UNIVERSITY OF ILLINOIS AT URBANA-CHAMPAIGN

AI-Hardware Co-Design: Advancing Quality, Productivity, and Reliability

Deming Chen

Abel Bliss Professor of Engineering

Department of Electrical and Computer Engineering

University of Illinois Urbana-Champaign

IWLS

Hong Kong, 5/30/2026

AI is at an Inflection Point

- AI progress has been driven by ***scaling***: larger models, more data, more compute
- This trajectory is increasingly **unsustainable**:
  - Exploding cost and energy consumption
  - Long development cycles and fragile systems
  - Growing gap between research ideas and deployable solutions
- **Key message:**

The future of AI will not be decided by algorithms or hardware alone but by how intelligently they are co-designed.

The Limits of “Scale First, Fix Later”

- **The dominant paradigm today:**
  - Design AI models first
  - Map them onto hardware later
  - Patch inefficiencies with compilers and heuristics

- **This creates:**
  - Mismatch between model structure and hardware realities
  - Massive data movement and energy waste
  - Unpredictable performance and reliability gaps

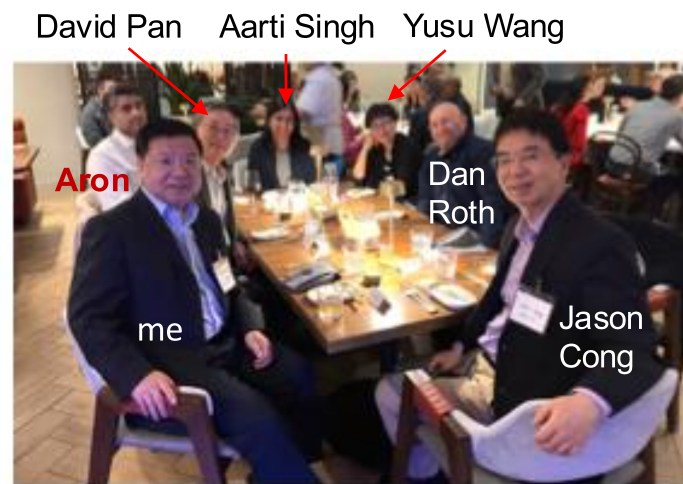
- **Key observation:**

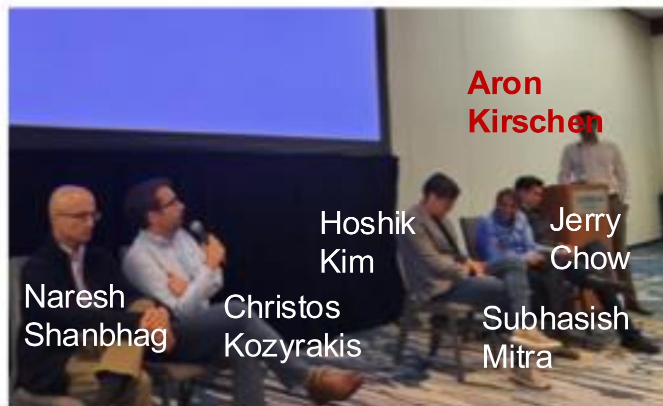
Top-down flow/optimization cannot compensate for fundamentally misaligned design decisions.

AI–HW "Co-Design" as a First-Principles Necessity

- **AI–HW co-design is not an optimization trick. It is a necessity.**
- **Co-design means:**
  - Algorithms shaped by memory, data movement, and energy
  - Hardware shaped by model structure, sparsity, precision, and adaptivity
  - Compilers and systems as *active co-design agents*, not glue
  - The co-design space with these dimensions is co-searched comprehensively
- **Outcome:**
  - Higher *effective quality per watt* and *intelligence per joule*
  - Faster design iteration and deployment
  - Predictable, reliable system behavior

NSF Workshop on “AI+HW 2035: Shaping the Next Decade”





https://publish.illinois.edu/ai-hw-workshop

Vision Paper: https://arxiv.org/abs/2603.05225

October 16–17, 2025

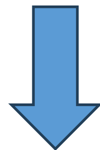
Outline of This Talk

- **The A^{3}C^3 Method** – AI+HW co-design by construction
  - A^{3}C^3-based solutions won AI competition awards
- Let's talk about **Synthesis**!
  - HLS and AI-to-hardware compiler
- **Proof2Silicon** – provable spec-to-silicon flow
- **AI Agents-Driven Synthesis & Future Research**

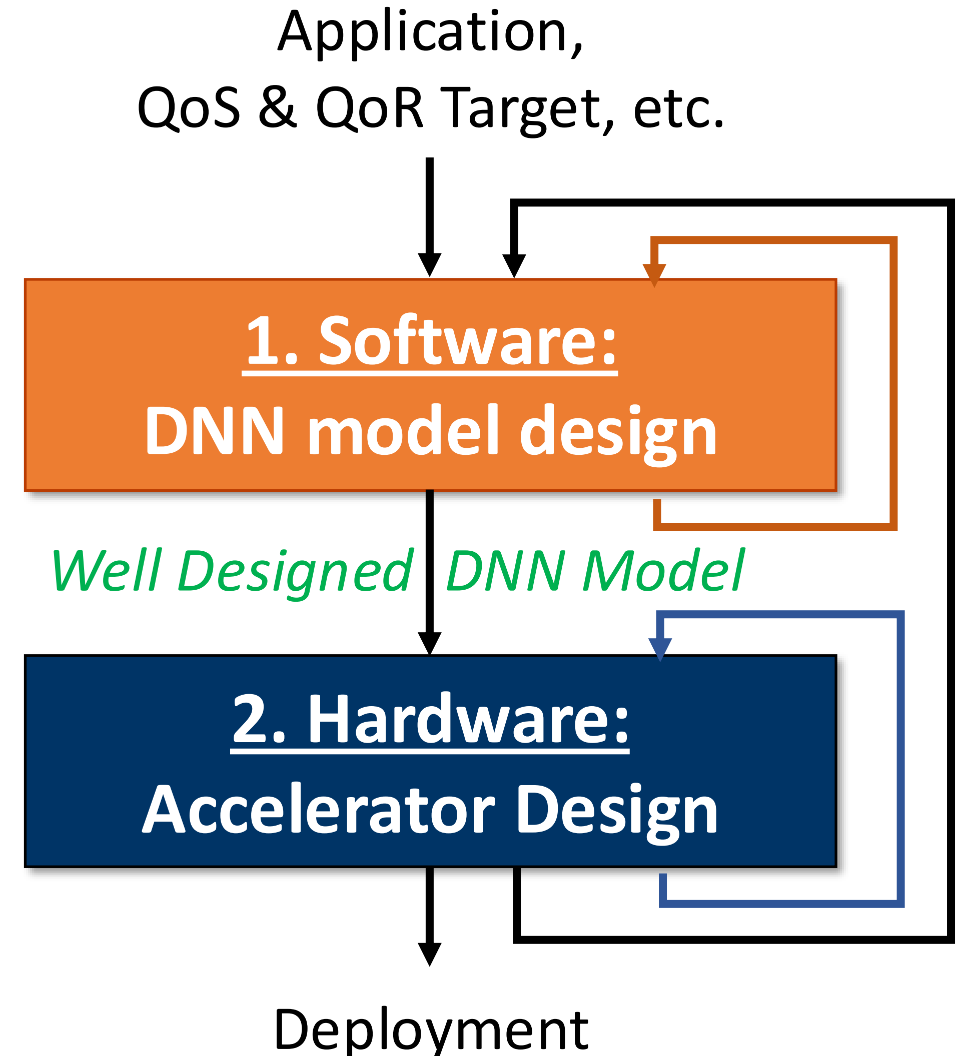
Drawbacks of Conventional DNN Design and Deployment

Long iterative procedure, tedious engineering efforts, especially for:

- Resource limited edge devices for Physical AI
- Strict performance requirement
  - E.g., faster than 35 FPS for real-time video processing



Sub-optimal (hardware-unfriendly) DNN models and accelerators



A^3C^3 Method Proposed in ICSICT 2018

Inputs

Performance Constraints:
Speed > 20 FPS
Power < 5W
Throughput > 20 GOPS
etc.

FPGA Resources:
BRAM, DSP,
LUTs, etc.

Target Application:
Image Classification / Recognition / Object Detection, etc.

IP Pool
Depth-wise 3x3 … Conv 1x1 … Conv 5x5
Max Pooling … Relu … MM Multiply

Configuration Data
Parallelism = 16:
LUTs (5000)
DSPs (18)
Parallelism = 32:
LUTs (12000)
DSPs (42)
…

Co-Design Engine

Outputs

Solution 1

Configured IPs
- Conv 1x1: Parallelism=32
- Conv 3x3: Parallelism=64
- Max pooling (str=2), etc.

