2021 DESIGN AND VERIFICATION™ DVCON CONFERENCE AND EXHIBITION UNITED STATES

VIRTUAL | MARCH 1-4, 2021

Early Design and Validation of an AI Accelerator's System Level Performance Using an HLS Design Methodology

Wenbo Zheng | Senior HLS AE

Siemens EDA, a part of Siemens Digital Industries Software

SIEMENS

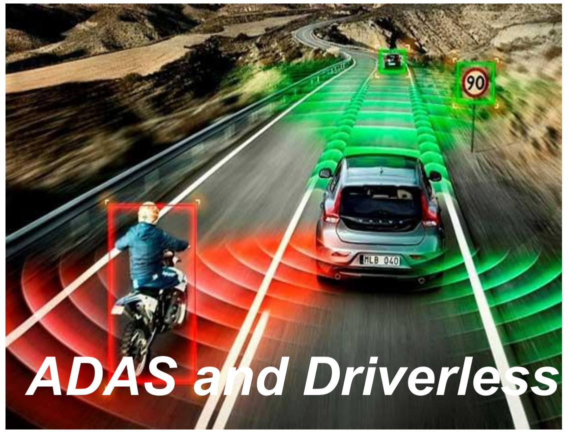
accellera SYSTEMS INITIATIVE™

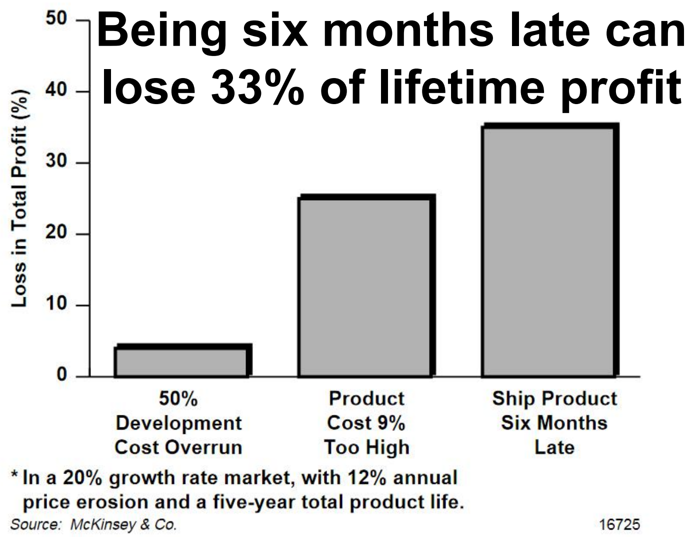
Agenda

- Challenges in Designing AI/ML Hardware
- Introduction to MatchLib
- Convolutional Neural Network (CNN) Overview
- Early Performance Analysis of CNN Convolution
- Architectural Refinement
- Synthesis and RTL Verification

AI/ML Application Challenges

- Algorithmic Complexity
  - Growing faster than the ability of RTL designers to code and verify
- Memory Architecture Complexity
  - Efficient data movement is key for power, performance and area
- RTL Verification Costs Increasing
  - Increased design complexity increases bugs introduced during hand-coding of RTL
  - RTL regressions involve server farms, electricity cost, licenses and time
- Slips in Design Schedule Kills Total Profit
  - Finding bugs during system integration is too late



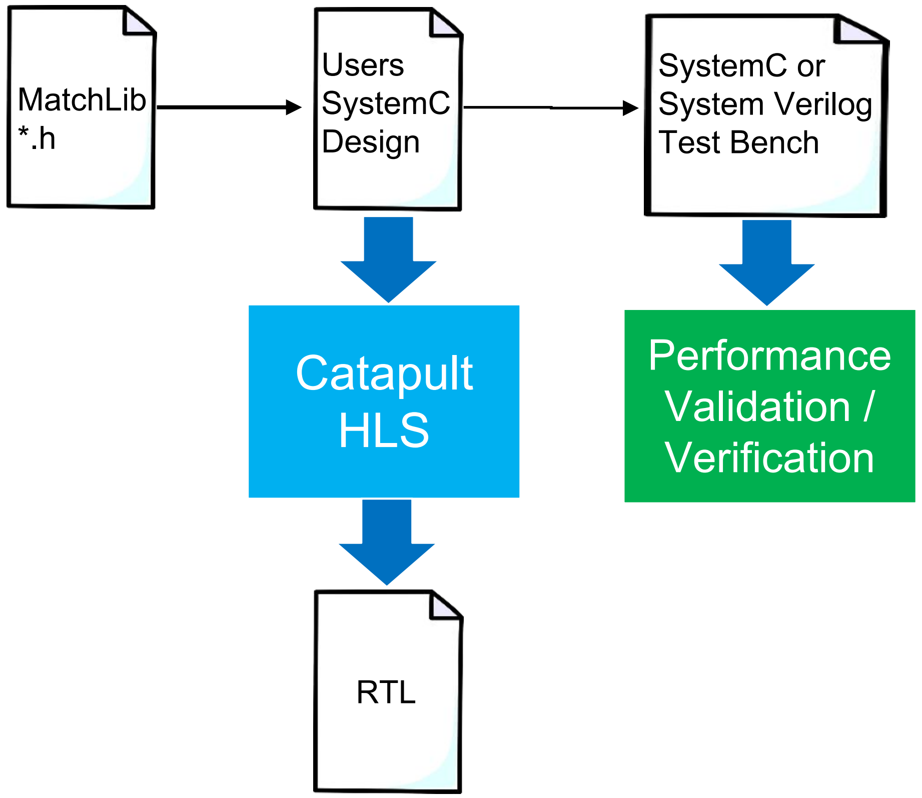


* In a 20% growth rate market, with 12% annual price erosion and a five-year total product life.
Source: McKinsey & Co.

Introduction to MatchLib

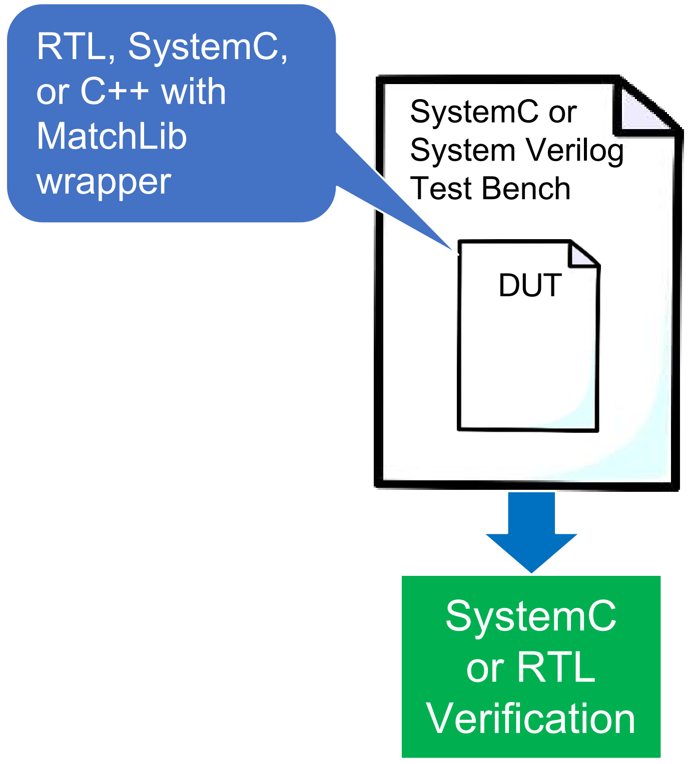
What is MatchLib?

- MatchLib is a SystemC library of commonly used functions and components developed in partnership with NVIDIA
  - Interfaces, FIFOs, interconnect, etc.
- Allows most design and verification of SoCs to be performed in SystemC
  - Verify system-level performance earlier
  - 30x faster than RTL for timing-accurate simulation
  - Easily plugs into existing DV flows
- Enables control oriented design
  - Easy-to-use throughput accurate modelling
  - DMA, Arbiters, Bus I/F


MatchLib *.h
Users SystemC Design
SystemC or System Verilog Test Bench
Catapult HLS
Performance Validation / Verification
RTL

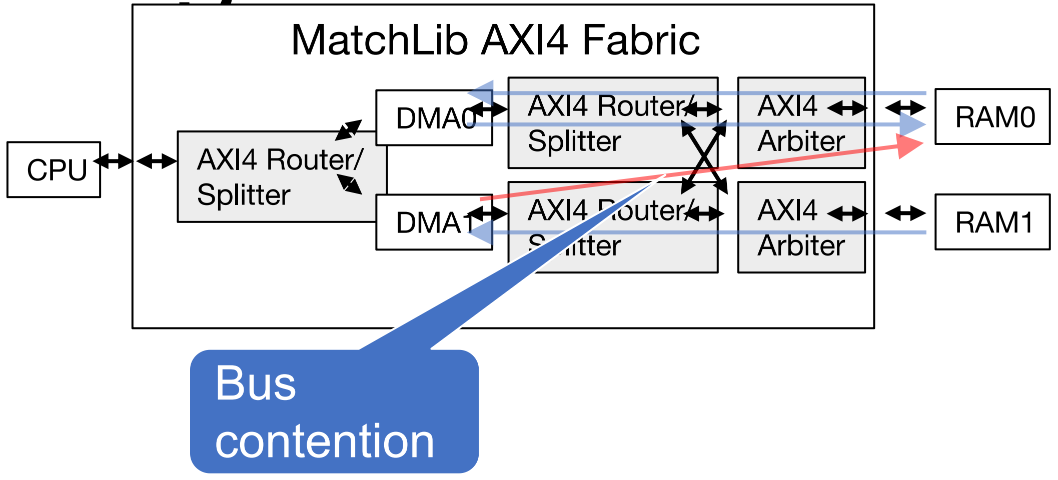
MatchLib + HLS Enables an Efficient Verification Flow

