<typename T0, typename T1, typename T2, int W, int N>
class pe_array_classN{
    pe_array_class<T0,T1,T2,W,3> pe_array[N];
    ac_array<ac_int<W,false>,N> ifmap_kernel_i;
```

Array of 3 1x3 PE convolutions , N=3

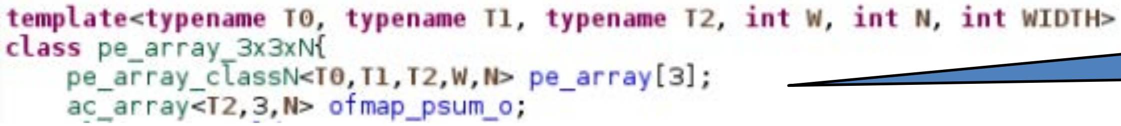


Need 3 Rows simultaneously (windowing)

3x3 result

Multiple Matrices of PEs

- Multiple output channels produced from single input channel
  - N=64 gives 576 parallel multiplications
  - ~690 Billions ops/sec @600MHz
- Minimal routing congestion
- Significantly less memory bandwidth required
- Still must accumulate partial sums in local memory



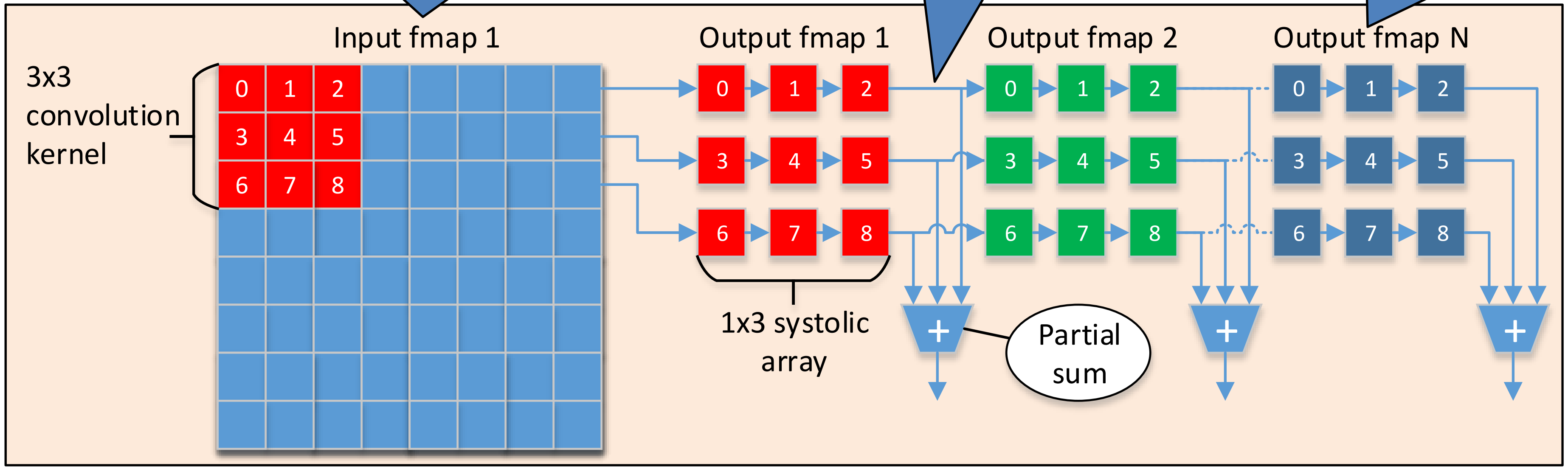
```
template<typename T0, typename T1, typename T2, int W, int N, int WIDTH>
class pe_array_3x3xN{
    pe_array_classN<T0,T1,T2,W,N> pe_array[3];
    ac_array<T2,3,N> ofmap_psum_o;
```

3 element array of Array of 1x3 PEs, N=64

Single input channel

Input channel data is reused

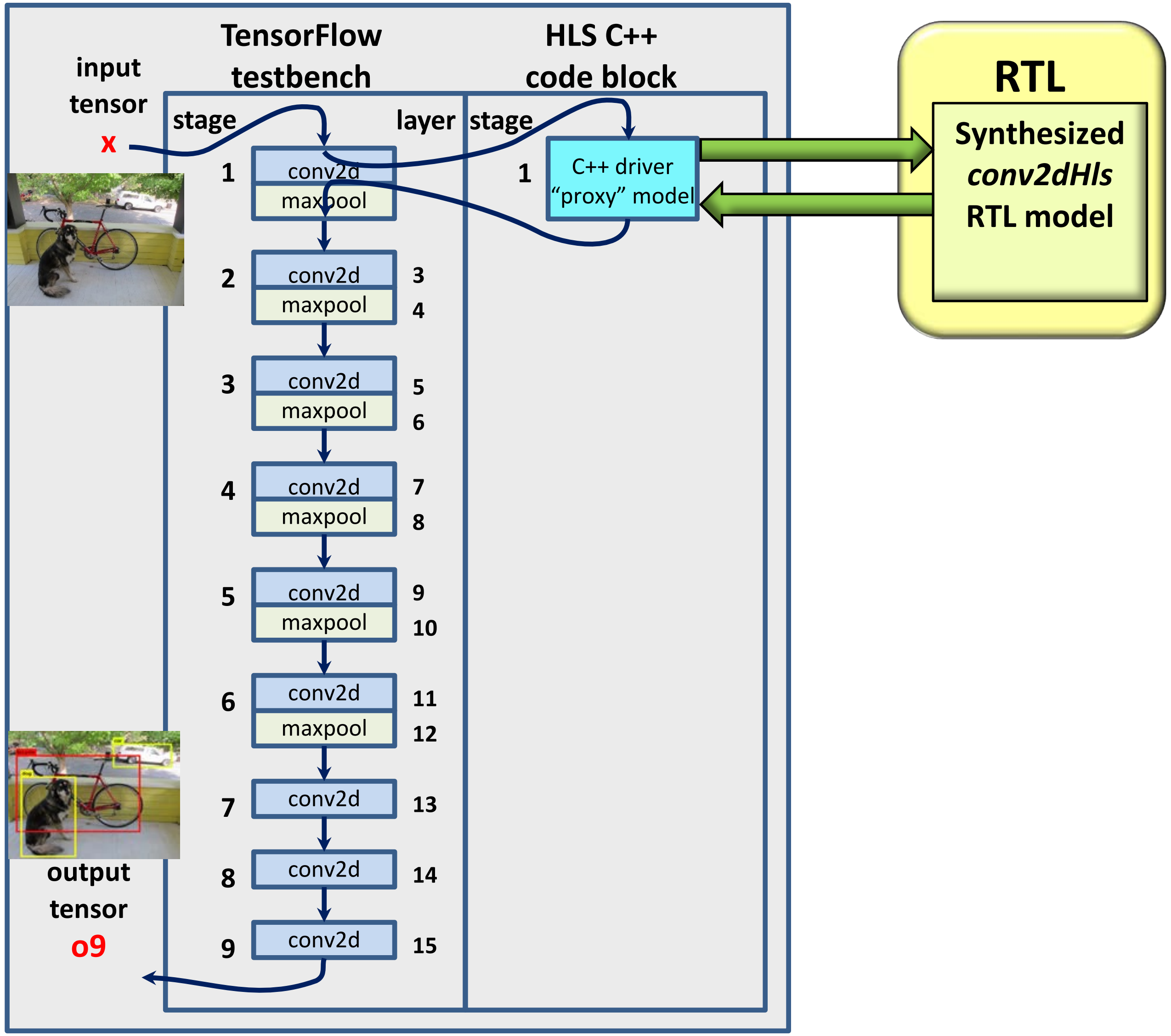
Multiple output channels



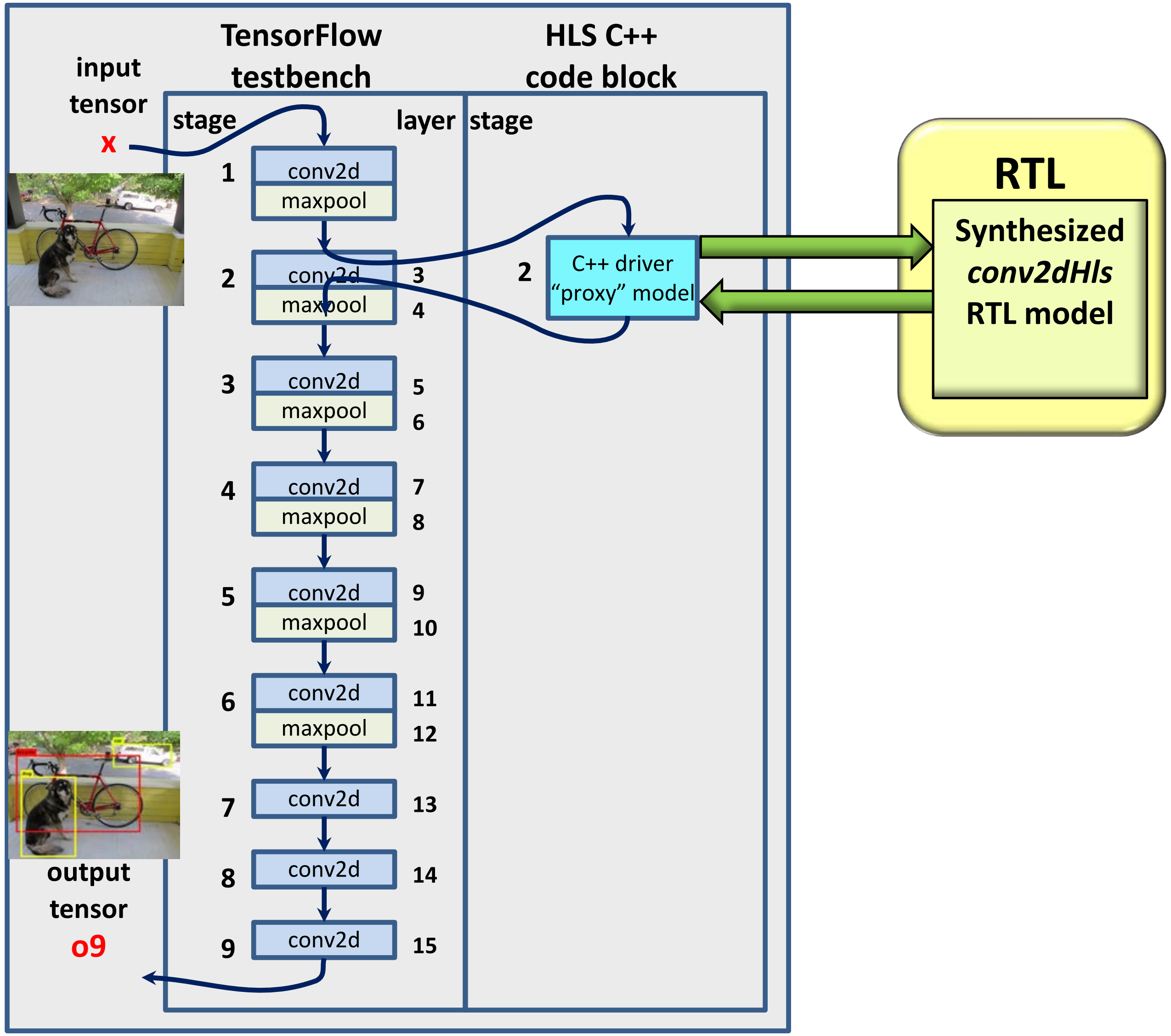
RTL Creation

- Once the architecture is determined, high level synthesis can be used to create the RTL implementation of the component
  - Interface synthesis creates bus connections for master and slave interfaces
- As RTL is created it can be dynamically verified
  - Stimulus can be captured from execution of Python with algorithmic C
  - Reponses from RTL compared with responses from algorithmic C

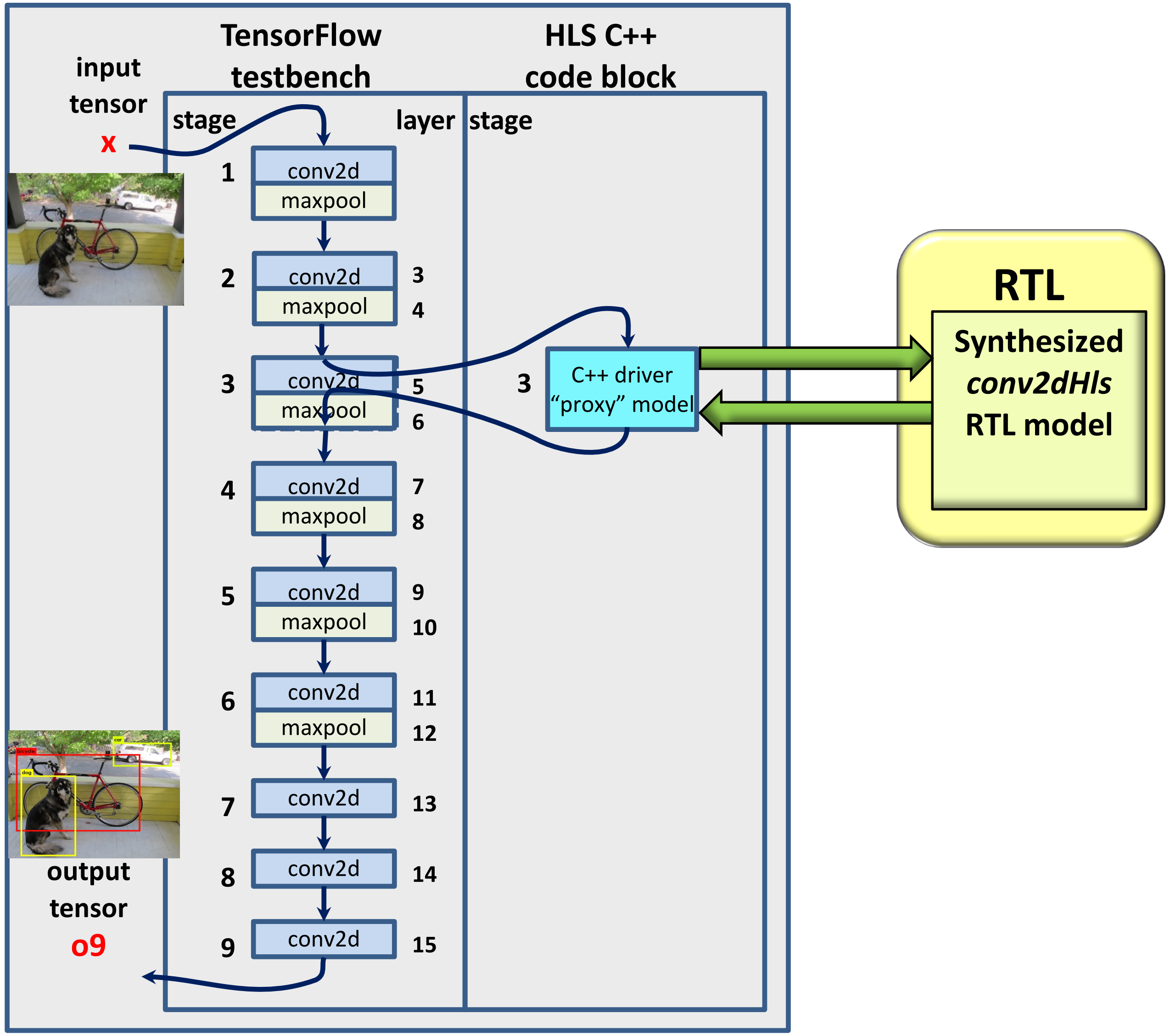
Replace One Layer at a Time



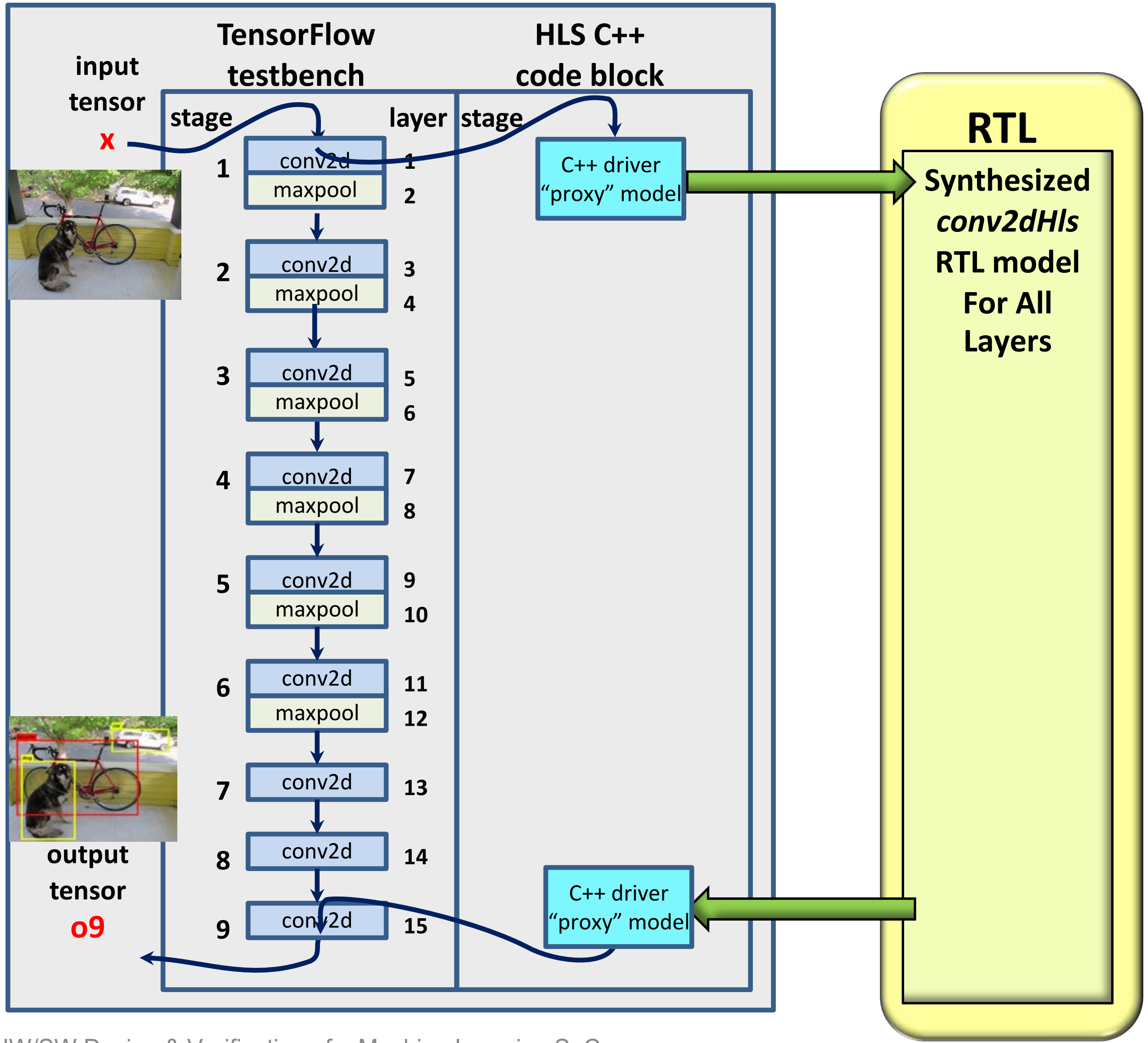
Replace One Layer at a Time



Replace One Layer at a Time

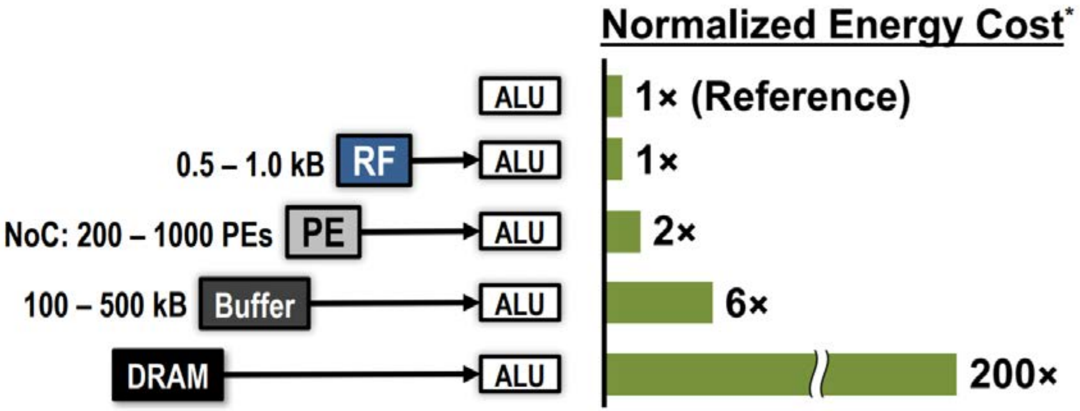


Replace All Layers



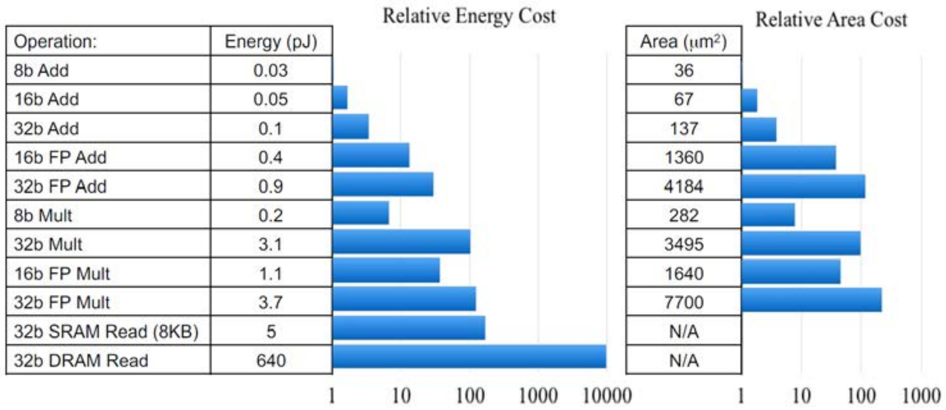
Power Considerations

- Keep data local
  - Very important for ASIC
- Floating-point is costly
  - Used in training of networks
  - Not needed in network inference engine
- Doesn't need to be 2^x bit-widths
  - Processors are fixed bit-width
- 8-bit integer multiplier is 27 times smaller and uses 19 times less energy then a 32-bit floating point multiplier



*NVIDIA 2017

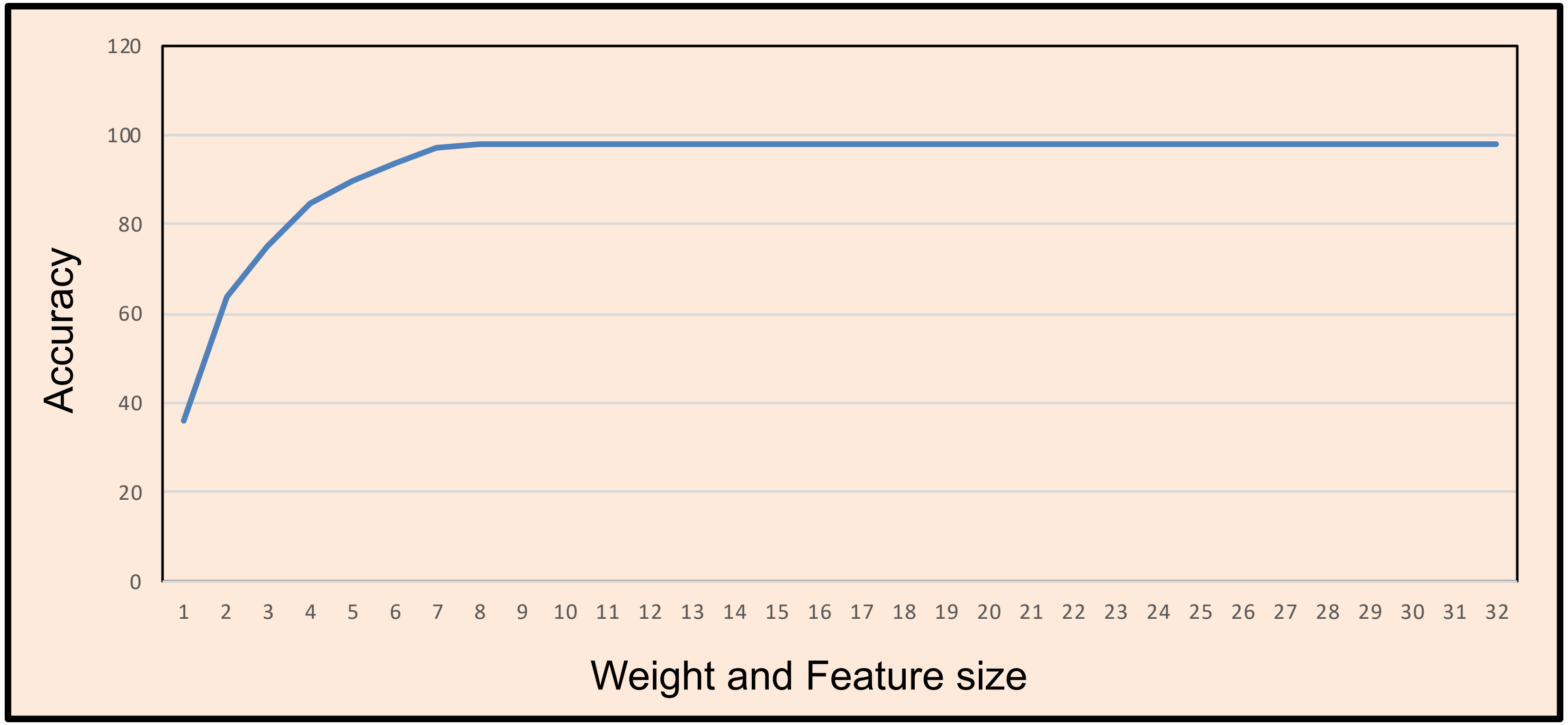
Cost of Operations



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

Accuracy vs. Bit Width for CNN

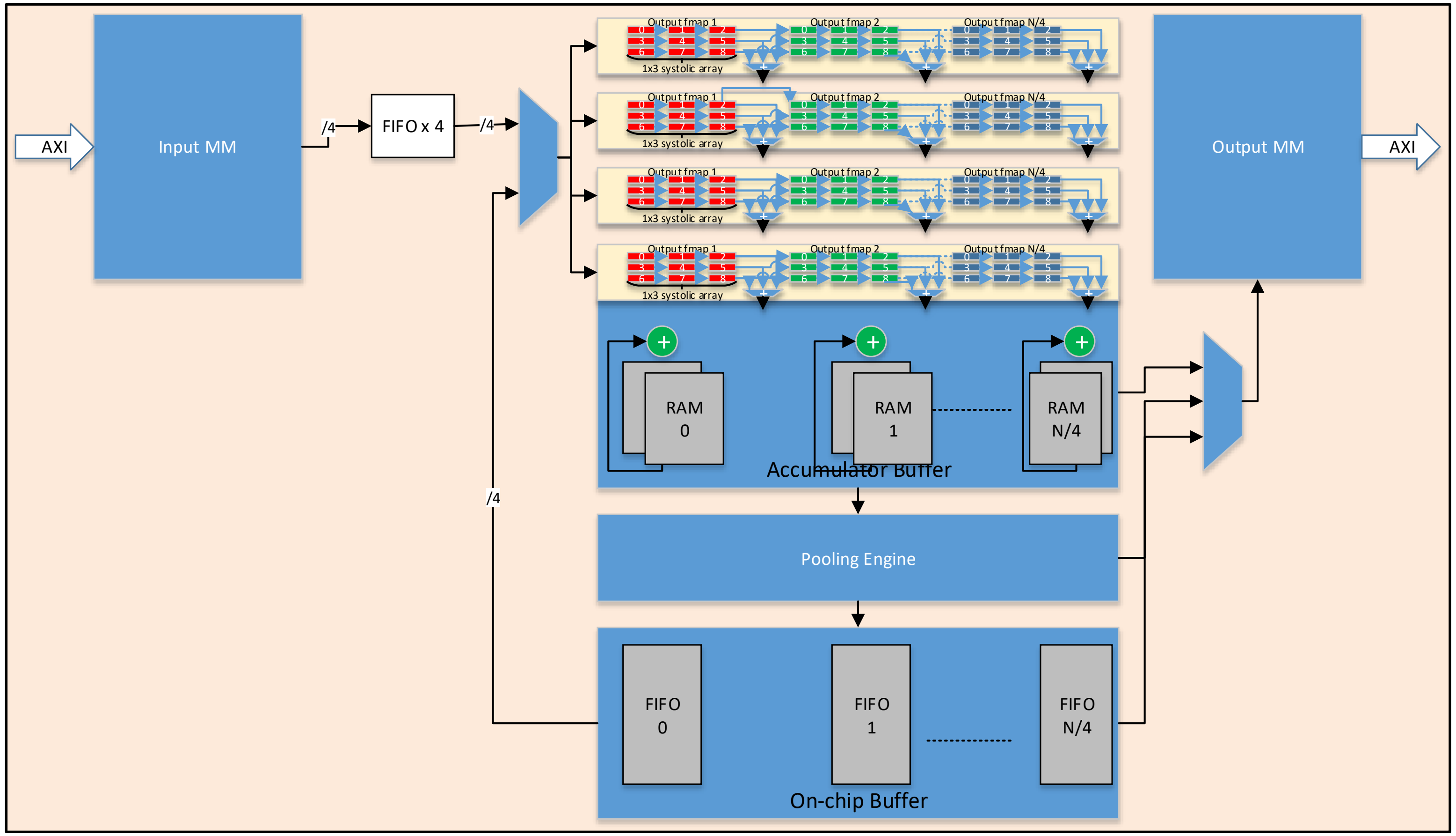


For ResNET

- 32-bit weights improves accuracy by less than 0.1% over 8-bit weights

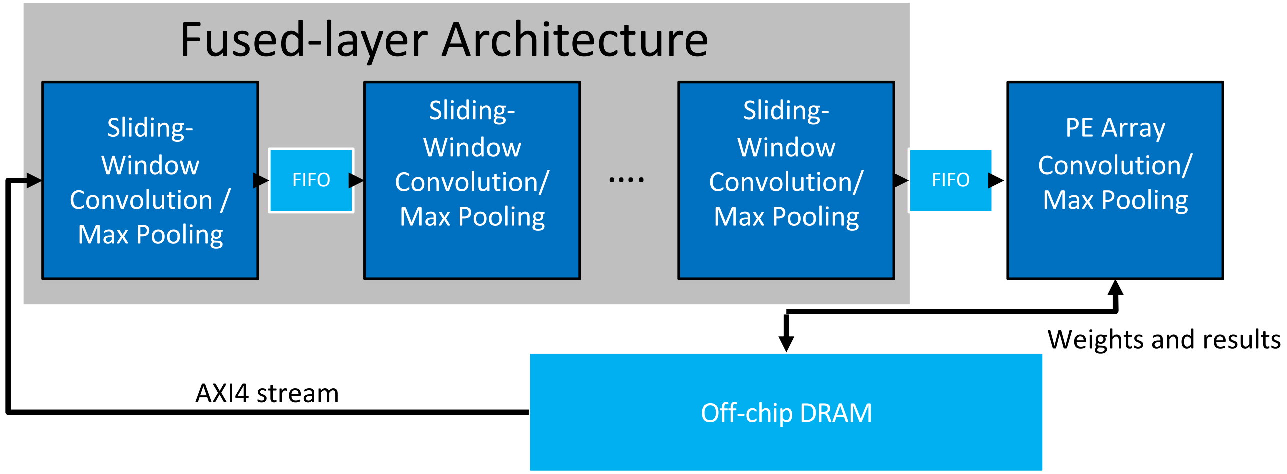
Architectural Exploration with HLS

- Original PE array was sub-optimal
- Process multiple input channels simultaneously
  - 4 PE arrays
  - Better utilization of AXI4MM bandwidth
  - Reduce on-chip memory by 4x
- Recoded in a few days
  - Evaluated PPA



Hybrid Architecture for Lowest Power

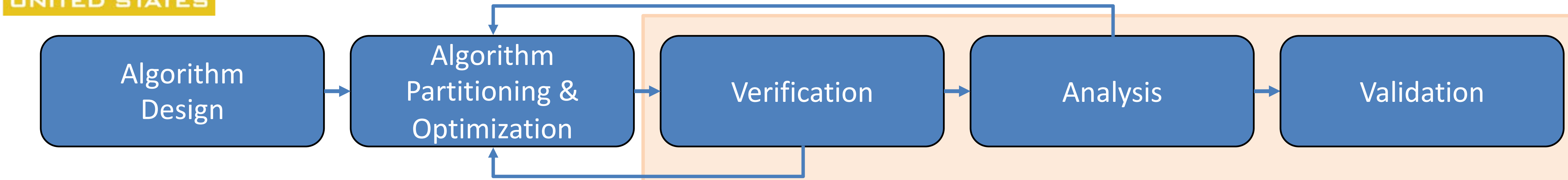
- Earlier layers can be processed together
  - Fused layer architectures don’t need all of the feature-map data from a previous layer to process the current layer
  - Keeping the data on-chip gives much lower power consumption
  - Works well for smaller number of input/output feature maps
- Later layers need a different kind of architecture
  - Large number of feature maps and weights
  - PE array architectures work well