Neural Network Structure
- depth-wise conv layers
- total 20 layers
- maximum 512 channels

FPGA Implementation
- 8 bit weight, 16 bit activation
- 3 instances of Conv 3x3, etc.
- folded structure

Solution 2

Configured IPs
- Conv 3x3: Parallelism=64
- Conv 5x5: Parallelism=128
- Avg pooling (str=1), etc.

Neural Network Structure
- depth-wise conv layers
- total 16 layers
- maximum 256 channels

FPGA Implementation
- 8 bit weight, 8 bit activation
- 1 instance of Conv 5x5, etc.
- folded structure

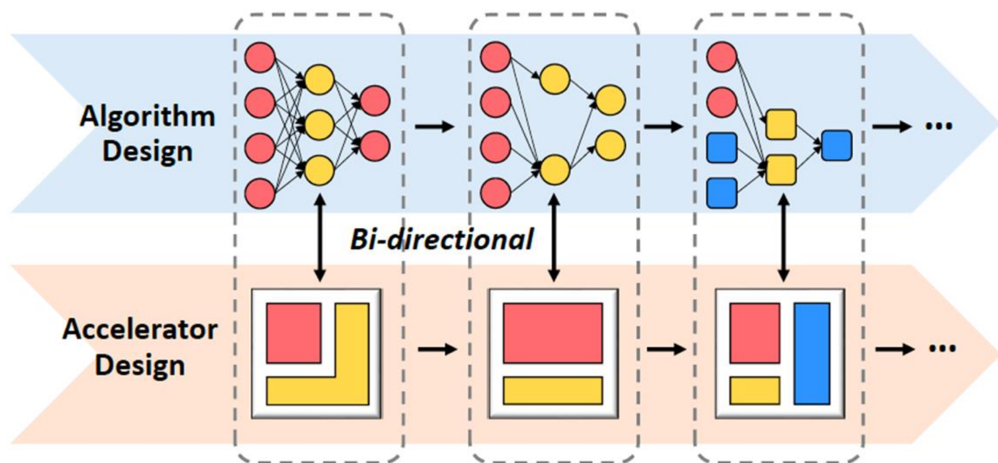
…

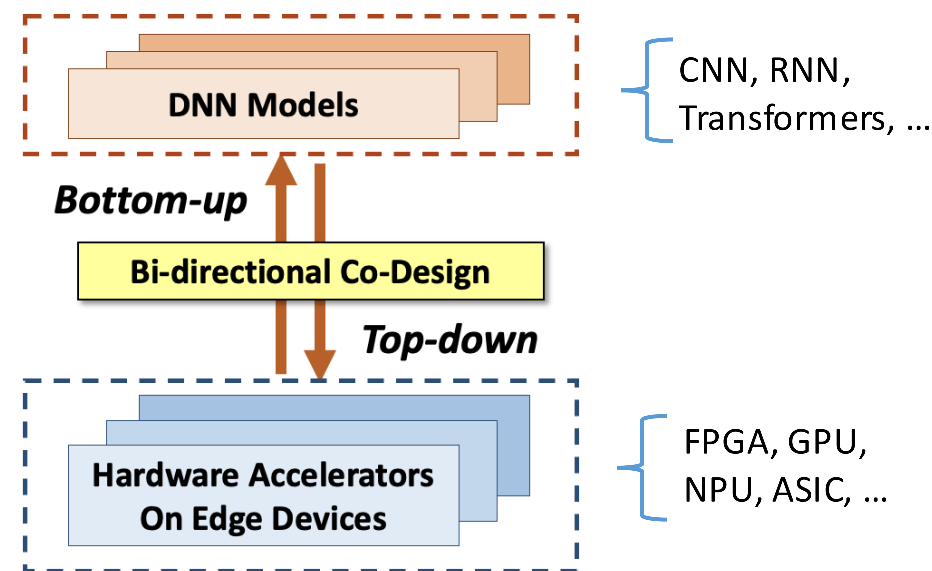
- The objective of the proposed method is: “*to automatically generate both DNN models and their corresponding implementations as pairs*” and “*shorten the DNN development time by avoiding tedious iterations between DNN model and its accelerator designs.*”

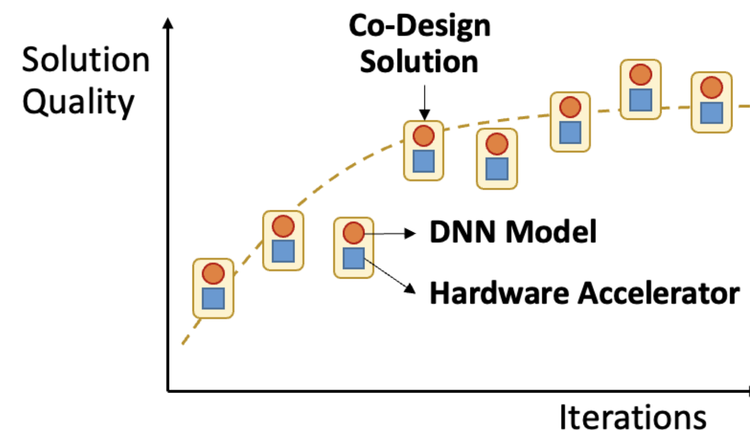
[ICSICT 2018]: Callie Hao and Deming Chen, “Deep Neural Network Model and FPGA Accelerator Co-design: Opportunities and Challenges”, 2018.

A^3C^3 – AI Algorithm & Accelerator Co-design, Co-search, and Co-generation

- Both the AI algorithm and accelerator design spaces are **parameterized** and **co-searched** simultaneously (e.g., through gradient descent).
- Both the AI model and its accelerator are **co-generated** as a friendly pair.







[ICSICT'18, DAC'19, ICCAD'19, MLSys'20, DAC'20, GLSVLSI'20, FPGA'20, AICAS'21, NeurIPS'21, DAC'21, ASPDAC'22, ICCV'23, DAC'24, ICML'24, …]

The A^3C^3 Framework Materialized in DAC 2019

Challenges

- High quality DNN design and its accelerator design are both challenging
- A^3C^3 significantly expands design space
  - Difficult to converge
  - Requires novel search methodology

Co-Design Flow Inputs
- Target ML task
- FPGA device (resource)
- Performance targets

Bundle-Arch: DNN template
DW-Conv
Conv
Pool-ing
Conv 3x3
Conv 5x5
Pool-ing
Relu
Bundle 1
2
n
Configurable IP Pool

Auto-DNN: DNN Exploration
Step 1: Bundle/DNN Modeling
Resource & Latency Model
Step 2: Bundle Evaluation
Coarse-Grained
Fine-Grained
Top-N promising Bundles
Step 3: Hardware-aware DNN search and update
DNN Initialization <bund$_i$, Stk, PF, Qt>
Resource Constraints & Target Latency
Stochastic Coordinate Descent (SCD) Unit
Latency/Resource Model
DNN Model Update
DNN candidates
Constraints met

Auto-HLS: Accelerator Generation
Auto-Bundle Generation
Auto-DNN Generation
Synthesizable C code
High Level Synthesis Tool
DNN resource and latency
+
FPGA Design

Tile-Arch: Accelerator template
data tiling
IP pipelining
Time
1 2 3 4

DNN Training Framework
Training Script
DNN Accuracy

Co-Design Flow Outputs
Software: DNN Model
Hardware: FPGA Design

[DAC 2019]