- MatchLib Involves DV Team at Every Stage of the Design Process
  - Early access to HW
- Smaller, faster models
  - C++/SystemC model is typically about 1/10 or less than the size of the RTL model
  - Easy to verify and debug
- C++/SystemC testbench reuse
  - Catapult makes it possible to automatically use same testbench for SystemC and Verilog models
- You can put thin Verilog wrapper around SystemC DUT
  - If using SV UVM DV flow, enables SV DV effort to start much earlier (even before any HLS)

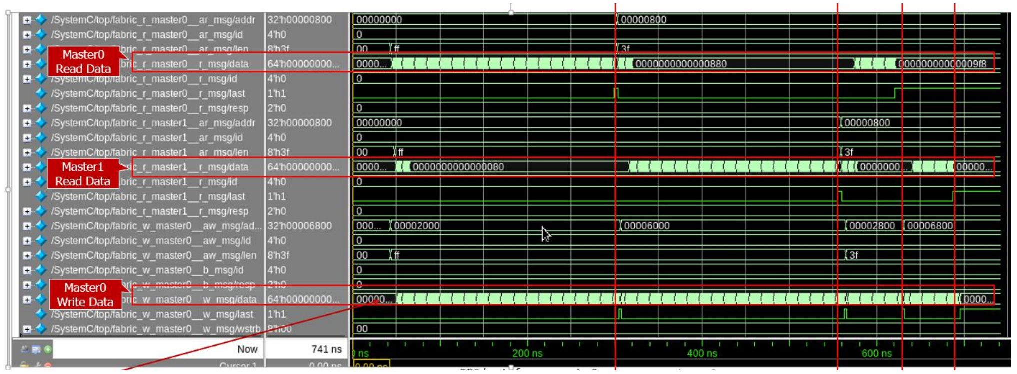

RTL, SystemC, or C++ with MatchLib wrapper
SystemC or System Verilog Test Bench
DUT
SystemC or RTL Verification

MatchLib Reduces Risks in Modern Digital Design

- Today's HW designs often process huge sets of data, with large intermediate results.
  - Machine Learning, Computer Vision, 5G Wireless
- Hard part is often managing the movement of data in the chip across all scenarios
  - Memory/interconnect architecture often has more impact on power/performance than the design of the computation units themselves
- Evaluating and verifying memory/interconnect architecture at RTL level is often not feasible
  - Too late in design cycle
  - Too much work to evaluate multiple candidate architectures
  - The most difficult/costly HW (& HW/SW) problems are found during system integration
  - If integration first occurs in RTL, it is very late and problems are very costly
- **MatchLib + SystemC HLS lets integration occur early when fixing problems is much cheaper**



MatchLib AXI4 Fabric SystemC Simulation

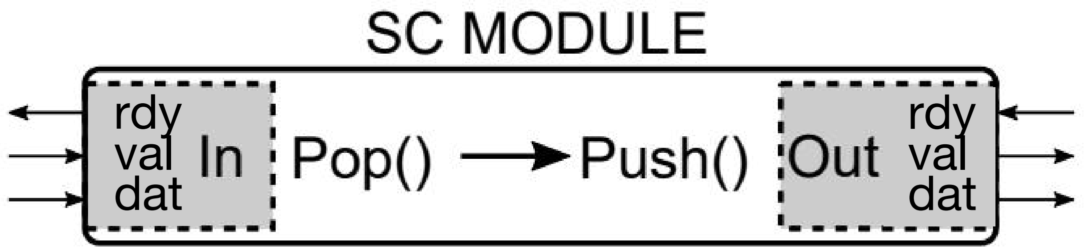


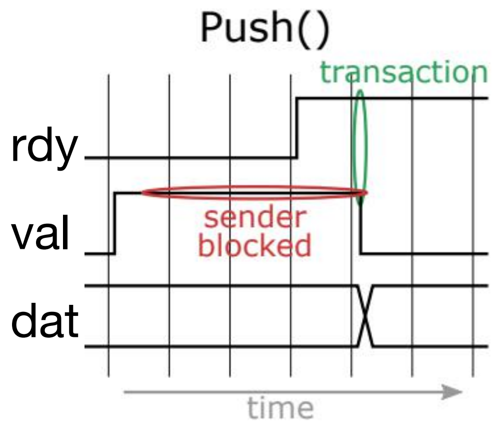
```
/SystemC/top/fabric_r_master0__ar_msg/addr   32'h00000800
/SystemC/top/fabric_r_master0__ar_msg/id     4'h0
bric_r_master0__ar_msg/len                   8'h3f
bric_r_master0__r_msg/data                   64'h00000000...
/SystemC/top/fabric_r_master0__r_msg/id      4'h0
/SystemC/top/fabric_r_master0__r_msg/last    1'h1
/SystemC/top/fabric_r_master0__r_msg/resp    2'h0
/SystemC/top/fabric_r_master1__ar_msg/addr   32'h00000800
/SystemC/top/fabric_r_master1__ar_msg/id     4'h0
/SystemC/top/fabric_r_master1__ar_msg/len    8'h3f
ric_r_master1__r_msg/data                    64'h00000000...
bric_r_master1__r_msg/id                     4'h0
/SystemC/top/fabric_r_master1__r_msg/last    1'h1
/SystemC/top/fabric_r_master1__r_msg/resp    2'h0
/SystemC/top/fabric_w_master0__aw_msg/ad...  32'h00006800
/SystemC/top/fabric_w_master0__aw_msg/id     4'h0
/SystemC/top/fabric_w_master0__aw_msg/len    8'h3f
/SystemC/top/fabric_w_master0__b_msg/id      4'h0
bric_w_master0__b_msg/resp                   2'h0
bric_w_master0__w_msg/data                   64'h00000000...
/SystemC/top/fabric_w_master0__w_msg/last    1'h1
/SystemC/top/fabric_w_master0__w_msg/wstrb   8'h00
Now                                          741 ns
Cursor 1                                     0.00 ns
```

Using MatchLib Connections

- MatchLib's Connections is a library and API of latency-insensitive channels
- All components of this library are synthesizable using HLS
- Connections library consists of ports and channels
  - Port implements data/ready/valid protocol
- Connections are templatized for data type T

Type	Name	Description
Port	In<T>	In port with Pop() and PobNB() methods
Port	Out<T>	Out port with Push() and PushNB() methods
Channel	Combinational<T>	Combinationally connects ports with Pop(), PobNB(), Push(), and PushNB() methods





Using MatchLib Connections - Example

```
#include <mc_connections.h>
class dut : public sc_module {
 public:
  sc_in<bool> CCS_INIT_S1(clk);
  sc_in<bool> CCS_INIT_S1(rst_bar);
  Connections::Out<uint32> CCS_INIT_S1(out1);
  Connections::In <uint32> CCS_INIT_S1(in1);

  SC_CTOR(dut) {
    SC_THREAD(main);
    sensitive << clk.pos();
    async_reset_signal_is(rst_bar, false);
  }
```

```
 rivate:
  void main() {
    out1.Reset();
    in1.Reset();
    wait();
    #pragma hls_pipeline_init_interval 1
    #pragma pipeline_stall_mode flush
    while (1) {
      uint32_t t = in1.Pop();
      out1.Push(t + 0x100);
    }
  }
};
```