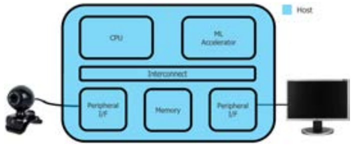

Fused-layer Architecture
Sliding-Window Convolution / Max Pooling
FIFO
Sliding-Window Convolution/ Max Pooling
....
Sliding-Window Convolution/ Max Pooling
FIFO
PE Array Convolution/ Max Pooling
Weights and results
AXI4 stream
Off-chip DRAM

From Block to Full SoC

VERIFICATION

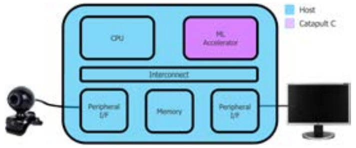
Our Story in Five Steps





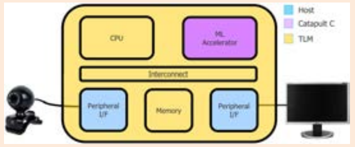
- Tiny YOLO algorithm, written in Python, executed in TensorFlow on a desktop or laptop as stand alone
- It inferences a camera input and it displays processed output on a screen
  - Verify algorithm works properly

- Speed ≃ 0.4 sec/inference

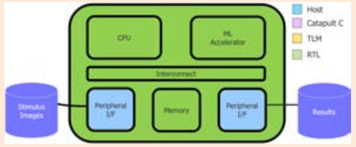


- Manual conversion of Tiny YOLO to C for High-Level Synthesis
  - Target wide variety of implementation architectures without re-coding
  - Common testbench for different abstraction levels
  - Automated creation of bus interfaces to surrounding system

- Speed ≃ 4 sec/inference



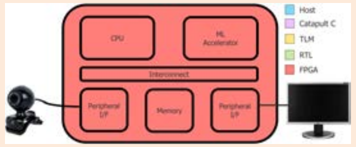
- Block-level verification at C and RT levels with a reusable verification environment
- Exploiting hybrid platform to maximize flexibility in verification
- And, enable earliest SW development and SW-driven verification
- Utilize HW-assisted verification for large dataset tests and full SoC verification



- Early & continuous power, performance analysis from algorithm through full SoC
- Utilize hybrid to focus analysis at block or broader levels
- Execute platform with same software stack from Hybrid platform
  - Realistic Performance
  - Accurate Power
  - Functional Coverage

Speed:

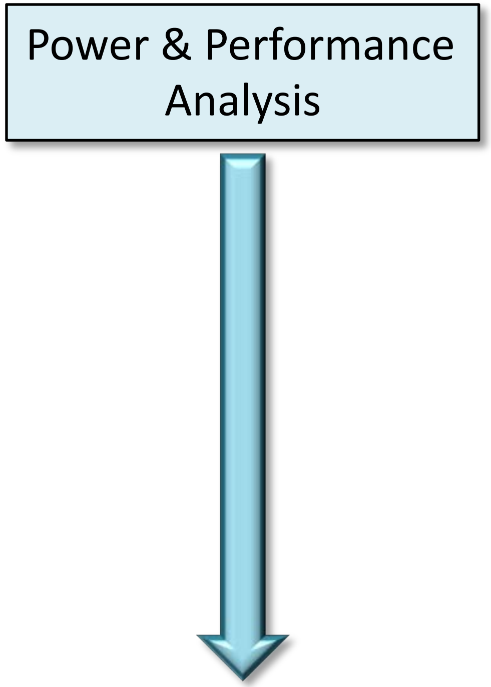
- 21,000 sec/inf RTL SW sim
- 10 sec/inf emulation
- 0.03 sec/inf prototype



- Block-level validation in SoC context with hybrid
- Prototype full SOC
  - Enable complete SW stack & system validation
  - Using real-world stimulus
- Pre-Si Validation
  - Connect to real interfaces, at speed
  - Prepare post-Si validation environment, tests and debug capabilities

Progression of Verification

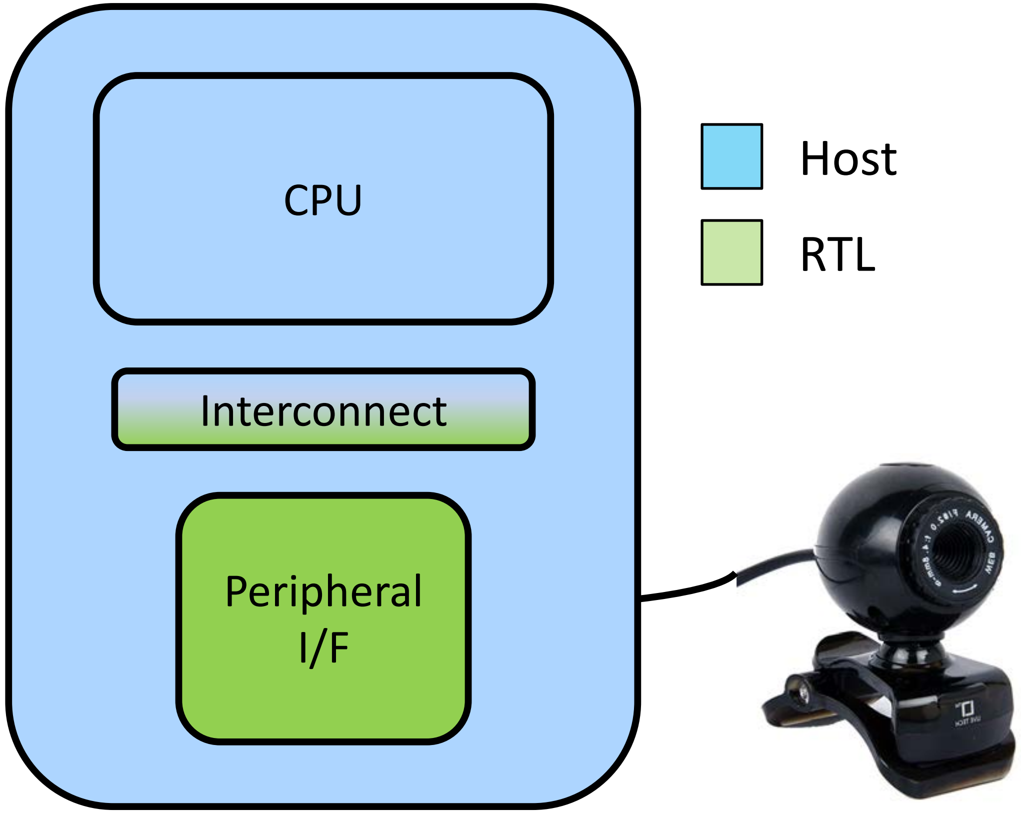
- **IP Block Verification at RTL**
- Earliest SW Enablement (SoC context)
- IP Block Validation Leveraging hybrid
  - Using a SW enabled flow for power and performance
- Full SoC Verification & validation
  - Focus on power & performance analysis



Power & Performance Analysis

IP Block Validation – Peripherals

- Objective: Ensure IP block functions correctly in SoC context
- Requires CPU subsystem
  - RTL or Virtual (Hybrid)
  - Driver/FW driven testing – the SW is key to the SoC context
- Environment
  - ICE is typical to validate ”plugfest” level compatibility with external world
  - Virtual may be used for subset or all tests
  - Post-silicon validation & debug environment functionality



Creating a portable test harness reusing environment from HLS C++ Verification

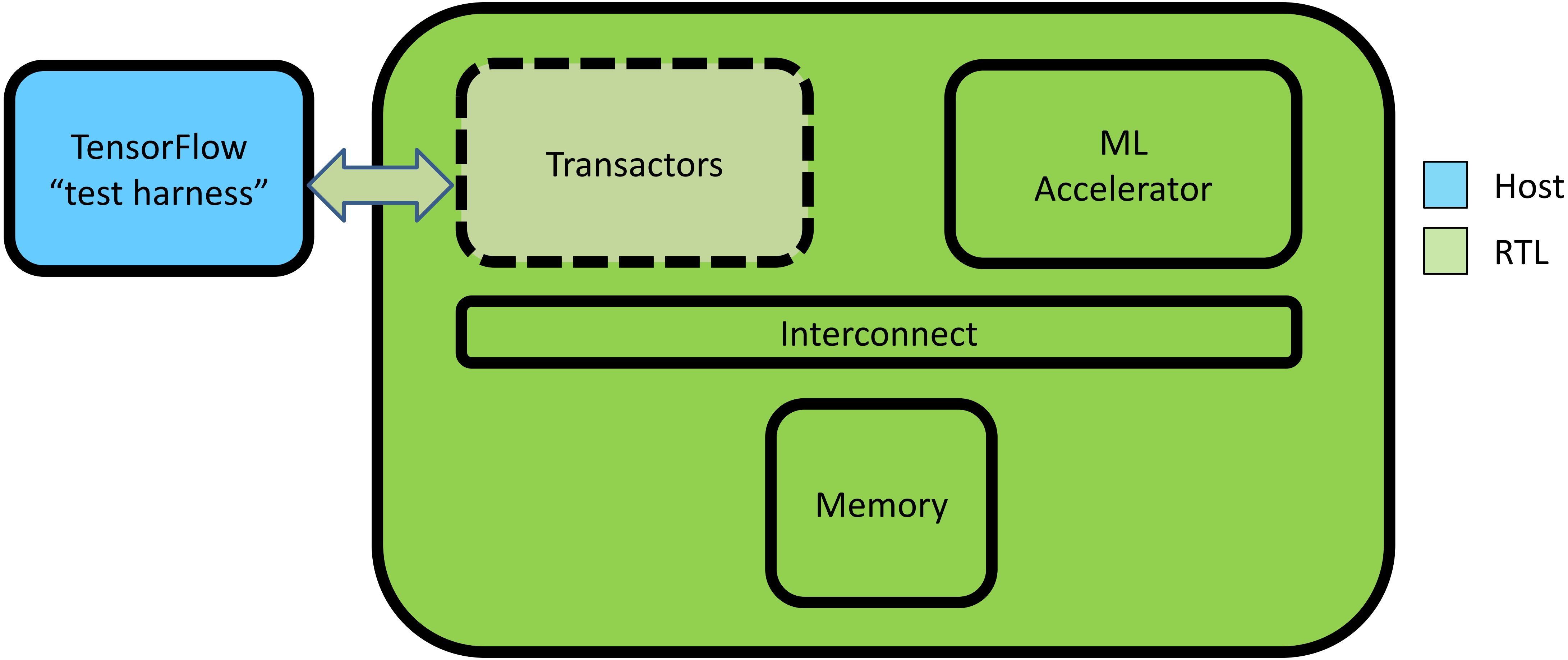
- Object recognition is test dataset intensive verification
  - Perfect application of HW-assisted to accelerate block-level RTL verification

- Create an environment for the TensorFlow framework and its host O/S

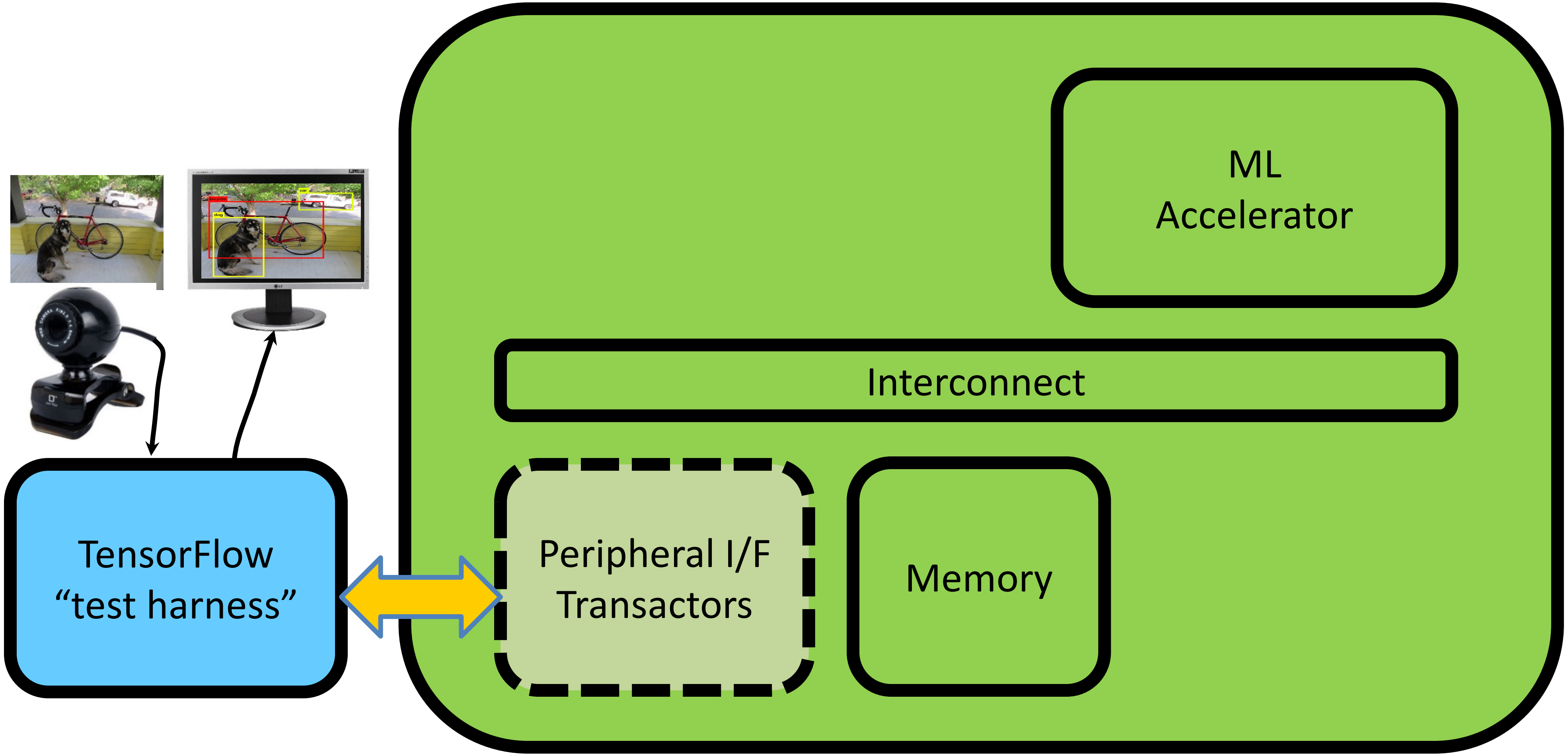
