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
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
SOFTWARE TO SYSTEMS

Lauro Rizzatti
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

- The algorithms must be accelerated in hardware
  - 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
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

* 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

1. **Algorithm Design**
   - Tiny YOLO algorithm, written in Python, executed in TensorFlow on a desktop or laptop as stand alone.
   - It infers a camera input and displays processed output on a screen.
   - Verify algorithm works properly.
   - Speed = 0.4 sec/inference

2. **Algorithm Partitioning & Optimization**
   - 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

3. **Verification**
   - 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.

4. **Analysis**
   - 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

5. **Validation**
   - 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
```python
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(f'Elapsed time : {strtime} secs')
```

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
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
')
Replace Layers one at a Time

TensorFlow testbench

1. conv2d
   maxpool
2. conv2d
   maxpool
3. conv2d
   maxpool
4. conv2d
   maxpool
5. conv2d
   maxpool
6. conv2d
   maxpool
7. conv2d
   maxpool
8. conv2d
   maxpool
9. conv2d

HLS C++ code block

1. conv2dHls
   maxpoolHls
2. conv2dHls
   maxpoolHls

input tensor x

output tensor o9
Replace Layers one at a Time

TensorFlow testbench

<table>
<thead>
<tr>
<th>stage</th>
<th>layer</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td></td>
</tr>
<tr>
<td>9</td>
<td></td>
</tr>
</tbody>
</table>

HLS C++ code block

<table>
<thead>
<tr>
<th>stage</th>
<th>layer</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td></td>
</tr>
<tr>
<td>9</td>
<td></td>
</tr>
</tbody>
</table>

input tensor X

output tensor o9
Replace Layers one at a Time

TensorFlow testbench

HLS C++ code block

input tensor

output tensor

Stage 1
- conv2d
- maxpool

Stage 2
- conv2d
- maxpool

Stage 3
- conv2d
- maxpool

Stage 4
- conv2d
- maxpool

Stage 5
- conv2d
- maxpool

Stage 6
- conv2d
- maxpool

Stage 7
- conv2d
- maxpool

Stage 8
- conv2d
- maxpool

Stage 9
- conv2d
- maxpool

Stage 10
- conv2dHls
- maxpoolHls

Stage 11
- conv2dHls
- maxpoolHls

Stage 12
- conv2dHls
- maxpoolHls

Stage 13
- conv2dHls
- maxpoolHls

Stage 14
- conv2dHls
- maxpoolHls

Stage 15
- conv2dHls
- maxpoolHls
Then Replace All Layers

```
input
tensor
```

```
output
tensor
```
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
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)
Verification of RTL