Connections inputs and outputs

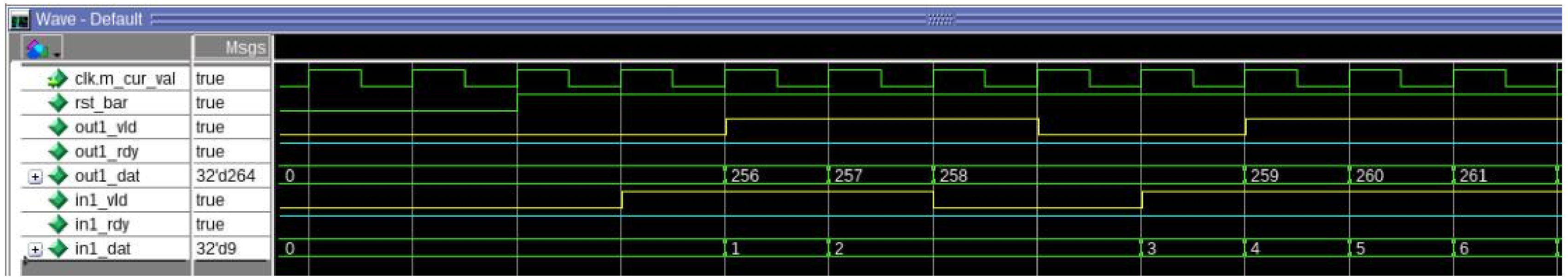
Connections write

Connections read

Simulating Performance Before Synthesis

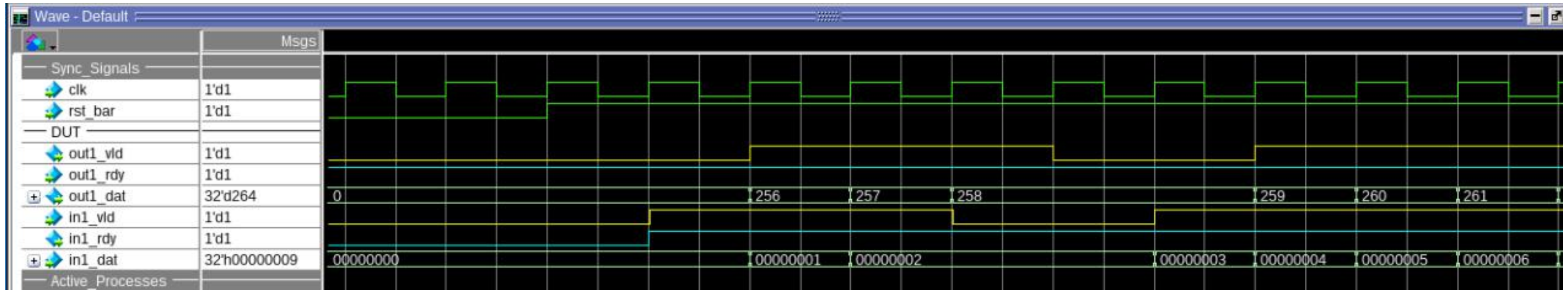
- Pre-HLS and Post-HLS simulation throughput are the same
- There can be differences in latency

Pre-HLS Simulation



```
Wave - Default
                 Msgs
clk.m_cur_val    true
rst_bar          true
out1_vld         true
out1_rdy         true
out1_dat         32'd264
in1_vld          true
in1_rdy          true
in1_dat          32'd9
```

Post-HLS Simulation



```
Wave - Default
                 Msgs
Sync_Signals
clk              1'd1
rst_bar          1'd1
DUT
out1_vld         1'd1
out1_rdy         1'd1
out1_dat         32'd264
in1_vld          1'd1
in1_rdy          1'd1
in1_dat          32'h00000009
Active_Processes
```

Modelling Bus I/F With MatchLib

- MatchLib provides high-quality implementations of AXI4 master and slave interfaces
- Users can also model custom bus interfaces using MatchLib
  - This example models a simple read bus I/F with burst
  - Testbench models 2-cycle overhead for initiating a new burst

```
class dut : public sc_module {
 public:
  …
  void main() {
    wait();
    #pragma hls_pipeline_init_interval 1
    #pragma pipeline_stall_mode flush
    while (1) {
      go.sync_in();
      uint32 addr = addr_offset_csr.read();
      uint32 burst_size = burst_csr.read();
      read_addr_chan.Push(addr);
      read_burst_size.Push(burst_size);
      do {
        uint32 data = read_data_chan.Pop();
        read_data_out.Push(data);
      } while (--burst_size !=0);
  }
```

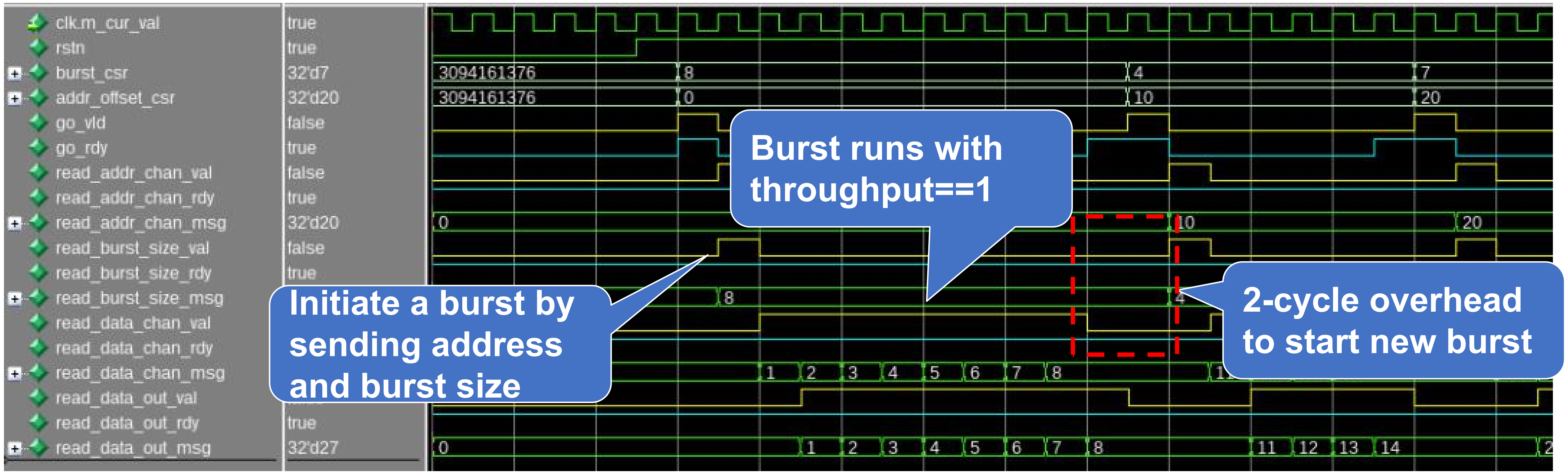
Initiate a burst by sending address and burst size

Read "burst_size" data from bus I/F

Modelling Bus I/F With MatchLib

- Small burst size and/or non-consecutive addresses will hurt performance by injecting dead cycles

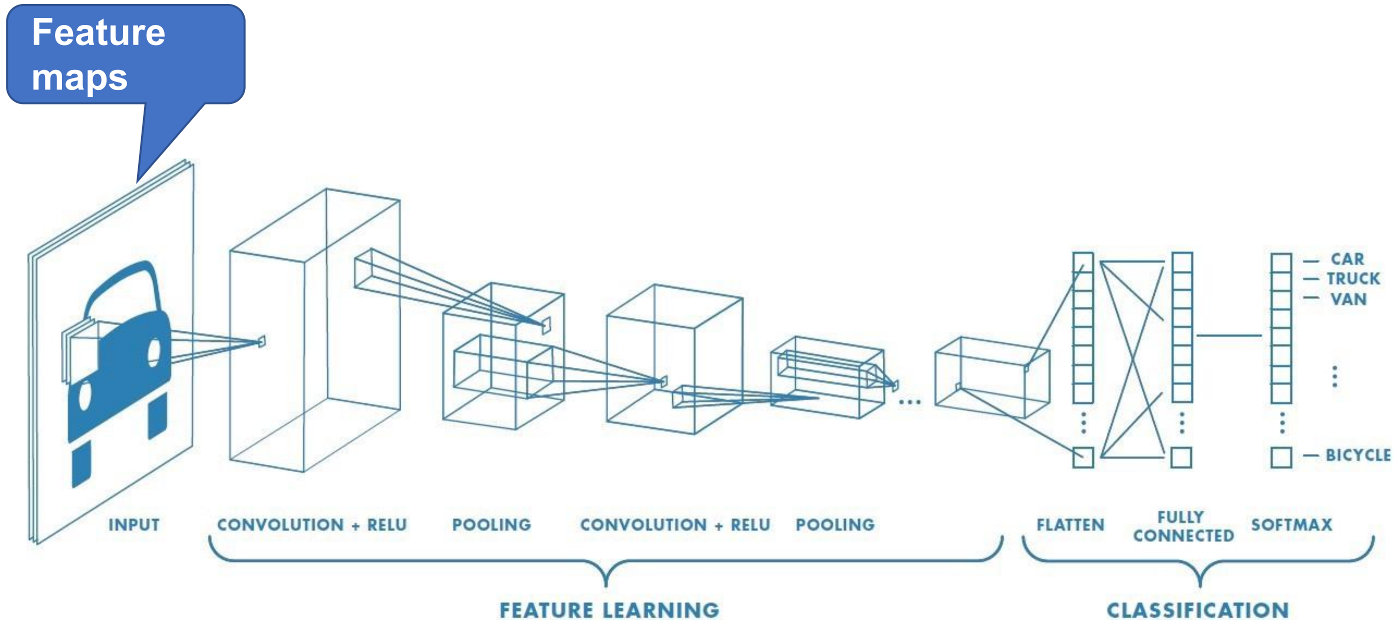
Pre-HLS Simulation



Convolutional Neural Network Overview

Convolutional Neural Network Overview

- Mostly Convolutional layers
  - Majority of computation done here (over 99%)
  - Majority of memory traffic
  - Bias and activation functions
- Pooling layers
  - Reduce feature map size
- Fully connected
  - Classifcation
- Softmax
  - normalize class probabilities



CNN Convolution – conv2d

- CNN convolutional layers have multiple input and output feature maps
- Each output feature map is a sum of separate convolutions across all input feature maps

