Effective SW/HW Co-Design of Specialized ML Accelerators using HLS

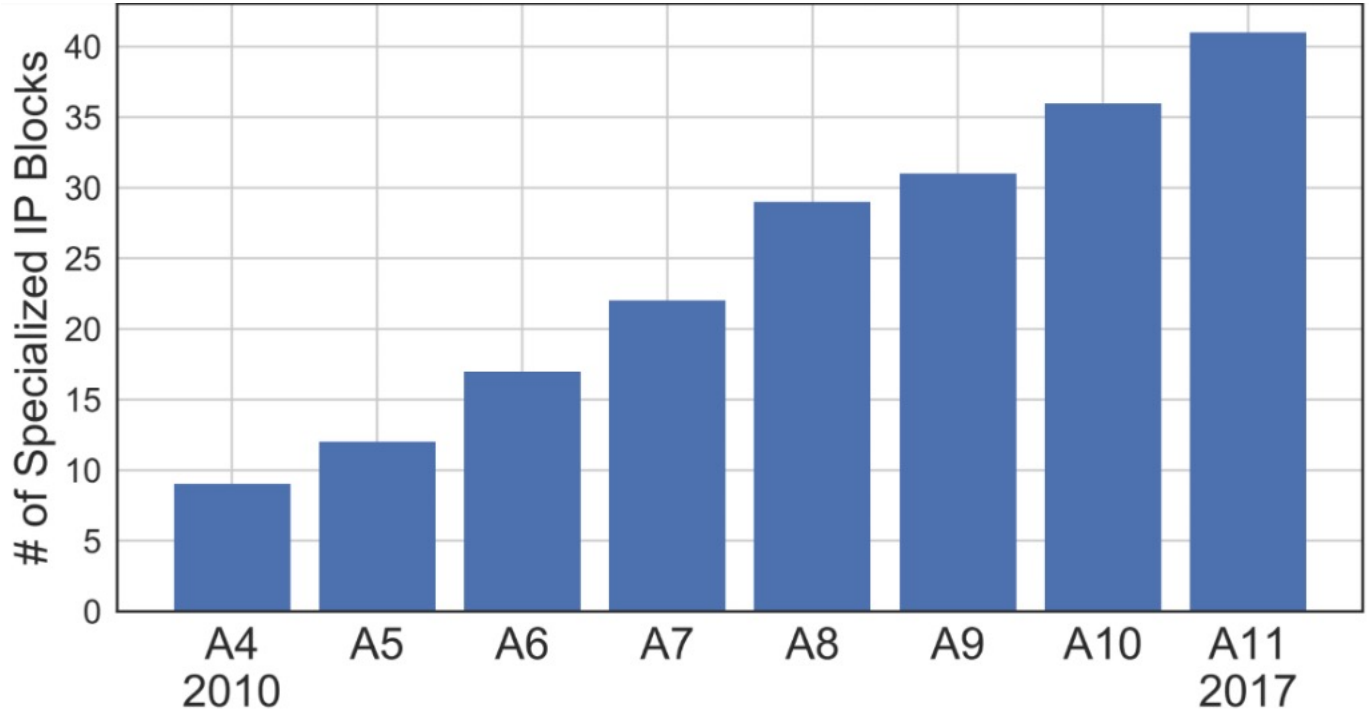
Thierry Tambe, Daniel Yang, Yeongil Ko, Jaylen Wang

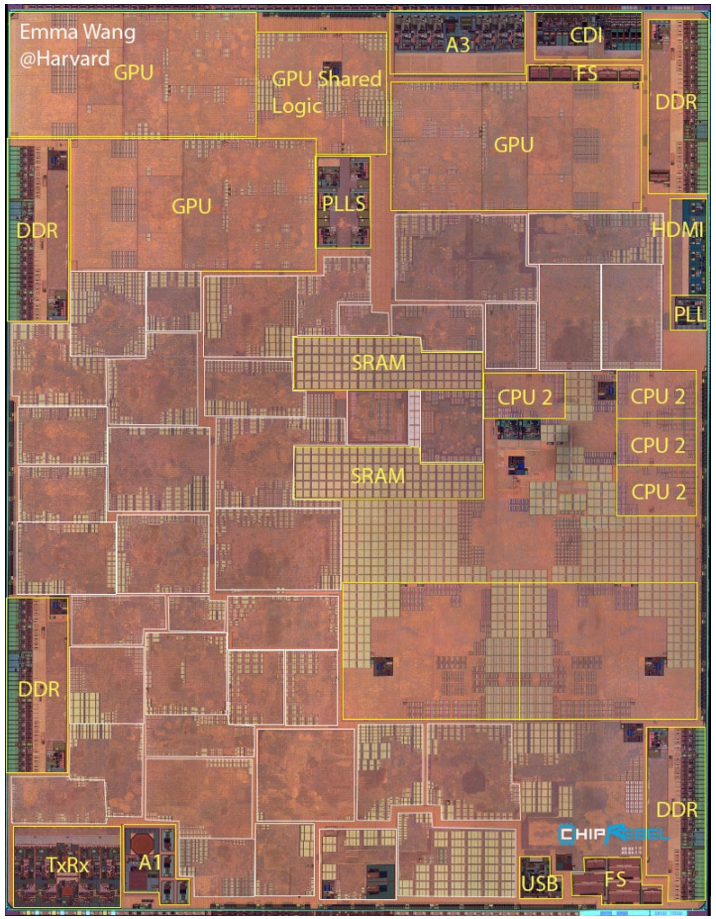
David Brooks, Gu-Yeon Wei

Harvard University

Era of hardware specialization

Source: Sophia Shao - http://vlsiarch.eecs.harvard.edu/research/accelerators/die-photo-analysis/





Apple A11 SoC

Moore's Law Slowdown:

- Number of specialized IP blocks in commercial SoCs has steadily risen over the years.
- Domain-specific hardware accelerators increasingly being used to meet desired power and performance targets.
- Increasingly laborious SoC development efforts

How to improve designer productivity

- Create more portable or reusable IPs
  - Needs to be easy to port across process nodes and platforms
    - ASIC-to-ASIC, ASIC-to-FPGA, FPGA-to-ASIC
  - “Soft” by default to facilitate HW recalibration
    - Hard IP is often too inflexible

How to improve designer productivity

- Create more portable or reusable IPs
  - Needs to be easy to port across process nodes and platforms
    - ASIC-to-ASIC, ASIC-to-FPGA, FPGA-to-ASIC
  - “Soft” by default to facilitate HW recalibration
    - Hard IP is often too inflexible
- Raise the level of abstraction for chip design and verification
  - Quicker evaluation of HW architectural tradeoffs
  - Reduced design and verification cycles

How to improve designer productivity

- Create more portable or reusable IPs
  - Needs to be easy to port across process nodes and platforms
    - ASIC-to-ASIC, ASIC-to-FPGA, FPGA-to-ASIC
  - “Soft” by default to facilitate HW recalibration
    - Hard IP is often too inflexible
- Raise the level of abstraction for chip design and verification
  - Quicker evaluation of HW architectural tradeoffs
  - Reduced design and verification cycles

High-Level Synthesis (HLS)

How to improve designer productivity

- Create more portable or reusable IPs
  - Needs to be easy to port across process nodes and platforms
    - ASIC-to-ASIC, ASIC-to-FPGA, FPGA-to-ASIC
  - “Soft” by default to facilitate HW recalibration
    - Hard IP is often too inflexible
- Raise the level of abstraction for chip design and verification
  - Quicker evaluation of HW architectural tradeoffs
  - Reduced design and verification cycles

High-Level Synthesis (HLS)

↓

Object-Oriented Programming

↓

Help reduce entry barrier for SW / compiler engineers

How to improve designer productivity

- Create more portable or reusable IPs
  - Needs to be easy to port across process nodes and platforms
    - ASIC-to-ASIC, ASIC-to-FPGA, FPGA-to-ASIC
  - “Soft” by default to facilitate HW recalibration
    - Hard IP is often too inflexible
- Raise the level of abstraction for chip design and verification
  - Quicker evaluation of HW architectural tradeoffs
  - Reduced design and verification cycles

High-Level Synthesis (HLS)

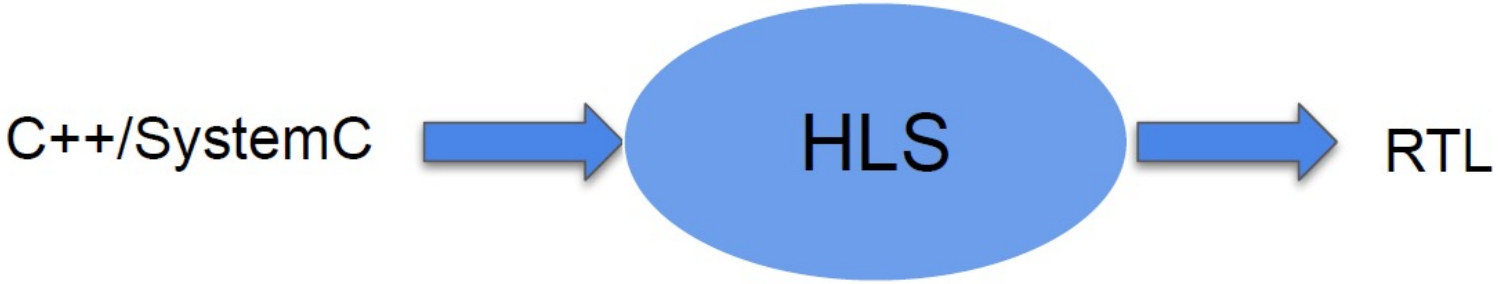
↓ Object-Oriented Programming

↓ Help reduce entry barrier for SW / compiler engineers

↓ HLS tools have vastly matured over the years

↓ Proven in multiple industry and academia chip tapeouts

Observed Benefits



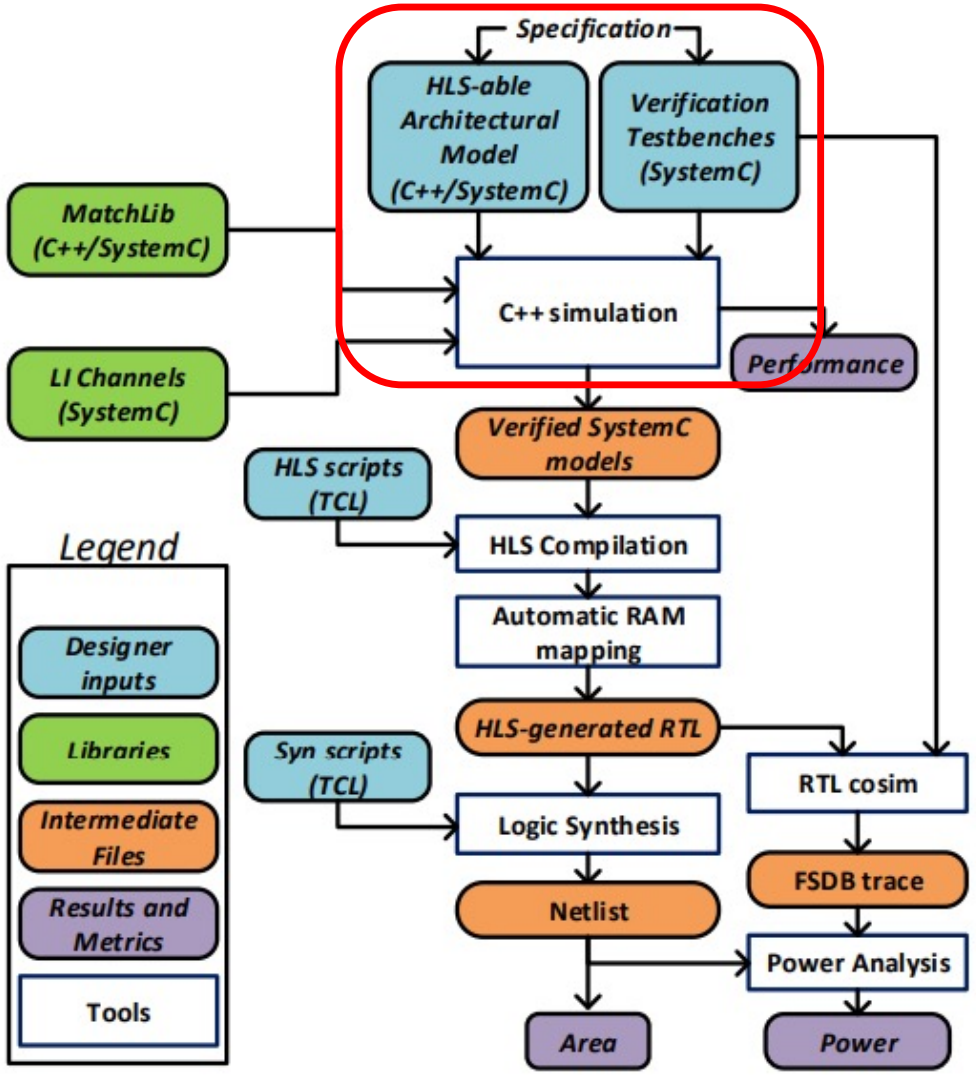
1. High level abstraction of HW and testbenches
   - Typically, much less code to write in C++/SystemC vs. handcrafted RTL
2. Rapid experiments on different design parameters with little overhead
3. High verification throughput:
   - Majority of our bugs are caught before running HLS
   - C++ testbench re-used to verify HLS-generated RTL
4. Enabled tapeout of our research test chips in ~4 months after starting source code development.