- **Dynamic verification**
  - Common input to algorithm and RTL
  - Compares output from RTL 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

```cpp
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;
}
```
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

```c
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-Tiny Profile

<table>
<thead>
<tr>
<th>layer</th>
<th>filters</th>
<th>size</th>
<th>input</th>
<th>output</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 conv</td>
<td>16</td>
<td>3 x 3 / 1</td>
<td>416 x 416 x 3</td>
<td>416 x 416 x 16</td>
</tr>
<tr>
<td>1 max</td>
<td>2 x 2 / 2</td>
<td>208 x 208 x 16</td>
<td>208 x 208 x 16</td>
<td>0.399 BFLOPs</td>
</tr>
<tr>
<td>2 conv</td>
<td>32</td>
<td>3 x 3 / 1</td>
<td>208 x 208 x 16</td>
<td>208 x 208 x 32</td>
</tr>
<tr>
<td>3 max</td>
<td>2 x 2 / 2</td>
<td>104 x 104 x 32</td>
<td>104 x 104 x 32</td>
<td>0.399 BFLOPs</td>
</tr>
<tr>
<td>4 conv</td>
<td>64</td>
<td>3 x 3 / 1</td>
<td>104 x 104 x 32</td>
<td>104 x 104 x 64</td>
</tr>
<tr>
<td>5 max</td>
<td>2 x 2 / 2</td>
<td>52 x 52 x 64</td>
<td>52 x 52 x 64</td>
<td>0.399 BFLOPs</td>
</tr>
<tr>
<td>6 conv</td>
<td>128</td>
<td>3 x 3 / 1</td>
<td>52 x 52 x 64</td>
<td>52 x 52 x 128</td>
</tr>
<tr>
<td>7 max</td>
<td>2 x 2 / 2</td>
<td>26 x 26 x 128</td>
<td>26 x 26 x 128</td>
<td>0.399 BFLOPs</td>
</tr>
<tr>
<td>8 conv</td>
<td>256</td>
<td>3 x 3 / 1</td>
<td>26 x 26 x 128</td>
<td>26 x 26 x 256</td>
</tr>
<tr>
<td>9 max</td>
<td>2 x 2 / 2</td>
<td>13 x 13 x 256</td>
<td>13 x 13 x 256</td>
<td>0.399 BFLOPs</td>
</tr>
<tr>
<td>10 conv</td>
<td>512</td>
<td>3 x 3 / 1</td>
<td>13 x 13 x 256</td>
<td>13 x 13 x 512</td>
</tr>
<tr>
<td>11 max</td>
<td>2 x 2 / 2</td>
<td>13 x 13 x 512</td>
<td>13 x 13 x 512</td>
<td>0.399 BFLOPs</td>
</tr>
<tr>
<td>12 conv</td>
<td>1024</td>
<td>3 x 3 / 1</td>
<td>13 x 13 x 512</td>
<td>13 x 13 x 1024</td>
</tr>
<tr>
<td>13 conv</td>
<td>256</td>
<td>1 x 1 / 1</td>
<td>13 x 13 x 1024</td>
<td>13 x 13 x 256</td>
</tr>
<tr>
<td>14 conv</td>
<td>512</td>
<td>3 x 3 / 1</td>
<td>13 x 13 x 256</td>
<td>13 x 13 x 512</td>
</tr>
<tr>
<td>15 conv</td>
<td>255</td>
<td>1 x 1 / 1</td>
<td>13 x 13 x 512</td>
<td>13 x 13 x 255</td>
</tr>
<tr>
<td>16 yolo</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>17 route</td>
<td>13</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>18 conv</td>
<td>128</td>
<td>1 x 1 / 1</td>
<td>13 x 13 x 256</td>
<td>13 x 13 x 128</td>
</tr>
<tr>
<td>19 upsampling</td>
<td>2x</td>
<td>13 x 13 x 128</td>
<td>26 x 26 x 128</td>
<td></td>
</tr>
<tr>
<td>20 route</td>
<td>19 8</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>21 conv</td>
<td>256</td>
<td>3 x 3 / 1</td>
<td>26 x 26 x 128</td>
<td>26 x 26 x 256</td>
</tr>
<tr>
<td>22 conv</td>
<td>255</td>
<td>1 x 1 / 1</td>
<td>26 x 26 x 256</td>
<td>26 x 26 x 255</td>
</tr>
</tbody>
</table>

Loading weights from yolov3-tiny.weights...Done!
What is convolution?

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
TensorFlow 2d convolution

Feature Maps

Kernels

Output channels

∑

∑
TensorFlow 2d convolution

- Feature Maps
- Kernels
- Output channels
TensorFlow 2d convolution

Feature Maps

Kernels

Output channels
TensorFlow 2d convolution

Feature Maps

Kernels

Output channels
TensorFlow 2d convolution

Feature Maps

Kernels

Output channels
TensorFlow 2d convolution

Feature Maps
Kernels
Output channels
TensorFlow 2d convolution

Feature Maps

Kernels

Output channels
TensorFlow 2d convolution
TensorFlow 2d convolution

Feature Maps

Kernels

Output channels
TensorFlow 2d convolution

Feature Maps

Kernels

Output channels
TensorFlow 2d convolution

_feature Maps_  
Kernels  
Output channels

\[ \sum \]

**Feature Maps**

**Kernels**

**Output channels**
TensorFlow 2d convolution

Feature Maps

Kernels

Output channels

$\sum$
TensorFlow 2d convolution

Feature Maps

Kernels

Output channels
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

```c
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_window[i][j] * kernel[ic][oc][i][j];
                      }
                  }
            
            fmap_out[d][r][c] = acc;
          }
      }
  }
```

Feature Map Constant

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

Output Channel Constant Feature Map Constant

- `< Read feature map data stream >`
- `< Sliding window of feature map data >`
- `< stationary data over output channels >`
- `< 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

```c
2  regs[0] = new_value;
3  #pragma hls loop_unroll
4  for (i=N-1; i>0; i++) {
5      regs[i] = regs[i-1];
6  }
7
8```

Diagram of a shift register with states 0, 1, 2, ..., N-2, N-1.
## Parallel multipliers

|   | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | 16 | 17 | 18 | 19 | 20 | 21 | 22 | 23 |
|---|---|---|---|---|---|---|---|---|---|----|----|----|----|----|----|----|----|----|----|----|----|----|    |

![Diagram of parallel multipliers](image_url)
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.