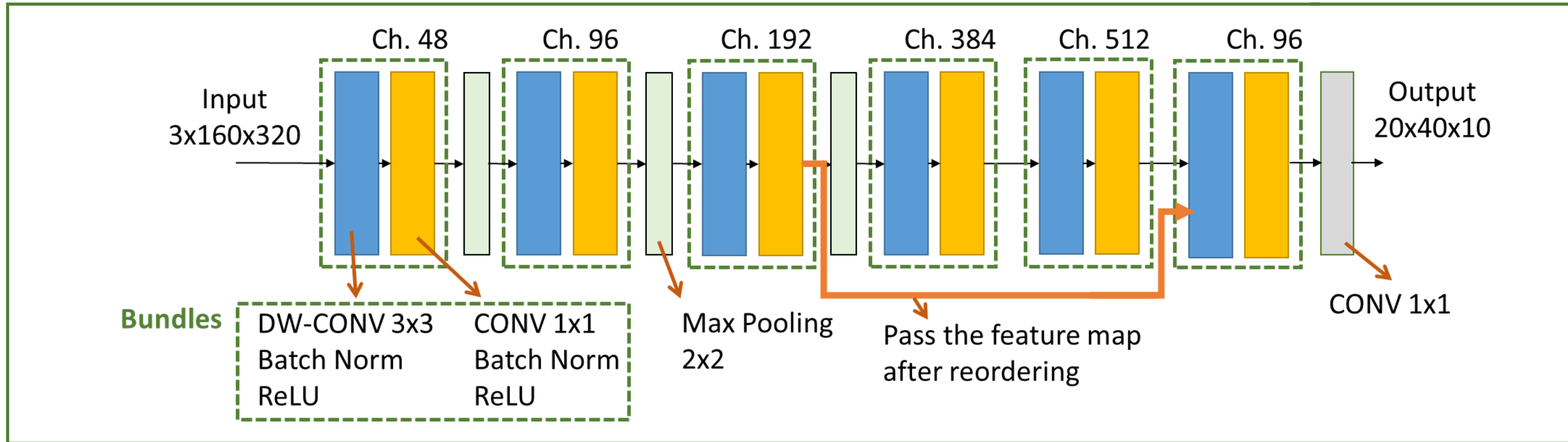
One Output of A^3C^3 : the SkyNet

➢ **Three Stages:**

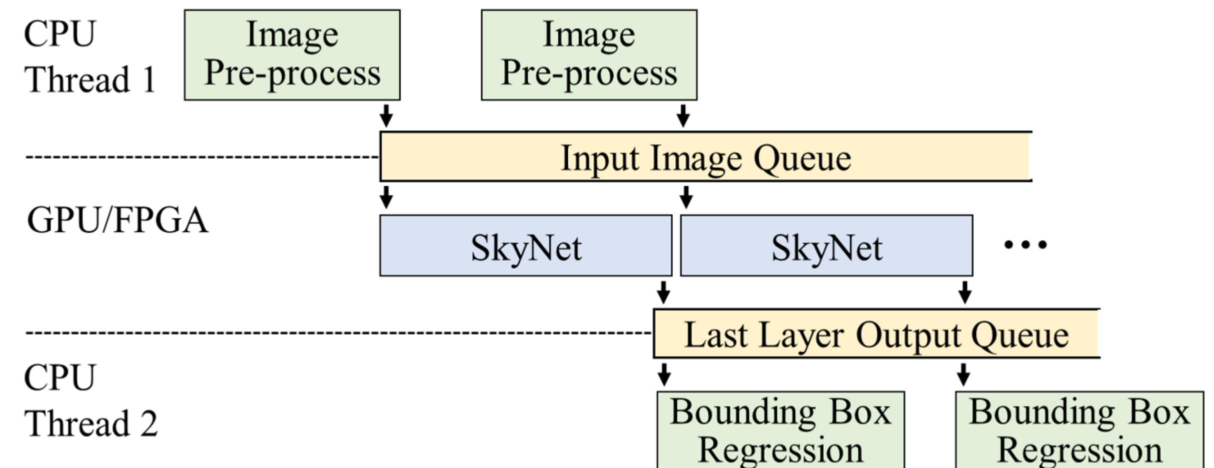
① Select **Basic Building Blocks** → ② Explore DNN and accelerator architectures based on **templates** → ③ Add features, fine-tuning and hardware deployment

[DAC SDC 2019] & [MLSys 2020]

The SkyNet DNN Architecture

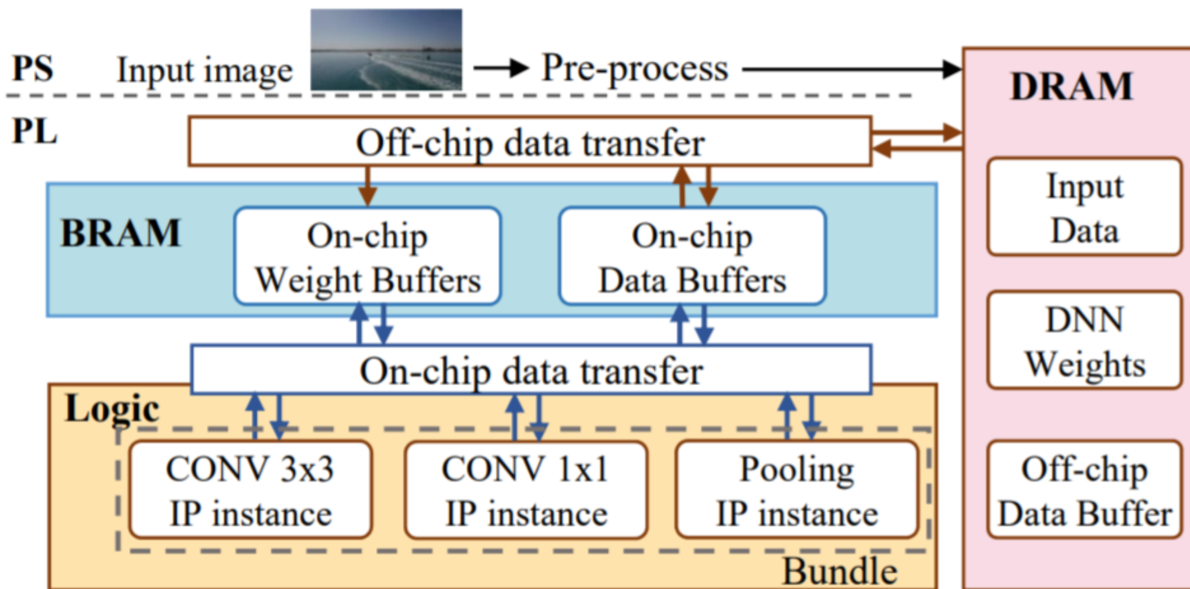


- 13 CONV with 0.4 million parameters
- For Embedded FPGA: Quantization, Batch, Tiling, Task partitioning
- For Embedded GPU: Task partitioning

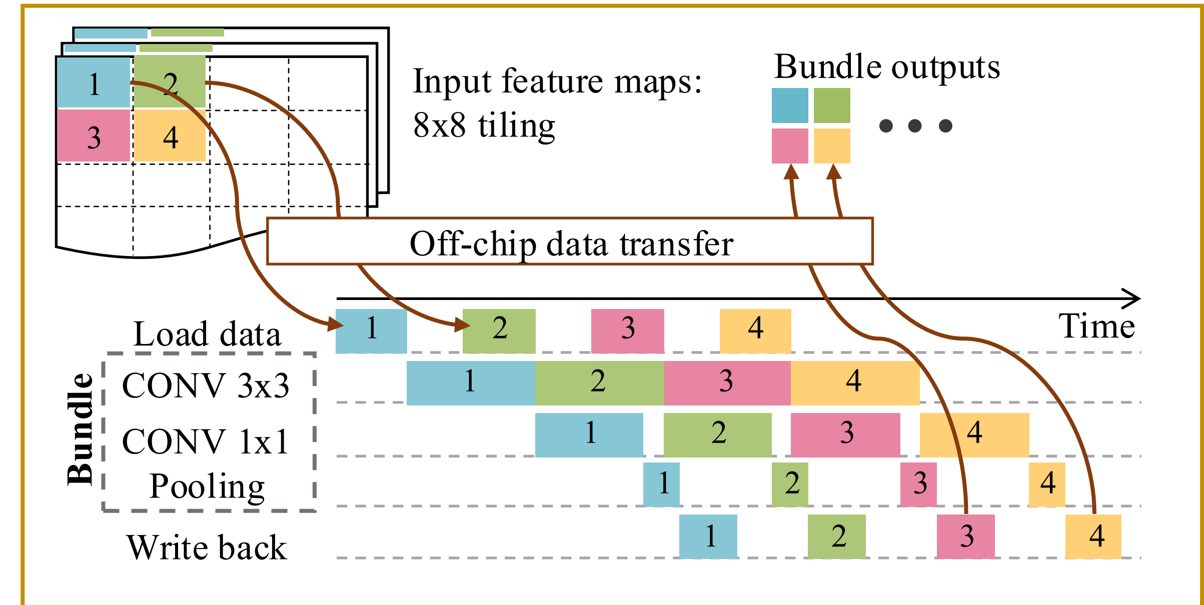


The HW Deployment of SkyNet

- Hardware (FPGA) accelerator using the proposed approach



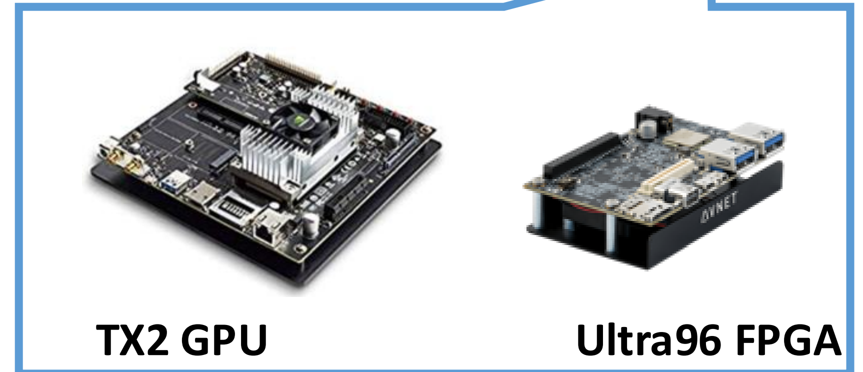
Overall system



Accelerator

Demo #1: Object Detection for Drones

- System Design Contest for ***low power object detection*** in the IEEE/ACM Design Automation Conference (DAC-SDC)