Four Main Advantages of HLS

- Modularity
- DSE Facilitation
- Fast Verification
- Easy Handshake with SW

Object-Oriented HLS Design Flow (OOHLS)

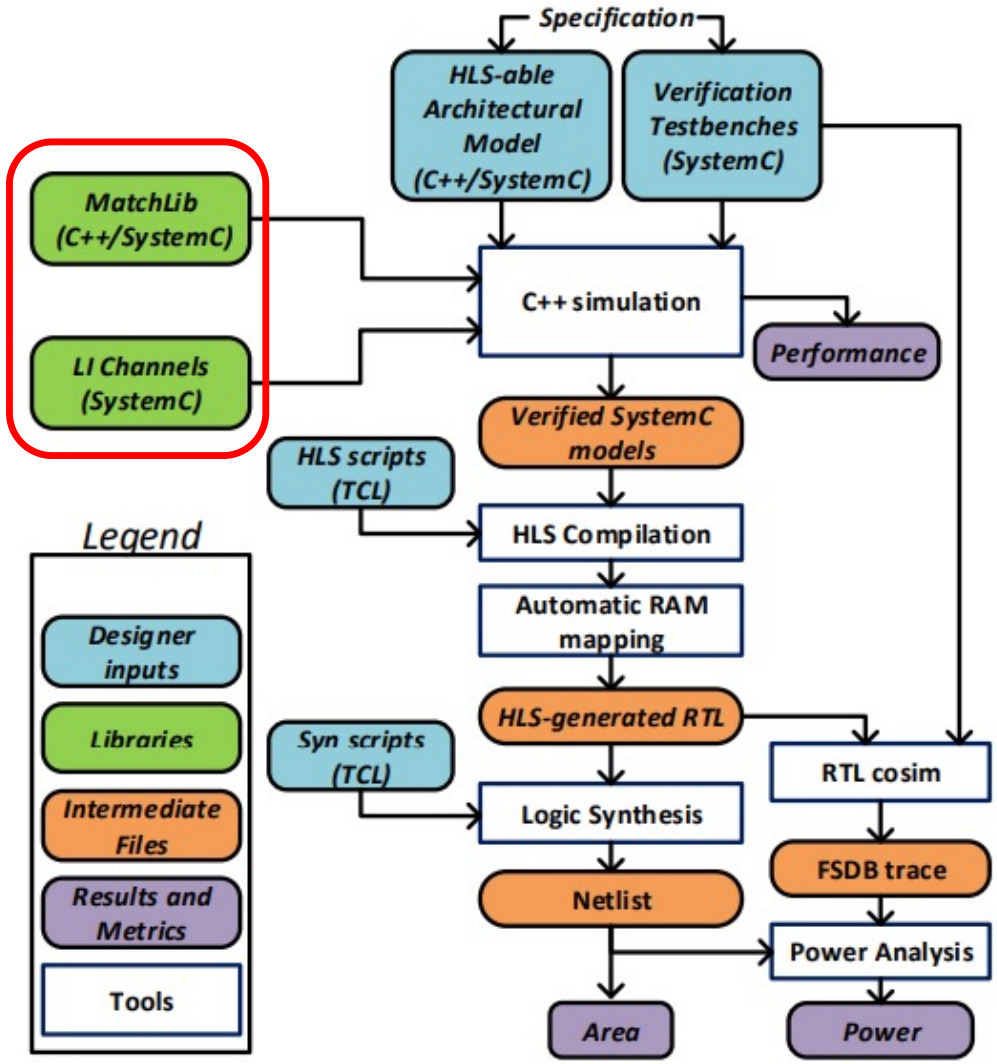
- Design and simulate at high level in C++/SystemC
  - C++ for untimed style abstraction
  - SystemC for “loosely-timed” abstraction
  - Fast C++ simulation



B. Khailany *et al.,* “A modular digital VLSI flow for high-productivity SoC design,” DAC, 2018

Object-Oriented HLS Design Flow (OOHLS)

- Leverage library of commonly-used HW components
- Communicate via Latency-Insensitive channels for flexible timing



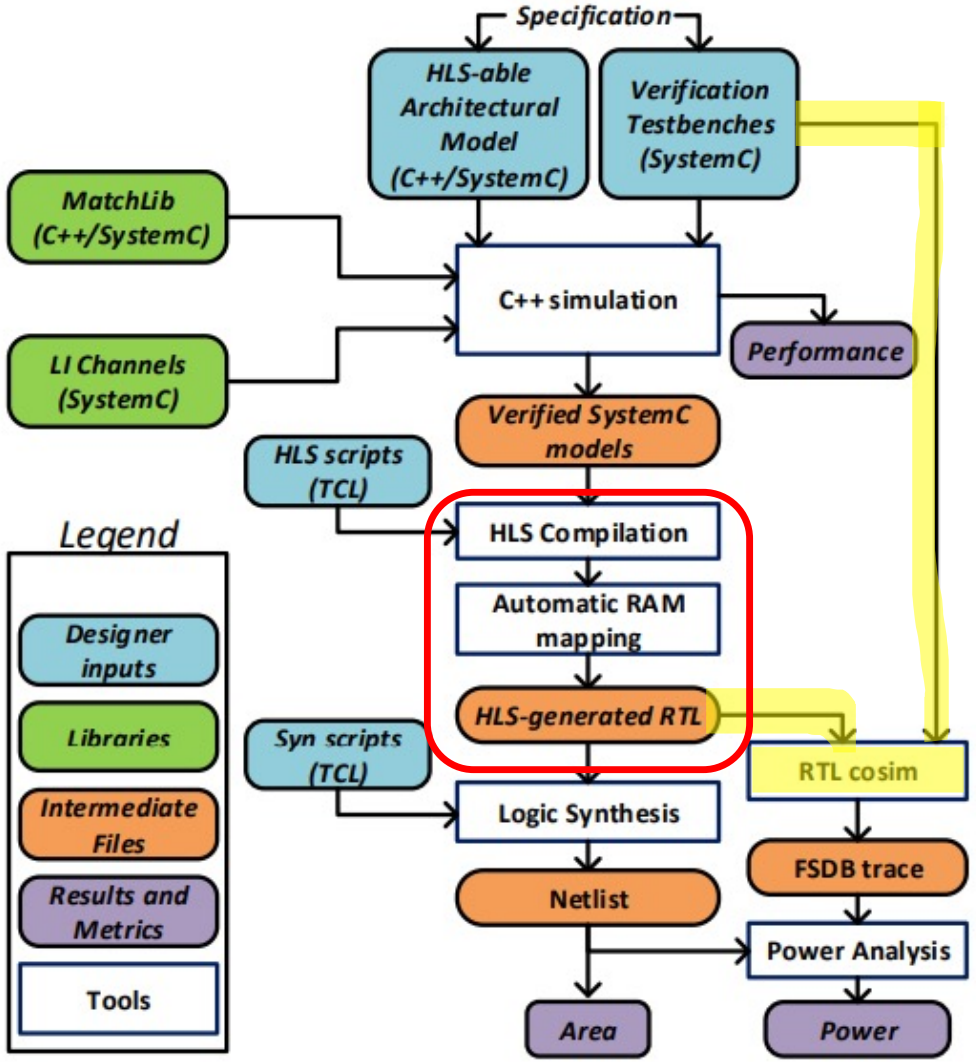
B. Khailany *et al.,* "A modular digital VLSI flow for high-productivity SoC design," DAC, 2018

Object-Oriented HLS Design Flow (OOHLS)

Fast Verification

DSE Facilitation

- HLS tools run compilation, pipelining, and scheduling optimizations to auto-generate RTL
- Reuse of C++/SystemC testbenches to verify HLS-generated RTL
- Cycle-level performance of C++/SystemC description within 3% of RTL



B. Khailany *et al.*, "A modular digital VLSI flow for high-productivity SoC design," DAC, 2018

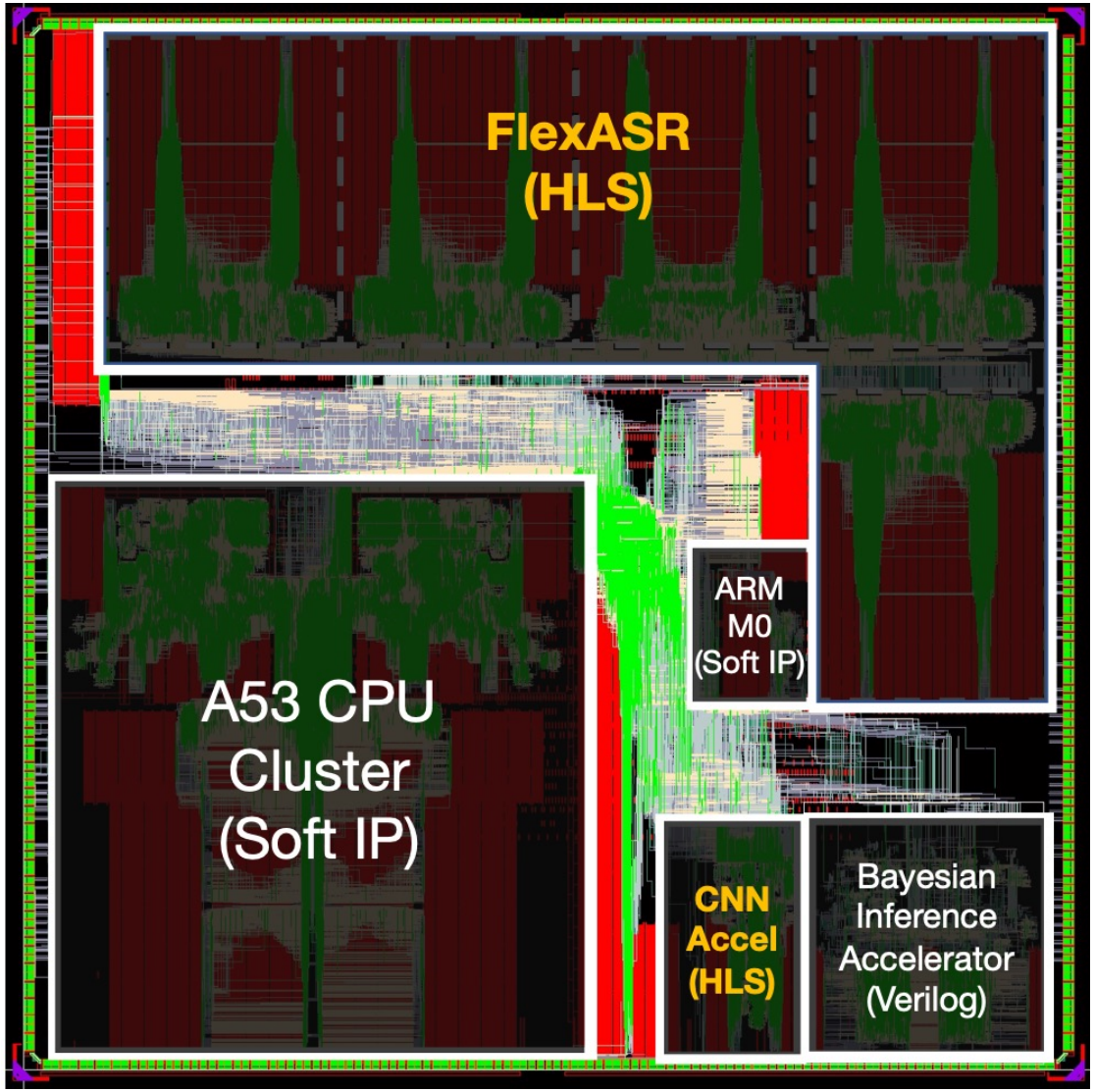
Goal of this talk

- Provide a concrete example of an OOHLS-inspired SW/HW co-design approach proven in many chip tapeouts
  - Logistic details
- Share our HLS experience with Siemens EDA's *Catapult HLS*
  - Challenges we encountered
  - Recommendations for achieving optimal PPA
  - Many learnings remain valid with other HLS tools
- We'll focus on SystemC-based design
  - Designers may code HW in C++ or SystemC depending on needs/objectives
  - C++ style bears high resemblance with SW C++ model (i.e. more abstract than SystemC)
    - May be good choice when datapath isn't too complex
    - Several C++ based designs are shipped with Catapult