Output feature map

Input feature map

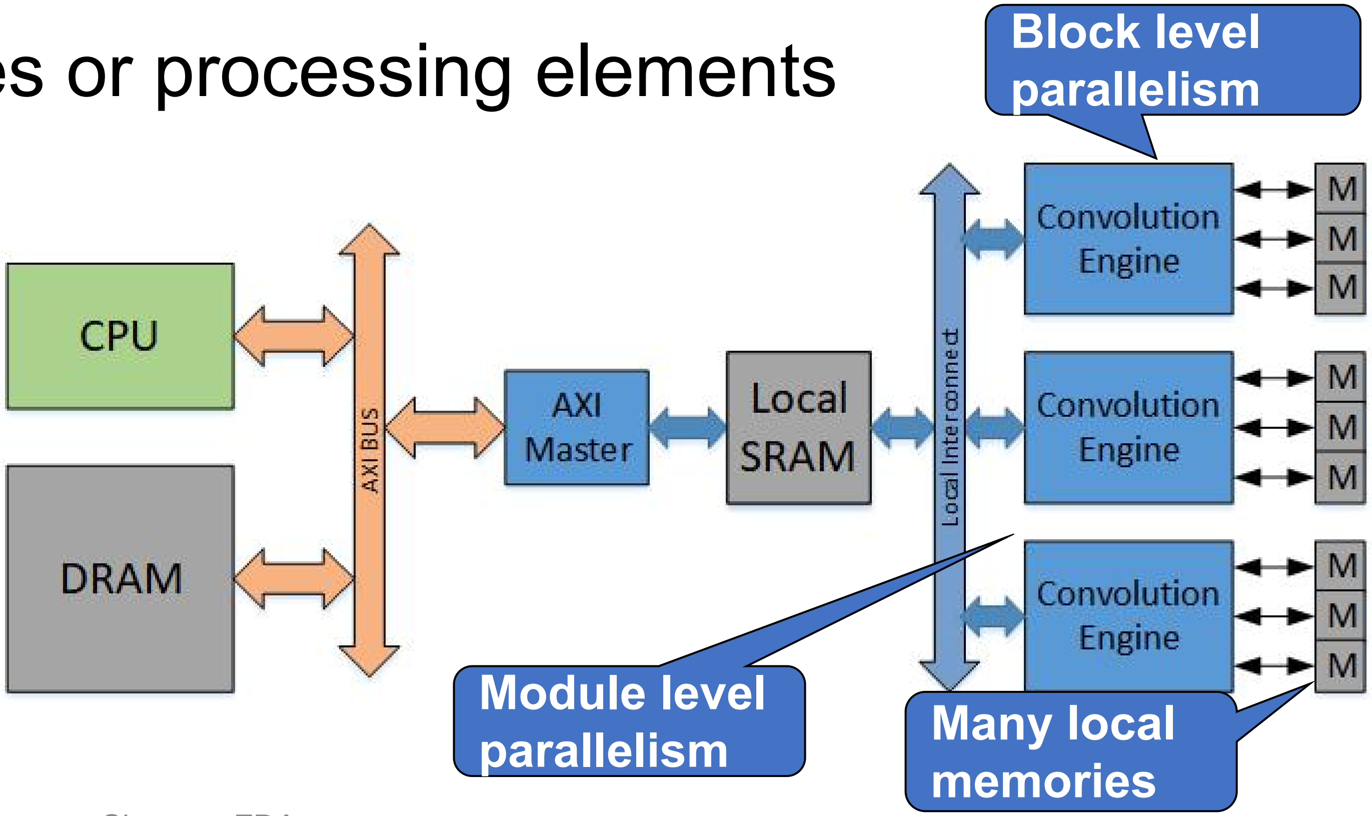
2-d convolution

```
OUT_FMAP:for(int oc=0;oc<OUT_FMAPS;oc++){
  IN_FMAP:for(int ic=0;ic<IN_FMAPS;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];
          }
        }
        if(<last input fmap>)
          fmap_out[oc][r][c] = acc[r][c];
      }
    }
  }
}
```

Lots of data movement for feature maps and weights

CNN Architectural Challenges

- Memory architectures need to leverage data reuse and parallelism
- May have multiple engines or processing elements
  - Block level parallelism
  - Module level parallelism
- Many local memories
- Complex interconnect

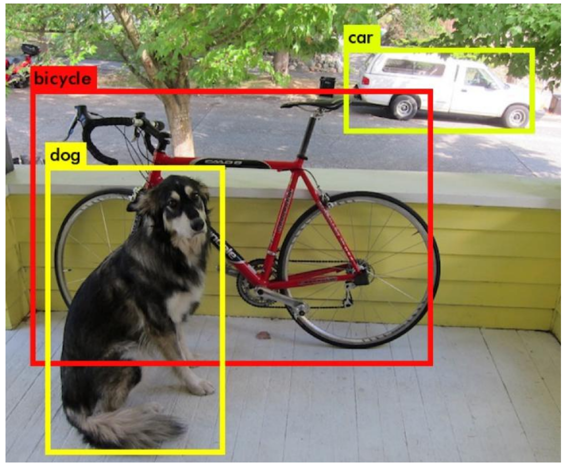


Early Performance Analysis of CNN Convolution

Siemens EDA

Design Goals

- Implement a CNN for object detection and classification
  - 9 layers
  - Mostly 3x3 convolution (9 multiply-acc)
  - 3.5 billion macs/inference
- Low power/performance Ring-doorbell type application
  - 1 inference/sec
- 500MHz clock frequency



Original Algorithmic Model of conv2d

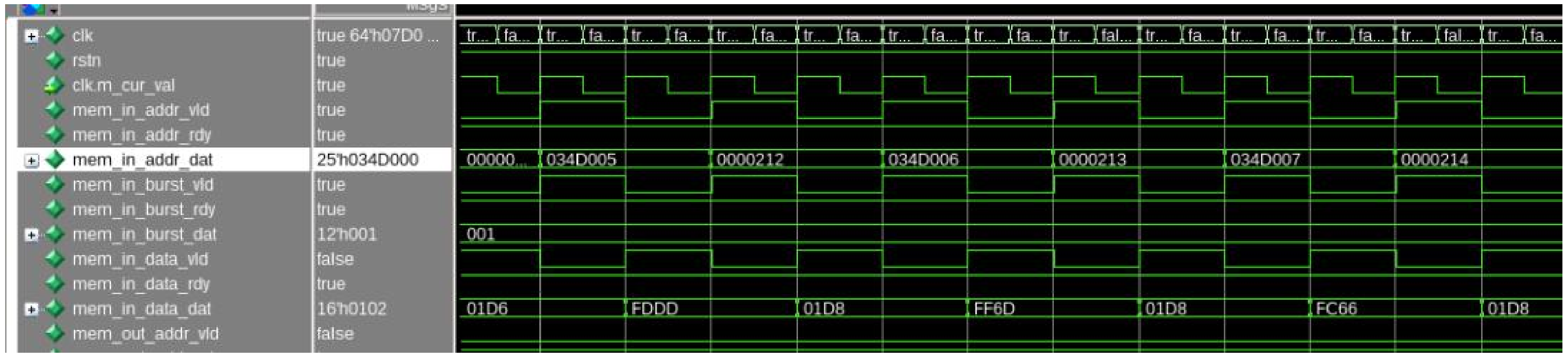
- Direct conversion of algorithm to HLS synthesizable bit-accurate model
- Generic bus interfaces with burst
  - Read burst size limited to one due to non-sequential addressing
  - Writes of feature maps can sustain large burst size
- No opportunity for parallelism

```
OFM:for(int ofm=0; ofm<OUT_FMAP; ofm++) {
  IFM:for (int ifm=0; ifm<IN_FMAP; ifm++) {
    ROW:for (int r=0; r<MAX_HEIGHT; r++) {
      COL:for (int c=0; c<MAX_WIDTH; c++) {
        K_X:for (int kr=0; kr<KSIZE; kr++) {
          K_Y:for (int kc=0; kc<KSIZE; kc++) {
            int ridx = r + kr - KSIZE/2;
            int cidx = c + kc - KSIZE/2;
            <zero pad>
            data_idx=rdoffset+ifm*ht*wt+ridx*wt+cidx;
            mem_in_addr.Push(data_idx);
            mem_in_burst.Push(1);
            data = mem_in_data.Pop();
            <weight read bus transaction>
            acc += data*mem_in_data.Pop();
          }
        }
        acc_buf[r][c] += acc; ...
<Copy feature maps to system memory>
```

Algorithmic Model Results

- SystemC simulation run time took very long (~ 2 hours)
  - Context switching due to non-sequential memory accesses
  - Redundant memory accesses
- Simulation time was ~14 seconds to simulate 1 inference
- Poor design
  - No need to go any further