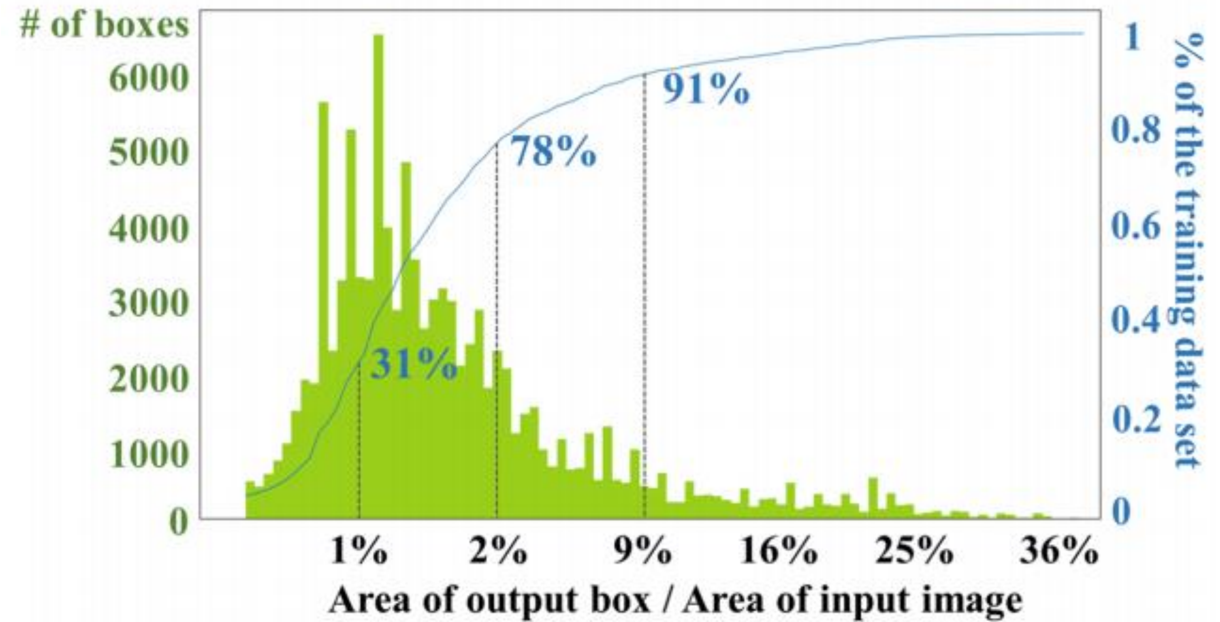
- DAC-SDC targets single object detection for real-life UAV applications

  Images contain 95 categories of targeted objects (most of them are small)

- Comprehensive evaluation: ***accuracy, throughput, and energy consumption***

Demo #1: DAC-SDC Dataset

- The distribution of target relative size compared to input image



31% targets < 1% of the input size

91% targets < 9% of the input size

Demo #1: SkyNet Results for DAC-SDC 2019

➢ Evaluated by 50k images in the official test set

1st SkyNet-FPGA

2.3X faster

3rd

1st

2nd

1st SkyNet-GPU

3rd

2nd

10.1% more accurate

2nd

1st

3rd

3rd

2nd

IoU

0.75
0.70
0.65
0.60
0.55
0.50
0.45

5 15 25 35 45 55 65 FPS

■ GPU-19	▲ GPU-18
● FPGA-19	● FPGA-18

Designs using TX2 GPU

Designs using Ultra96 FPGA

Demo #2: Generic Object Tracking in the Wild

- We extend SkyNet to real-time tracking problems
- We use a large-scale high-diversity benchmark called ***Got-10K***
  - ***Large-scale:*** 10K video segments with 1.5 million labeled bounding boxes
  - **Generic:** 560+ classes and 80+ motion patterns (better coverage than others)

animal

brachiation

jumping

person

surfing

skiing

[From Got-10K]

Demo #2: Results from Got-10K

- Evaluated using two state-of-the-art trackers with single 1080Ti

SiamRPN++ with different backbones

Backbone	AO	$SR_{0.50}$	$SR_{0.75}$	FPS
AlexNet	0.354	0.385	0.101	52.36
ResNet-50	0.365	0.411	0.115	25.90
SkyNet	0.364	0.391	0.116	41.22

Similar AO, 1.6X faster vs. ResNet-50

SiamMask with different backbones

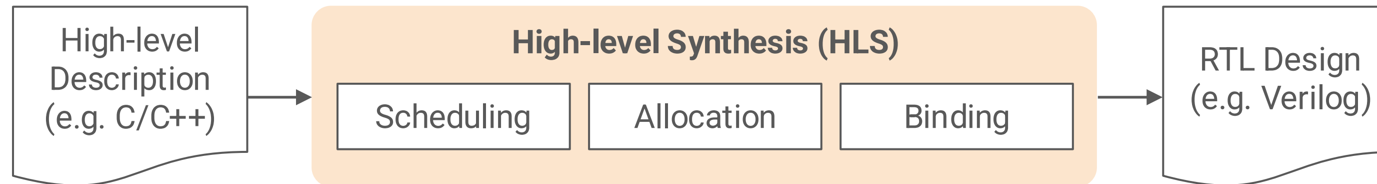
Backbone	AO	$SR_{0.50}$	$SR_{0.75}$	FPS
ResNet-50	0.380	0.439	0.153	17.44
SkyNet	0.390	0.442	0.158	30.15

Slightly better AO, 1.7X faster vs. ResNet-50

- AO (average overlap) is defined as the mean of IoU between prediction and ground truth bounding boxes
- SR (success rate) is defined as the proportion of predictions where the IoU is beyond some threshold.

Now, Let’s Talk about High-Level Synthesis

Conventional High-level Synthesis (HLS)



High-level Synthesis (HLS) is great

- **Reduce design complexity:** Code density can be reduced by 7x - 8x moving from RTL to C/C++ [1]
- **Improve design productivity:** Get to working designs faster and reduce time-to-market [1]
- **Identify performance-area trade-offs:** Implement design choices quickly and avoid premature optimization [2]

Designing HLS accelerator is still challenging

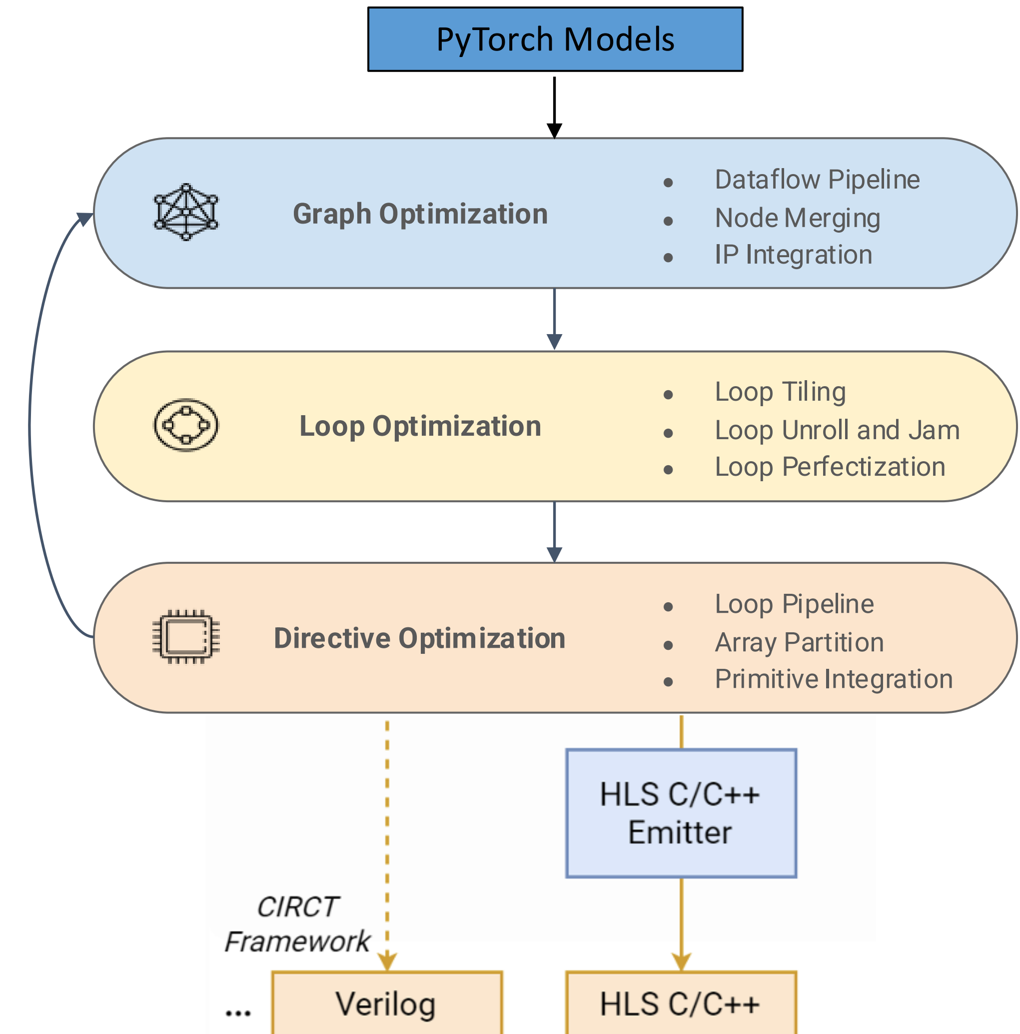
- **Friendly to experts:** Rely on the designers writing 'good' code to achieve high design quality [3]
- **Large design space:** Different combinations of applicable optimizations for large-scale designs [2]
- **Correlation of design factors:** It is difficult for human to discover the complicated correlations [4]