Examples of HLS-based designs @ Harvard

- Speech Recognition -- https://github.com/harvard-acc/FlexASR
- Natural Language Processing -- https://github.com/Harvard-acc/EdgeBERT/tree/main/hw
- Systolic / TPU-like
- ConvNets
- Homomorphic Encryption
- Spike Sorting
- 3D Mapping
- Crypto Mining

We taped out SoCs with integrated HLS-generated IP



- HLS allowed us to explore, and quickly iterate on the main differentiating features/blocks of SoC.
- Synthesizable interfaces (AXI or custom) available to merge HLS blocks with rest of the SoC.

T. Tambe et al., A 25mm^2 SoC for IoT Devices with 18ms Noise Robust Speech-to-Text Latency via Bayesian Speech Denoising and Attention-Based Sequence-to-Sequence DNN Speech Recognition in 16nm FinFET. ISSCC, 2021.

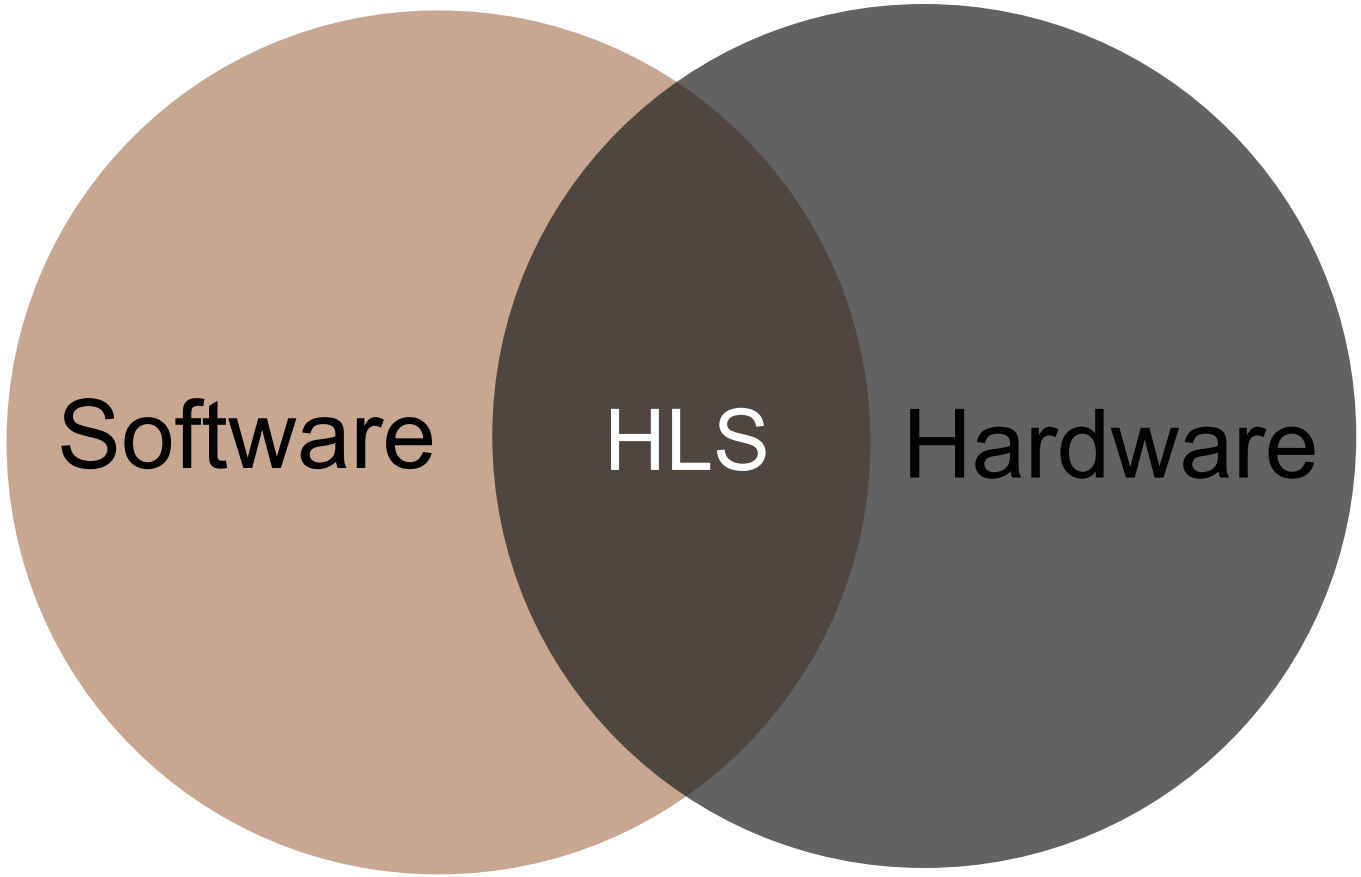
Outline

- Background and Motivation
- SW / HW Co-Design and Co-Verification Flow
- Our HLS experience
- Recommendations

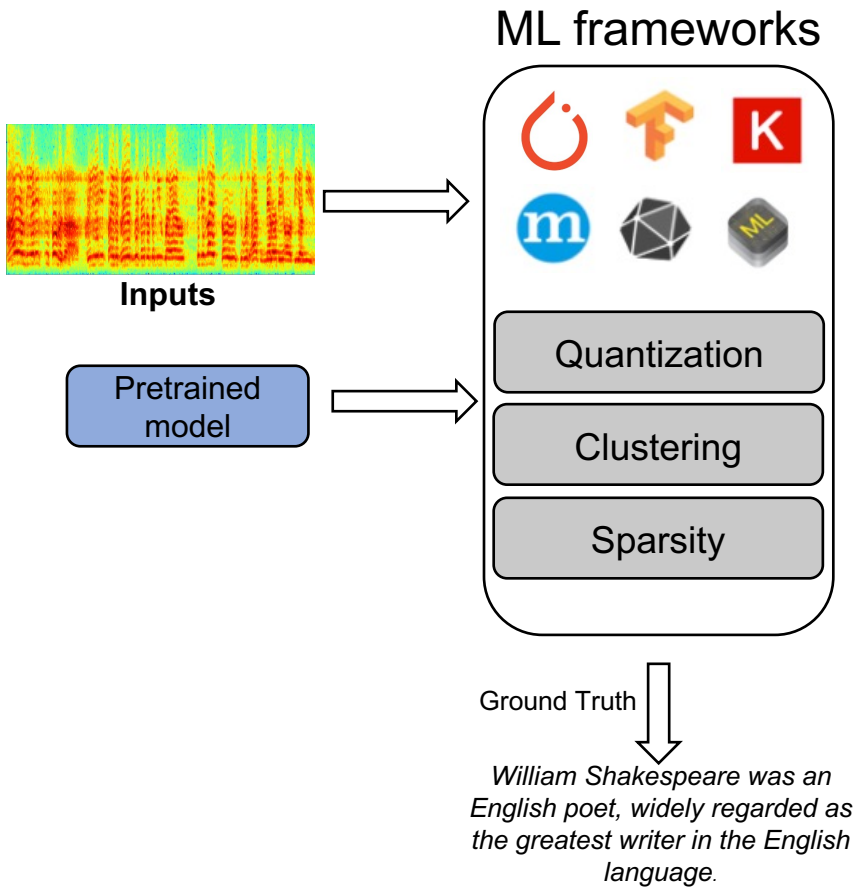
Outline

- Background and Motivation
- **SW / HW Co-Design and Co-Verification Flow**
- Our HLS experience
- Recommendations

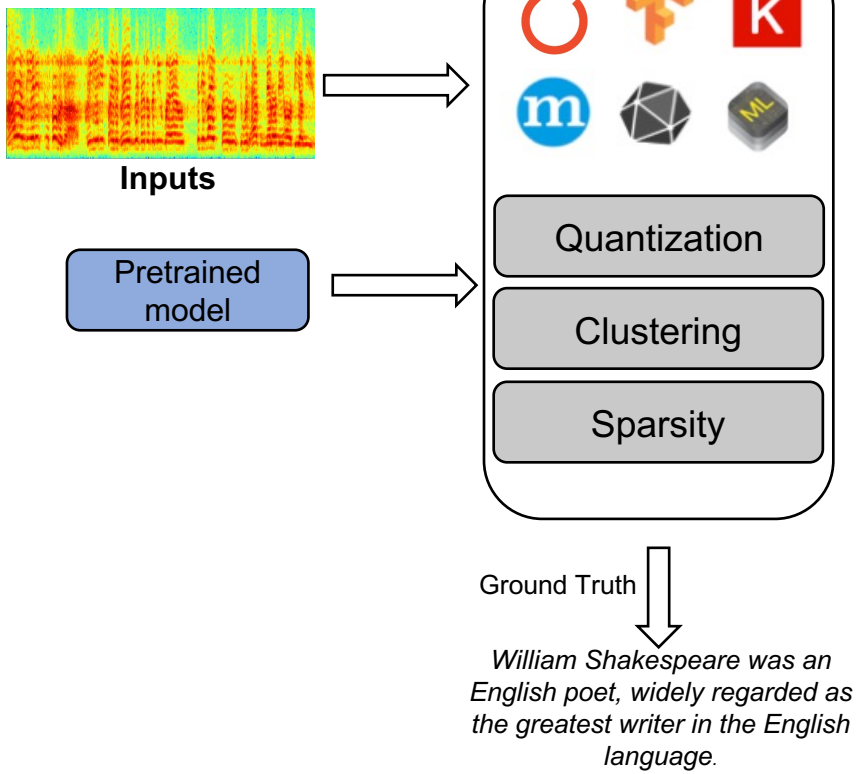
Closing the *SW/HW Co-Design Loop* with HLS