- Reuse the environment for the HLS C++ verification

TensorFlow test harness for RTL

Minimal SoC Subset for Verification



TensorFlow test harness for RTL



🟦 Host

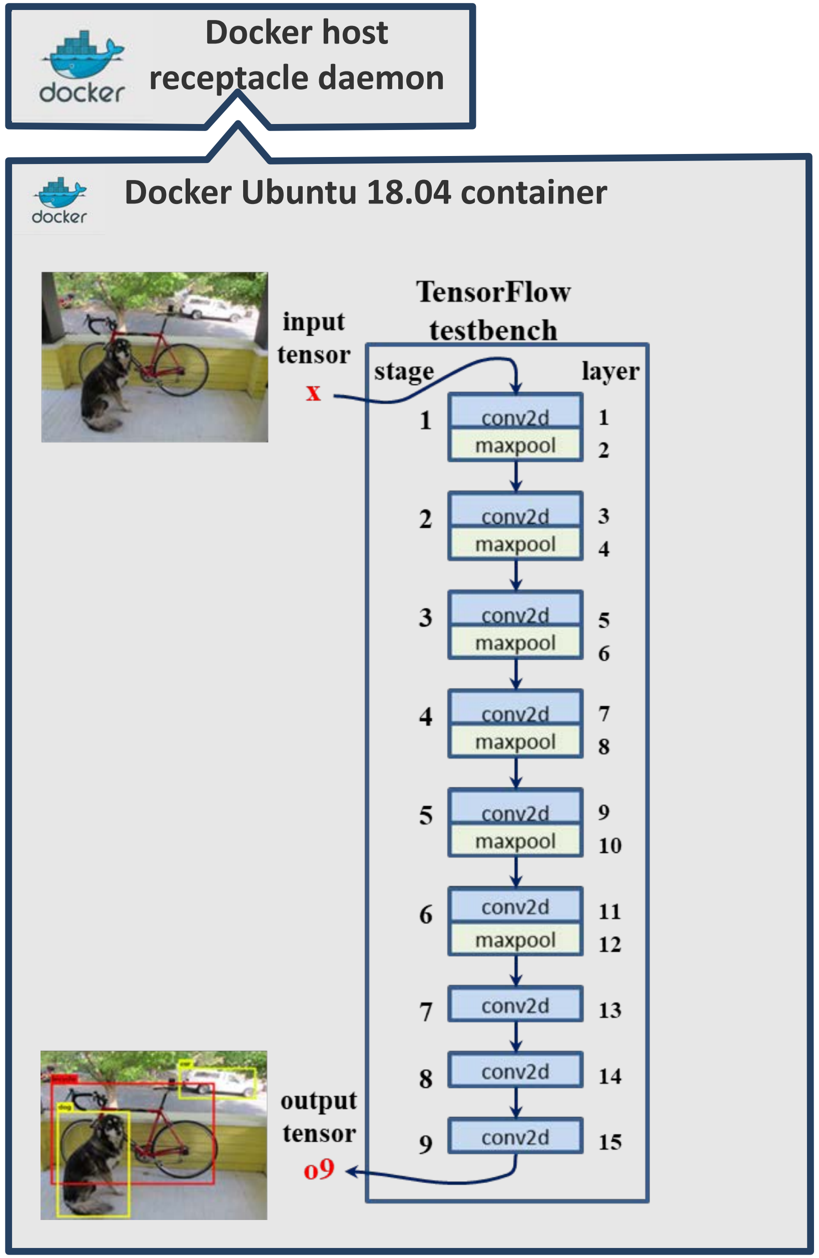
🟪 Catapult C

🟨 TLM

🟩 RTL

Encapsulated(containerized) pure TensorFlow environment

- Runs original YoloTiny design in a pure TensorFlow environment
- Encapsulates entire **Ubuntu 18.04** host O/S and **TensorFlow** framework into a *Docker* container
- This simplifies complex installation process for AI frameworks and makes them easily portable and reusable among different hosts



docker Docker host receptacle daemon

docker Docker Ubuntu 18.04 container

input tensor x

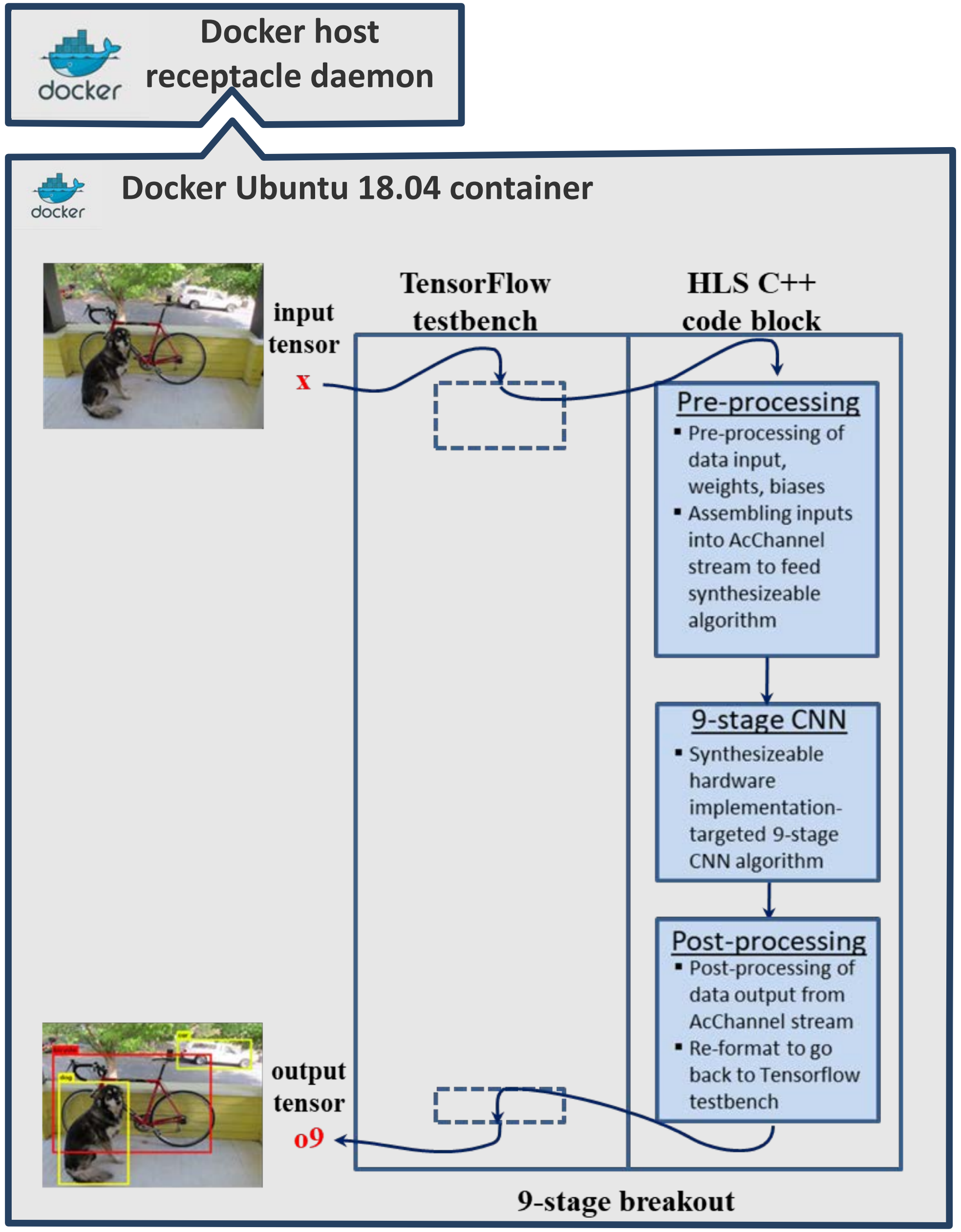
TensorFlow testbench

stage		layer
1	conv2d	1
	maxpool	2
2	conv2d	3
	maxpool	4
3	conv2d	5
	maxpool	6
4	conv2d	7
	maxpool	8
5	conv2d	9
	maxpool	10
6	conv2d	11
	maxpool	12
7	conv2d	13
8	conv2d	14
9	conv2d	15

output tensor o9

Test harness for TensorFlow + HLS C++validation

- Replace original 9 stages of the CNN algorithm with HLS compliant C++ implementing equivalent algorithm
- Still test new C++ code prototypes in the context of original TensorFlow framework
  - Input image stream, weights loading, and final output processing kept in Python/TensorFlow front end
- Pre-verify the synthesizeable code before generating RTL from it



Docker host receptacle daemon

Docker Ubuntu 18.04 container

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

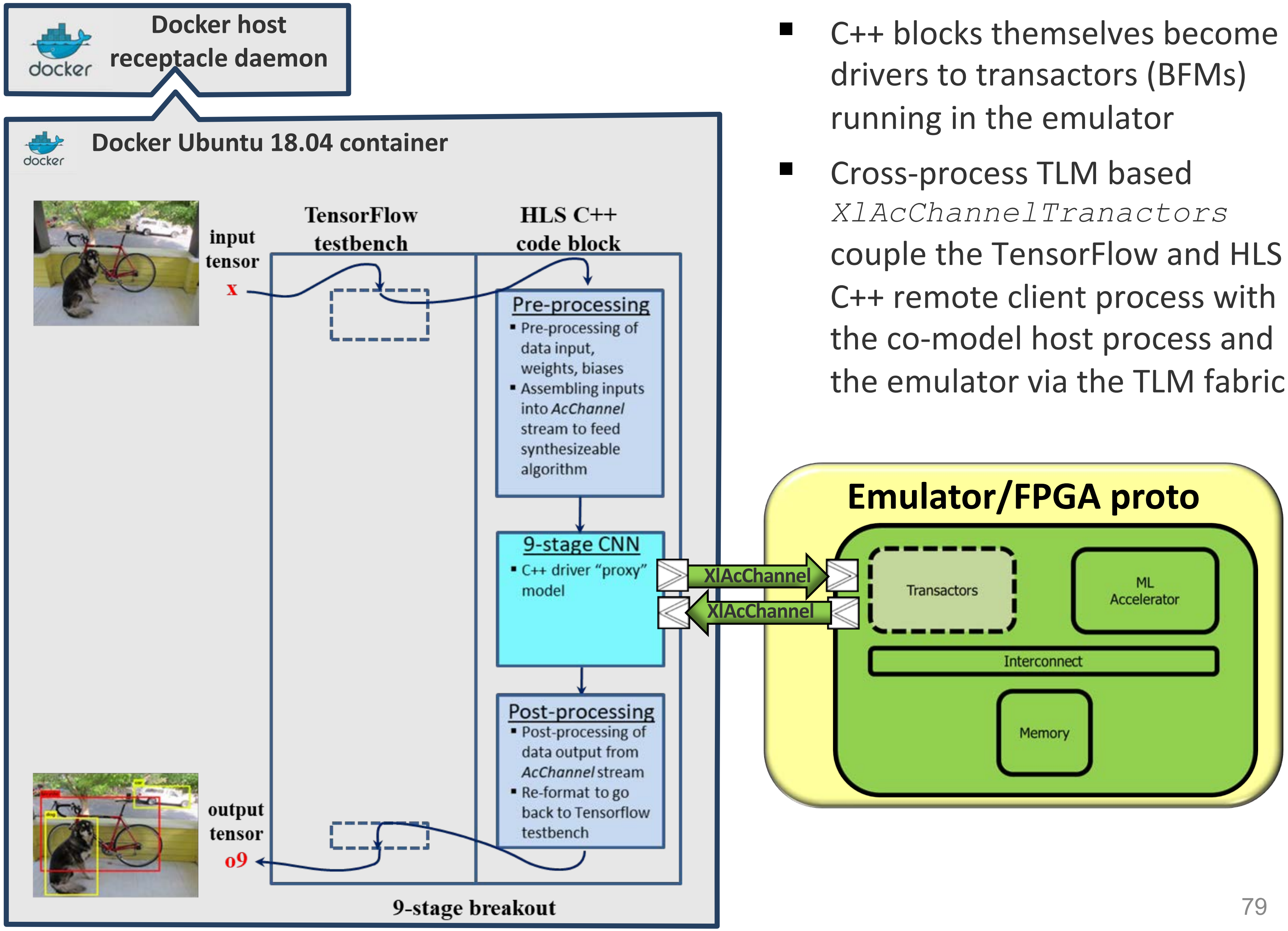
9-stage CNN
- Synthesizeable hardware implementation-targeted 9-stage CNN algorithm

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

- Reuses same Docker container image shown previously as portable “test harness” to house host O/S and Python/TensorFlow framework along with the HLS C++ implementation

Test harness for TensorFlow + RTL validation on emulator

- Replace original 9 stages of the CNN algorithm with C++ coupled to RTL using transactors
- Validates synthesized RTL ML core in the context of original TensorFlow framework
- Provides convenient platform for power/performance analysis of the ML core itself



Docker host receptacle daemon

Docker Ubuntu 18.04 container

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

XlAcChannel

XlAcChannel

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

output tensor o9

9-stage breakout

Emulator/FPGA proto

Transactors

ML Accelerator

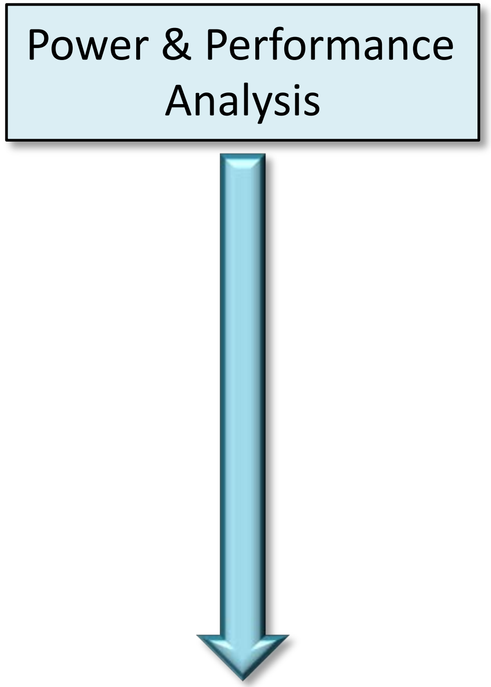
Interconnect

Memory

- C++ blocks themselves become drivers to transactors (BFMs) running in the emulator
- Cross-process TLM based `XlAcChannelTranactors` couple the TensorFlow and HLS C++ remote client process with the co-model host process and the emulator via the TLM fabric

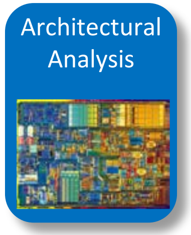
Progression of Verification

- IP Block Verification at RTL
- **Earliest SW Enablement (SoC context)**
- IP Block Validation Leveraging hybrid
  - Using a SW enabled flow for power and performance
- Full SoC Verification & validation
  - Focus on power & performance analysis



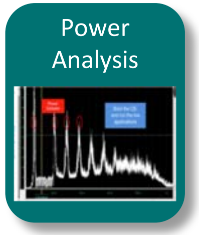
Power & Performance Analysis

What is SW Enabled Verification?