Pre-HLS Simulation

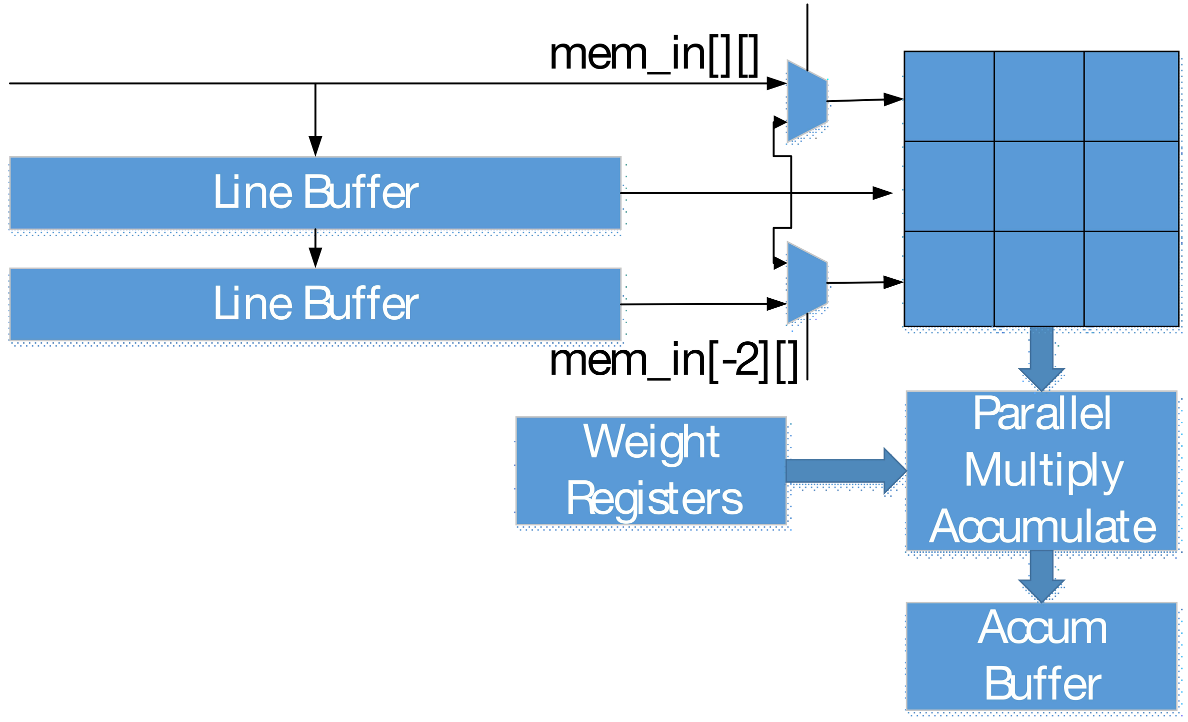


clk	true 64'h07D0 ...
rstn	true
clk.m_cur_val	true
mem_in_addr_vld	true
mem_in_addr_rdy	true
mem_in_addr_dat	25'h034D000
mem_in_burst_vld	true
mem_in_burst_rdy	true
mem_in_burst_dat	12'h001
mem_in_data_vld	false
mem_in_data_rdy	true
mem_in_data_dat	16'h0102
mem_out_addr_vld	false

Architectural Refinement

On-chip Buffering and Windowing

- SystemC designs must be architected for efficient data movement and reuse
  - Improved simulation performance
  - Will allow HLS to extract parallelism
- Sliding-window architecture allows feature map data reuse
- Weight register cache read once for each input/output feature map computation



On-chip Buffering and Windowing

- 9 weight bursts
  - Stored in register cache
- Feature maps burst a row at a time
  - Could also burst entire feature map
- Sliding window architecture allows K_X and K_Y to execute in one clock cylce

```
OFM:for(int ofm=0;ofm<OUT_FMAP;ofm++){
  IFM:for(int ifm=0;ifm<IN_FMAP;ifm++){
    mem_in_addr.Push(weight_idx);
    mem_in_burst.Push(9);
    <cache weights>
    ROW:for(int r=0;r<MAX_HEIGHT+1;r++){
      data_idx=read_offset+ifm*height*width+r*width;
      if(r != height){
        mem_in_addr.Push(data_idx);
        mem_in_burst.Push(width);
      }
      COL:for(int c=0;c<MAX_WIDTH+1;c++){
        if(r != height && c != width)
          data[0] = mem_in_data.Pop();
          <sliding window architecture>
          K_X:for(int kr=0;kr<KSIZE;kr++){
            K_Y:for(int kc=0;kc<KSIZE;kc++){
              acc += window[kr][kc]*weights[kr][kc];
            }
          }
```

On-chip Buffering and Windowing Results

- Simulation results
  - Design goal met with simulation time of 0.864 secs
  - Pre-hls simulation runtime 34 minutes for 1 inference
- All other operations run in software
  - Bias, RELU, max pooling, etc.
  - SystemC testbench runs in zero time
- What can MatchLib and SystemC tell us about the system-level performance?

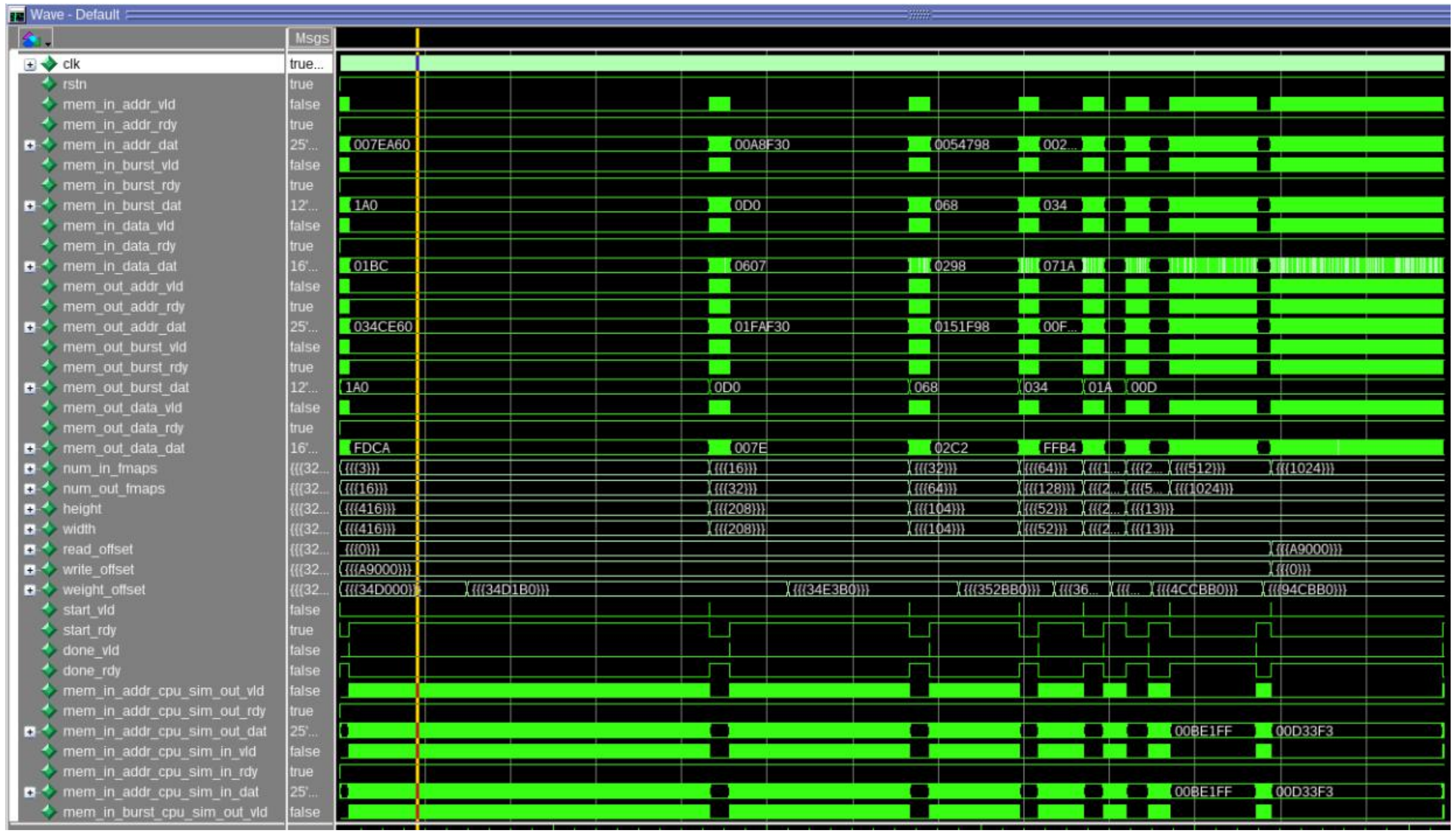
CPU Software Function Calls

```
preprocessing()
setup layer parameters()
start_conv2d()
<HW executing>
wait_for_done()
bias_add();
leakyRelu()
max_pooling()
post_processing()
```

Interaction with the CPU

- Target hardware platform
  - System memory is shared between the CPU and the conv2d accelerator
  - There is no CPU cache
- SystemC testbench models arbitrated memory between CPU and ML accelerator
  - Approximated CPU instruction execution and memory access time
- The performance of the accelerator is throttled by the CPU
  - Simulation took 2.6 secs
  - Simulation runtime 63 minutes
  - Time spent converting from fixed-point to float