Develop models using ML frameworks



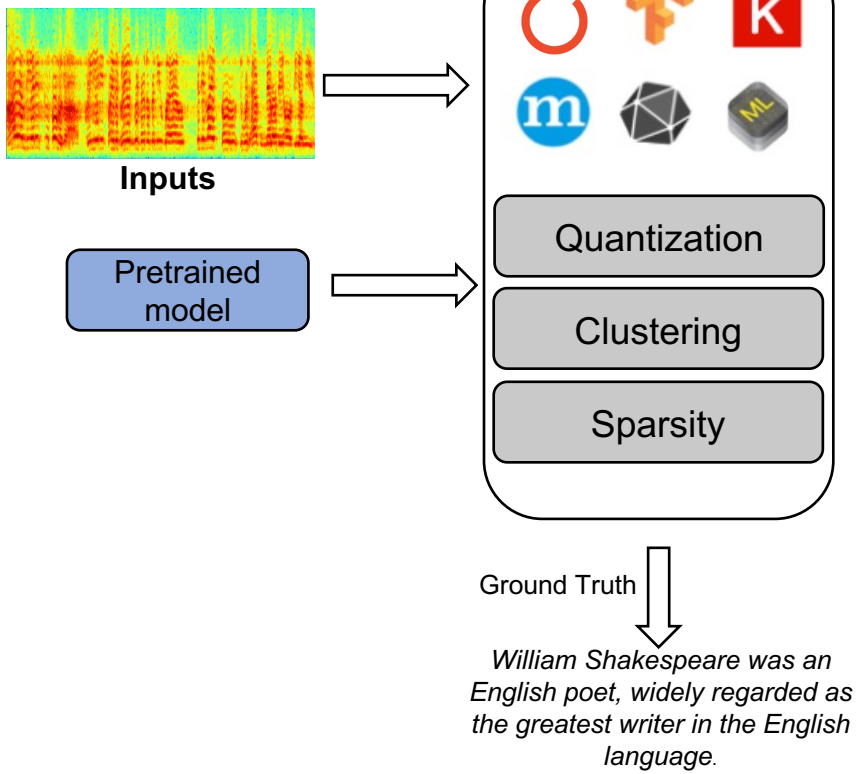
Model the application in hardware

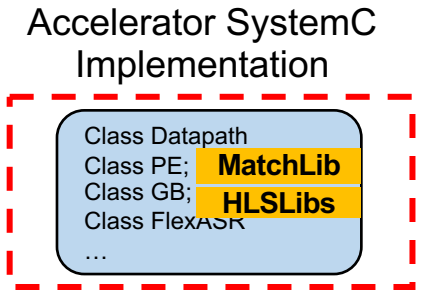


Accelerator SystemC Implementation

Class Datapath
Class PE
Class GB
Class FlexASR
…

HW libraries save design time





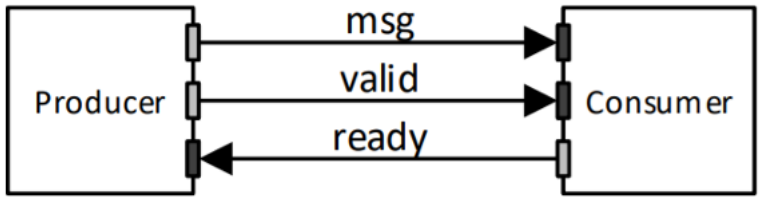
Save time by using already available synthesizable components!

MatchLib – A Synthesizable HW Library

https://github.com/NVlabs/matchlib

We extensively use:

- Latency Insensitive Channels
- Matchlib Integers
- Arbitrated Scratchpad
- AXI Interface



Latency insensitive connection channel

▼ Stateless datapath blocks - implemented as functions	
Nvhls_int	Integer library with built-in support for sc_int and ac_int datatypes
Crossbar	Configurable nxn crossbar datapath
One_hot_to_bin	One-hot to binary convertor
▼ Loosly-timed units with state - implemented as classes	
Arbiter	Configurable n inputs arbiter
MemArray	Abstract Memory Class
FIFO	Configurable FIFO class
Nvhls_vector	Vector helper container with vector operations
Connections	Modular IO supporting latency-insensitive channels
ArbitratedCrossbar	Crossbar with conflict arbitration and queuing
ArbitratedScratchpad	Scratchpad memories with arbitration and queuing
ReorderBuffer	Out-of-order writes into queue, in-order reads
▼ Timed units - implemented as sc_module	
WHVCRouter	Wormhole router with virtual channels
SerDes	N-bit packets to/from M cycles of (N/M)-bit packets
Cache	Direct-mapped Cache design
Scratchpad	Banked Memory Array with Crossbar
FlitMplex	Mux multiple input channels to single output channel
FlitDeMplex	DeMux single input channel to one of multiple output channels
AXI	Master/Slave Interfaces & bridges for AXI interconnect
▼ Misc utilities and auxiliary constructs	
TypeToBits	Convert any datatype to bit-vector
Nvhls_packet	Configurable packet and flit classes
Comptrees	Compile-time minmax tree
DebugPrint	Debug print statements
Nvhls_array	A variant of synthesizable array implementation
Assertions	Macros for synthesizable and non-synthesizable assertions
Marshaller	Marshaller is used to automatically convert types to logic vector and vice versa
StaticMax	StaticMax Class: returns the larger value between two unsigned integers
BitUnion2	BitUnion2 class: A union class to hold two Marshallers
Nvhls_module	Matchlib Module class: a wrapper of sc_module with tracing and stats support
Tracer	Tracer class to dump simulation stats to an output stream
NVHLSVerify	Verification co-simulation support
▼ Non-synthesizable components suitable for testbench construction	
Pacer	Random stall generator
Set_random_seed	Set random seed
Gen_random_payload	Generate Random payload of any type
Get_rand	Generate Random integer value of desired width

HLSLibs Synthesizable Math Functions

https://github.com/hlslibs/

FUNCTION TYPE	FUNCTION CALL
ABSOLUTE VALUE	ac_abs()
DIVISION	ac_div()
NORMALIZATION	ac_normalize()
RECIPROCAL	ac_reciprocal_pwl()
LOGARITHM BASE E	ac_log_pwl()
	ac_log_cordic()
LOGARITHM BASE 2	ac_log2_pwl()
	ac_log2_cordic()
EXPONENT BASE E	ac_exp_pwl()
	ac_exp_cordic()
EXPONENT BASE 2	ac_pow2_pwl()
	ac_exp2_cordic()
SQUARE ROOT	ac_sqrt_pwl()
	ac_sqrt()
INVERSE SQUARE ROOT	ac_inverse_sqrt_pwl()
SINE/COSINE	ac_sincos()
	ac_cos_cordic()
	ac_sin_cordic()
	ac_sincos_cordic()
TANGENT	ac_tan_pwl()
INVERSE TRIG	ac_atan_pwl()
	ac_arccos_cordic()
	ac_arcsin_cordic()
	ac_arctan_cordic()
SHIFT LEFT/RIGHT	ac_shift_left
	ac_shift_right
HYPERBOLIC TANGENT	ac_tanh_pwl()
SIGMOID	ac_sigmoid_pwl()
SOFTMAX	ac_softmax_pwl()

LSTM RNN

$$
\begin{aligned}
i_t &= \sigma(W_{ii}x_t + b_{ii} + W_{hi}h_{t-1} + b_{hi}) \\
f_t &= \sigma(W_{if}x_t + b_{if} + W_{hf}h_{t-1} + b_{hf}) \\
g_t &= \tanh(W_{ig}x_t + b_{ig} + W_{hg}h_{t-1} + b_{hg}) \\
o_t &= \sigma(W_{io}x_t + b_{io} + W_{ho}h_{t-1} + b_{ho}) \\
c_t &= f_t \odot c_{t-1} + i_t \odot g_t \\
h_t &= o_t \odot \tanh(c_t)
\end{aligned}
$$