Architectural Analysis

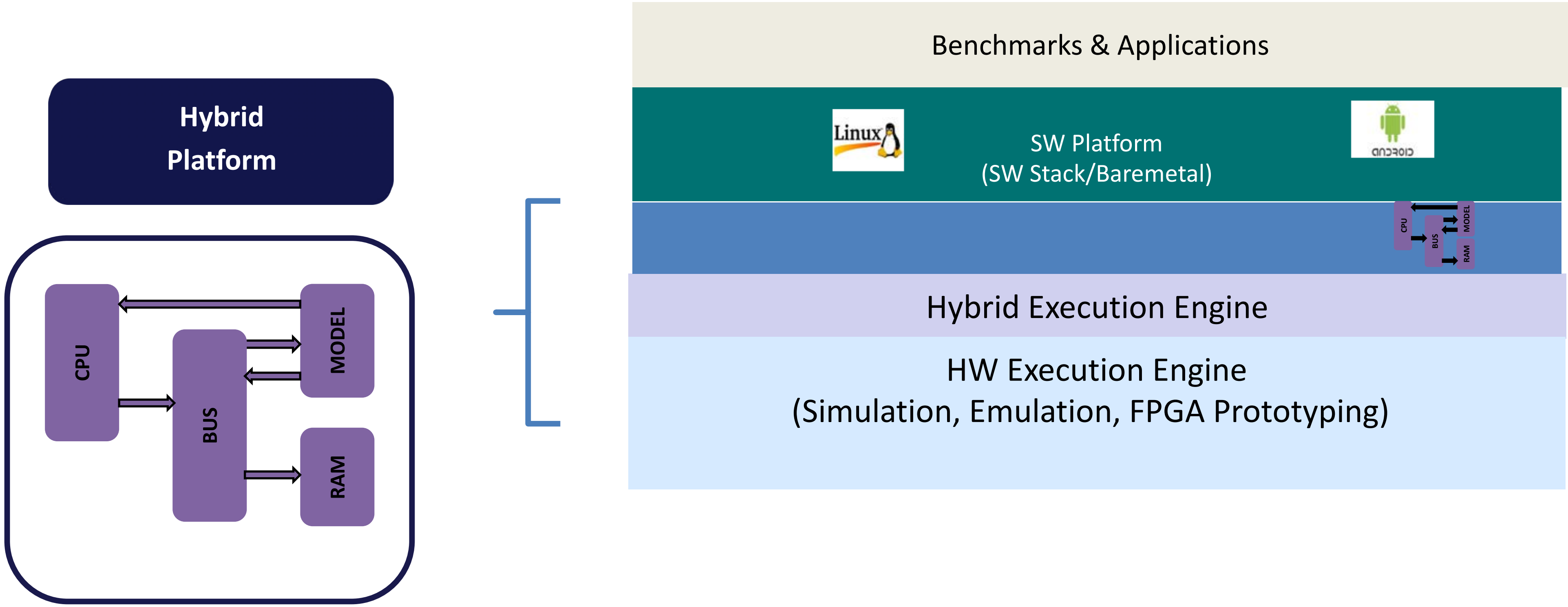
Performance Analysis



Power Analysis

SoC Validation

SW Development



Benchmarks & Applications

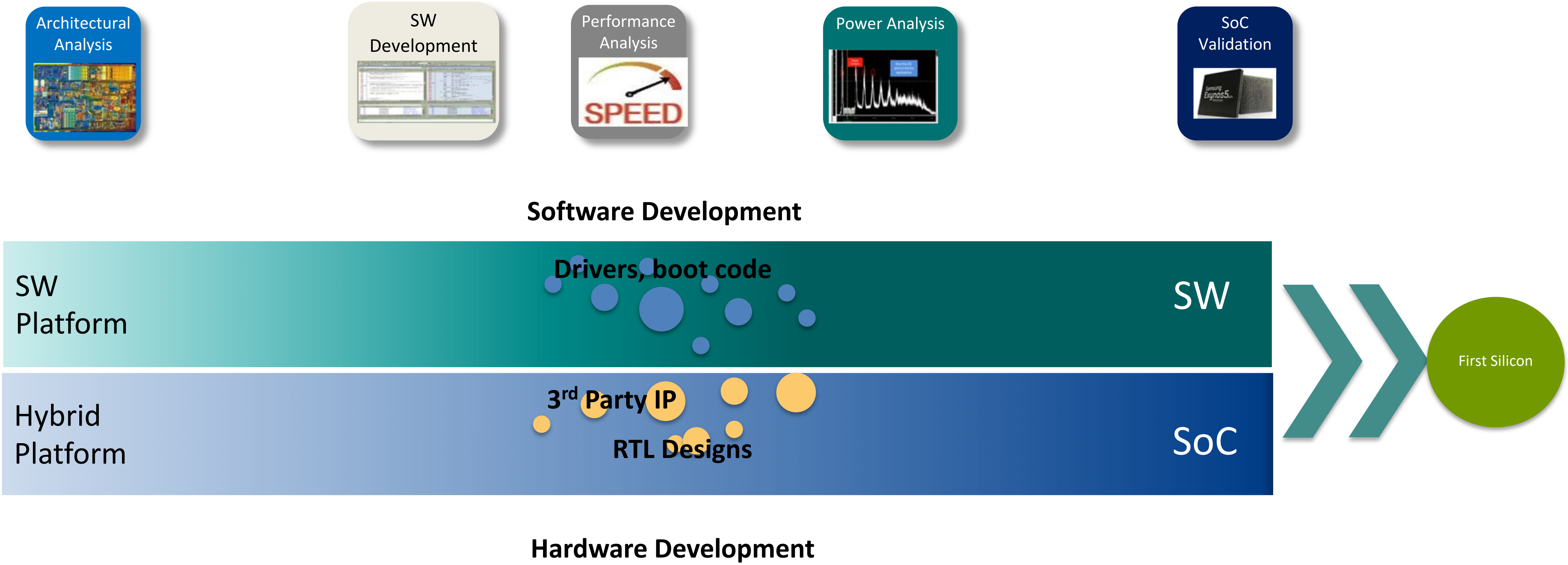
SW Platform
(SW Stack/Baremetal)

Hybrid Execution Engine

HW Execution Engine
(Simulation, Emulation, FPGA Prototyping)

Hybrid Platform

Hybrid Verification in a SW Enabled flow



Architectural Analysis

SW Development

Performance Analysis

Power Analysis

SoC Validation

Software Development

SW Platform

Drivers, boot code

SW

Hybrid Platform

3rd Party IP

RTL Designs

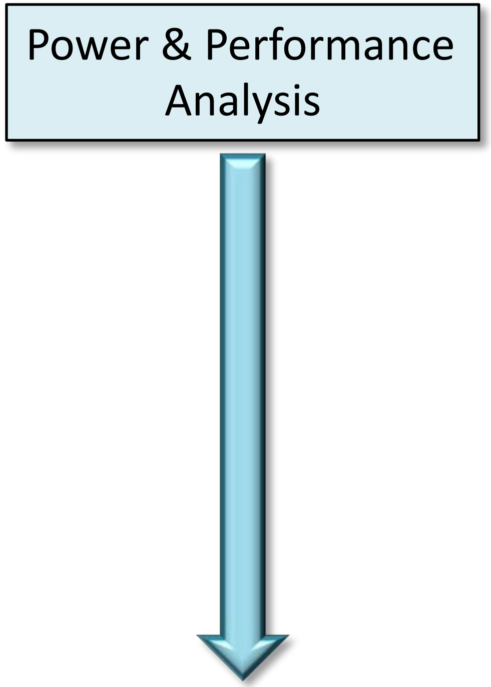
SoC

First Silicon

Hardware Development

Progression of Verification

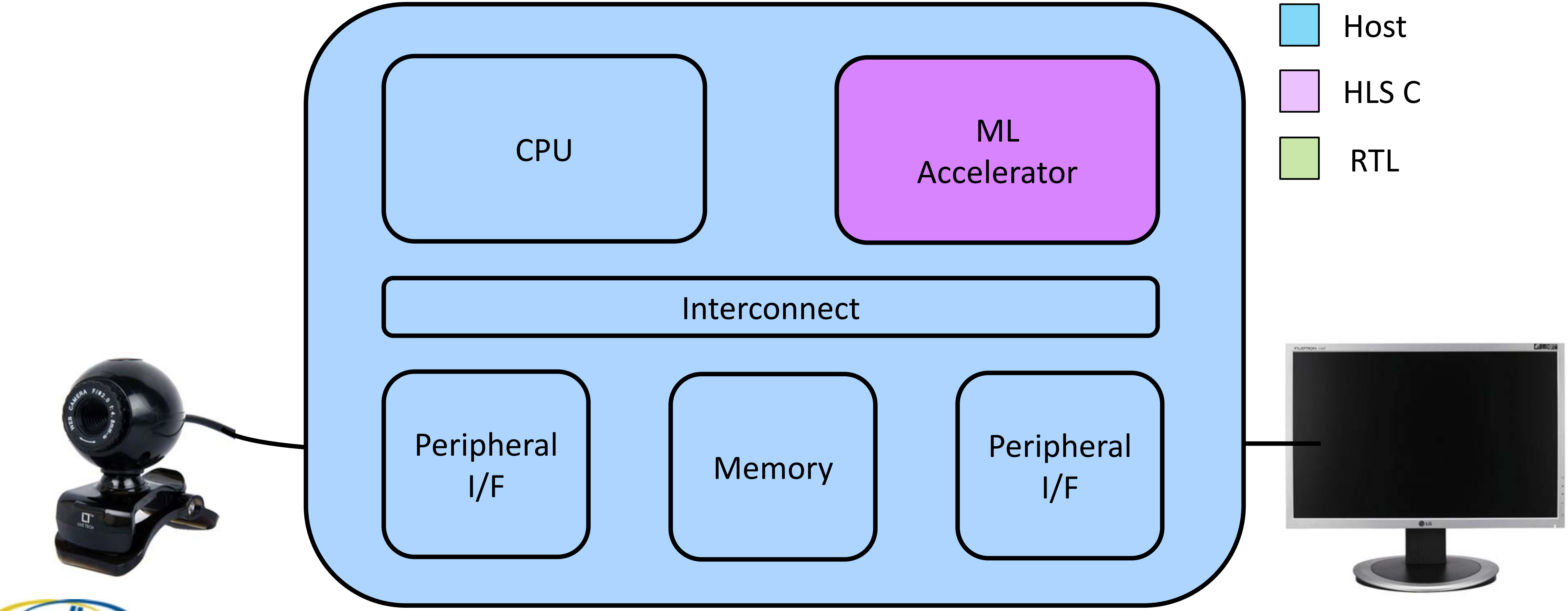
- IP Block Verification at RTL
- Earliest SW Enablement (SoC context)
- **IP Block Validation Leveraging hybrid**
  - Using a SW enabled flow for power and performance
- Full SoC Verification & validation
  - Focus on power & performance analysis



Power & Performance Analysis

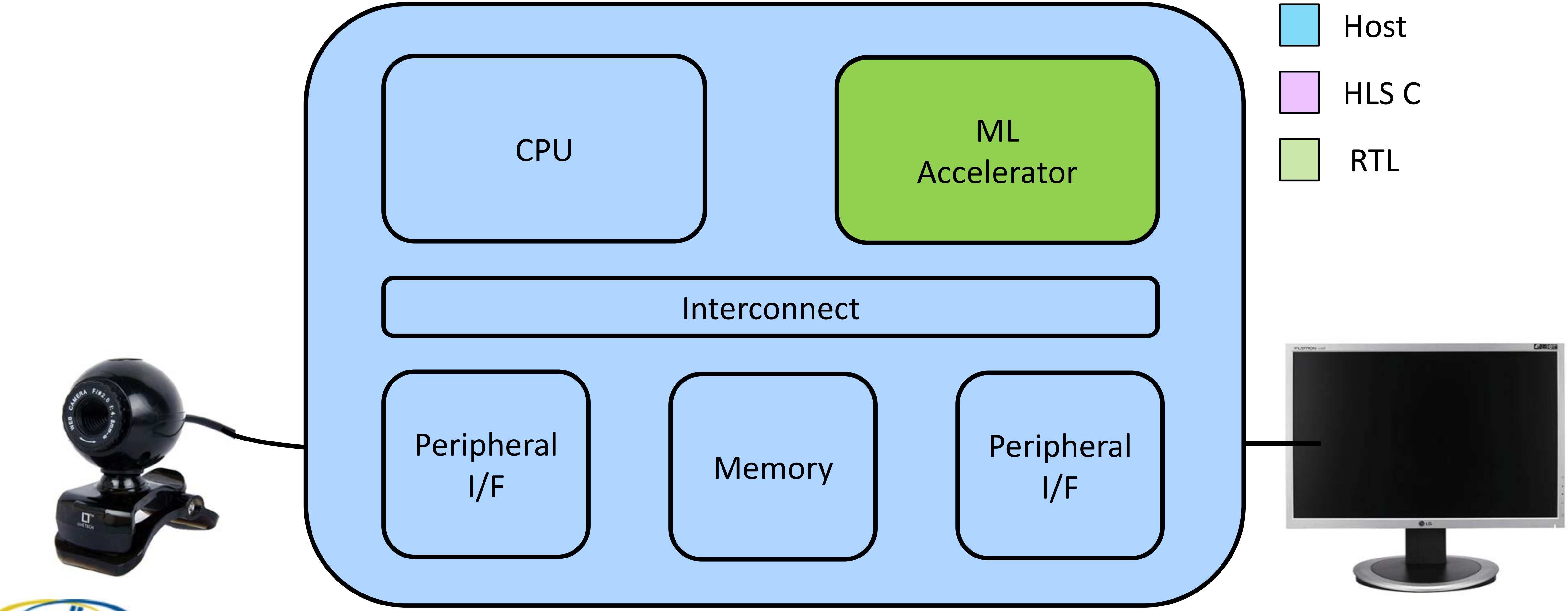
Hybrid Enables Mixing Abstractions

Flexibility in Verifying, Analyzing HW & Enabling SW



Hybrid Enables Mixing Abstractions

Flexibility in Verifying, Analyzing HW & Enabling SW

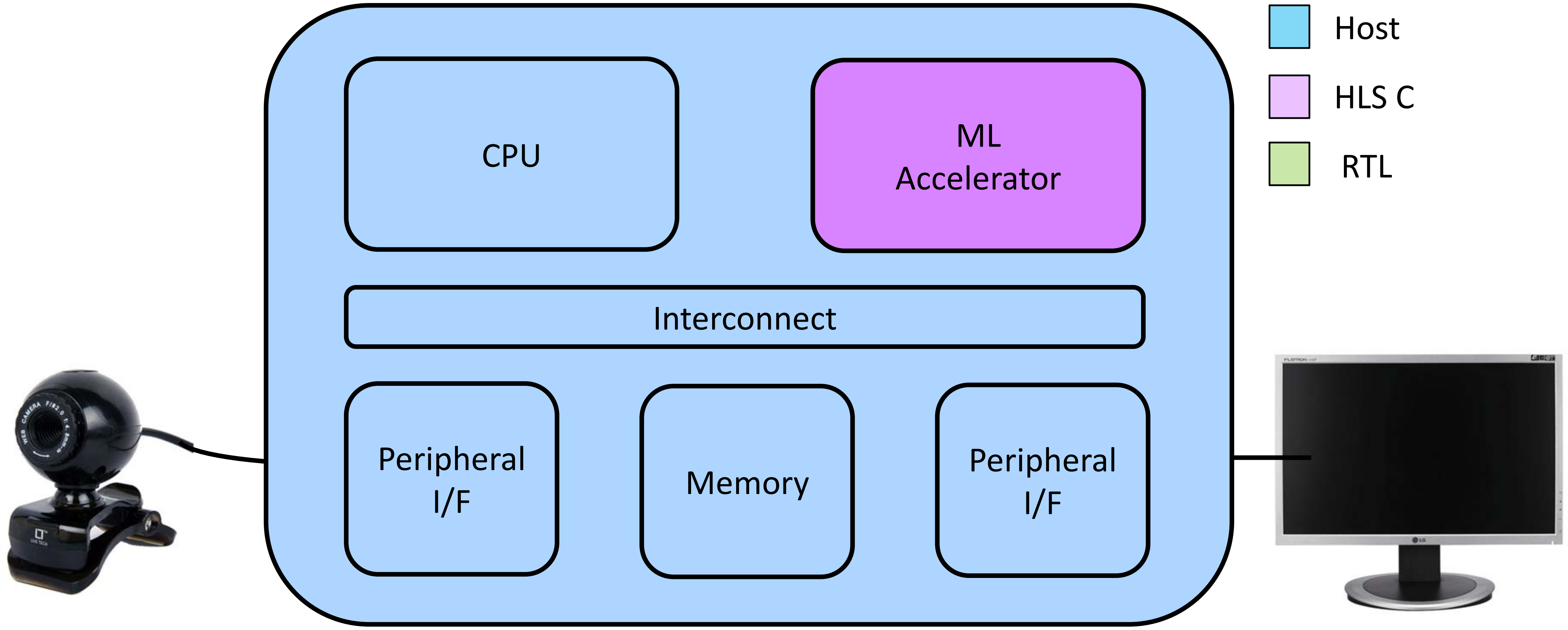


🟦 Host

🟪 HLS C

🟩 RTL

Hybrid Platform-HLS C



CPU

ML Accelerator

Interconnect

Peripheral I/F

Memory

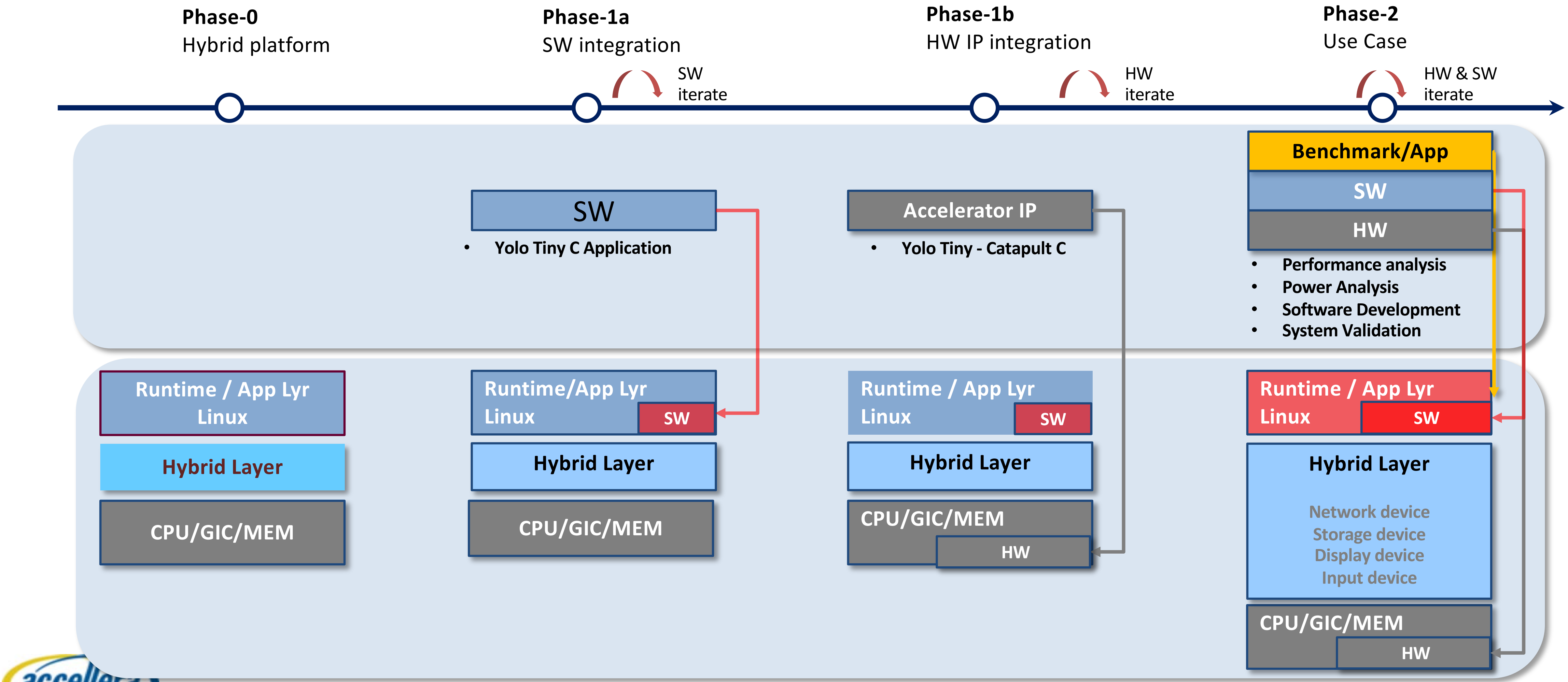
Peripheral I/F

🟦 Host

🟪 HLS C

🟩 RTL

IP Integration- HLS C



Phase-0
Hybrid platform

Phase-1a
SW integration

Phase-1b
HW IP integration

Phase-2
Use Case

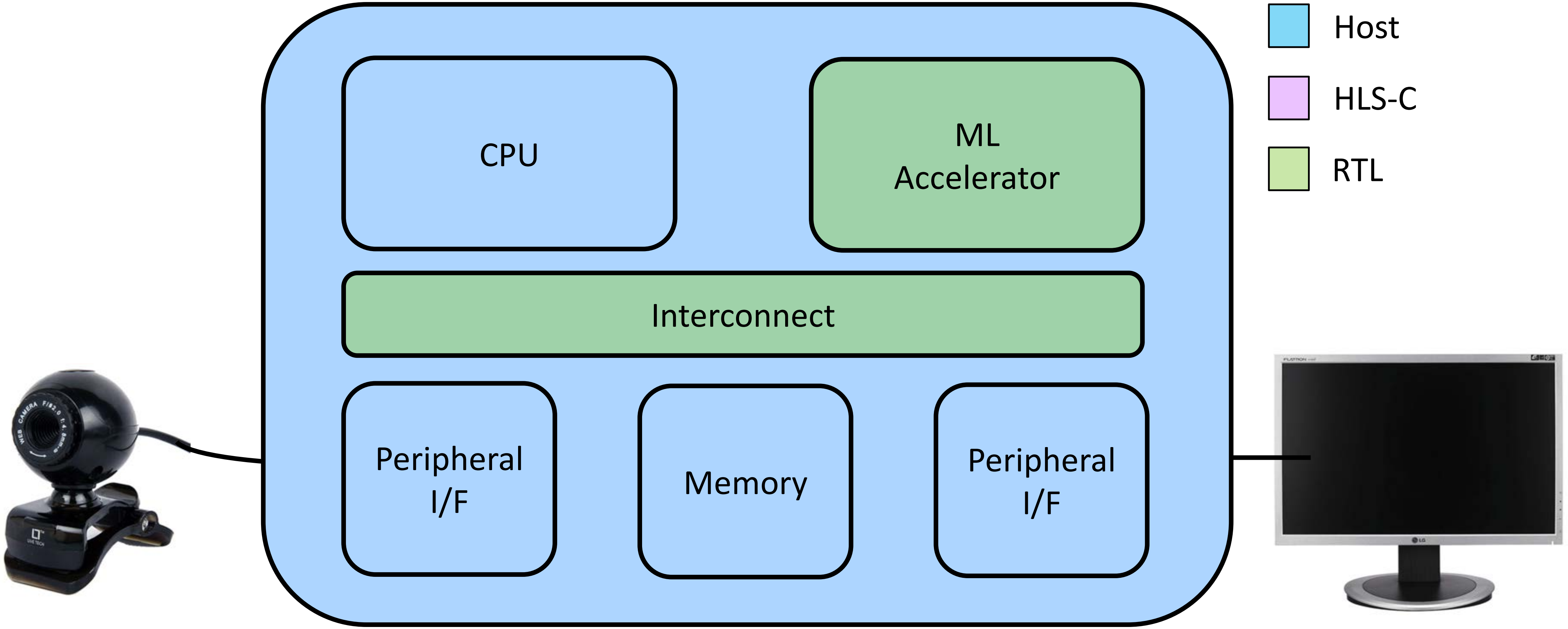
SW iterate

HW iterate

HW & SW iterate

- Yolo Tiny C Application
- Yolo Tiny - Catapult C
- Performance analysis
- Power Analysis
- Software Development
- System Validation

HLS-C is Synthesized to RTL



CPU

ML Accelerator

Interconnect

Peripheral I/F

Memory

Peripheral I/F

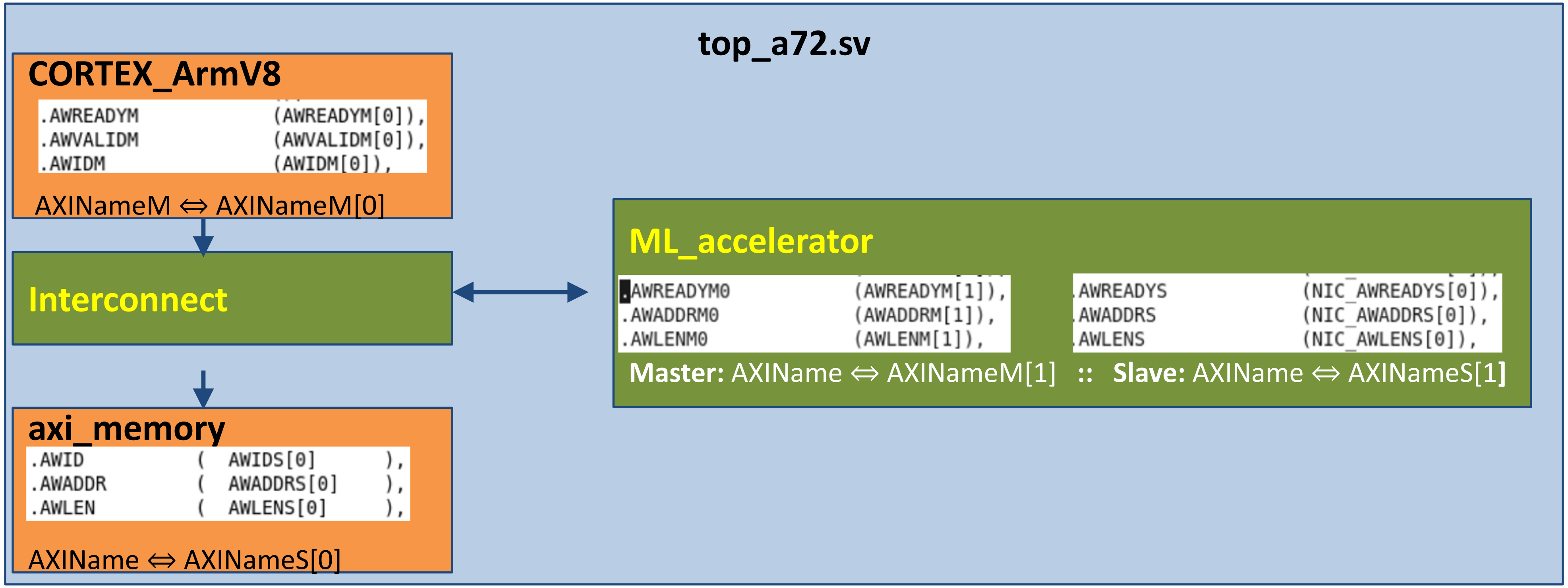
🟦 Host

🟪 HLS-C

🟩 RTL

Insertion of RTL

- Connect Tiny Yolo RTL to Interconnect
  - master port to <AXIName>M[1], to drive memory transactions
  - slave port to NIC_<AXINAME>S[0] to accept CPU transactions.