SystemC Simulation of One Inference



Siemens EDA

Fusing Computational Layers

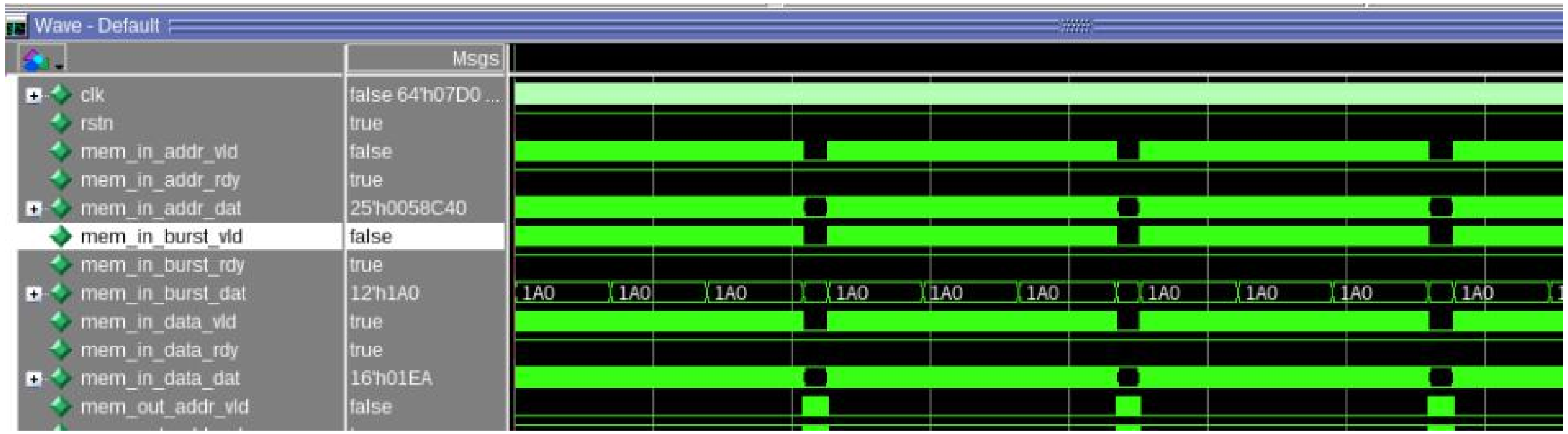
- Move Bias, ReLU, and max pooling into the accelerator
  - Cost little more in hardware area
- Can be coded into the design where feature map data is copied back to system memory
- Design simulates in .9 secs for one inference
- Pre-HLS simulation runtime 30 minutes

```
<Get bias from system memory>
ROW_CPY:for (int r=0; r<MAX_HEIGHT+1; r++) {
  <setup burst size>
  mem_out_addr.Push(out_idx);
  mem_out_burst.Push(burst_size);
  COL_CPY:for (int c=0; c<MAX_WIDTH+1; c++) {
    add_bias = acc_buf[r][c] + bias;
    if (relu)
      if (add_bias < 0)
        add_bias = add_bias * SAT_TYPE(0.1);
    …
    if(pool){
      <max_pooling>
      mem_out_data.Push(max);
    }else
      mem_out_data.Push(add_bias);
}
```

Optimizing for Power

- Memory bus is 100% utilized by the ML accelerator
- Input feature maps are re-read from system memory for each output feature map computation
  - System memory accesses are an order of magnitude larger for power consumption compared to on-chip SRAM

Pre-HLS Simulation



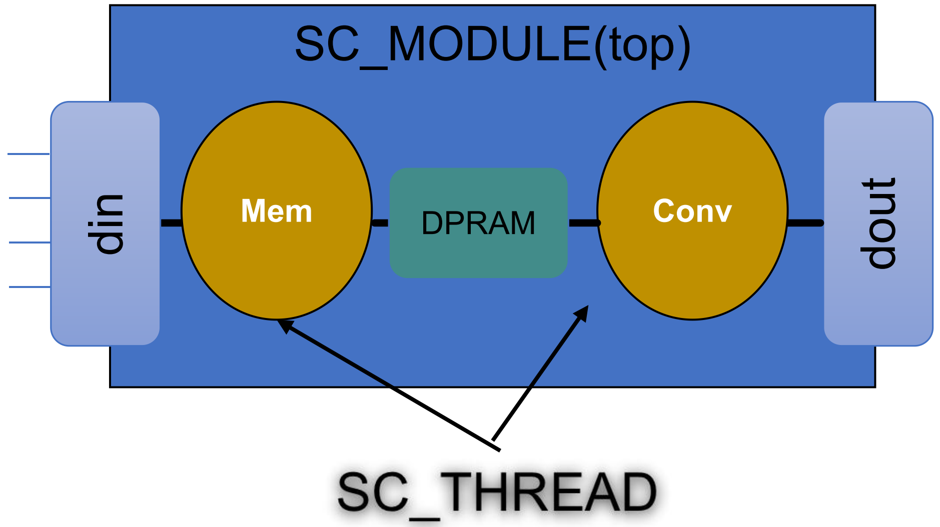
Wave - Default

	Msgs
clk	false 64'h07D0 ...
rstn	true
mem_in_addr_vld	false
mem_in_addr_rdy	true
mem_in_addr_dat	25'h0058C40
mem_in_burst_vld	false
mem_in_burst_rdy	true
mem_in_burst_dat	12'h1A0
mem_in_data_vld	true
mem_in_data_rdy	true
mem_in_data_dat	16'h01EA
mem_out_addr_vld	false

Siemens EDA

Adding On-chip Buffering

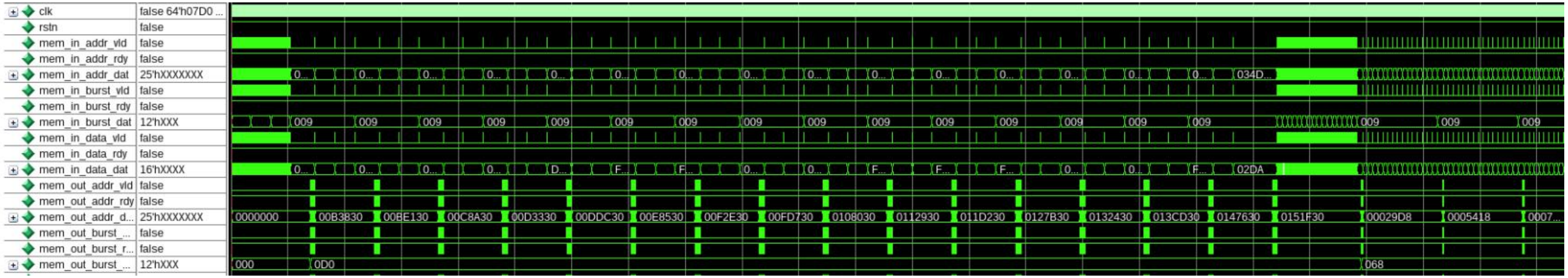
- Buffer feature maps on-chip
  - ~800 KB for full buffering
- Split design into multiple processes
  - Memory read process to access system memory
  - Convolution, bias, ReLu, and max pooling process
  - Shared instantiated SystemC feature map memory between processes



Adding On-chip Buffering

- Simulation finished in .93 secs for one inference
- Simulation runtime was 20 minutes

Pre-HLS Simulation

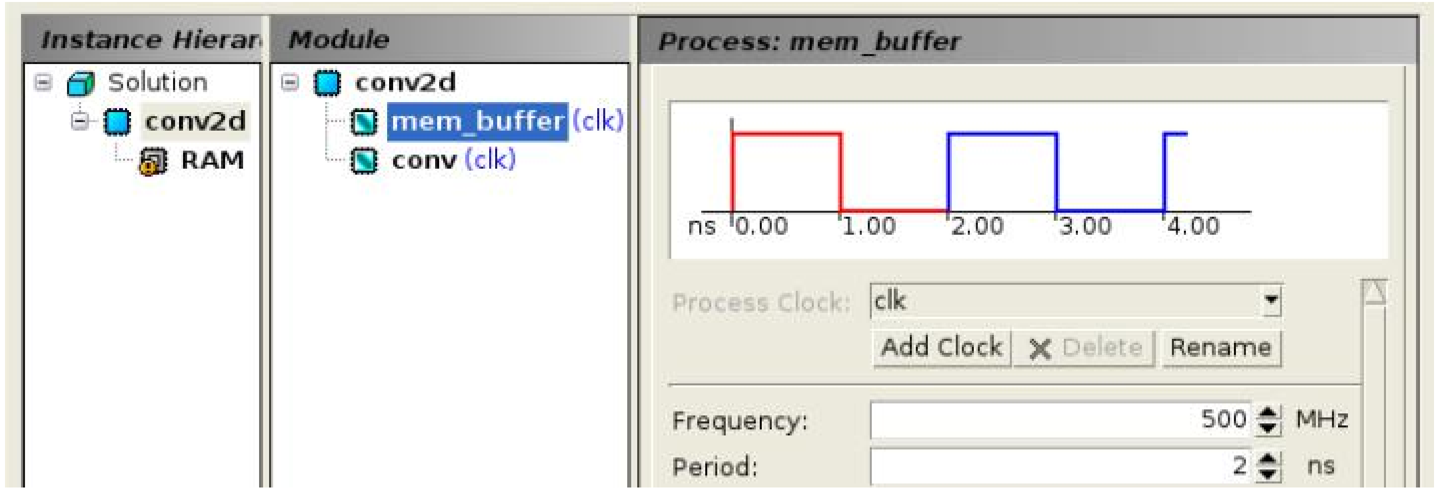


```
clk                  false 64'h07D0 ...
rstn                 false
mem_in_addr_vld      false
mem_in_addr_rdy      false
mem_in_addr_dat      25'hXXXXXXX
mem_in_burst_vld     false
mem_in_burst_rdy     false
mem_in_burst_dat     12'hXXX
mem_in_data_vld      false
mem_in_data_rdy      false
mem_in_data_dat      16'hXXXX
mem_out_addr_vld     false
mem_out_addr_rdy     false
mem_out_addr_d...    25'hXXXXXXX
mem_out_burst_...    false
mem_out_burst_r...   false
mem_out_burst_...    12'hXXX
```