ac_fixed
ac_tanh_pwl
ac_sigmoid_pwl

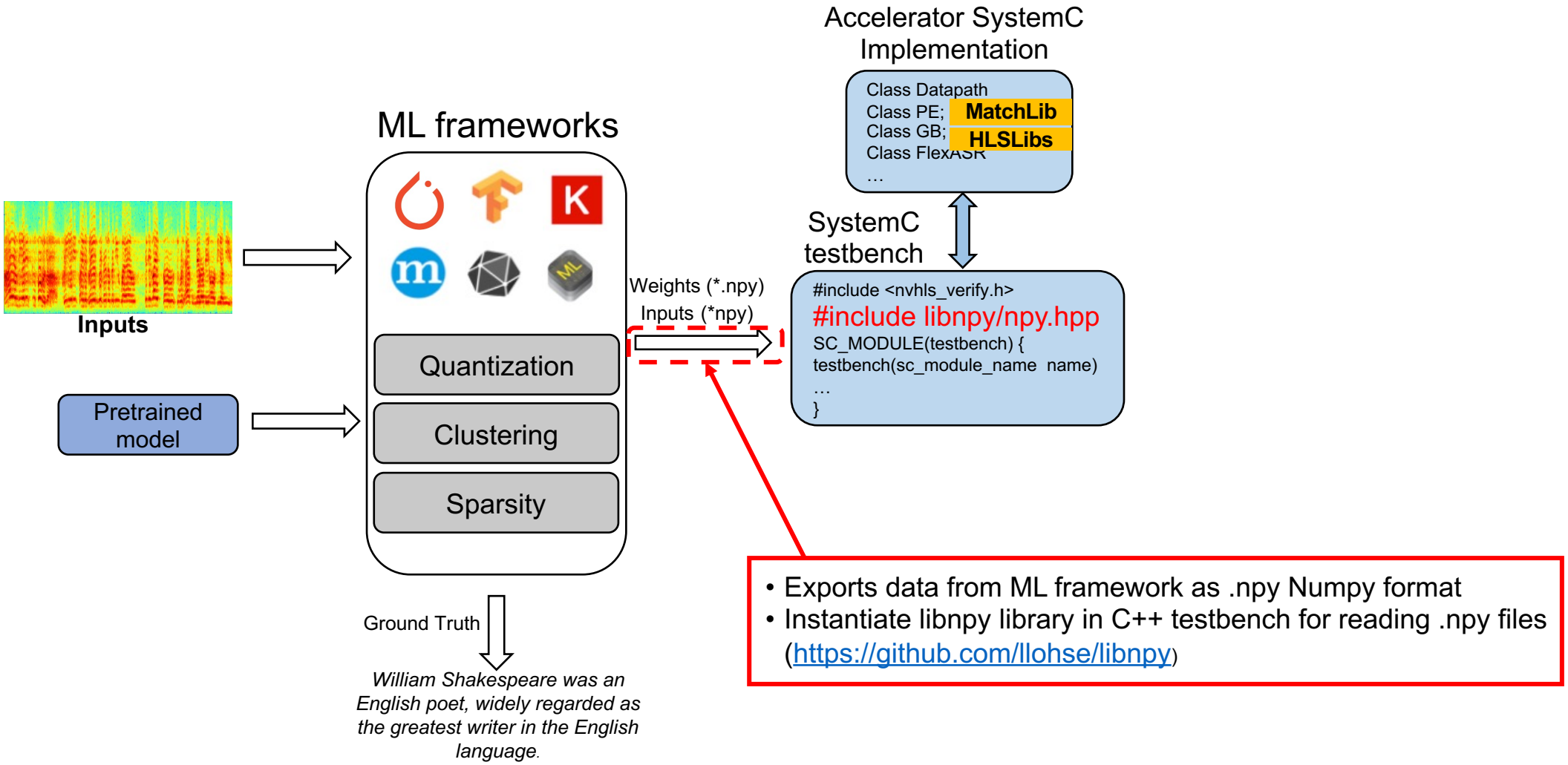
LayerNorm

$$y = \frac{x - E[x]}{\sqrt{Var[x] + \epsilon}} * \gamma + \beta$$

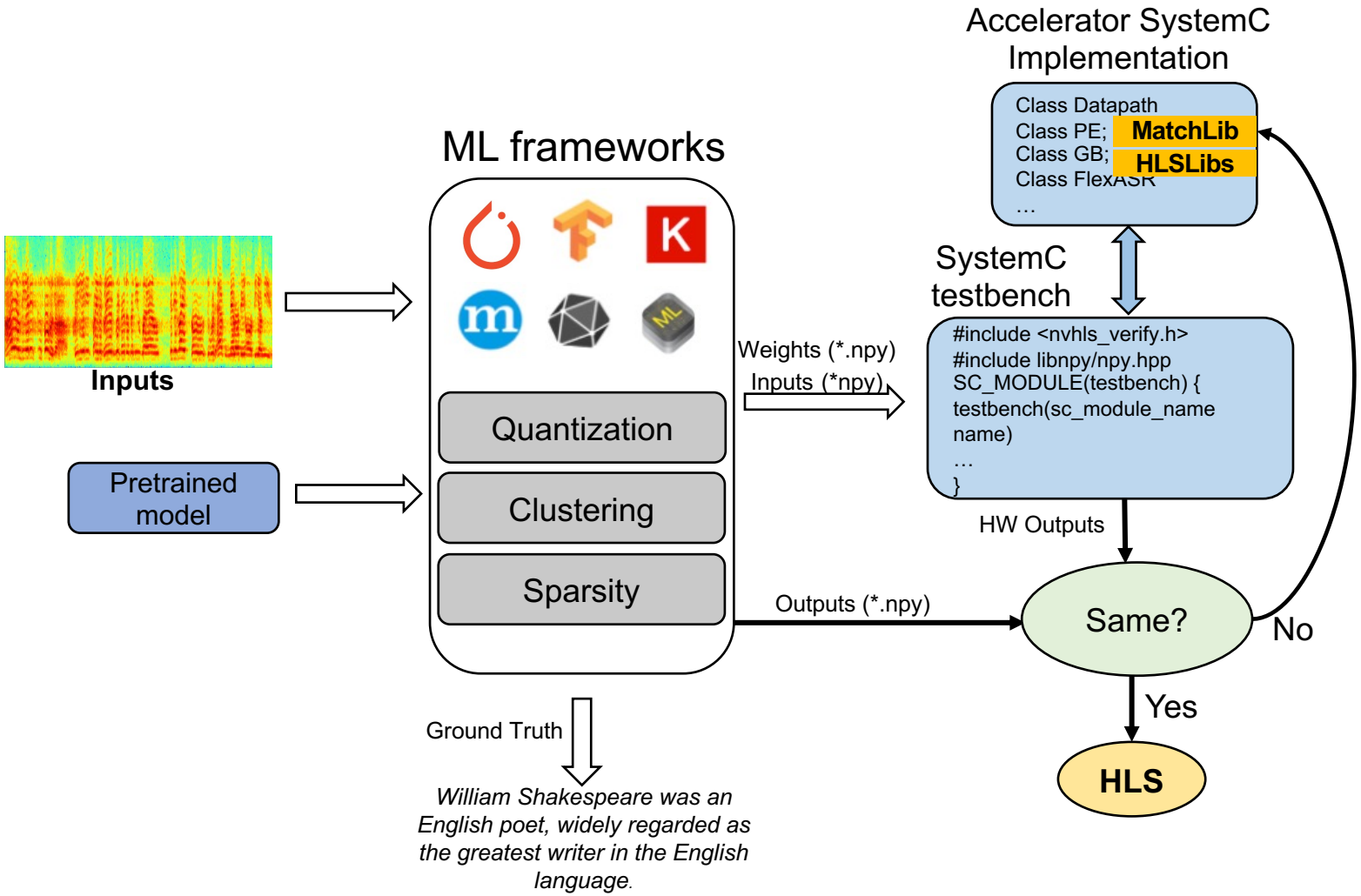
ac_fixed
ac_inverse_sqrt_pwl

Export ML data to SystemC testbench

Easy Handshake with SW



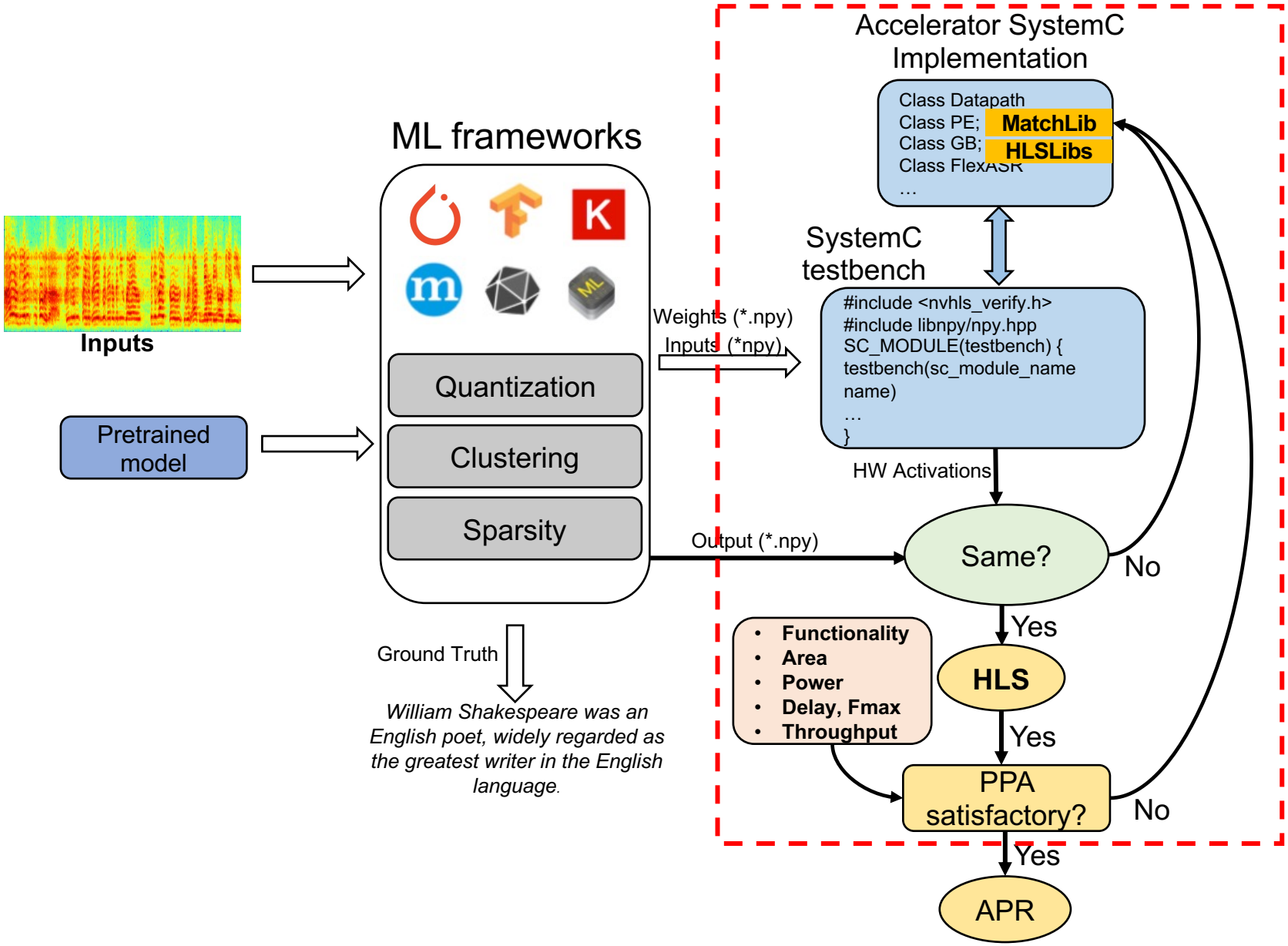
Check HW against SW output



Iterate design until PPA is met

Fast Verification

DSE Facilitation



Looped inside Catapult HLS environment

Tunable HW Parameters

- # of PEs
- MAC vector size
- Scratchpad size
- Datapath precision
- # of pipeline stages

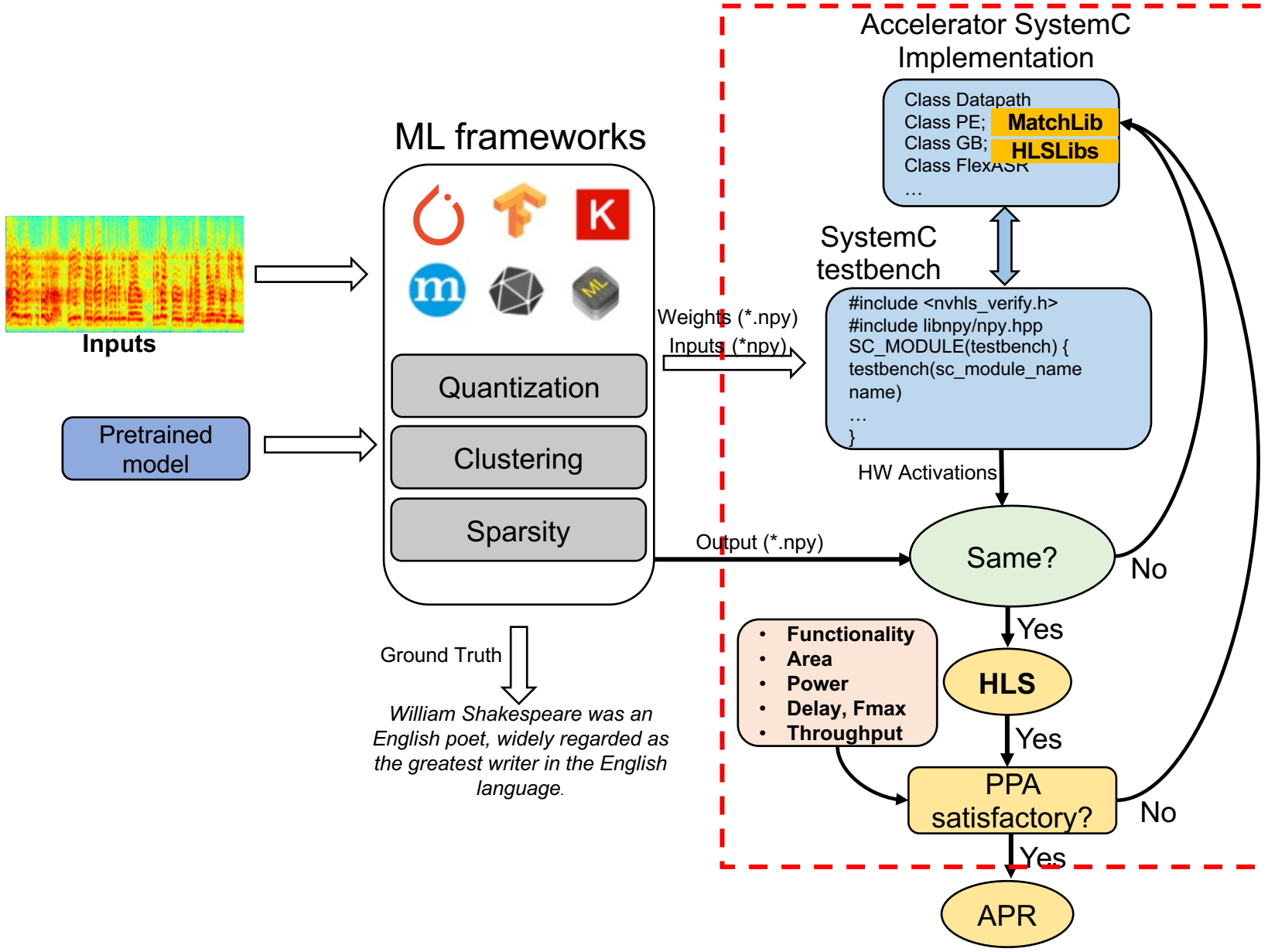
SW / HW Co-Design and Co-Verification Flow