top_a72.sv

CORTEX_ArmV8

```
.AWREADYM            (AWREADYM[0]),
.AWVALIDM            (AWVALIDM[0]),
.AWIDM               (AWIDM[0]),
```

AXINameM ⇔ AXINameM[0]

Interconnect

axi_memory

```
.AWID            (  AWIDS[0]      ),
.AWADDR          (  AWADDRS[0]    ),
.AWLEN           (  AWLENS[0]     ),
```

AXIName ⇔ AXINameS[0]

ML_accelerator

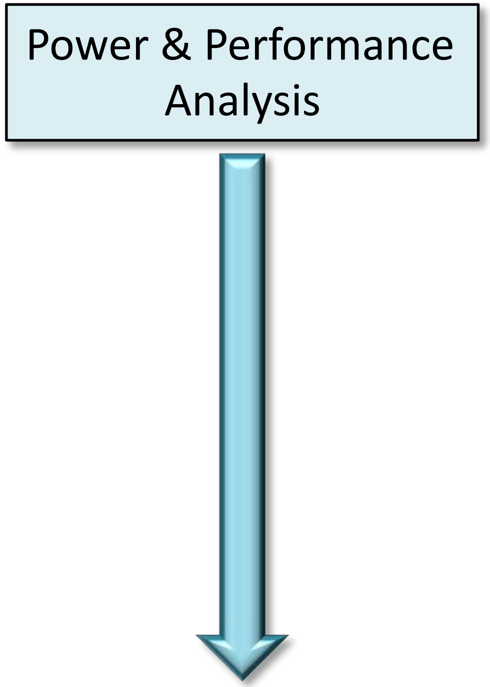
```
.AWREADYM0              (AWREADYM[1]),
.AWADDRM0               (AWADDRM[1]),
.AWLENM0                (AWLENM[1]),
```

```
.AWREADYS               (NIC_AWREADYS[0]),
.AWADDRS                (NIC_AWADDRS[0]),
.AWLENS                 (NIC_AWLENS[0]),
```

Master: AXIName ⇔ AXINameM[1] :: **Slave:** AXIName ⇔ AXINameS[1]

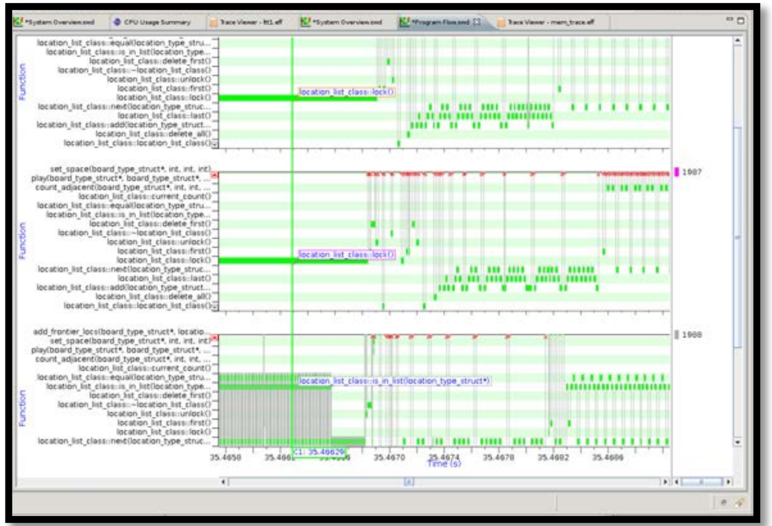
Progression of Verification

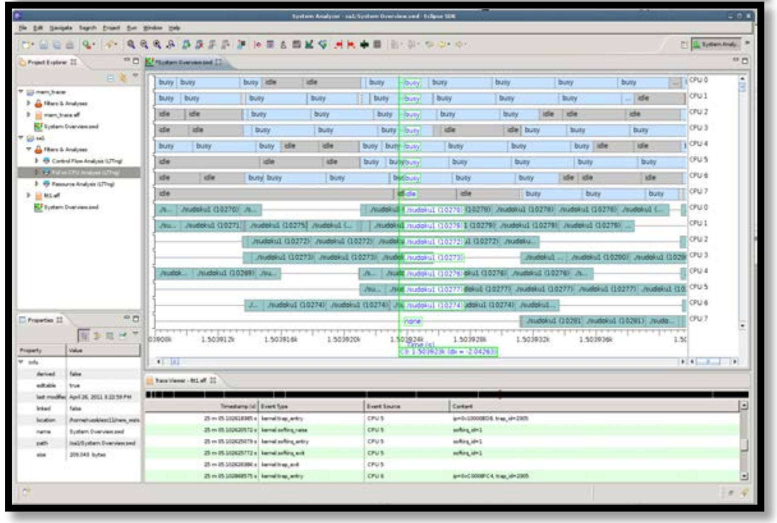
- IP Block Verification at RTL
- Earliest SW Enablement (SoC context)
- **IP Block Validation Leveraging hybrid**
  - **Using a SW enabled flow for power and performance**
- Full SoC Verification & validation
  - Focus on power & performance analysis


Power & Performance Analysis

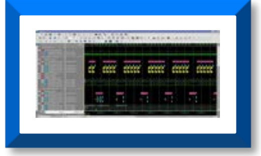
System Analyzer

- Collects data from embedded processors during runs
  - Simulation
  - Emulation
  - FPGA prototype
- Collects data from hardware monitors
  - User defined, SLA monitors
- Post processing resports and views
  - Standard reporting for common buses and interfaces
    - Bus utilization
    - Communication latencies
    - Bus traffic correlated with software activity
  - Transaction tracing
  - Facilities for user defined reporting
  - Data stored in SQL database





SIMULATION

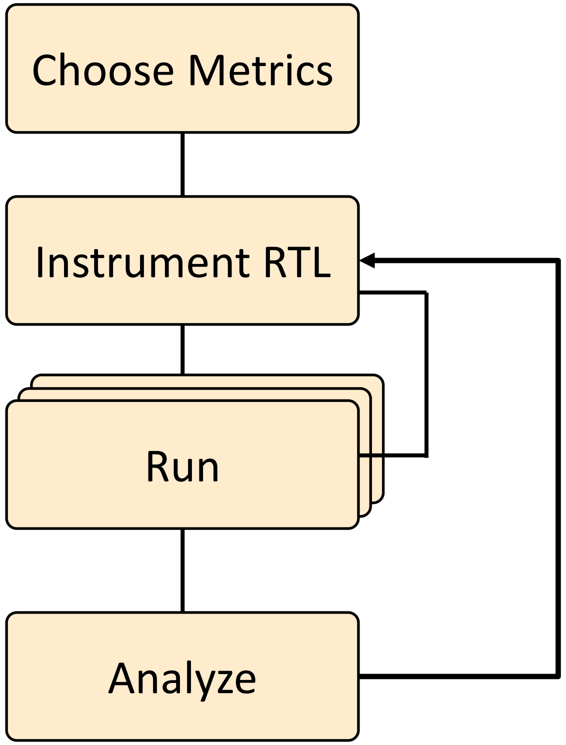


EMULATION

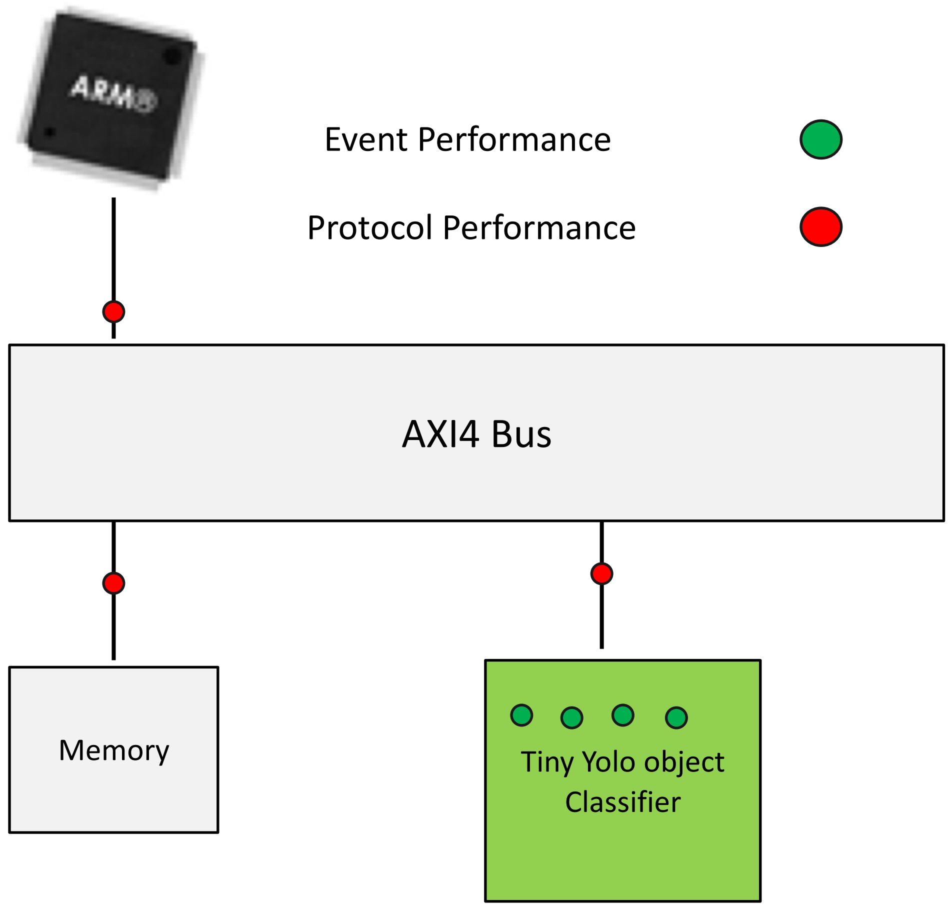
FPGA PROTO

Capturing IP Performance – SW Enabled Methodology

- Add additional interesting metrics
  - Identify data paths conducive to obtaining metrics
  - Event probes, counters, triggers, trace buffers, protocol-specific bus monitors, etc.
  - Choose and place along identified paths; probe other points to capture additional details
- Apply stimulus
  - Run applications and benchmarks
  - Capture only during performance measurement window of interest via triggers
- Analyze
  - Establish pass/fail thresholds for
  - Filter results and track progress: pass/fail checks, comparisons to previous results, etc.
  - Manage results: across regressions, test categories, design changes, configuration settings



IP Integration w/Performance Validation

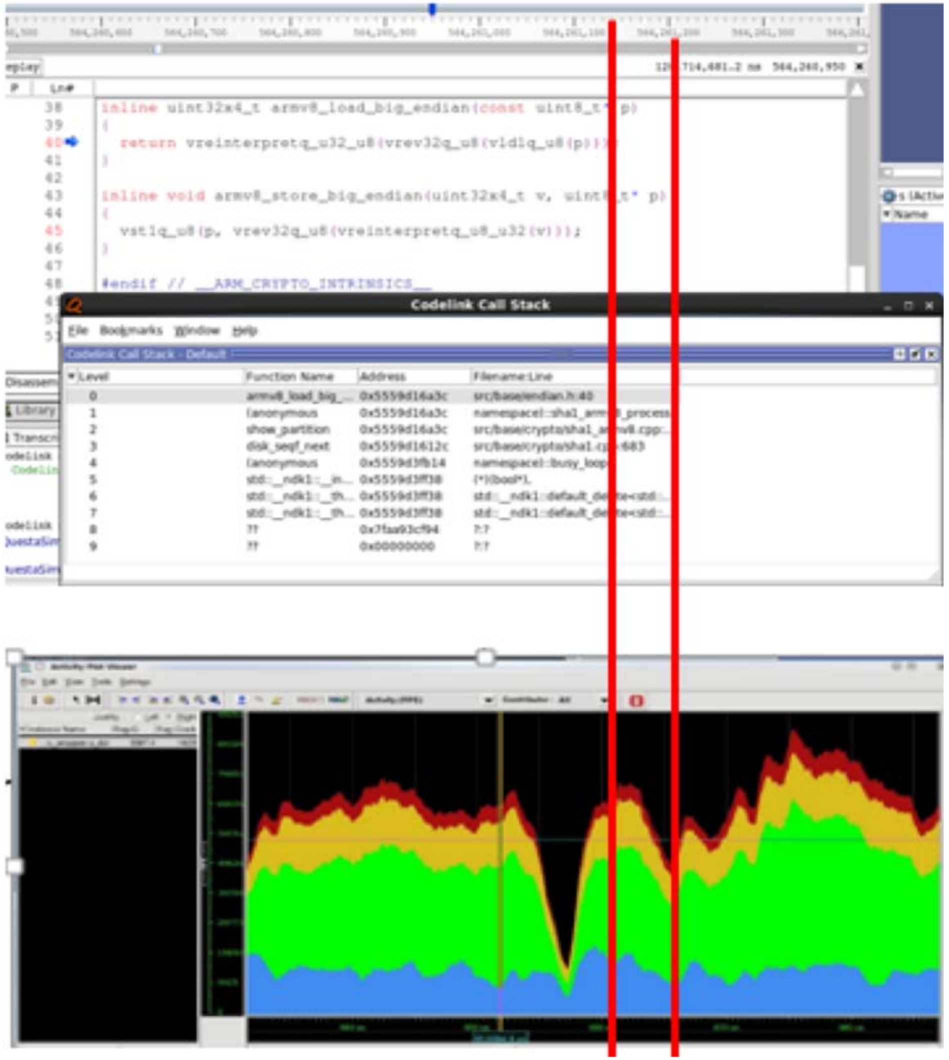


- Measure event-based metrics
  - Bus utilization
  - Bus wait/stall statistics
- Get full analysis of standard protocols
  - Transaction latency over time
  - Cache-state tracking
  - Duration by transaction type
  - Associate snoops and memory accesses to original request
  - Drill-down to individual transactions as needed
- Monitor User-defined events
  - PMU monitoring, FSMs, FIFO levels, other design points not requiring protocol knowledge
- Correlation between HW & Real world SW

Power Analysis

Ip Integration w/Power Analysis

- Identify design hotspots from Yolo Tiny RTL
- Visually drill down into design hierarchies of concern
- Identify mistakenly active power domains (Power estimation + UPF)
- Correlation of data between activity plot and Yolo Tiny C application running on Linux



```
inline uint32x4_t armv8_load_big_endian(const uint8_t* p)
{
  return vreinterpretq_u32_u8(vrev32q_u8(vld1q_u8(p)));
}

inline void armv8_store_big_endian(uint32x4_t v, uint8_t* p)
{
  vst1q_u8(p, vrev32q_u8(vreinterpretq_u8_u32(v)));
}

#endif // __ARM_CRYPTO_INTRINSICS__
```

Codelink Call Stack

File Bookmarks Window Help

Codelink Call Stack - Default

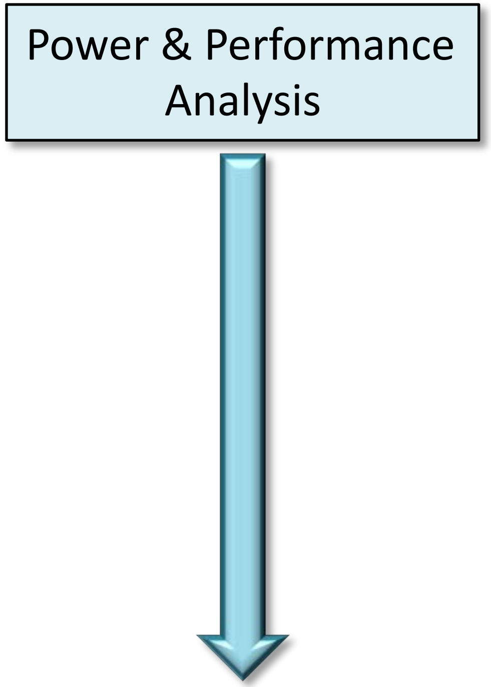
Level	Function Name	Address	Filename:Line
0	armv8_load_big_...	0x5559d16a3c	src/base/endian.h:40
1	(anonymous	0x5559d16a3c	namespace)::sha1_armv8_process
2	show_partition	0x5559d16a3c	src/base/crypto/sha1_armv8.cpp:...
3	disk_seqf_next	0x5559d1612c	src/base/crypto/sha1.cpp:683
4	(anonymous	0x5559d3fb14	namespace)::busy_loop
5	std::__ndk1::__in...	0x5559d3ff38	(*)(bool*),
6	std::__ndk1::__th...	0x5559d3ff38	std::__ndk1::default_delete<std:...
7	std::__ndk1::__th...	0x5559d3ff38	std::__ndk1::default_delete<std:...
8	??	0x7faa93cf94	?:?
9	??	0x00000000	?:?