[1] Cong, Jason, et al. "High-level synthesis for FPGAs: From prototyping to deployment." TCAD. 2011.
[2] Schafer, Benjamin Carrion, and Zi Wang. "High-level synthesis design space exploration: Past, present, and future." TCAD. 2019.
[3] Sohrabizadeh, Atefeh, et al. "AutoDSE: Enabling software programmers to design efficient FPGA accelerators." TODAES. 2022.
[4] Yu, Mang, Sitao Huang, and Deming Chen. "Chimera: A hybrid machine learning-driven multi-objective design space exploration tool for fpga high-level synthesis." IDEAL. 2021.

HLS + MLIR = **ScaleHLS**

The Next Wave of HLS

- New HLS solutions translating AI models to customized AI accelerators automatically.
  - By adopting PyTorch as input for AI designs (instead of traditional C/C++ for HLS), the lines of code and design simulation time can be reduced by about **10x** and **100x**, respectively.
- Such AI model-to-RTL flows pave the way for a new wave of HLS
  - Drive the high-productivity designs of AI circuits with high density, high-energy efficiency, low cost, and short design cycle.
  - Can be expanded to other non-AI domains.
- Challenges for such next-gen HLS solutions
  - Ensuring the correctness of the high-level design
  - Accommodating accurate low-level timing/energy information
  - Handling the complexity of 3D and/or chiplet-based design
  - Achieving scalability and quality
  - …

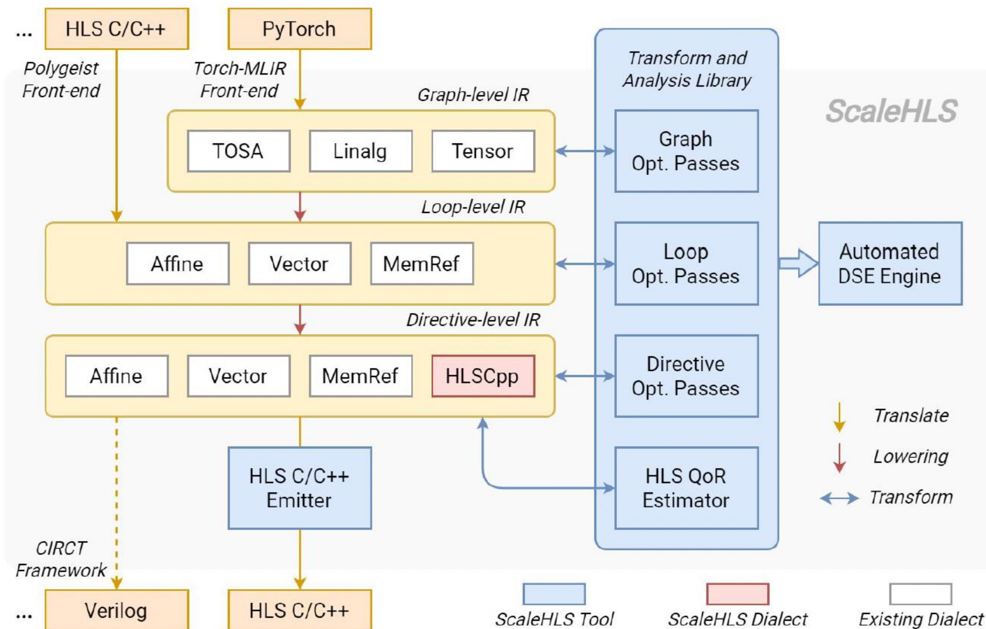


The ScaleHLS Framework [1] [2] [3]

https://github.com/hanchenye/scalehls

[1] Ye, Hanchen, et al., "ScaleHLS: A New Scalable High-Level Synthesis Framework on Multi-Level Intermediate Representation", HPCA, 2022.
[2] Ye, Hanchen, et al., "Invited: ScaleHLS, a Scalable High-Level Synthesis Framework with Multi-level Transformations and Optimizations", DAC, 2022.
[3] Ye, Hanchen, et al., "High-level Synthesis for Domain Specific Computing", ISPD, 2023.

The PyTorch-to-AI Accelerator Compiler



Represent It!

Graph-level IR: TOSA, Linalg, and Tensor dialect.

Loop-level IR: Affine and Memref dialect. Can leverage the transformation and analysis libraries applicable in MLIR.

Directive-level IR: HLSCpp, Affine, and Memref.

Optimize It!

Optimization Passes: Cover the graph, loop, and directive levels. Solve optimization problems at the 'correct' abstraction level. 💪

QoR Estimator: Estimate the latency and resource utilization through IR analysis.

Explore It!

Transform and Analysis Library: Parameterized interfaces of all optimization passes and the QoR estimator. A playground of DSE. 🚀

Automated DSE Engine: Find the Pareto-frontier of the throughput-area trade-off design space.

Enable End-to-end Flow!

HLS C Front-end: Parse C programs into MLIR.

HLS C/C++ Emitter: Generate synthesizable HLS designs for downstream tools, such as Vivado HLS.

- **For the first time**, we can directly take large PyTorch models, go through multiple levels of optimizations, and generate high-quality hardware designs automatically.
- Outperforms an RTL IP-based designs.
- Open-source project: **4000+ downloads** so far from researchers and industrial practitioners around the world.

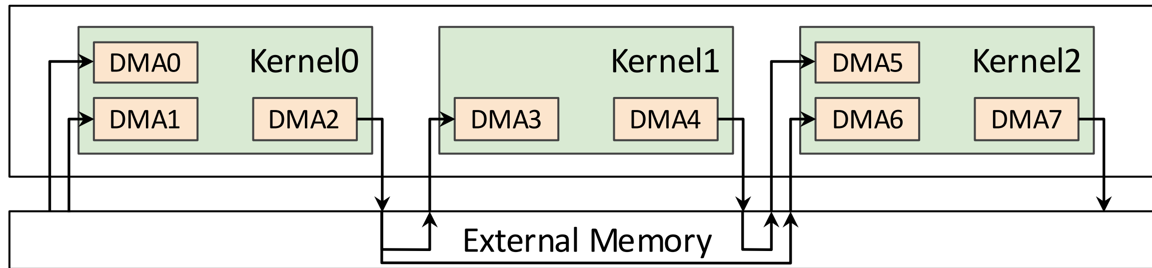
[HPCA'22, DAC'22, DAC'23, TRETS'23, ISPD'23, ASPLOS'24, DAC'24]

- **Lead student, Hanchen Ye, won the IEEE/ACM DAC Ph.D. Forum First Place Winner Award in 2023**
- **Story about this: https://ece.illinois.edu/newsroom/news/59100**

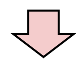
- So far, we’ve only dealt with smaller models

- How about the Memory Wall problem when we are facing large models, such as LLMs?