Modularity

Fast Verification

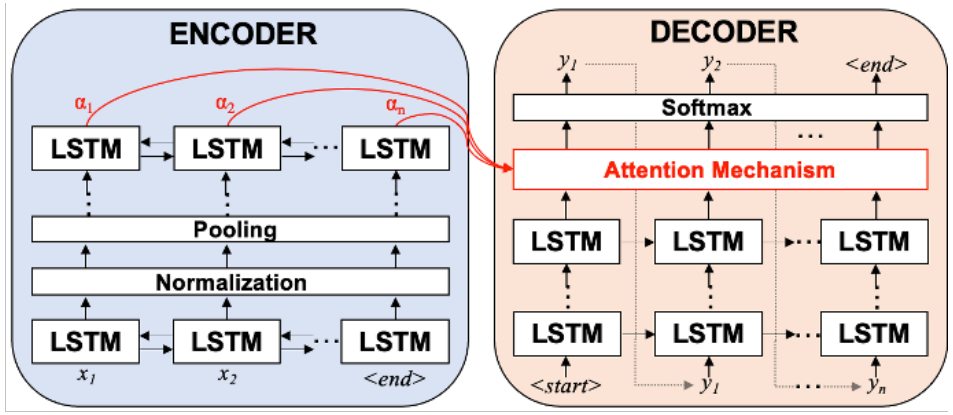
DSE Facilitation

Easy Handshake with SW

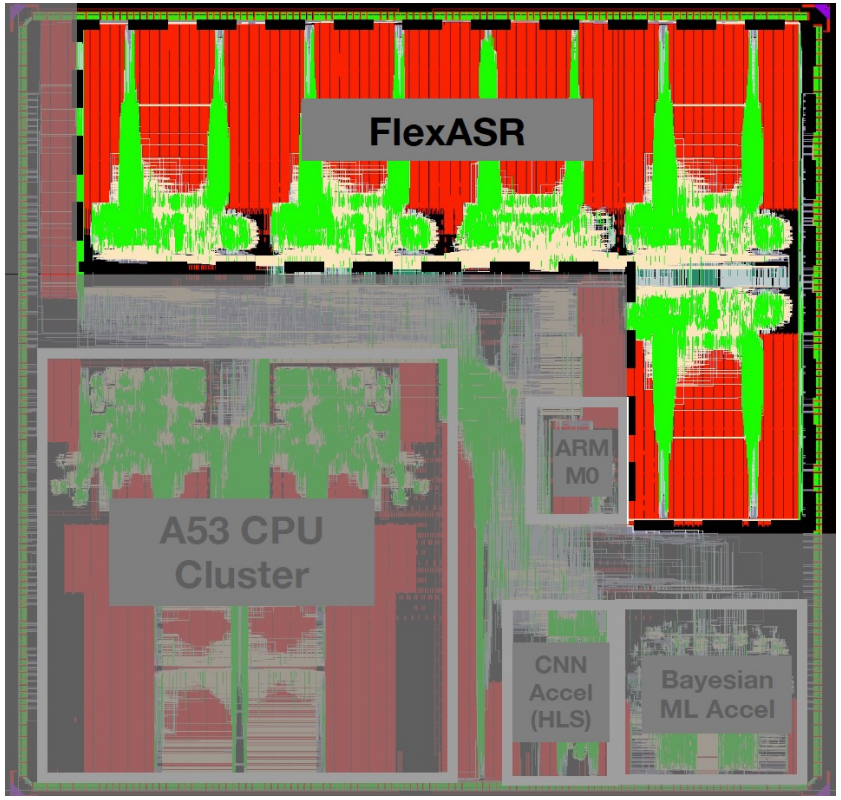


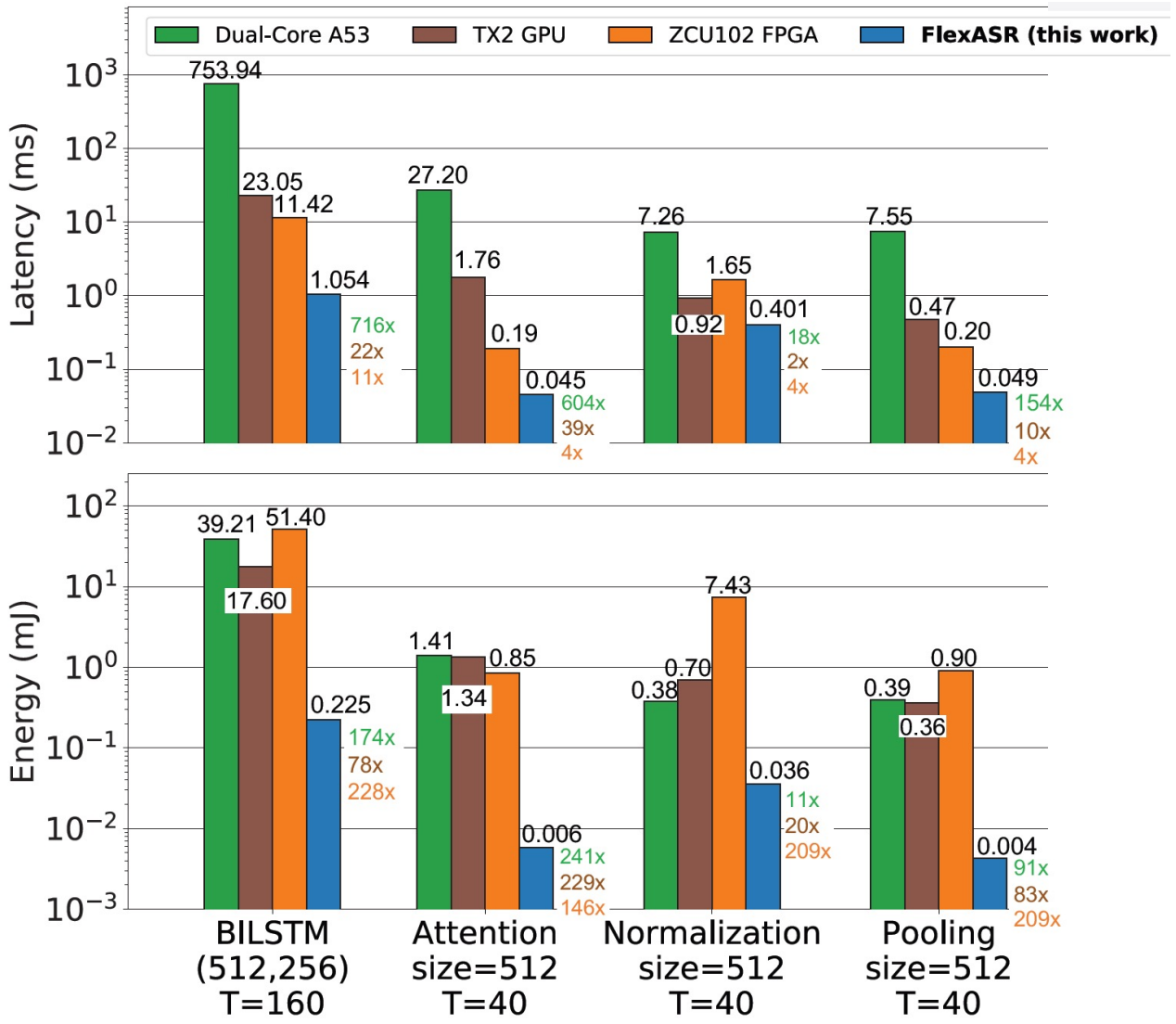
HLS Post-Silicon Results: *FlexASR*

Orders-of-Magnitude Speedup and Energy Savings over CPU, GPU, and FPGA



Attention-based Seq2Seq Model





T. Tambe et al., A 25mm² SoC for IoT Devices with 18ms Noise Robust Speech-to-Text Latency via Bayesian Speech Denoising and Attention-Based Sequence-to-Sequence DNN Speech Recognition in 16nm FinFET. ISSCC, 2021.

HLS Post-Silicon Results: *Cache-Coherent Accelerator (CCA)*

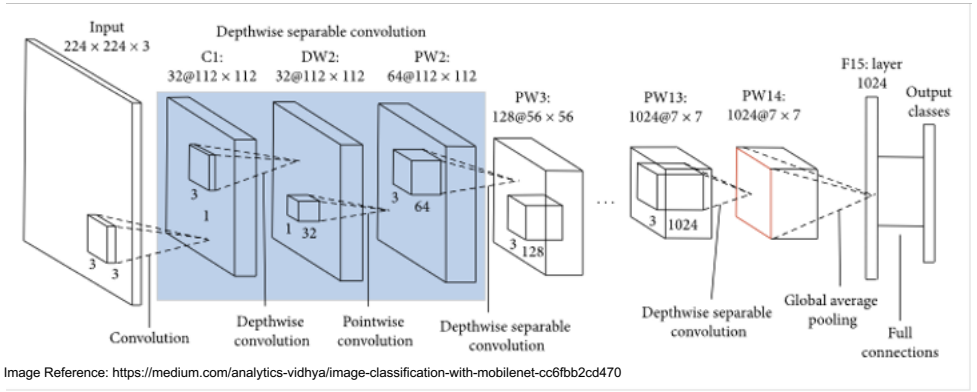
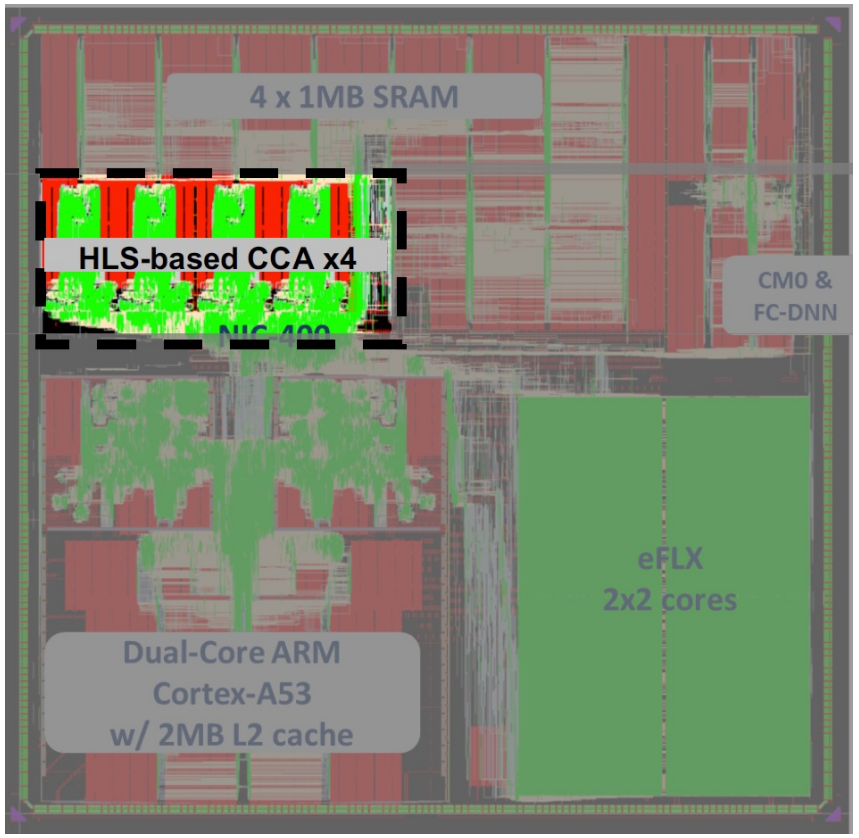
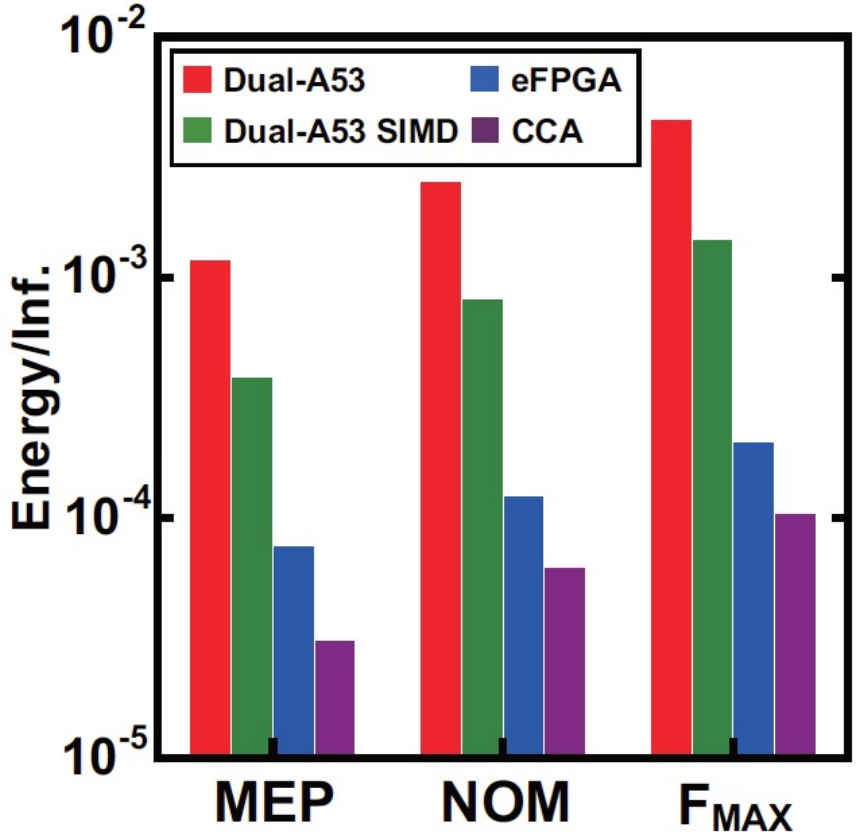