Synthesis and RTL Verification

Constraining the Design in Catapult HLS

- Design targeted a 45nm Catapult sample library
- Catapult Design Mapping
  - Shows the SystemC interconnect memory mapped to a dual-port SRAM
  - Shows the two process, mem_buffer and conv, with a 500MHz clock



Process Clock: clk

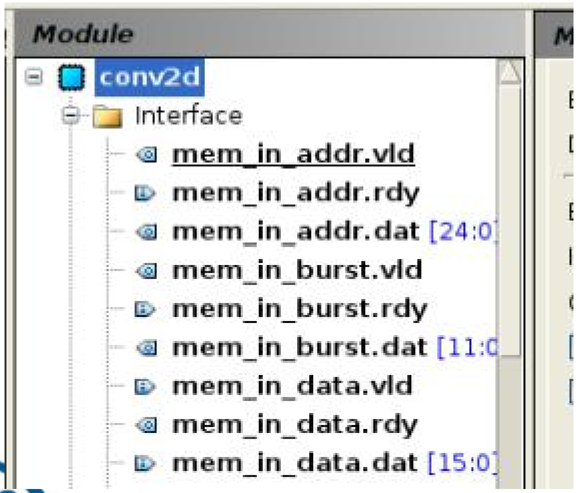
Frequency: 500 MHz

Period: 2 ns

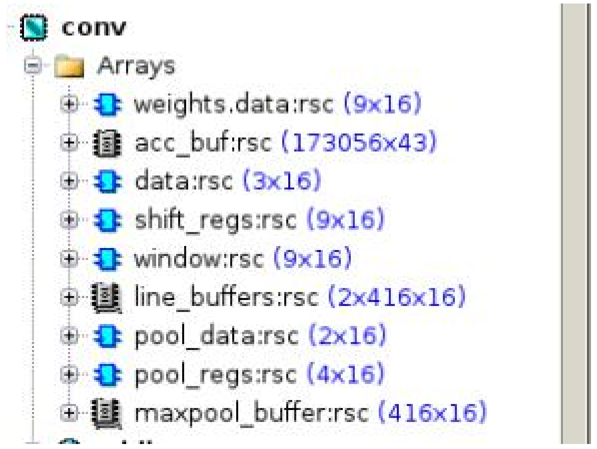
Siemens EDA

Constraining the Design in Catapult HLS

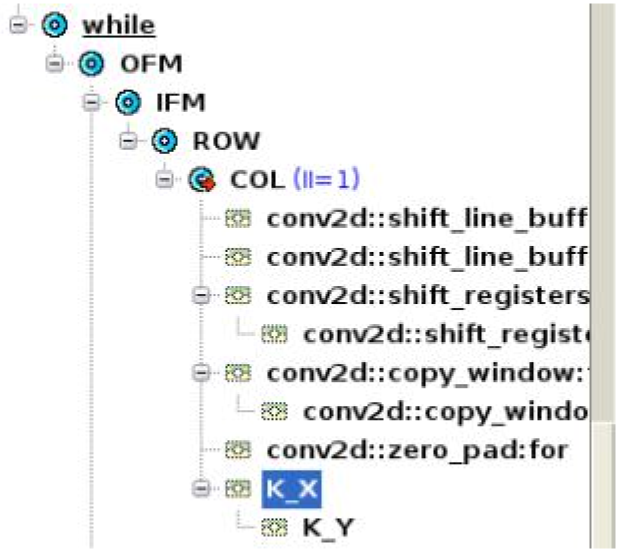
- Catapult Architectural Constraints
  - MatchLib Connections interfaces synthesized to dat/rdy/vld ports
  - Internal arrays in “convolution block” for accum and line buffers mapped to SRAM
  - KX/KY multiply-accumulate loops unrolled for parallel multiplication



```
Module
conv2d
  Interface
    mem_in_addr.vld
    mem_in_addr.rdy
    mem_in_addr.dat [24:0
    mem_in_burst.vld
    mem_in_burst.rdy
    mem_in_burst.dat [11:0
    mem_in_data.vld
    mem_in_data.rdy
    mem_in_data.dat [15:0
```



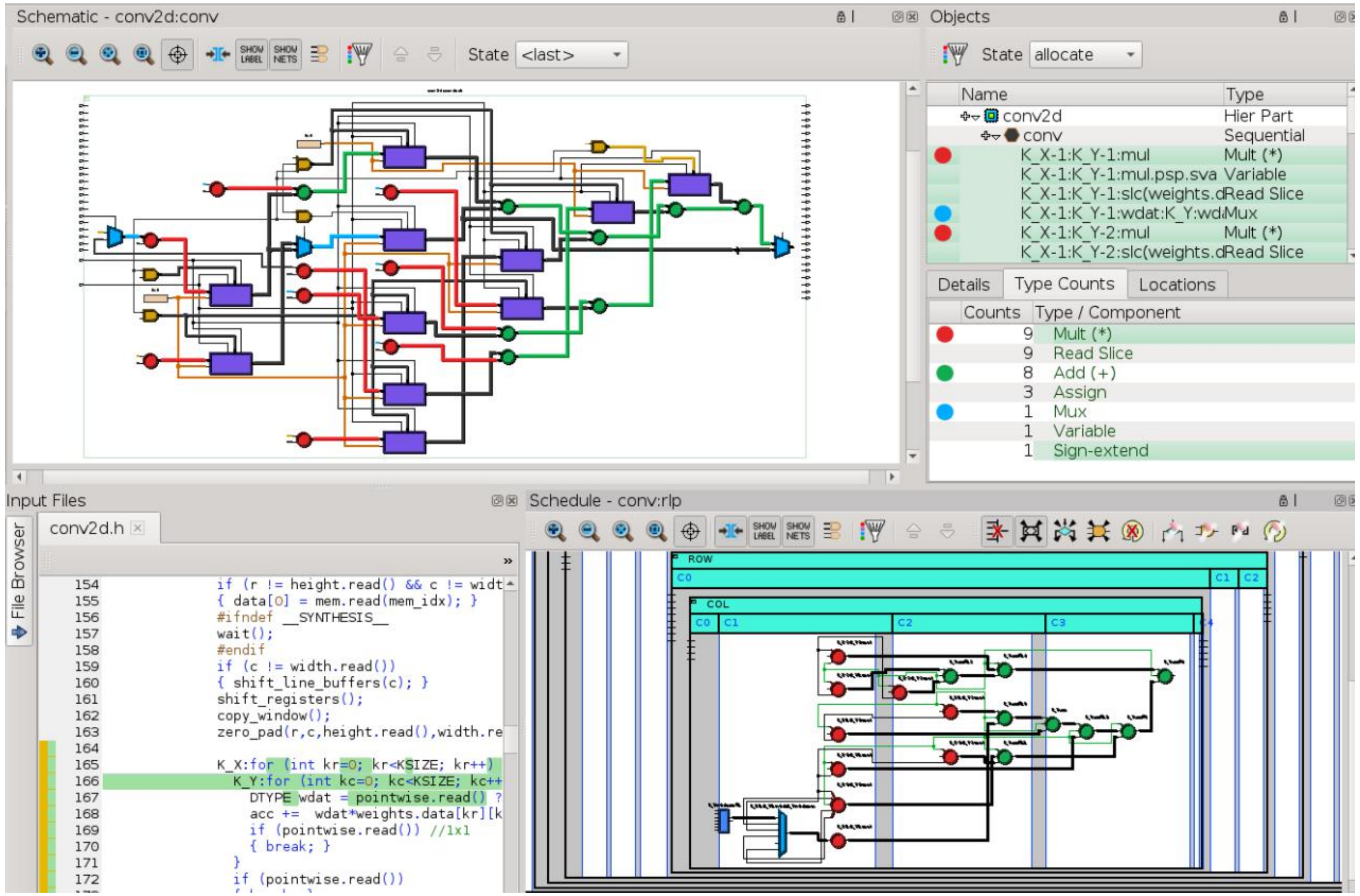
```
conv
  Arrays
    weights.data:rsc (9x16)
    acc_buf:rsc (173056x43)
    data:rsc (3x16)
    shift_regs:rsc (9x16)
    window:rsc (9x16)
    line_buffers:rsc (2x416x16)
    pool_data:rsc (2x16)
    pool_regs:rsc (4x16)
    maxpool_buffer:rsc (416x16)
```



```
while
  OFM
    IFM
      ROW
        COL (II=1)
          conv2d::shift_line_buff
          conv2d::shift_line_buff
          conv2d::shift_registers
            conv2d::shift_regist
          conv2d::copy_window:
            conv2d::copy_windo
          conv2d::zero_pad:for
          K_X
            K_Y
```

Analyzing the Generated Hardware

- Catapult Design Analyzer shows how the SystemC + constraints was synthesized to RTL



	Name	Type
	conv2d	Hier Part
	conv	Sequential
●	K_X-1:K_Y-1:mul	Mult (*)
	K_X-1:K_Y-1:mul.psp.sva	Variable
	K_X-1:K_Y-1:slc(weights.d	Read Slice
●	K_X-1:K_Y-1:wdat:K_Y:wd	Mux
●	K_X-1:K_Y-2:mul	Mult (*)
	K_X-1:K_Y-2:slc(weights.d	Read Slice