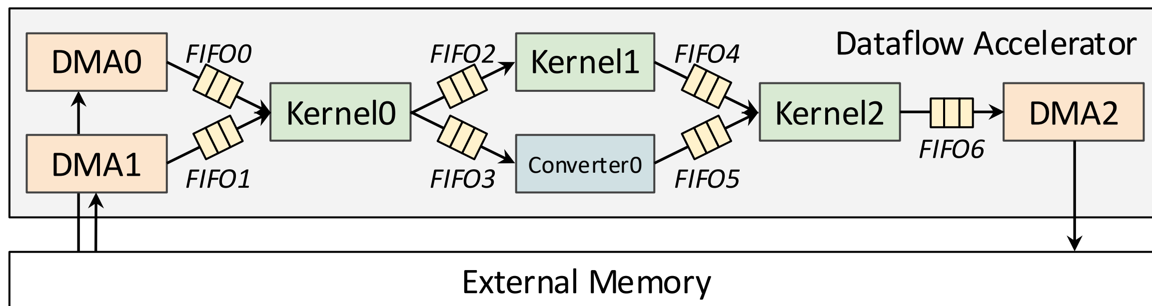
Reduce the Chance to Access Off-Chip Memory



(a) A buffer-based dataflow accelerator

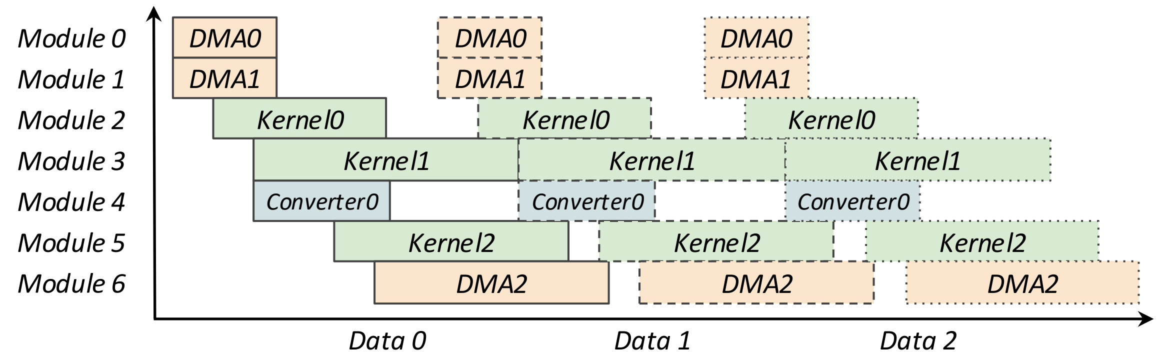


Stream-based Kernel Fusion



(b) A stream-based dataflow accelerator

- Intermediate results are communicated through **on-chip ping-pong buffers**
- If on-chip memory resources are not enough, external memory is used
- Lead to frequent external memory access



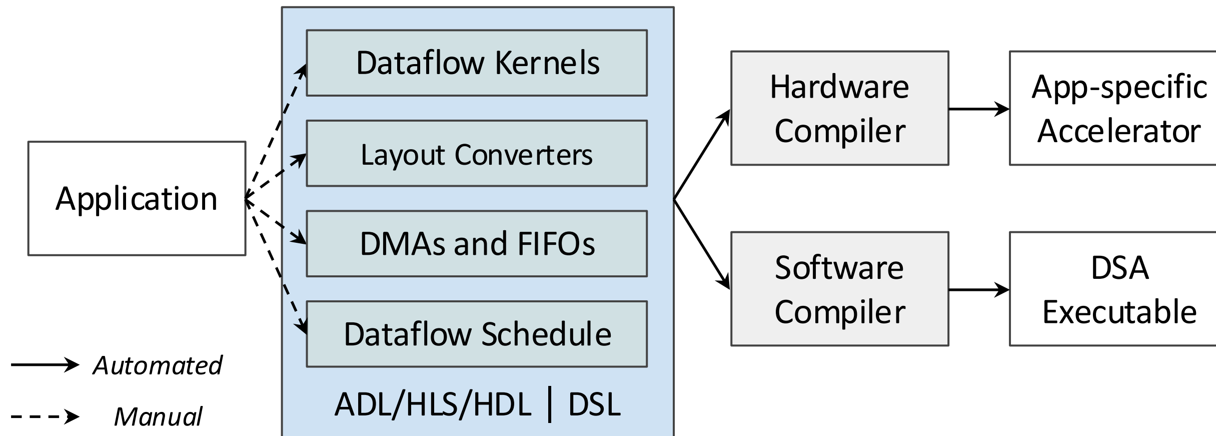
(c) Schedule of the stream-based dataflow accelerator

Reduce on-chip buffer utilization ✓

Reduce external memory access ✓

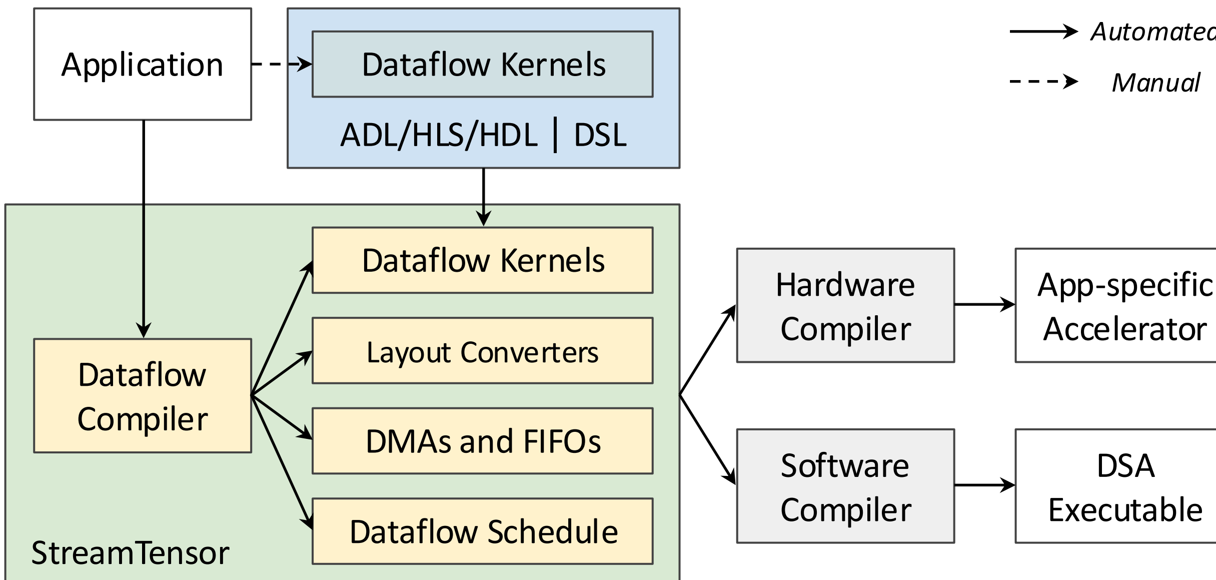
Reduce overall latency & improve throughput ✓

Dataflow Accelerator Design Paradigm



- Manual dataflow kernels, layout converters, DMAs, and FIFOs design
- Difficult to comprehend the optimal solutions of the pitfalls
- Low design productivity

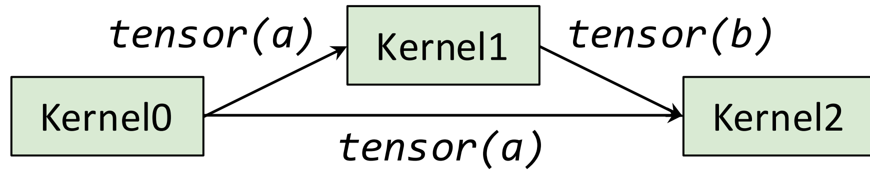
Paradigm Shift

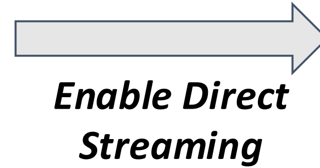


- Automate the generation of layout converters, DMAs, and FIFOs
- Resolve pitfalls through systematic design space exploration
- Support auto-tuned or hand-written kernel integration