Image Reference: https://medium.com/analytics-vidhya/image-classification-with-mobilenet-cc6fbb2cd470

MobileNet CNN

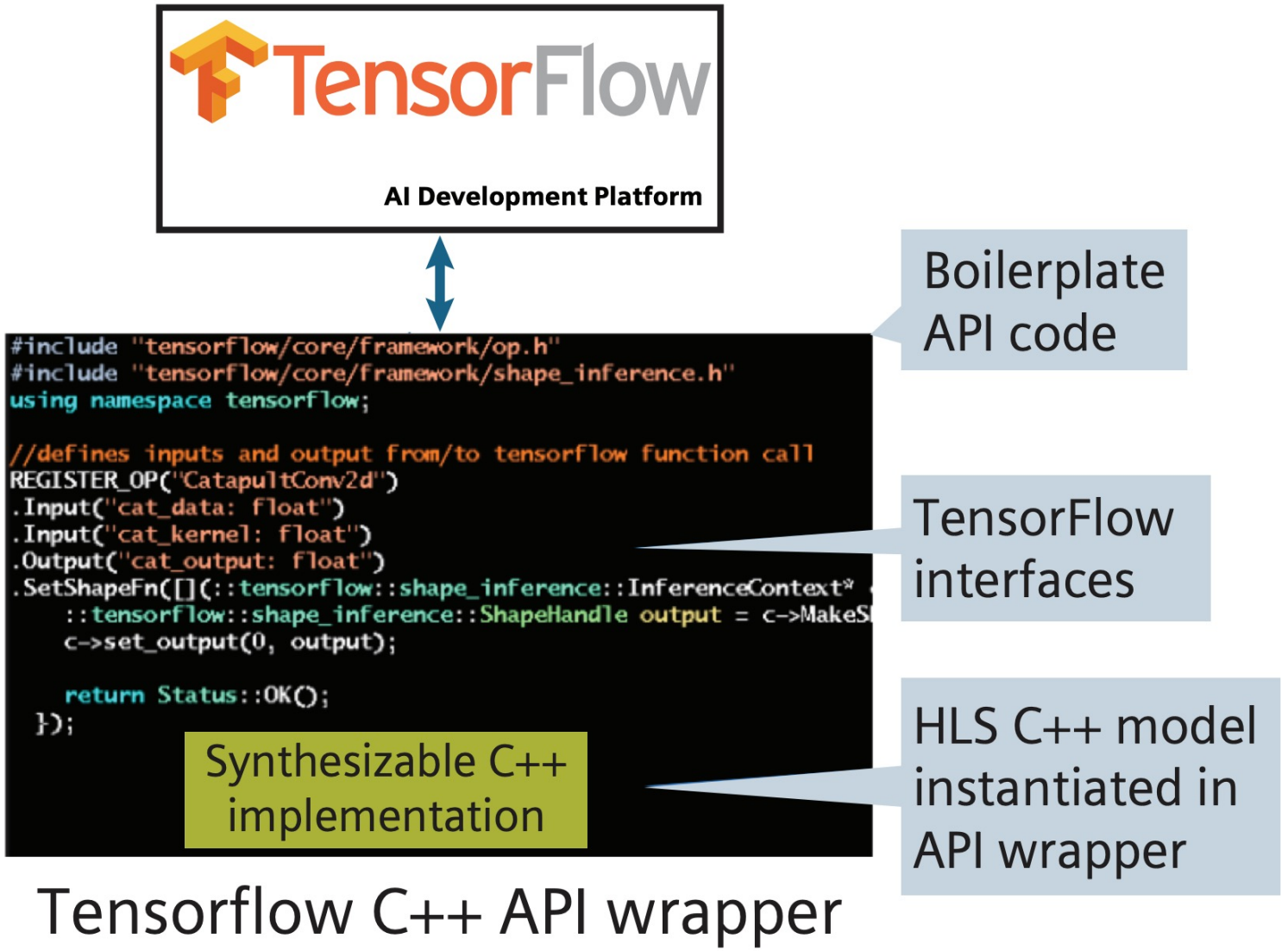




15x and 2x smaller energy/inf over SIMD and eFPGA, respectively

P. N. Whatmough et al., A 16nm 25mm^2 SoC with a 54.5x Flexibility-Efficiency Range from Dual-Core Arm Cortex-A53 to eFPGA and Cache-Coherent Accelerators. Symposium on VLSI Circuits, 2019

Alternative Approach: Plug HLS design directly into the ML Framework



Source: M. Fingeroff, *Machine Learning at the edge: using HLS to optimize power and performance*, Siemens White Paper, 2020

Outline

- Background and Motivation
- SW / HW Co-Design and Co-Verification Flow
- **Our HLS experience**
  - **Challenges**
  - **Pitfalls to avoid**
- Recommendations

Lots of trial and error to achieving synthesizable and fully functional HW

Easy to write compiling code that fails RTL verification

- Ex: pushing and popping same channel in same case statement
- Ex: Designing FSM transitions with case statements without specifying the next state in the default case

Missing next state transition
C++ sim passes but RTL sim may fail

```
void UpdateFSM() {
  FSM next_state;

  switch (state) {
    case IDLE: {
      next_state = PRE;
      break;
    }

    case PRE: {
      next_state = MAC;
      break;
    }

    case MAC: {
      //...
      if (is_output_end) {
        next_state = OUT;
      }
      else {
        next_state = MAC;
      }
      break;
    }

    case OUT: {
      //...
      if (is_output_end) {
        next_state = IDLE;
      }
      else {
        next_state = PRE;
      }
      break;
    }

    default: {

      break;
    }
  }
  state = next_state;
}
```

Lots of trial and error to achieving optimal PPA

Easy to write compile code with unoptimized PPA

- Ex: Single SC_THREAD() executing long datapaths
- Ex: Operation directly on output reference argument can lead to worse PPA

Simple coding tweaks impact PPA
Example: Vector Accumulator

```
#pragma hls_design ccore
#pragma hls_ccore_type combinational
inline void VSum (const spec::ActVectorType in, spec::ActScalarType& out) {
  #pragma hls_unroll yes
  for (int i = 0; i < 16; i++) {
    out += in[i];
  }
}
```


```
module VSum (
  in_data, out_in, out_out
);
  input [319:0] in_data;
  input [19:0] out_in;
  output [19:0] out_out;
  wire [24:0] nl_out_out;

  // Interconnect Declarations for Component Instantiations
  assign nl_out_out = (in_data[299:280]) + (in_data[319:300]) + out_in + (in_data[19:0])
      + (in_data[39:20]) + (in_data[59:40]) + (in_data[79:60]) + (in_data[179:160])
      + (in_data[199:180]) + (in_data[219:200]) + (in_data[239:220]) + (in_data[99:80])
      + (in_data[119:100]) + (in_data[139:120]) + (in_data[159:140]) + (in_data[259:240])
      + (in_data[279:260]);
  assign out_out = nl_out_out[19:0];
endmodule
```

Additional arguments!
Not what we want

Simple coding tweaks impact PPA
Example: Vector Accumulator

```
#pragma hls_design ccore
#pragma hls_ccore_type combinational
inline void VSum (const spec::ActVectorType in, spec::ActScalarType& out) {
  #pragma hls_unroll yes
  for (int i = 0; i < 16; i++) {
    out += in[i];
  }
}
```

HLS

```
module VSum (
  in_data, out_in, out_out
);
  input [319:0] in_data;
  input [19:0] out_in;
  output [19:0] out_out;
  wire [24:0] nl_out_out;

  // Interconnect Declarations for Component Instantiations
  assign nl_out_out = (in_data[299:280]) + (in_data[319:300]) + out_in + (in_data[19:0])
      + (in_data[39:20]) + (in_data[59:40]) + (in_data[79:60]) + (in_data[179:160])
      + (in_data[199:180]) + (in_data[219:200]) + (in_data[239:220]) + (in_data[99:80])
      + (in_data[119:100]) + (in_data[139:120]) + (in_data[159:140]) + (in_data[259:240])
      + (in_data[279:260]);
  assign out_out = nl_out_out[19:0];
endmodule
```

Additional arguments!
Not what we want

```
#pragma hls_design ccore
#pragma hls_ccore_type combinational
inline void VSum (const spec::ActVectorType in, spec::ActScalarType& out) {
  spec::ActScalarType out_tmp = 0;
  #pragma hls_unroll yes
  for (int i = 0; i < 16; i++) {
    out_tmp += in[i];
  }
  out = out_tmp;
}
```

HLS

Result stored to temporary variable

```
module VSum (
  in_data, out_0
);
  input [319:0] in_data;
  output [19:0] out_0;
  wire [23:0] nl_out_0;

  // Interconnect Declarations for Component Instantiations
  assign nl_out_0 = (in_data[19:0]) + (in_data[39:20]) + (in_data[59:40]) + (in_data[79:60])
      + (in_data[99:80]) + (in_data[119:100]) + (in_data[139:120]) + (in_data[159:140])
      + (in_data[259:240]) + (in_data[279:260]) + (in_data[299:280]) + (in_data[319:300])
      + (in_data[179:160]) + (in_data[199:180]) + (in_data[219:200]) + (in_data[239:220]);
  assign out_0 = nl_out_0[19:0];
endmodule
```

Correct RTL implementation!

Simple coding tweaks impact PPA
Example: Vector Accumulator

```
#pragma hls_design ccore
#pragma hls_ccore_type combinational
inline void VSum (const spec::ActVectorType in, spec::ActScalarType& out) {
  #pragma hls_unroll yes
  for (int i = 0; i < 16; i++) {
    out += in[i];
  }
}
```

HLS

```
module VSum (
  in_data, out_in, out_out
);
  input [319:0] in_data;
  input [19:0] out_in;
  output [19:0] out_out;
  wire [24:0] nl_out_out;
  ...
  assign out_out = nl_out_out[19:0];
endmodule
```

Additional arguments!
Not what we

```
#pragma hls_design ccore
#pragma hls_ccore_type combinational
inline void VSum (const spec::ActVectorType in, spec::ActScalarType& out) {
  spec::ActScalarType out_tmp = 0;
  #pragma hls_unroll yes
  for (int i = 0; i < 16; i++) {
    out_tmp += in[i];
  }
  out = out_tmp;
}
```

HLS

```
module VSum (
  in_data, out_0
);
```

Result stored to temporary variable

Correct RTL implementation!

Area = **511 µm²**
Power = **22 µW**

12nm PPA @ 500MHz
- Same latency and same result
- 10% smaller area
- 27% smaller power

Area = **460 µm²**
Power = **16 µW**

Outline

- Background and Motivation
- SW / HW Co-Design and Co-Verification Flow
- Our HLS experience
- **Recommendations**

1) Make heavy use of HW library components

- MatchLib and HLSLibs libraries boost productivity

FlexASR example

Component from MatchLib	Description	Number of instantiations
Connections	Modular IO supporting LI channels	>500
Scratchpad	Banked memory array with crossbar	14
AXI Components	Master/Slave interfaces and bridges for AXI interconnect	>100
Arbitrated Crossbar	Crossbar with conflict arbitration and queuing	1
nvhls_vector	Vector helper container w/ vector operations	> 400
nvhls_array	Array helper container	10

Component from HLSLibs	Description	Number of instantiations
ac_sigmoid_pwl	Piecewise linear (PWL) function for Sigmoid	1
ac_tanh_pwl	PWL function for Tanh	1
ac_exp_pwl	PWL function for Natural Exponential	1
ac_div	Computes quotient of Division	1
ac_inverse_sqrt_pwl	PWL function of Inverse Square Root	1
ac_fixed	Numerical package that controls the behavior of fixed-point type	>10
ac_float	Numerical package that controls the behavior of floating-point type	>30

2) Parameterize key HW features

Source code example

```
namespace spec {
  const int kNumPE = NUM_PES;
  const int kVectorSize = VECTOR_SIZE;
  const int  scratchpad_size = MEM_SIZE;
```

```
CFLAGS = -DHLS_ALGORITHMICC -DNUM_PES=4 -DVECTOR_SIZE=16 -DMEM_SIZE=4096
include ../cmod_Makefile

all: sim_test

run:
  ./sim_test

sim_test: $(wildcard *.h) $(wildcard *.cpp)
  $(CC) -o sim_test $(CFLAGS) $(USER_FLAGS) $(wildcard *.cpp) $(BOOSTLIBS) $(LIBS)

sim_clean:
  rm -rf *.o sim_*
```

Makefile for C++ sim

Additional examples here: https://github.com/harvard-acc/EdgeBERT/blob/main/hw/cmod

3) Avoid overly complex SC_THREADs

- *Initiation interval* (II) is the # of cycles between successive operations.
- II=1 is desirable for high throughput.
- HLS may be incapable of synthesizing loops with complex/long datapaths at target clock frequency unless II is relaxed.