Hybrid-Enabled Block-Level Validation Summary

- Applications and benchmarks optimize HW and SW together
  - Performance Analysis
  - Power Analysis
- Platform evolves to deliver optimal solution
- Enabling a software-driven design methodology
- Bridge the discontinuity between different levels of abstraction

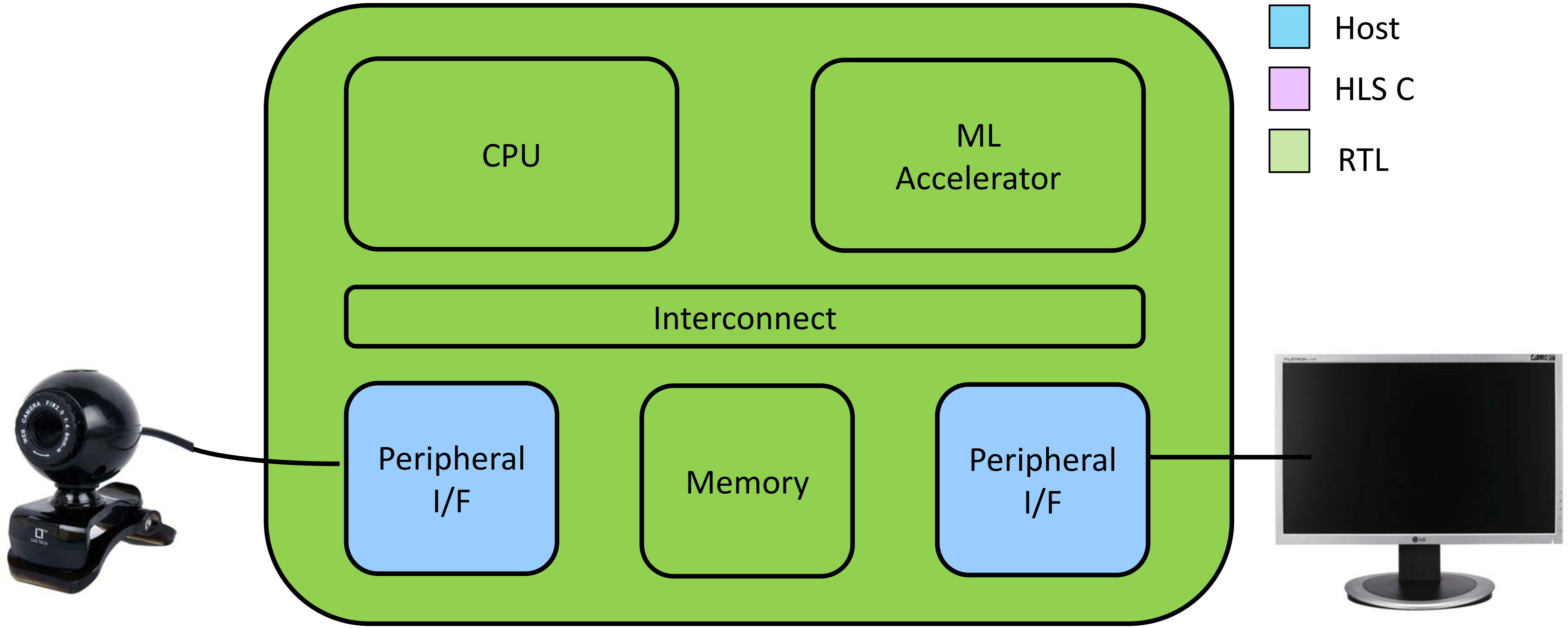
Progression of Verification

- IP Block Verification at RTL
- Earliest SW Enablement (SoC context)
- IP Block Validation Leveraging hybrid
  - Using a SW enabled flow for power and performance
- **Full SoC Verification & validation**
  - **Focus on power & performance analysis**



Power & Performance Analysis

System Integration with full RTL



CPU

ML Accelerator

Interconnect

Peripheral I/F

Memory

Peripheral I/F

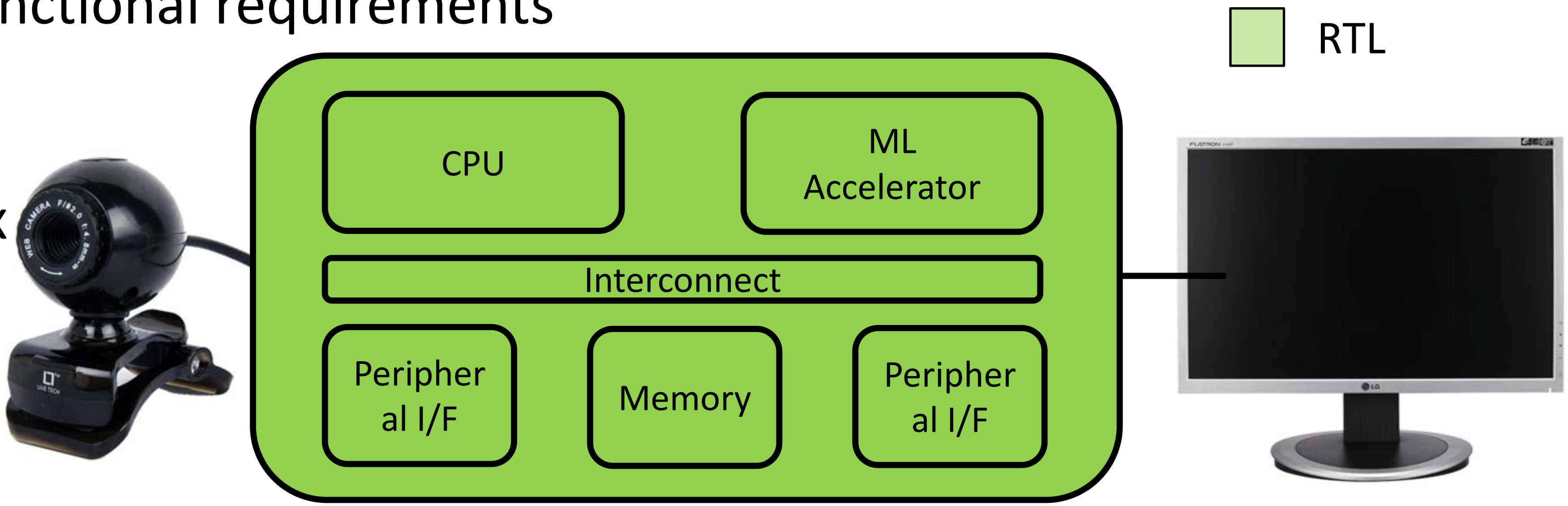
🟦 Host

🟪 HLS C

🟩 RTL

SoC Integration

- Full RTL
  - Although some users are moving to hybrid (almost) everything
  - Leading edge of largest chips being designed
- Objectives:
  - Ensure the fully integrated SoC functions properly, at least through initialization-reset, usually OS boot
- Analysis of non-functional requirements
  - Power
  - Performance
- Verification of DFx Instrumentation

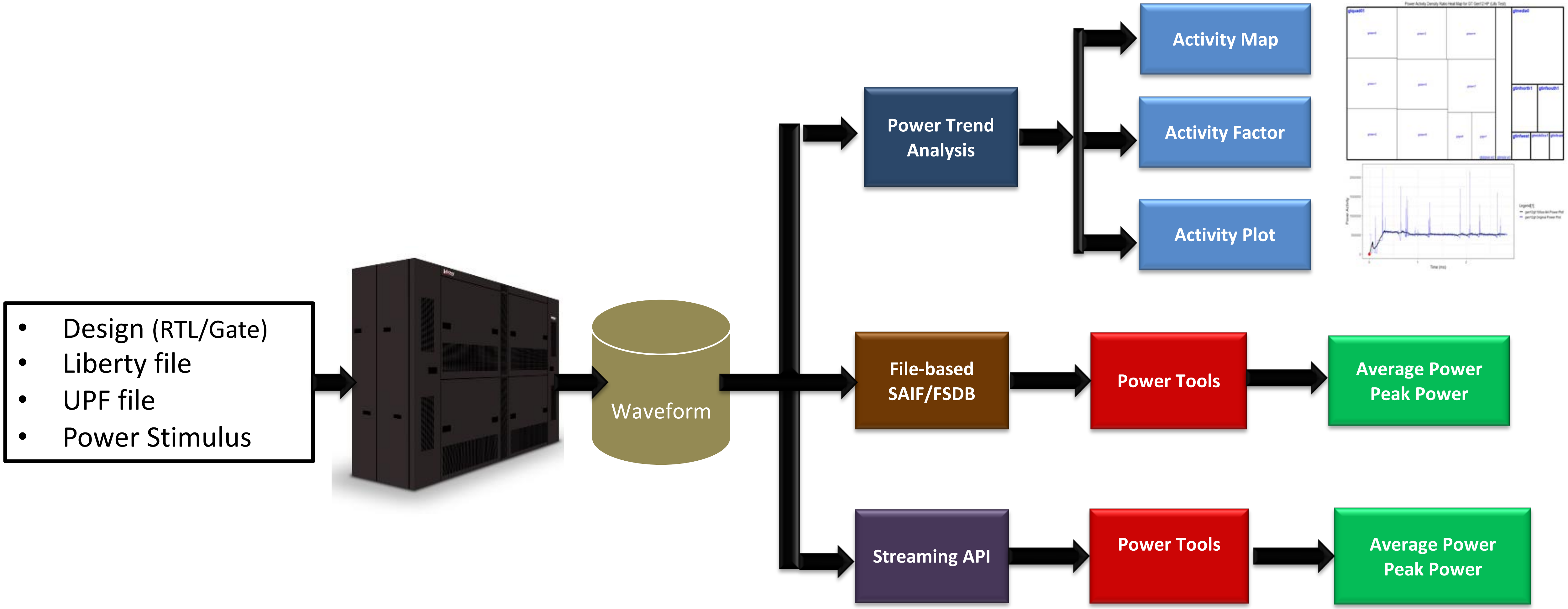


Key user requirements for power analysis

1	**Power w/ Real-world Scenarios/SW:** Early power trend analysis at full SoC while running real world user scenarios and software
2	**Accurate RTL/GL average and peak power:** Generate accurate average and peak power numbers in target application environment with RTL and gate level netlist
3	**Power Optimization:** Identify potential power optimization opportunities early in design cycle for architectural tradeoffs
4	**Low-Power Control via HW/SW:** UPF-based low power verification with power controls coming from SW applications

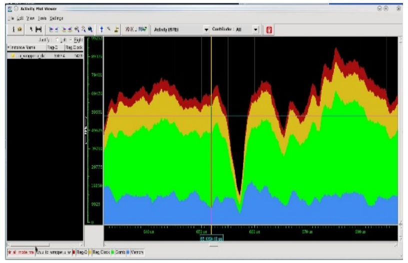
Typical power app

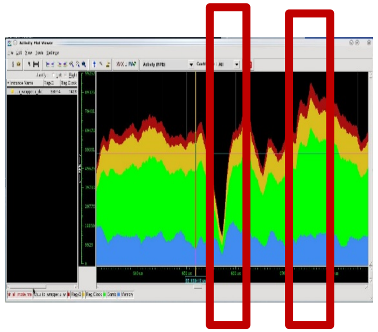
How it works

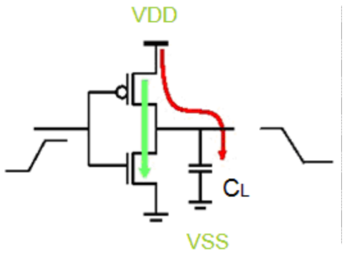


- Design (RTL/Gate)
- Liberty file
- UPF file
- Power Stimulus

Complete Power Solution

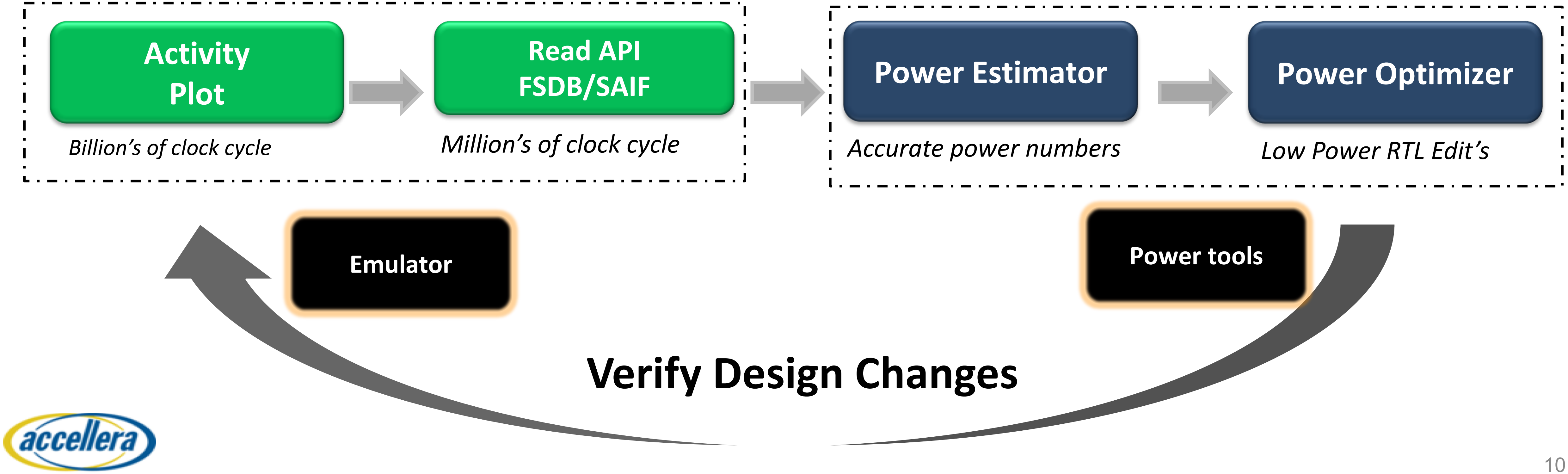






Dynamic power is only when switching

```
always@(posedge clk)
   q<=d;
        ⬇
always@(posedge clk)
  if(en)
  q<=d;
```



Billion’s of clock cycle

Million’s of clock cycle

Accurate power numbers

Low Power RTL Edit’s

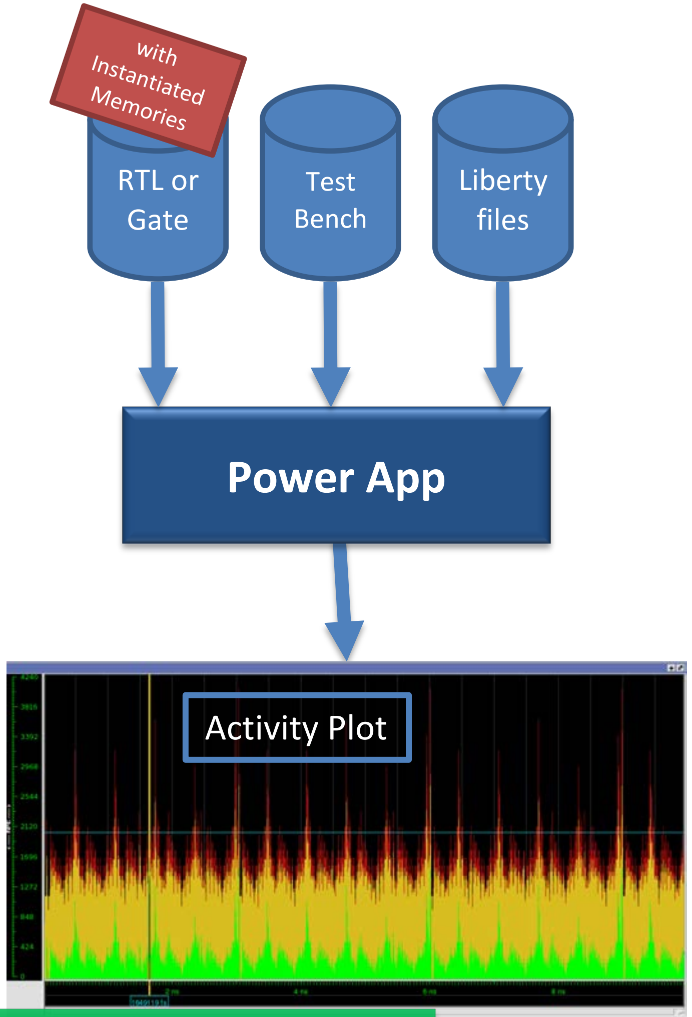
Verify Design Changes

Early Power Trend Analysis

Activity Plot

- Activity Plot
  - Generate very fast power profile for logic and memory
  - Very high correlation with actual power graphs
  - Identify power peaks, valleys and di/dt
  - DvFS what if analysis
  - Verify power domain ON/OFF via UPF

- Enabling technology with emulation
  - Capacity to handle large SoC
  - 100% visibility of all the design signals
  - Fast waveform upload
  - Accurate modeling of power components @ RTL (clock gating, multi-bit flop, voltage scaling, read liberty files)
  - Top down GUI based power analysis

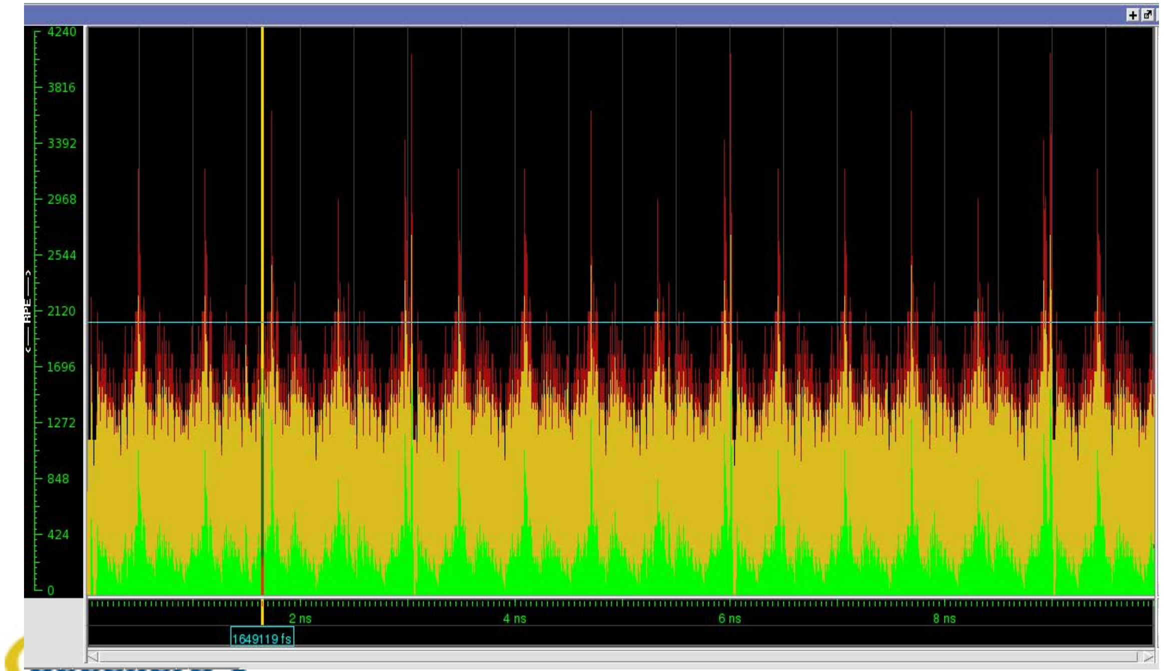


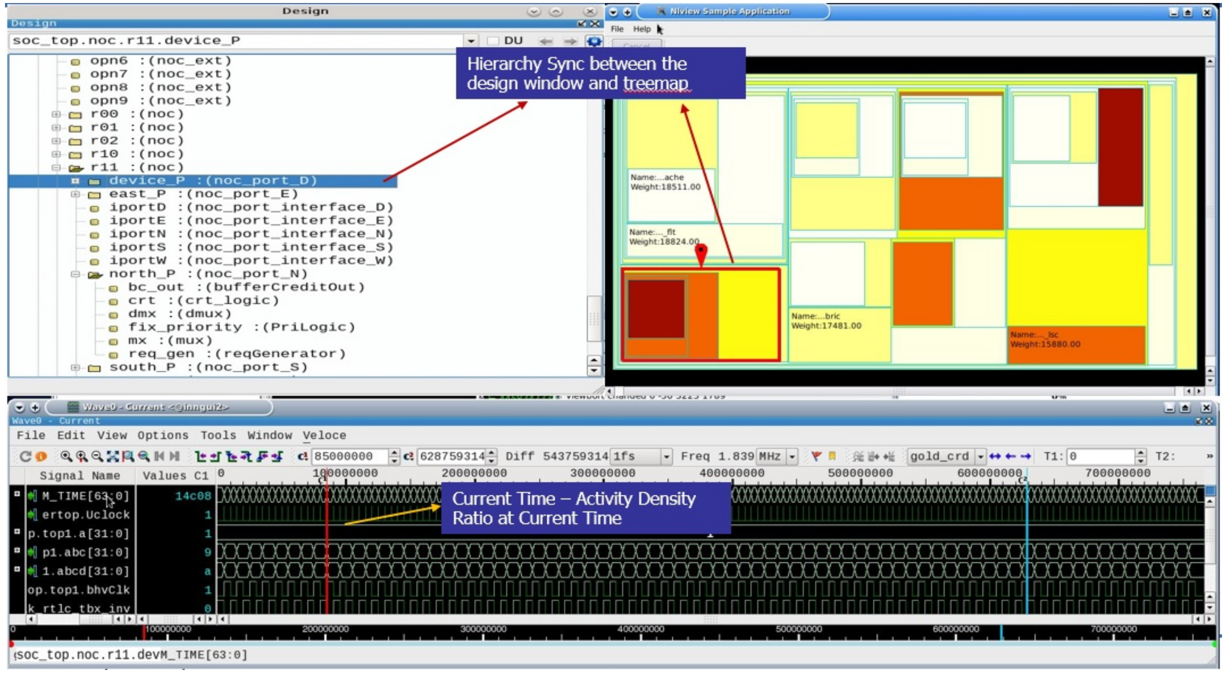
Enables very fast Power profiling at full-SoC while running very long customer scenarios

Hotspot Analysis

Activity Map

- Identify design hotspots for representative scenarios
- Visual drill down into design hierarchies of concern
- Identify mistakenly active power domains *(Power estimation + UPF)*
- Time synced with activity plot and waveform




Design
DU
File Help
Wave0 - Current
File Edit View Options Tools Window Veloce
Signal Name
Values C1
Name:...ache Weight:18511.00
Name:..._flt Weight:18824.00
Name:...bric Weight:17481.00
Name:..._lsc Weight:15880.00

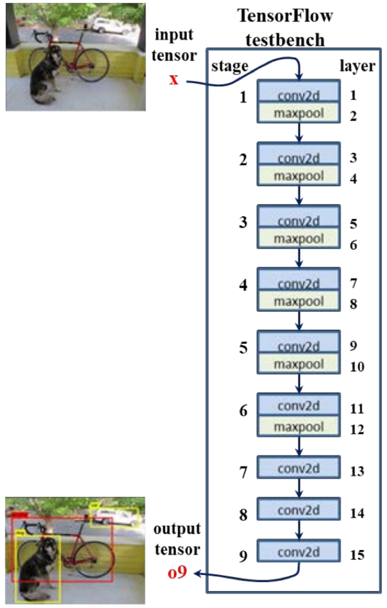
```
soc_top.noc.r11.device_P
      opn6 :(noc_ext)
      opn7 :(noc_ext)
      opn8 :(noc_ext)
      opn9 :(noc_ext)
    r00 :(noc)
    r01 :(noc)
    r02 :(noc)
    r10 :(noc)
    r11 :(noc)
      device_P :(noc_port_D)
      east_P :(noc_port_E)
        iportD :(noc_port_interface_D)
        iportE :(noc_port_interface_E)
        iportN :(noc_port_interface_N)
        iportS :(noc_port_interface_S)
        iportW :(noc_port_interface_W)
      north_P :(noc_port_N)
          bc_out :(bufferCreditOut)
          crt :(crt_logic)
          dmx :(dmux)
          fix_priority :(PriLogic)
          mx :(mux)
          req_gen :(reqGenerator)
      south_P :(noc_port_S)
```

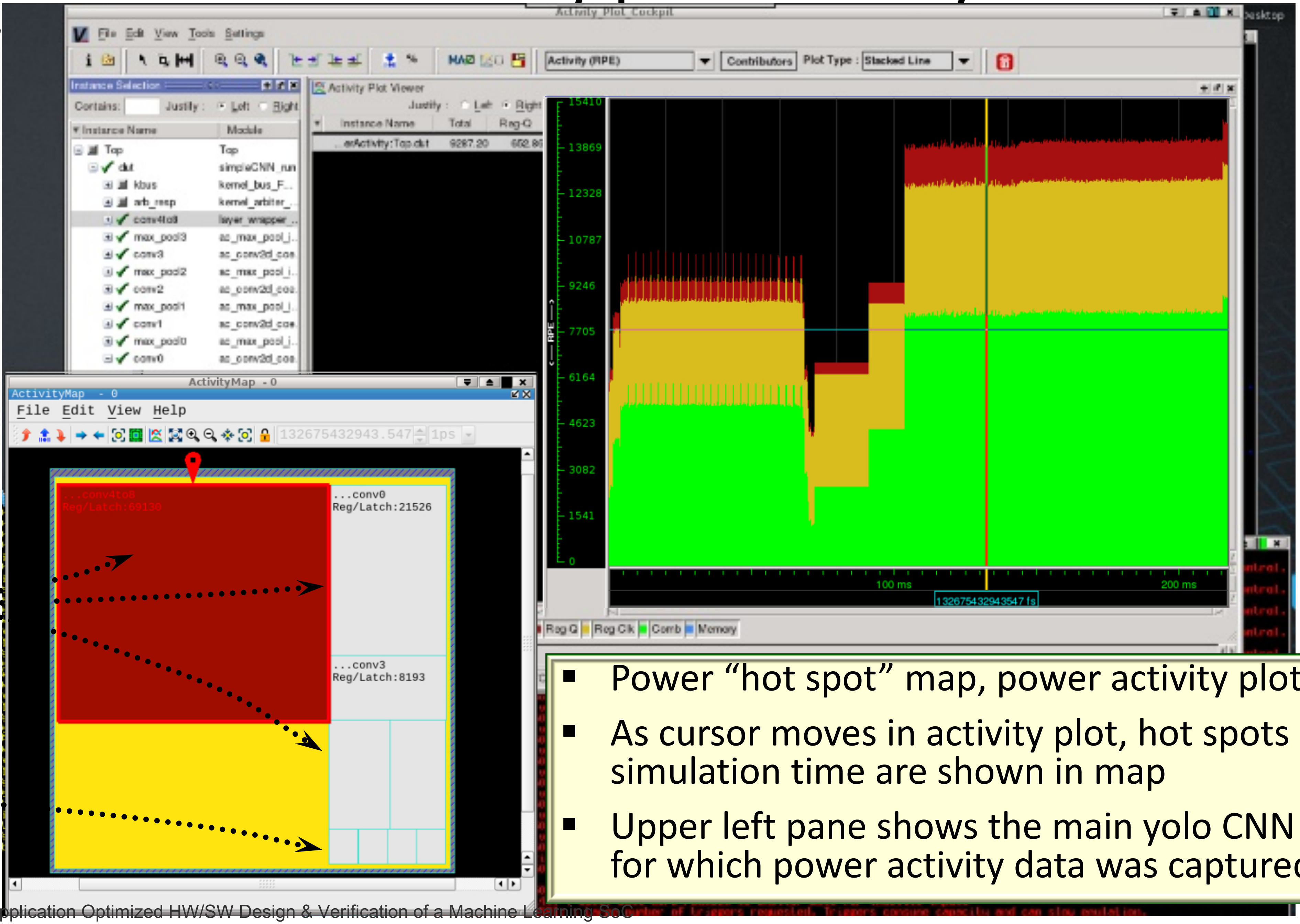
Hierarchy Sync between the design window and treemap

```
85000000  628759314  Diff 543759314 1fs  Freq 1.839 MHz  gold_crd  T1: 0  T2:
0  100000000  200000000  300000000  400000000  500000000  600000000  700000000
M_TIME[63:0]      14c08
ertop.Uclock      1
p.top1.a[31:0]    1
p1.abc[31:0]      9
1.abcd[31:0]      a
op.top1.bhvClk    1
k_rtlc_tbx_inv    0
soc_top.noc.r11.devM_TIME[63:0]
```

Current Time – Activity Density Ratio at Current Time

YoloTiny power analysis





Instance Name	Module
Top	Top
dut	simpleCNN_run
kbus	kernel_bus_F...
arb_resp	kernel_arbiter_...
conv4to8	layer_wrapper_...
max_pool3	ac_max_pool_i...
conv3	ac_conv2d_coe...
max_pool2	ac_max_pool_i...
conv2	ac_conv2d_coe...
max_pool1	ac_max_pool_i...
conv1	ac_conv2d_coe...
max_pool0	ac_max_pool_i...
conv0	ac_conv2d_coe...