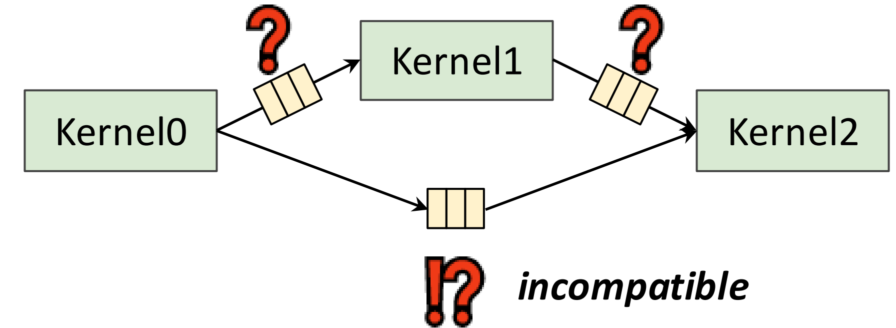
Motivation of Iterative Tensor Type

Tensor-level Graph

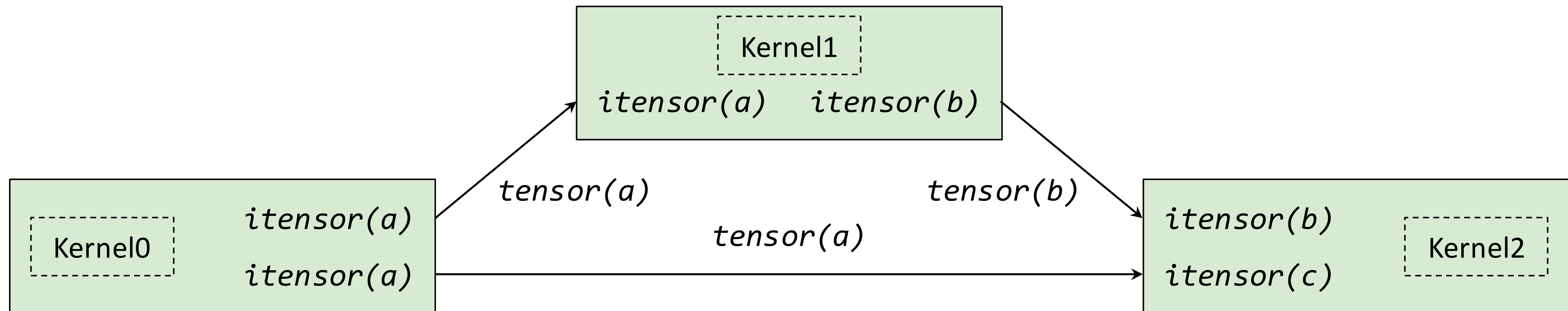




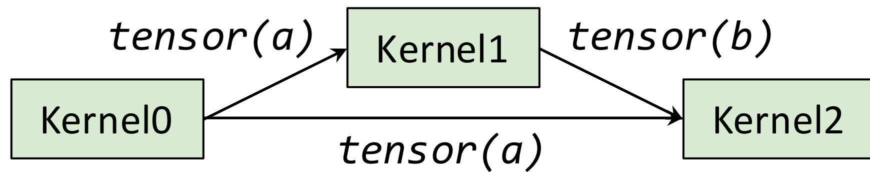
Dataflow Accelerator



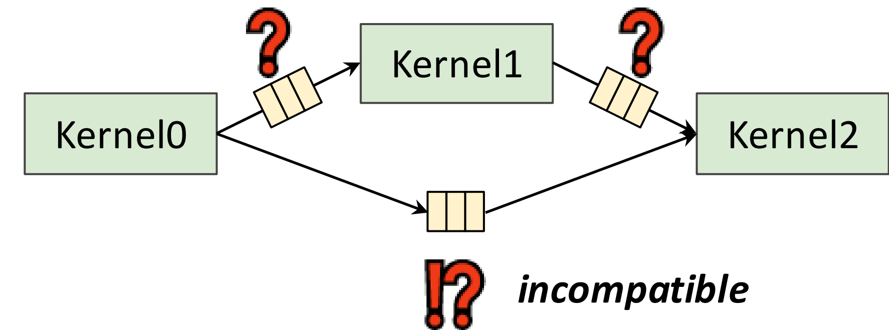
Iterative Tensor-level Graph



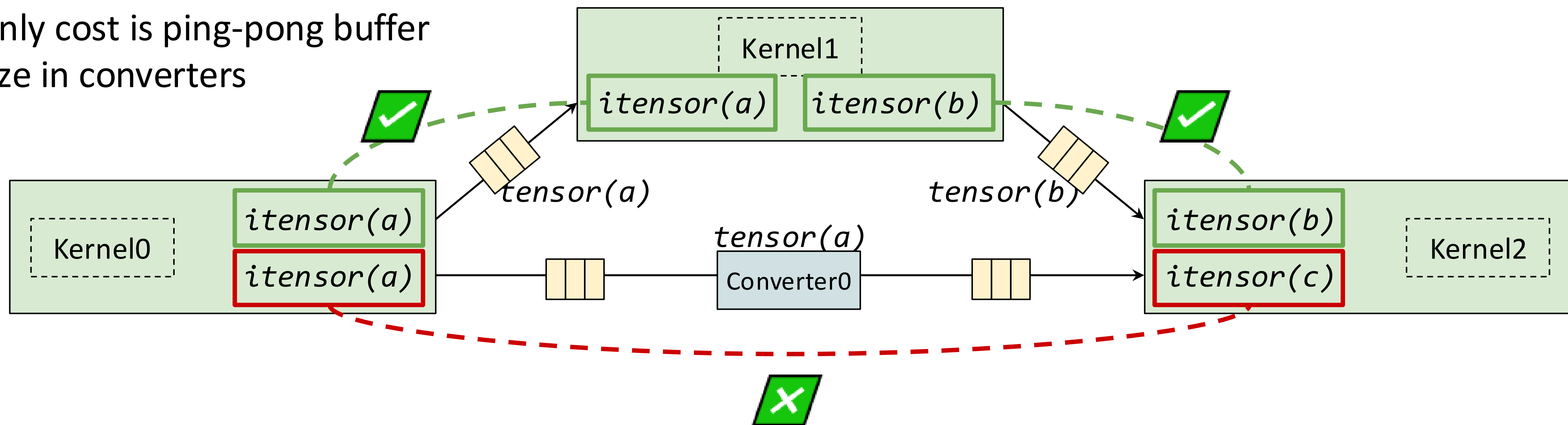
Motivation of Iterative Tensor Type (Cont.)



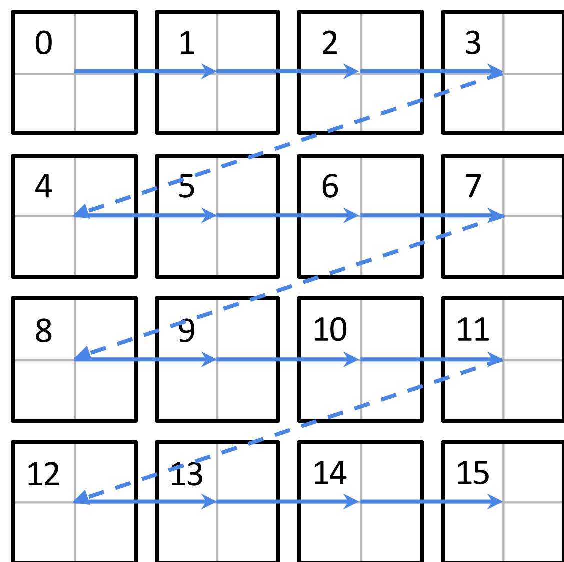
Enable Direct Streaming

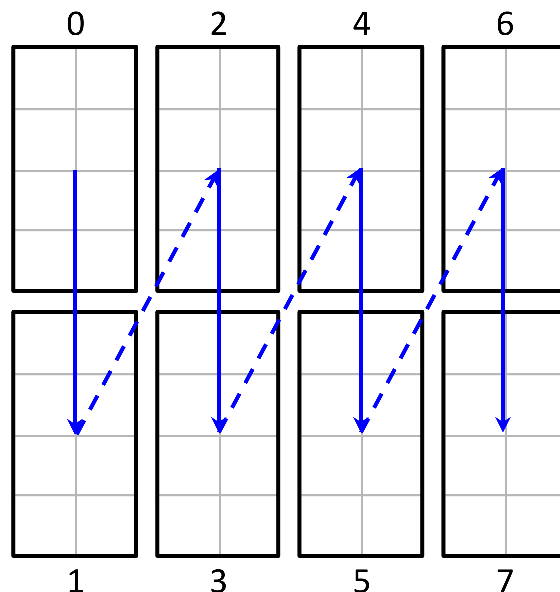


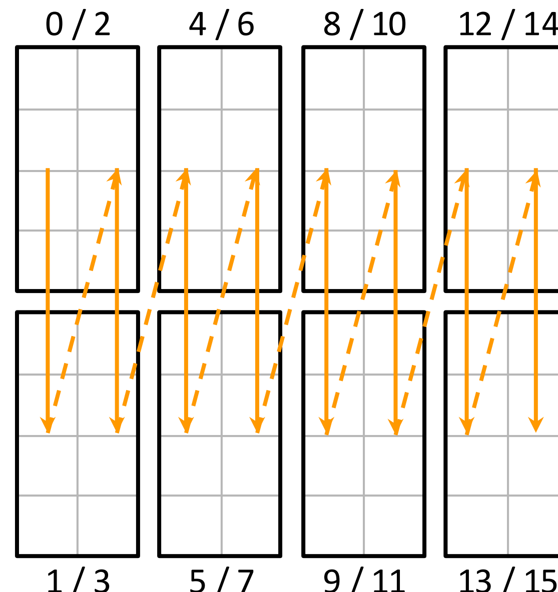
- By design, all kernels can be fused in a streaming manner
- Only cost is ping-pong buffer size in converters



Iterative Tensor Type (Cont.)