Long SC_THREAD degrades throughput

```
SC_THREAD(Run);

void Run() {
  input.Reset();

  #pragma hls_pipeline_init_interval 4
  while(1) {
    ....
    ....
    ....
    ....
    ....
    ....
    ....
    ....
    ....
    ....
    ....
    ....
    wait();
  }
}
```

Very long datapaths Requires II=4

Low throughput!

Multithreading improves throughput

```
SC_THREAD(Run);

void Run() {
  input.Reset();

  #pragma hls_pipeline_init_interval 4
  while(1) {
    ....
    ....
    ....
    ....
    ....
    ....
    ....
    ....
    ....
    ....
    ....
    ....
    wait();
  }
}
```

Very long datapaths Requires II=4

Low throughput!

```
SC_THREAD(Run0);
SC_THREAD(Run1);
SC_THREAD(Run2);

void Run0() {
  input_run0.Reset();
  #pragma hls_pipeline_init_interval 1
  while(1) {
    ....
    wait();
  }
}
void Run1() {
  input_run1.Reset();
  #pragma hls_pipeline_init_interval 1
  while(1) {
    ....
    wait();
  }
}
void Run2() {
  input_run2.Reset();
  #pragma hls_pipeline_init_interval 1
  while(1) {
    ....
    wait();
  }
}
```

Shorter datapaths with II=1

Higher throughput!

Example of Multithreading

Pooling Unit

- https://github.com/harvard-acc/FlexASR/blob/master/cmod/GBPartition/GBModule/LayerReduce/LayerReduce.h

```
SC_THREAD(LayerReduceRun);
sensitive << clk.pos();
async_reset_signal_is(rst, false);

        ⋮

void LayerReduceRun(){
  Reset();
  #pragma hls_pipeline_init_interval 4
  while(1) {
    DecodeAxi();
    PushAxiRsp();
    CheckStart();
    RunFSM();
    UpdateFSM();
    wait();
  }
}
```

Moved to separate thread

```
SC_THREAD(LayerReduceRun);
sensitive << clk.pos();
async_reset_signal_is(rst, false);

SC_THREAD(ConfigRun);
sensitive << clk.pos();
async_reset_signal_is(rst, false);

        ⋮

void LayerReduceRun(){
  ResetFSM();
  #pragma hls_pipeline_init_interval 1
  while(1) {
    RunFSM();
    UpdateFSM();
    wait();
  }
}

void ConfigRun(){
  Reset();
  #pragma hls_pipeline_init_interval 1
  while(1) {
    DecodeAxi();
    PushAxiRsp();
    CheckStart();
    wait();
  }
}
```

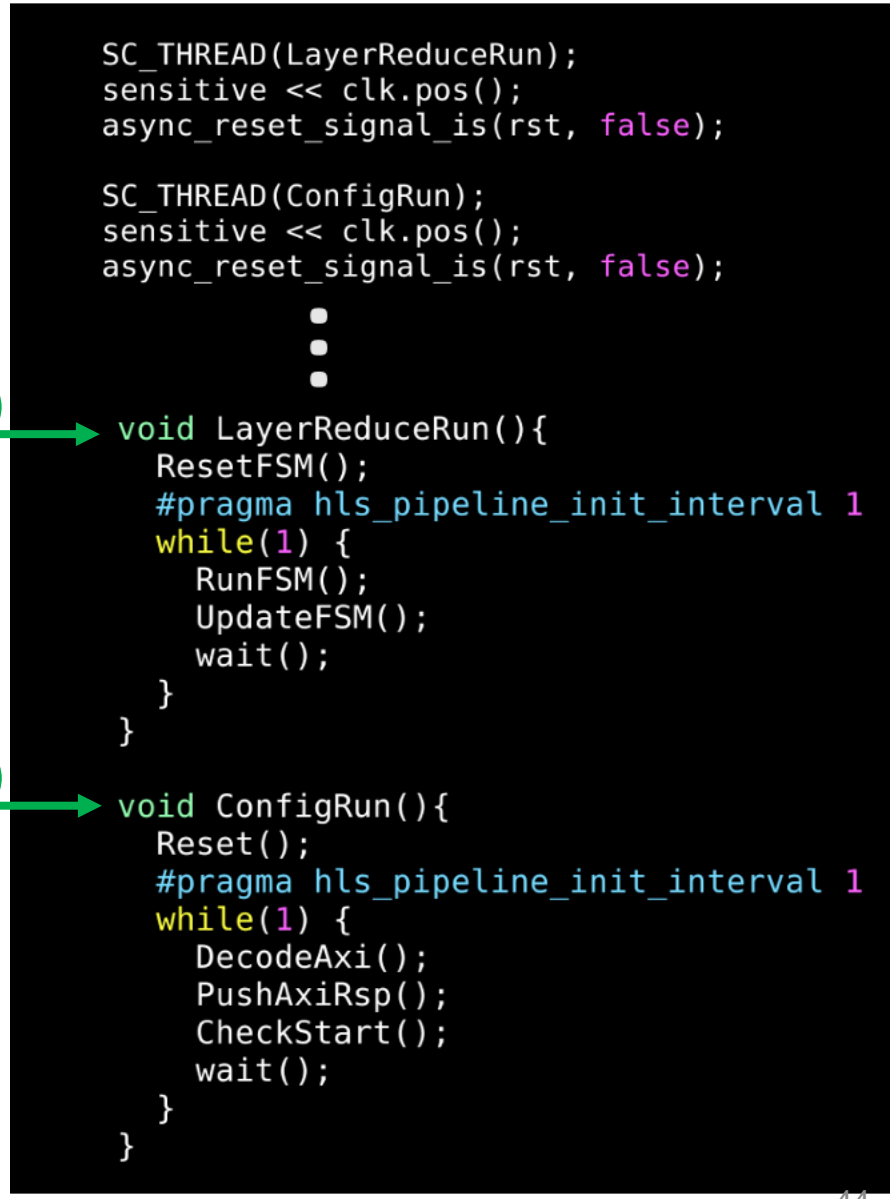
12nm PPA @ 500MHz

- **2.5x faster!**
- 1.1x higher area
- 1.05x higher power

However, multi-threaded designs present challenges

Common Problem:
Variable ‘xyz’ cannot be shared in multiple threads.

Solution:
Use combinational channels to exchange data between SC_THREADs :
E.g., **Connections::Combinational<bool>** start_trig;

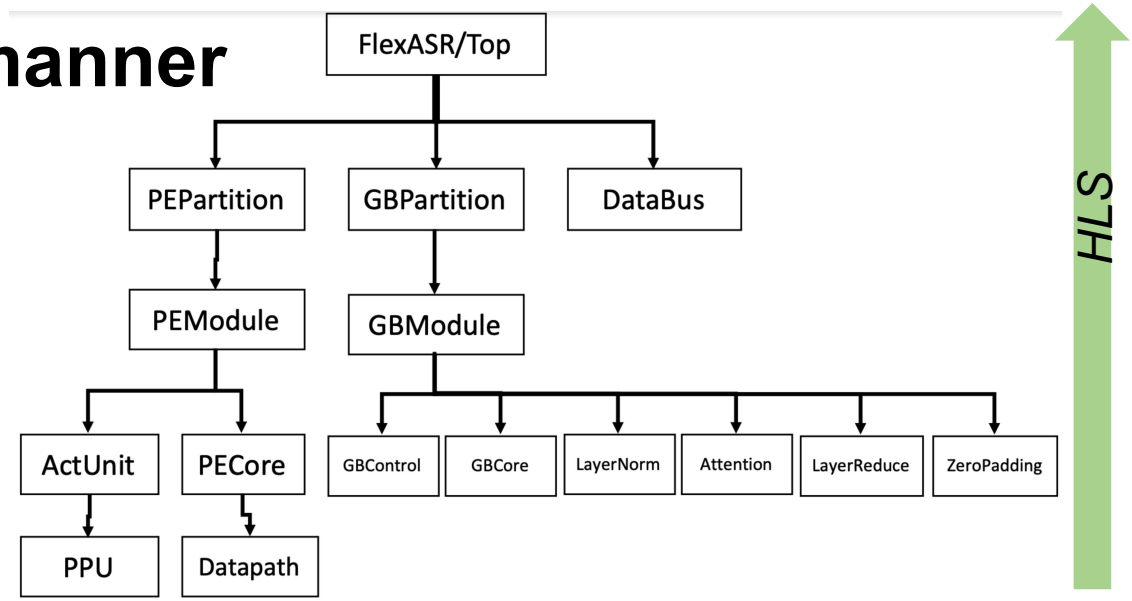


4) Enable random stalling during testing

- Often, design bugs/hangs are only observed post-HLS.
- Random stalling of ports/channels during SystemC/C++ simulation may catch these elusive bugs prior to HLS.
- In MatchLib *connections*, this is done with the **-DCONN_RAND_STALL** flag.

5) Some other recommendations

- **Code large design in a hierarchical manner**
  - Verify modules in a bottom-up fashion
  - Synthesize and optimize each module individually from bottom to top



- **Make heavy use of function templates to make code more legible**

Ex:

```
template <unsigned int kAddressWidth>
class PEManager {
 public:
  typedef NVUINTW(kAddressWidth)        Address;
  typedef NVUINTW(kAddressWidth+1)      AddressPlus1;
```

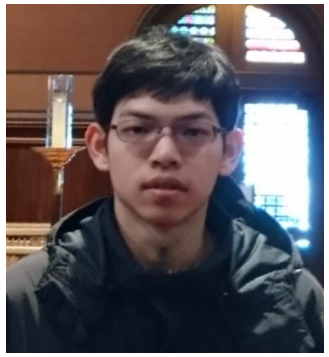
Tips on getting started with HLS

- Study open-source examples with supporting testbenches. Many are publicly available:
  - FlexASR: https://github.com/harvard-acc/FlexASR
  - EdgeBERT: https://github.com/Harvard-acc/EdgeBERT/tree/main/hw
  - Chimera: https://code.stanford.edu/kprabhu7/dnn-accelerator
  - Accelerra: https://forums.accellera.org/files/category/2-systemc/
  - MatchLib: https://github.com/NVlabs/matchlib/tree/master/cmod/examples
  - HLSLibs: https://hlslibs.org/
  - Many examples shipped with Catapult under `/Mgc_home/shared/examples`

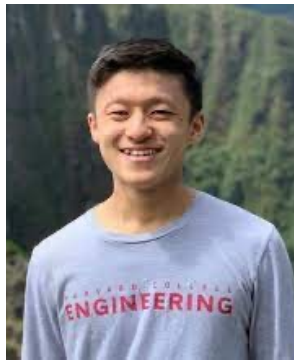
- HLS bluebook provides a lot of great advice on proper syntax for various functionalities and design constraints
  - Source: https://cse.usf.edu/~haozheng/teach/cda4253/doc/hls/hls_bluebook_uv.pdf

Acknowledgements


Thierry Tambe

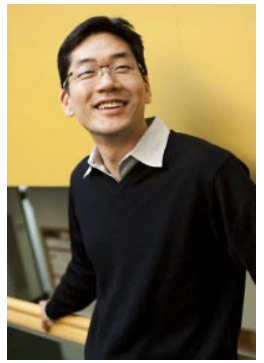

Daniel Yang


Yeongil Ko



Jaylen Wang


Prof. David Brooks


Prof. Gu-Yeon Wei

SIEMENS
Catapult HLS

NVIDIA
NVIDIA ASIC & VLSI Research Group

ada
Applications Driving Architectures
SRC JUMP Program

DARPA
DARPA
CRAFT & DSSoC

NSF
NSF# 1704834
NSF# 1718160

Questions?