Instance Name	Total	Reg-Q
...enActivity:Top.dut	9287.20	652.86

...conv4to8
Reg/Latch:69130

...conv0
Reg/Latch:21526

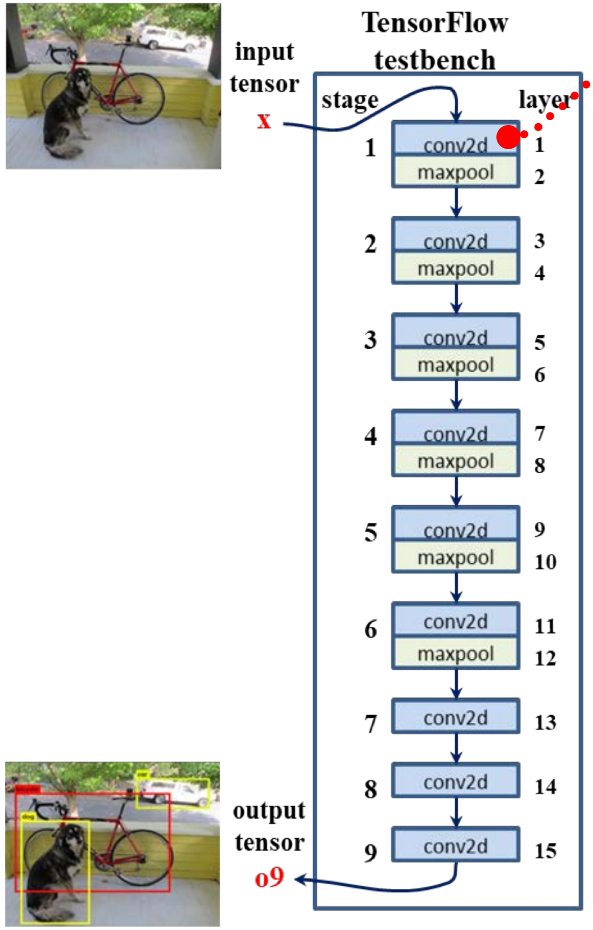
...conv3
Reg/Latch:8193

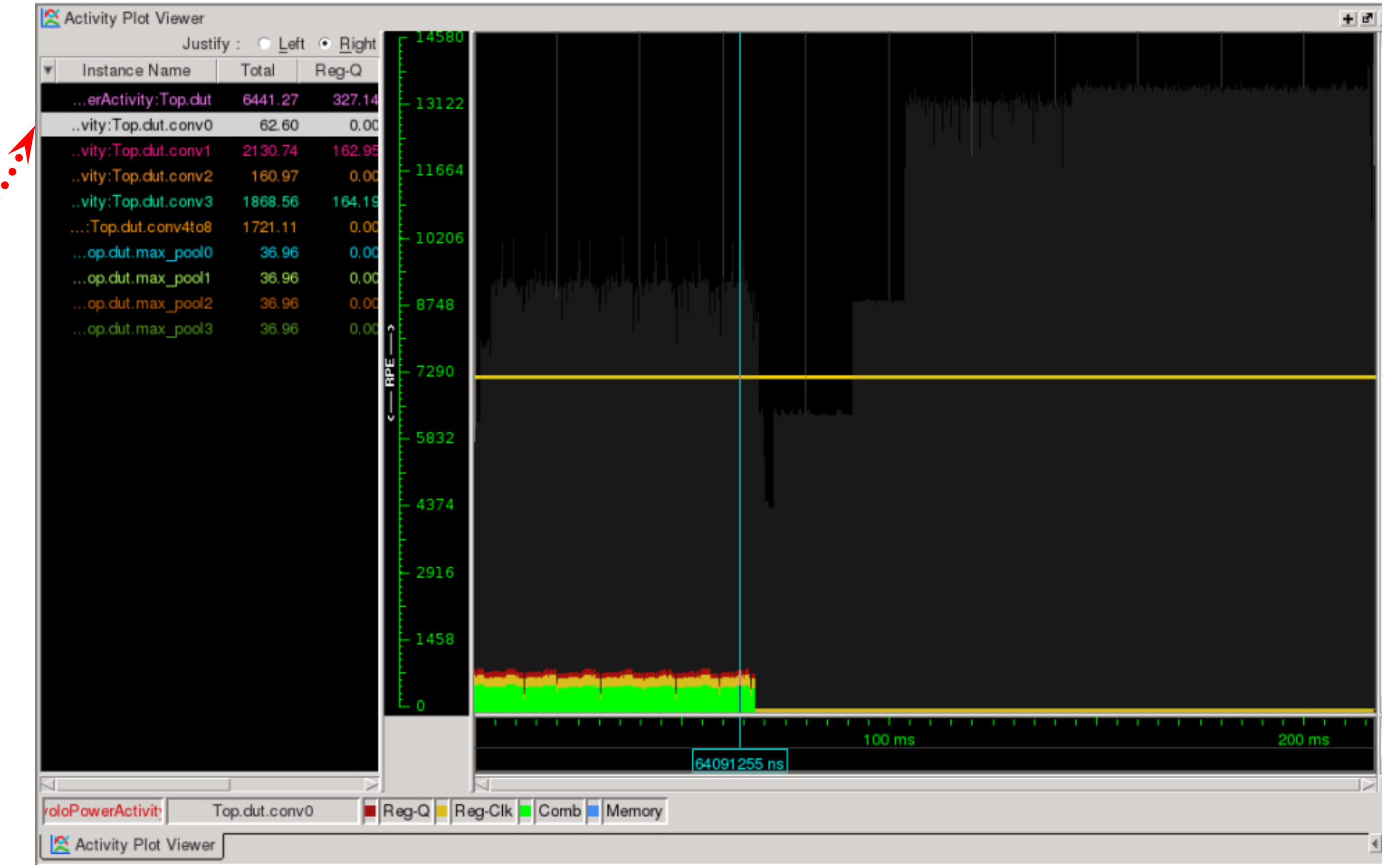
conv 0-8
layer "hot spot"
tiles

max_pool 0-3
layer "hot spot"
tiles

- Power "hot spot" map, power activity plot
- As cursor moves in activity plot, hot spots at that simulation time are shown in map
- Upper left pane shows the main yolo CNN layers for which power activity data was captured

YoloTiny Power Analysis

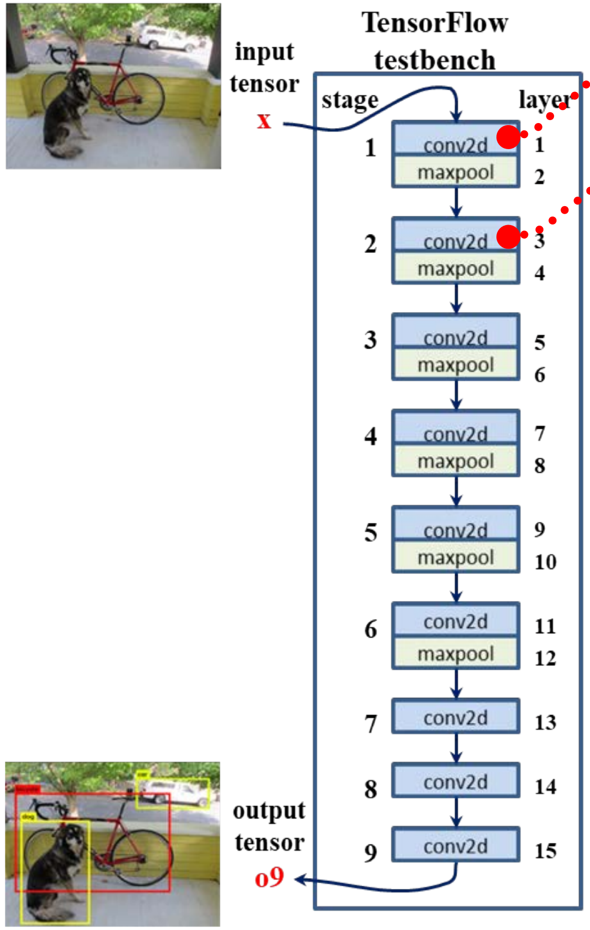


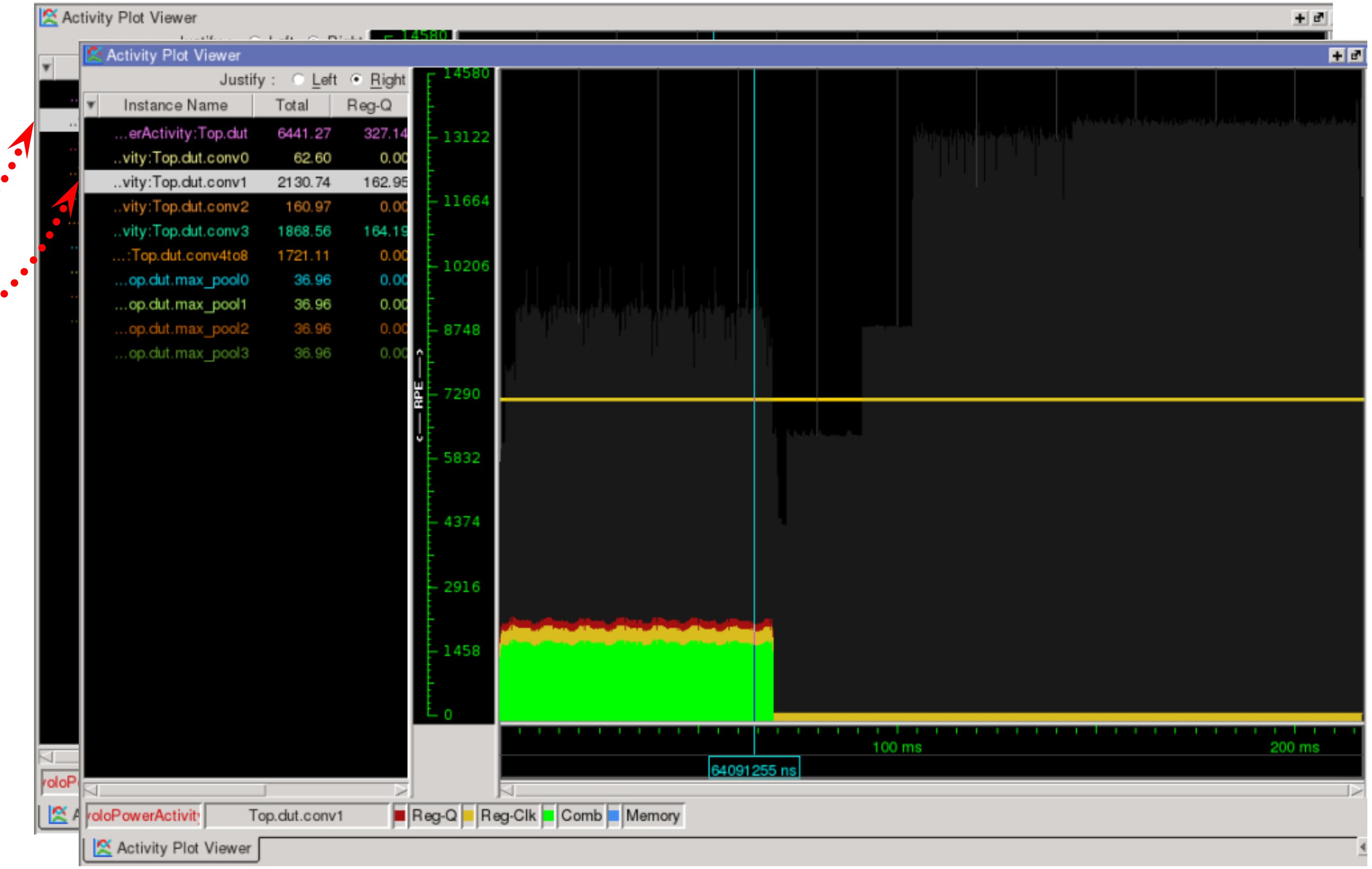


Instance Name	Total	Reg-Q
...erActivity:Top.dut	6441.27	327.14
..vity:Top.dut.conv0	62.60	0.00
..vity:Top.dut.conv1	2130.74	162.95
..vity:Top.dut.conv2	160.97	0.00
..vity:Top.dut.conv3	1868.56	164.19
...:Top.dut.conv4to8	1721.11	0.00
...op.dut.max_pool0	36.96	0.00
...op.dut.max_pool1	36.96	0.00
...op.dut.max_pool2	36.96	0.00
...op.dut.max_pool3	36.96	0.00

- Individual contributions are shown for highlighed modules in leftpane

YoloTiny Power Analysis

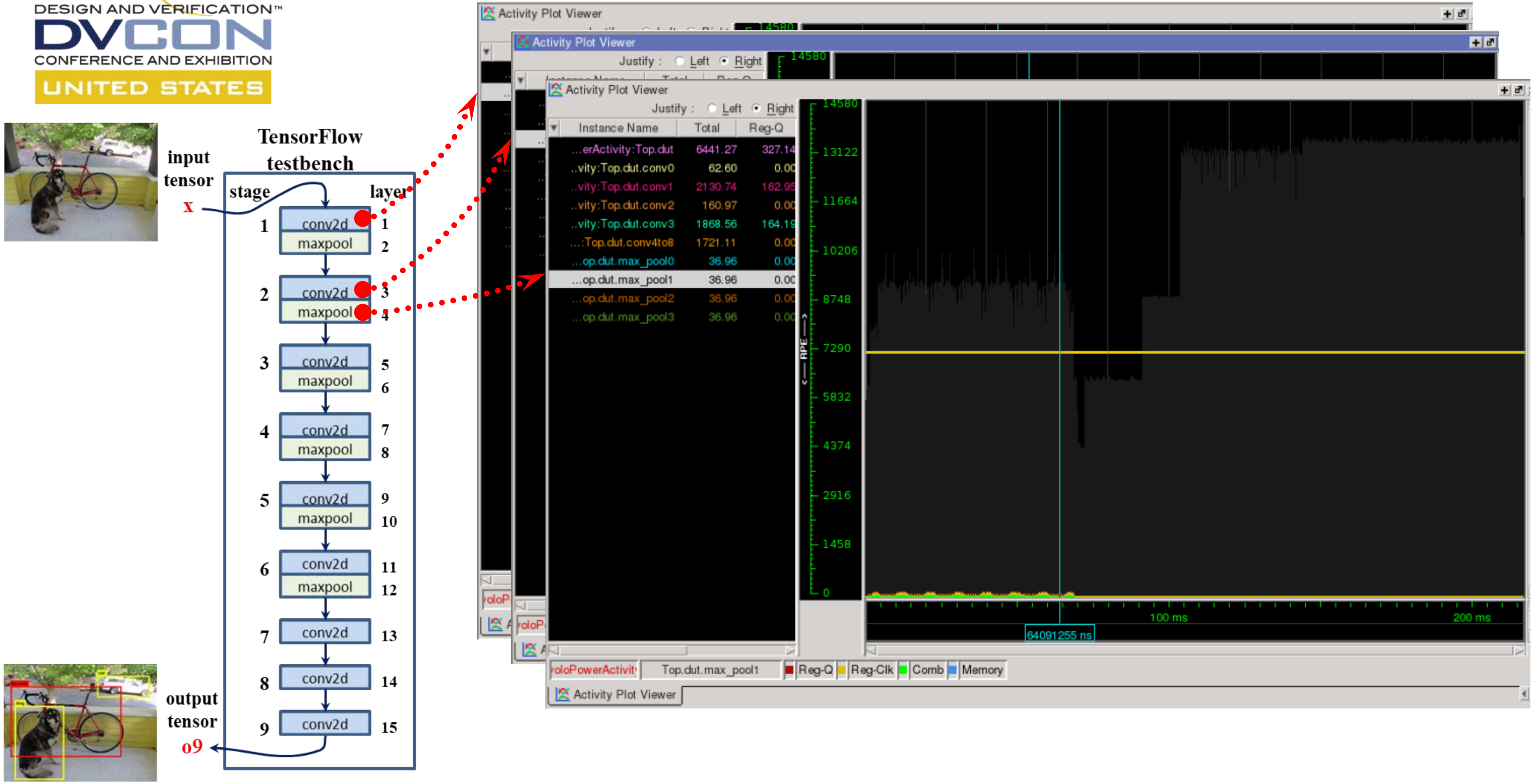




Instance Name	Total	Reg-Q
...erActivity:Top.dut	6441.27	327.14
..vity:Top.dut.conv0	62.60	0.00
..vity:Top.dut.conv1	2130.74	162.95
..vity:Top.dut.conv2	160.97	0.00
..vity:Top.dut.conv3	1868.56	164.19
...:Top.dut.conv4to8	1721.11	0.00
...op.dut.max_pool0	36.96	0.00
...op.dut.max_pool1	36.96	0.00
...op.dut.max_pool2	36.96	0.00
...op.dut.max_pool3	36.96	0.00

- Individual contributions are shown for highlighed modules in leftpane

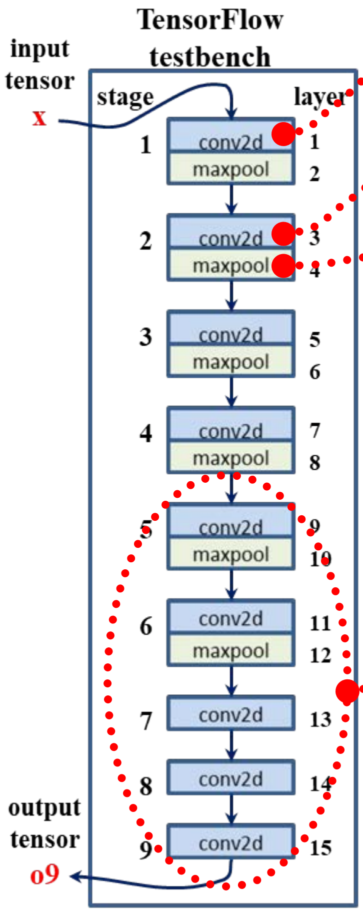
YoloTiny power analysis

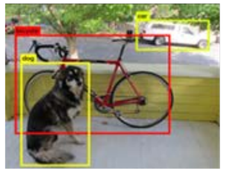


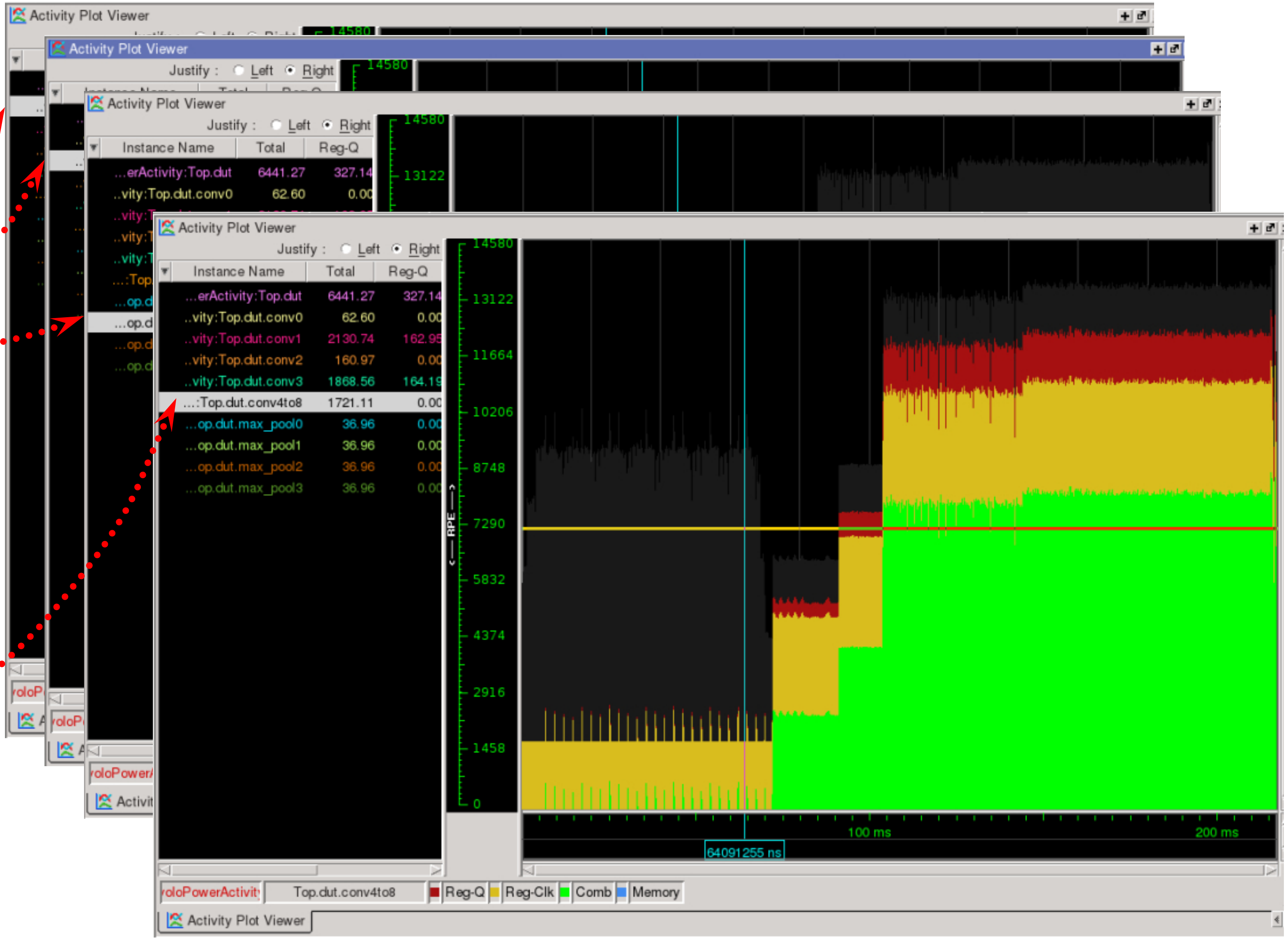
Instance Name	Total	Reg-Q
...erActivity:Top.dut	6441.27	327.14
..vity:Top.dut.conv0	62.60	0.00
..vity:Top.dut.conv1	2130.74	162.95
..vity:Top.dut.conv2	160.97	0.00
..vity:Top.dut.conv3	1868.56	164.19
...:Top.dut.conv4to8	1721.11	0.00
...op.dut.max_pool0	36.96	0.00
...op.dut.max_pool1	36.96	0.00
...op.dut.max_pool2	36.96	0.00
...op.dut.max_pool3	36.96	0.00

- Individual contributions are shown for highlighed modules in leftpane

YoloTiny power analysis







Activity Plot Viewer

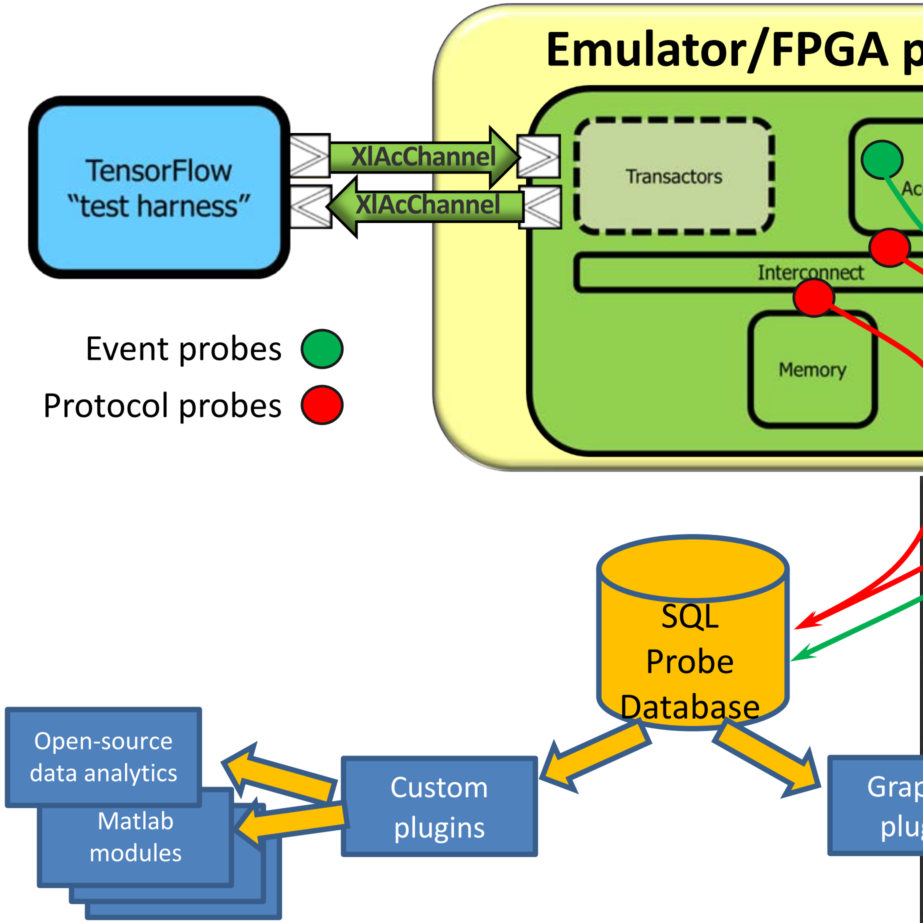
Justify : ○ Left ● Right

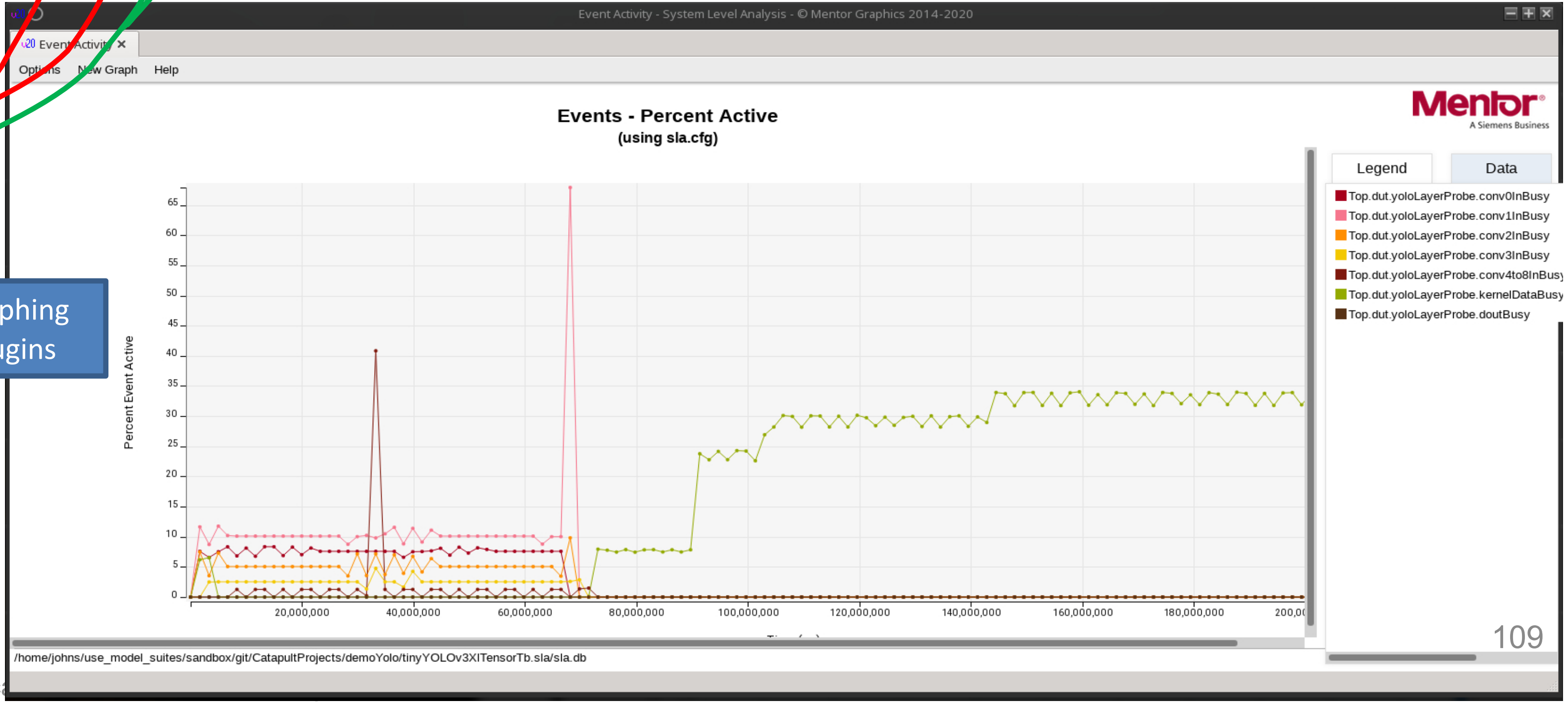
Instance Name	Total	Reg-Q
...erActivity:Top.dut	6441.27	327.14
..vity:Top.dut.conv0	62.60	0.00
..vity:Top.dut.conv1	2130.74	162.95
..vity:Top.dut.conv2	160.97	0.00
..vity:Top.dut.conv3	1868.56	164.19
...:Top.dut.conv4to8	1721.11	0.00
...op.dut.max_pool0	36.96	0.00
...op.dut.max_pool1	36.96	0.00
...op.dut.max_pool2	36.96	0.00
...op.dut.max_pool3	36.96	0.00

- Individual contributions are shown for highlighed modules in leftpane

YoloTiny performance analysis (System Level Analyzer)

- Emulator and FPGA proto platforms allow massive amounts of probed performance data collection in a relatively small amount of time
- Non-intrusive probing into DUT using SystemVerilog’s ‘bind’ construct
- When simulation occurs, captured probed events are efficiently directed into an SQL database
- Builtin collection of graphing plugins can be used to create displays of a variety metrics and visualizations
- Custom plugins are easy to create using standard SQL query commands interfacing to back-end tools such as open-source data analytics, Matlab, etc.



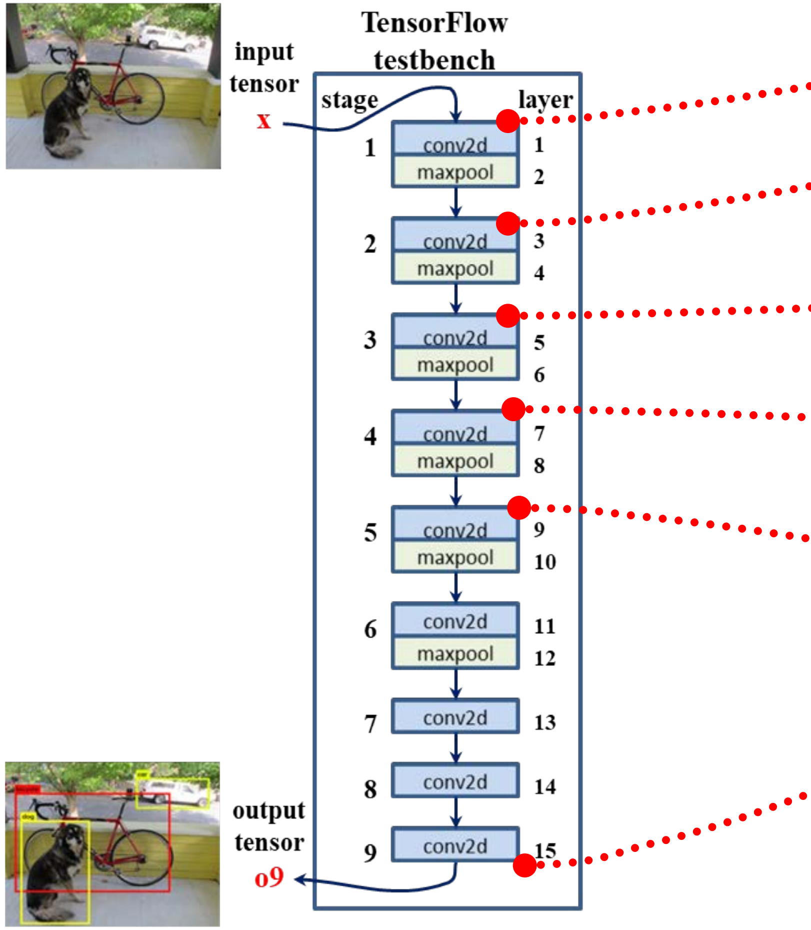


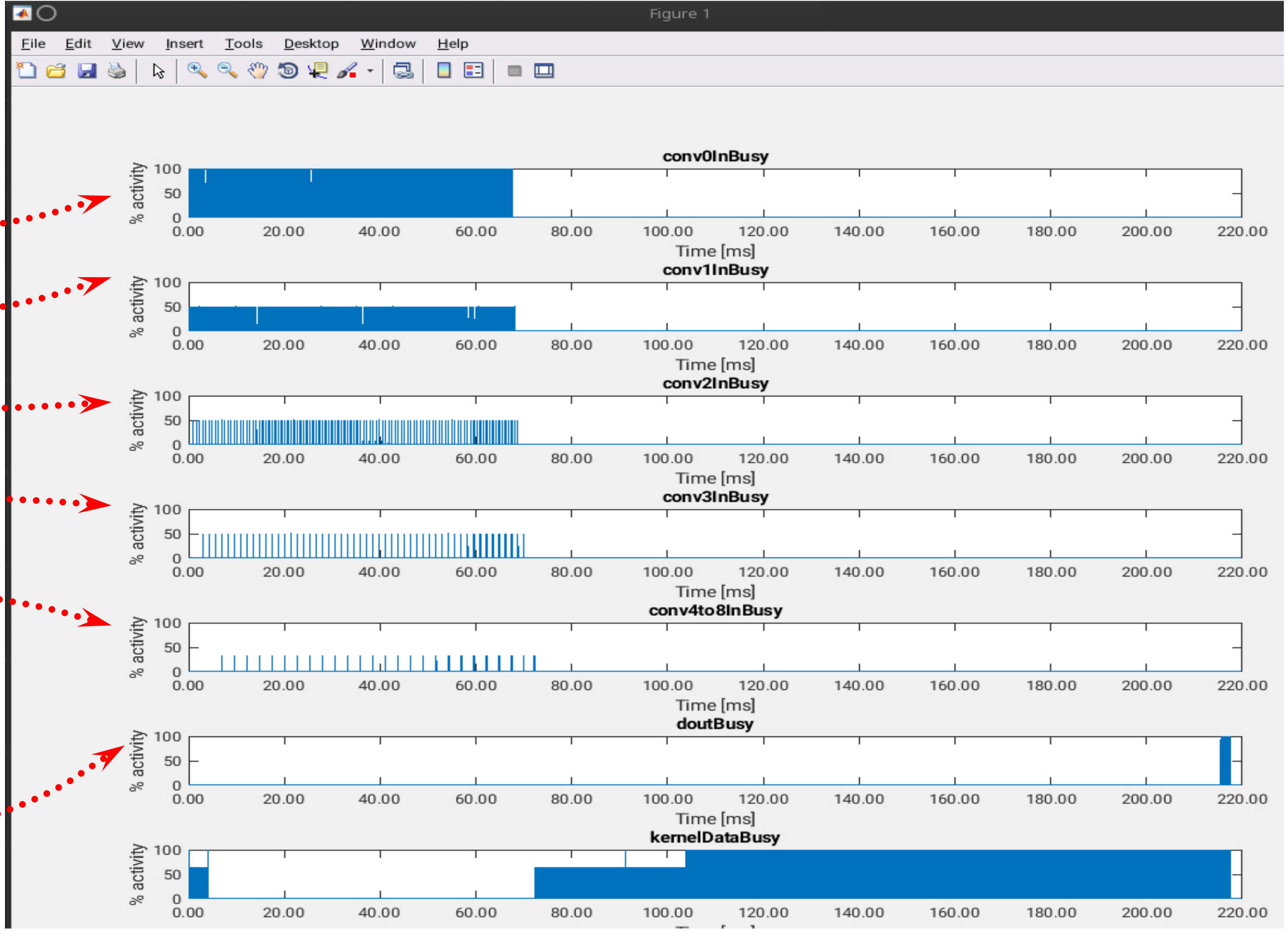
Events - Percent Active
(using sla.cfg)

- 🟥 Top.dut.yoloLayerProbe.conv0InBusy
- 🟥 Top.dut.yoloLayerProbe.conv1InBusy