Multi-channel Vertical Windowing

Pixel stream

Line Buffer

Line Buffer

Line Buffer

Line Buffer

Multi-channel Sliding Window
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

```cpp
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 core
  void CCS_BLOCK(run){ac_int<W,false> &x_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,x_in.template slc<T0::width>(0)); //weight and data sent
    if(ld)/\Load weight
    h.set_slc(0,x_in.template slc<T1::width>(0)); //slice read weight
    x_out = x_in; //forward weight
    }else{T//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 data
  }
}
```
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

1x3 PE Array convolution

Partial sum input

Shifted input
Matrix of PEs

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

```cpp
// template<typename T0, typename T1, typename T2, int W, int N>
class pe_array_classN{
    pe_array_class<T0,T1,T2,W> pe_array[N];
    ac_array<ac_int<W,false>,N> imap_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;
}
```
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
  - Responses from RTL compared with responses from algorithmic C
Replace One Layer at a Time

TensorFlow testbench

input tensor x

output tensor o9

HLS C++ code block

RTL

Synthesized conv2dHls RTL model

C++ driver "proxy" model
Replace One Layer at a Time

TensorFlow testbench

input tensor x

output tensor o9

HLS C++ code block

stage layer stage

1 con2d maxpool

2 con2d maxpool

3 con2d maxpool

4 con2d maxpool

5 con2d maxpool

6 con2d maxpool

7 con2d

8 con2d

9 con2d

10 con2d

11 con2d

12 con2d

13 con2d

14 con2d

15 con2d

C++ driver “proxy” model

Synthesized conv2dHls
RTL model

RTL
Replace One Layer at a Time

TensorFlow testbench

input tensor x

1. conv2d
   maxpool

2. conv2d
   maxpool

3. conv2d
   maxpool

4. conv2d
   maxpool

5. conv2d
   maxpool

6. conv2d
   maxpool

7. conv2d

8. conv2d

9. conv2d

output tensor o9

HLS C++ code block

stage

layer

1

2

3

4

5

6

7

8

9

10

11

12

13

14

15

C++ driver "proxy" model

RTL

Synthesized conv2dHls RTL model

input tensor x

output tensor o9

stage

layer

1

2

3

4

5

6

7

8

9

10

11

12

13

14

15
Replace All Layers

TensorFlow testbench

HLS C++ code block

RTL

Synthesized conv2dHls RTL model For All Layers

input tensor \( x \)

output tensor \( o_9 \)

1. conv2d
   maxpool

2. conv2d
   maxpool

3. conv2d
   maxpool

4. conv2d
   maxpool

5. conv2d
   maxpool

6. conv2d
   maxpool

7. conv2d

8. conv2d

9. conv2d

C++ driver “proxy” model

C++ driver “proxy” model
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 2x bit-widths
  - Processors are fixed bit-width
- 8-bit integer multiplier is 27 times smaller and uses 19 times less energy than a 32-bit floating point multiplier

*NVIDIA 2017

![Normalized Energy Cost Diagram]

![Cost of Operations Table]

Energy numbers are from Mark Horowitz: “Computing’s Energy Problem (and what we can do about it)”, ISSCC 2014
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
From Block to Full SoC

VERIFICATION
Our Story in Five Steps

Algorithm Design

Algorithm Partitioning & Optimization

Verification

Analysis

Validation

- 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

Tutorial: Application Optimized HW/SW Design & Verification of a Machine Learning SoC
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
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”**
- **ML Accelerator**
- **Interconnect**
- **Memory**

Diagram showing the integration of TensorFlow with Transactors, ML Accelerator, Interconnect, and Memory.
TensorFlow test harness for RTL

- Host
- Catapult C
- TLM
- RTL

TensorFlow "test harness"
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
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.

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.
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.

- 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
What is SW Enabled Verification?