Details | Type Counts | Locations

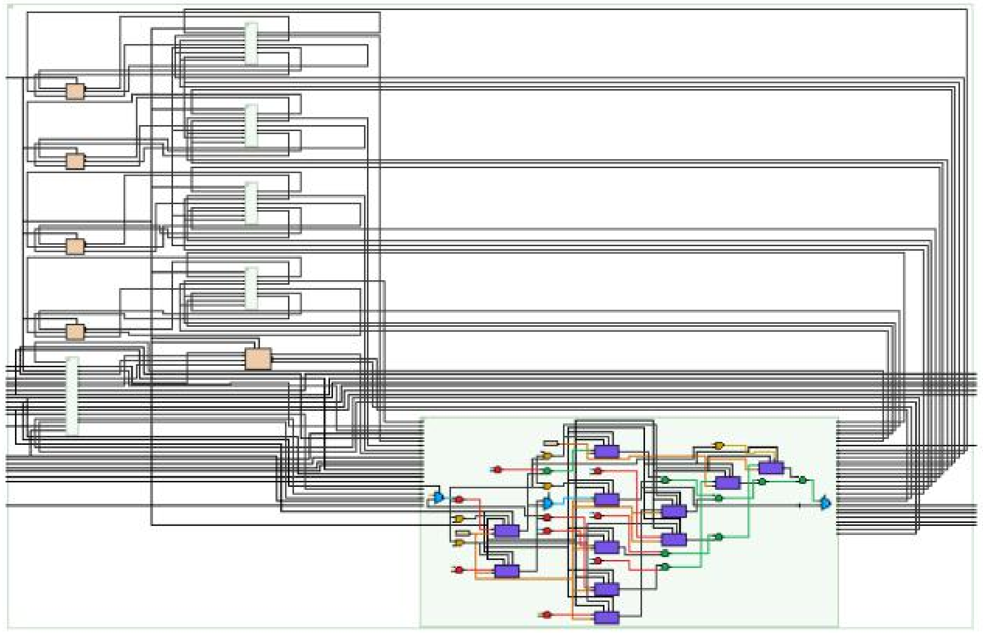
	Counts	Type / Component
●	9	Mult (*)
	9	Read Slice
●	8	Add (+)
	3	Assign
●	1	Mux
	1	Variable
	1	Sign-extend

```
        if (r != height.read() && c != widt
        { data[0] = mem.read(mem_idx); }
        #ifndef __SYNTHESIS__
        wait();
        #endif
        if (c != width.read())
        { shift_line_buffers(c); }
        shift_registers();
        copy_window();
        zero_pad(r,c,height.read(),width.re

        K_X:for (int kr=0; kr<KSIZE; kr++)
          K_Y:for (int kc=0; kc<KSIZE; kc++
            DTYPE wdat = pointwise.read() ?
            acc +=  wdat*weights.data[kr][k
            if (pointwise.read()) //1x1
            { break; }
          }
          if (pointwise.read())
```

Post-HLS Synthesis RTL Simulation Results

- Post-HLS simulation results were very close to the pre-HLS simulation
- Post-HLS simulation runtime was over 30x longer than the pre-HLS simulation
  - Not practical for simulating multiple frames of video



Simulation Type	Simulation Time (secs)	Simulation Runtime (mins)
Pre-HLS	0.93	20
Post-HLS	0.97	630

Next Architectural Refinement Steps

- Increase bus width to take advantage of more parallelism
  - Restructure code for more loop unrolling
  - Process multiple input and output feature maps in parallel
- Rewrite design to use “Loop Tiling” to reduce on-chip buffer requirements
  - Requires additional looping structure
- Rewrite the convolution to use a PE array architecture

Source Code Examples

- Source code examples and other tutorials can be found at:
  - https://hlslibs.org/
  - https://github.com/hlslibs

Customer Case Studies

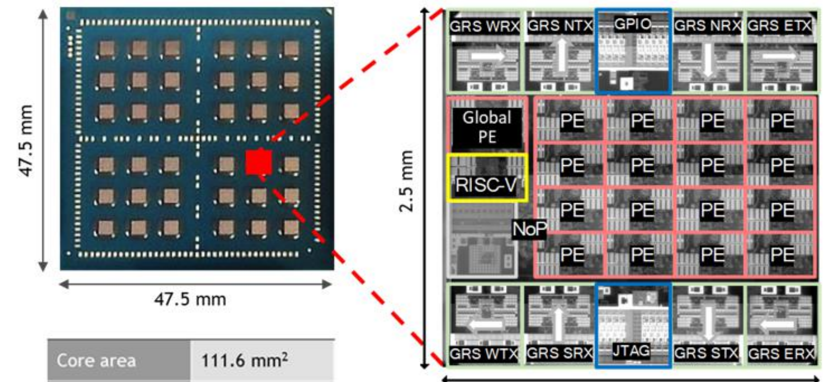
NVIDIA Research – Catapult HLS Key to Optimize AI Inferencing for Performance/Watt

- **AI/ML Inference SoC implemented entirely in C++ with HLS and Catapult**
- Enabled full SoC-level performance verification
  - 30X RTL, <2.6% difference from RTL in cycle count
- Performance/Power and hits the mark
  - 9.5 TOPS/watt in vanilla TSMC 16nm
  - Scales to 128TOPS
- **10X Productivity over manual RTL**
  - Spec-to-Tapeout in 6 months with < 10 engineers

ORDER OF MAGNITUDE LESS DESIGN EFFORT

“The whole RC18 chip was designed by fewer than ten engineers in six months, coded entirely in C++ using high-level synthesis.”

-- Bill Dally, Chief Scientist, NVIDIA
Hot Chips, Aug 2019



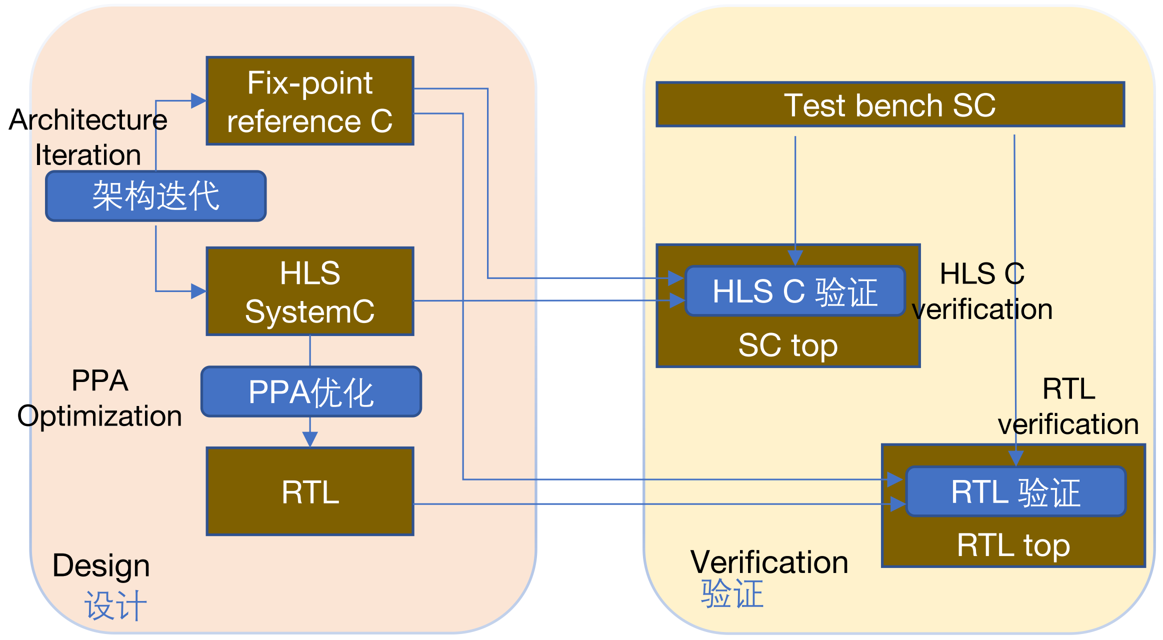
Core area	111.6 mm^2
Voltage	0.52-1.1 V
Frequency	0.48-1.8 GHz

RC18 Testchip

Horizon Robotics uses HLS to shorten the development cycle of Computer Vision algorithm to dedicated IP

- HLS reduced development cycle from 12 months to 6 months over hand-coded RTL
  - Included complete design, architecture and all verification through RTL closure
- HLS delivered PPA equal to hand-coded RTL
- HLS Design Advantages
  - Higher abstraction which greatly reduces coding workload
  - Catapult HLS provides large number of library functions
- HLS Verification Advantages
  - Biggest advantage is ability to compare C reference model with HLS C HW model
  - C level verification can completely solve functional verification
    - RTL is then just scheduling and interface related issues
  - RTL verification and C verification can reuse test stimulus

Horizon Robotics Design and verification process of CV dedicated IP based on HLS



Summary

- Increasing AI/ML algorithm complexity is making RTL verification more difficult
- MatchLib and SystemC allows designers to model and verify the true hardware performance, catching bugs early that would normally be exposed during system integration when it's too late
- MatchLib models can be directly synthesized to RTL and performance of the pre-hls and post-hls results are near identical
- Customers are using MatchLib today to solve the design challenges associated with building AI/ML hardware

Thank You!