- 🟧 Top.dut.yoloLayerProbe.conv2InBusy
- 🟨 Top.dut.yoloLayerProbe.conv3InBusy
- 🟥 Top.dut.yoloLayerProbe.conv4to8InBusy
- 🟩 Top.dut.yoloLayerProbe.kernelDataBusy
- 🟫 Top.dut.yoloLayerProbe.doutBusy

/home/johns/use_model_suites/sandbox/git/CatapultProjects/demoYolo/tinyYOLOv3XlTensorTb.sla/sla.db

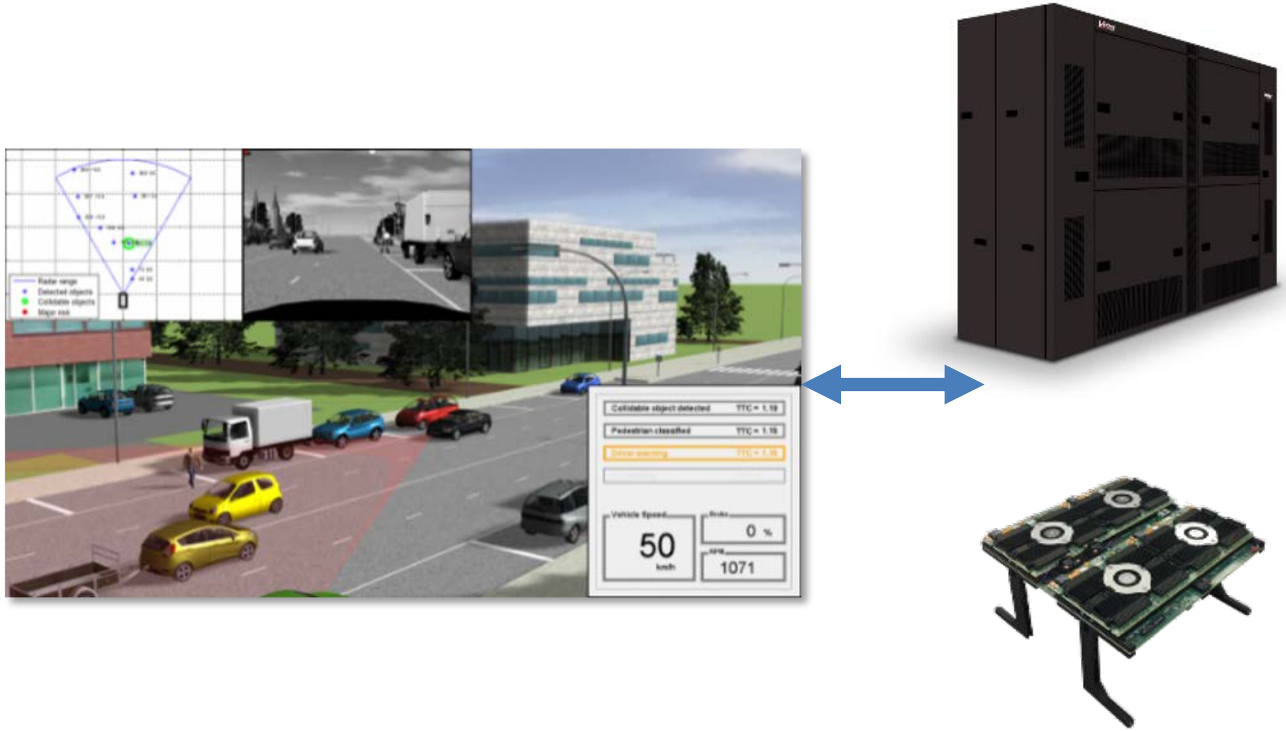
YoloTiny performance analysis (System Level Analyzer)





- Customized probe events defining activity into and out of each layer can be defined
- In this case a custom “plugin” was created to generate Matlab™ graphing plots

Pre-Si Validation – SoC’s Digital Twin

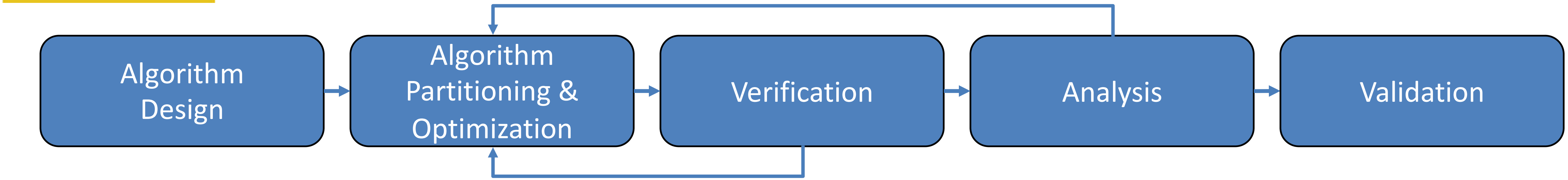


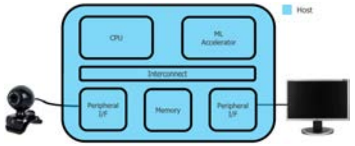
- Bring SW to alpha release state
  - Before 1st silicon
- Validate post-Si lab setup pre-Si
  - Including debug capabilities
- Begin validation testing
  - Billions of miles to validate ADAS!
  - Start pre-Si
- Use as demonstrators
  - Customers
  - Government regulators, …
- Debug issues uncovered in silicon

Summary

- In this tutorial we have shown:
  - How accelerating key algorithms in HW deliver application performance
  - Designing the algorithm in C++ to
    - Quickly explore power, performance, area of alternative algorithmic approaches
    - Verify the algorithm implemented in C++
    - Use high-level synthesis to implement the accelerator in RTL
  - Verified and validated the accelerator block
    - Enabling SW driven system design
    - Used accelerated simulation to cover deep test datasets
  - Verified and validated the full SoC
    - Validating power and performance of full SoC
    - SoC optimized in context of SW
  - We maximized reuse of block verification from C++ through RTL
    - Development environments and platforms evolve to maximize reuse
    - Work done at the block level, reused at SoC level

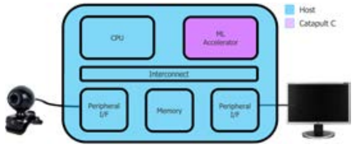
Our Story in Five Steps





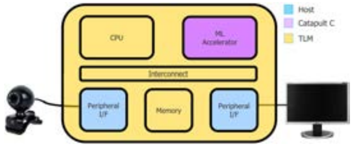
- Tiny YOLO algorithm, written in Python, executed in TensorFlow on a desktop or laptop as stand alone
- It inferences a camera input and it displays processed output on a screen
  - Verify algorithm works properly

- Speed ≃ 0.4 sec/inference

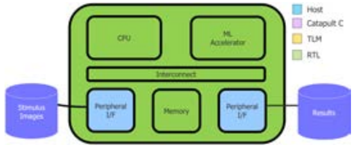


- Manual conversion of Tiny YOLO to C for High-Level Synthesis
  - Target wide variety of implementation architectures without re-coding
  - Common testbench for different abstraction levels
  - Automated creation of bus interfaces to surrounding system

- Speed ≃ 4 sec/inference



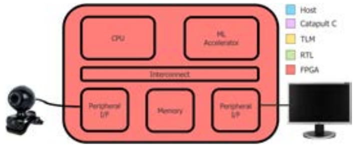
- Block-level verification at C and RT levels with a reusable verification environment
- Exploiting hybrid platform to maximize flexibility in verification
- And, enable earliest SW development and SW-driven verification
- Utilize HW-assisted verification for large dataset tests and full SoC verification



- Early & continuous power, performance analysis from algorithm through full SoC
- Utilize hybrid to focus analysis at block or broader levels
- Execute platform with same software stack from Hybrid platform
  - Realistic Performance
  - Accurate Power
  - Functional Coverage

Speed:

- 21,000 sec/inf RTL SW sim
- 10 sec/inf emulation
- 0.03 sec/inf prototype



- Block-level validation in SoC context with hybrid
- Prototype full SOC
  - Enable complete SW stack & system validation
  - Using real-world stimulus
- Pre-Si Validation
  - Connect to real interfaces, at speed
  - Prepare post-Si validation environment, tests and debug capabilities

CONCLUSION AND Q & A