Hybrid Platform

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)

CPU
BUS
MODEL
RAM

Tutorial: Application Optimized HW/SW Design & Verification of a Machine Learning SoC
Hybrid Verification in a SW Enabled flow

Software Development

SW Platform

Drivers, boot code

Hybrid Platform

3rd Party IP
RTL Designs

Hardware Development

First Silicon

SW Development
Performance Analysis
Power Analysis
SoC Validation
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
Hybrid Enables Mixing Abstractions

*Flexibility in Verifying, Analyzing HW & Enabling SW*

- CPU
- ML Accelerator
- Interconnect
- Peripheral I/F
- Memory
- Peripheral I/F

- Host
- HLS C
- RTL

Tutorial: Application Optimized HW/SW Design & Verification of a Machine Learning SoC
Hybrid Enables Mixing Abstractions

Flexibility in Verifying, Analyzing HW & Enabling SW

CPU
ML Accelerator
Interconnect
Peripheral I/F
Memory
Peripheral I/F

Host
HLS C
RTL

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

Benchmark/App

SW

HW

- Performance analysis
- Power Analysis
- Software Development
- System Validation

Runtime / App Lyr
Linux

Hybrid Layer

CPU/GIC/MEM

- Yolo Tiny C Application

Accelerator IP

- Yolo Tiny - Catapult C

Runtime / App Lyr
Linux

Hybrid Layer

CPU/GIC/MEM

- SW iterate

HW iterate

HW iterate

Software Development

System Validation

Hybrid Layer

Network device
Storage device
Display device
Input device

CPU/GIC/MEM

Tutorial: Application Optimized HW/SW Design & Verification of a Machine Learning SoC
HLS-C is Synthesized to 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.

```
CORTEX_ArmV8
  .AWREADYM (AWREADYM[0]),
  .AWVALIDM (AWVALIDM[0]),
  .AWIDM (AWIDM[0]),
AXINameM ⇔ AXINameM[0]

Interconnect

axi_memory
  .AWID  (AWID[0]),
  .AWADDR (AWADDR[0]),
  .AWLEN  (AWLEN[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 reports 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
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
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
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
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**
<table>
<thead>
<tr>
<th></th>
<th><strong>Key user requirements for power analysis</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td><strong>Power w/ Real-world Scenarios/SW:</strong> Early power trend analysis at full SoC while running <em>real world user scenarios and software</em></td>
</tr>
<tr>
<td>2</td>
<td><strong>Accurate RTL/GL average and peak power:</strong> Generate accurate average and peak power numbers in target application environment with RTL and gate level netlist</td>
</tr>
<tr>
<td>3</td>
<td><strong>Power Optimization:</strong> Identify potential power optimization opportunities early in design cycle for architectural tradeoffs</td>
</tr>
<tr>
<td>4</td>
<td><strong>Low-Power Control via HW/SW:</strong> UPF-based low power verification with power controls coming from SW applications</td>
</tr>
</tbody>
</table>
Typical power app

How it works

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

Waveform

- Power Trend Analysis
  - Activity Map
  - Activity Factor
  - Activity Plot

File-based SAIF/FSDB

Power Tools

Average Power
Peak Power

Streaming API

Power Tools

Average Power
Peak Power
Complete Power Solution

Activity Plot
- Billion’s of clock cycle

Read API FSDB/SAIF
- Million’s of clock cycle

Power Estimator
- Accurate power numbers

Power Optimizer
- Low Power RTL Edit’s

Verify Design Changes

always@(posedge clk)
q<=d;

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

Billion’s of clock cycle

Million’s of clock cycle

Power tools

Tutorial: Application Optimized HW/SW Design & Verification of a Machine Learning SoC
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
YoloTiny power analysis

- 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
Individual contributions are shown for highlighted modules in left pane.
YoloTiny Power Analysis

- Individual contributions are shown for highlighted modules in left pane.
YoloTiny power analysis

- Individual contributions are shown for highlighted modules in left pane.
Individual contributions are shown for highlighted modules in left pane.
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
- Built-in collection of graphing plugins can be used to create displays of a variety of 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.
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 1\textsuperscript{st} 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

Algorithm Design

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

Algorithm Partitioning & Optimization

- 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

Verification

- 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

Analysis

- 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

Validation

- 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