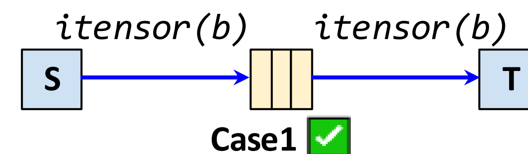


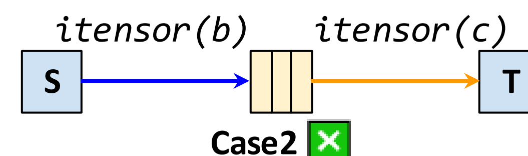
(a) `itensor<2x2xf32,`
`iter_space: [4,4]*[2,2],`
`iter_map: (d0,d1)->(d0,d1)>`

(b) `itensor<4x2xf32,`
`iter_space: [4,2]*[2,4],`
`iter_map: (d0,d1)->(d1,d0)>`

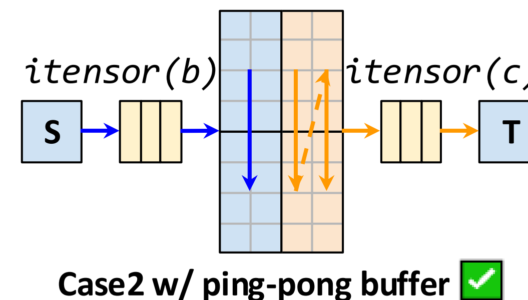
(c) `itensor<4x2xf32,`
`iter_space: [4,2,2]*[2,1,4],`
`iter_map: (d0,d1,d2)->(d2,d0)>`

`tensor<8x8xf32>`



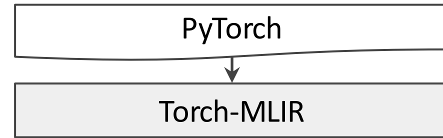


Insert Buffer



The minimal buffer size is inferred from `itensor` types, which are used as "cost" during kernel fusion

StreamTensor Framework Overview



StreamTensor Framework Overview

Passes

Linalg

```
convert_tensor_to_linalg
fuse_elementwise_ops*
fold_unit_extent_dims*
fuse_linalg_fill
```

PyTorch

Torch-MLIR

Linalg Optimization

Linalg Tiling

Design Spaces

Linalg Tiling Space

- Tiling factors
- Unrolling factors
- Vectorization factors
- Permutation

Algorithms

- Naive tiling
- Intensity-driven unroll and vectorization
- Heuristic permutation

StreamTensor Framework Overview

Passes

Linalg

```
convert_tensor_to_linalg
fuse_elementwise_ops*
fold_unit_extent_dims*
fuse_linalg_fill
```

```
convert_linalg_to_kernel
convert_tensor_to_kernel
```

PyTorch

Torch-MLIR

Linalg Optimization

Linalg Tiling

Linalg to Dataflow

Dataflow Kernel Fusion

Design Spaces

Linalg Tiling Space
- Tiling factors
- Unrolling factors
- Vectorization factors
- Permutation

Kernel Fusion Space
- Kernel fusion solution

Algorithms

- Naive tiling
- Intensity-driven unroll and vectorization
- Heuristic permutation

- Heuristic kernel fusion

StreamTensor Framework Overview

StreamTensor Framework Overview

Passes

Linalg

convert_tensor_to_linalg
fuse_elementwise_ops*
fold_unit_extent_dims*
fuse_linalg_fill

convert_linalg_to_kernel
convert_tensor_to_kernel

High-dataflow

pack_kernel_interface
lower_pack_unpack
widen_kernel_interface
materialize_kernel
fold_itensor
vectorize_itensor
ensure_itensor_single_use
convert_empty_to_instance
simplify_task_structure

strategy_merge_task
lower_itensor_to_stream
comprehensive_bufferize
fold_memref_alias_ops*

Low-dataflow

reduce_stream_depth
generate_directive
convert_linalg_to_loops*
affine_loop_normalize*
affine_simplify_structures*
materialize_directive

PyTorch
Torch-MLIR
Linalg Optimization
Linalg Tiling
Linalg to Dataflow
Dataflow Kernel Fusion
High-dataflow Optimization
Dataflow Resource Alloc
Bufferization
Low-dataflow Optimization

Design Spaces

Linalg Tiling Space
- Tiling factors
- Unrolling factors
- Vectorization factors
- Permutation

Kernel Fusion Space
- Kernel fusion solution

Resource Alloc Space
- Stream FIFO sizes
- Memory alloc solution
- Graph partition solution

Algorithms

- Naive tiling
- Intensity-driven unroll and vectorization
- Heuristic permutation

- Heuristic kernel fusion

(Profiling-based)
- LP FIFO sizing
- Heuristic memory alloc
- ILP graph partition

StreamTensor Compilation Pipeline

StreamTensor Framework Overview

StreamTensor Compilation Pipeline

PyTorch → Torch-MLIR → Linalg Optimization → ***Linalg Tiling*** → Linalg to Dataflow → ***Dataflow Kernel Fusion*** → High-dataflow Optimization → ***Dataflow Resource Alloc*** → Bufferization → Low-dataflow Optimization → HLS Codegen / Connectivity Codegen / Runtime Codegen

HLS Codegen → HLS C++

Connectivity Codegen → Link Connectivity Configuration

Runtime Codegen → Host C++, Binary Model Parameters

Passes

Linalg

- convert_tensor_to_linalg
- fuse_elementwise_ops*
- fold_unit_extent_dims*
- fuse_linalg_fill

High-dataflow

- convert_linalg_to_kernel
- convert_tensor_to_kernel

- pack_kernel_interface
- lower_pack_unpack
- widen_kernel_interface
- materialize_kernel
- fold_itensor
- vectorize_itensor
- ensure_itensor_single_use
- convert_empty_to_instance
- simplify_task_structure

Low-dataflow

- strategy_merge_task
- lower_itensor_to_stream
- comprehensive_bufferize
- fold_memref_alias_ops*

- reduce_stream_depth
- generate_directive
- convert_linalg_to_loops*
- affine_loop_normalize*
- affine_simplify_structures*
- materialize_directive

Func

- convert_dataflow_to_func
- generate_connectivity
- convert_memref_to_pointer
- generate_runtime_host

Design Spaces

Linalg Tiling Space

- Tiling factors
- Unrolling factors
- Vectorization factors
- Permutation

Kernel Fusion Space

- Kernel fusion solution

Resource Alloc Space

- Stream FIFO sizes
- Memory alloc solution
- Graph partition solution

Algorithms

- Naive tiling
- Intensity-driven unroll and vectorization
- Heuristic permutation

- Heuristic kernel fusion

(Profiling-based)

- LP FIFO sizing
- Heuristic memory alloc
- ILP graph partition

StreamTensor Framework Overview

Passes

Linalg

convert_tensor_to_linalg
fuse_elementwise_ops*
fold_unit_extent_dims*
fuse_linalg_fill

convert_linalg_to_kernel
convert_tensor_to_kernel

High-dataflow

pack_kernel_interface
lower_pack_unpack
widen_kernel_interface
materialize_kernel
fold_itensor
vectorize_itensor
ensure_itensor_single_use
convert_empty_to_instance
simplify_task_structure

Low-dataflow

strategy_merge_task
lower_itensor_to_stream
comprehensive_bufferize
fold_memref_alias_ops*

reduce_stream_depth
generate_directive
convert_linalg_to_loops*
affine_loop_normalize*
affine_simplify_structures*
materialize_directive

Func

convert_dataflow_to_func
generate_connectivity
convert_memref_to_pointer
generate_runtime_host

StreamTensor Compilation Pipeline

PyTorch
Torch-MLIR
Linalg Optimization
Linalg Tiling
Linalg to Dataflow
Dataflow Kernel Fusion
High-dataflow Optimization
Dataflow Resource Alloc
Bufferization
Low-dataflow Optimization
HLS Codegen | Connectivity Codegen | Runtime Codegen
HLS C++ | Link Connectivity Configuration | Host C++ | Binary Model Parameters
Vendor Compile (e.g., Vitis) | g++
Vendor Link (e.g., Vitis) | Host Executable
Device Executable / Bitstream
Device + Host

Design Spaces

Linalg Tiling Space
- Tiling factors
- Unrolling factors
- Vectorization factors
- Permutation

Kernel Fusion Space
- Kernel fusion solution

Resource Alloc Space
- Stream FIFO sizes
- Memory alloc solution
- Graph partition solution

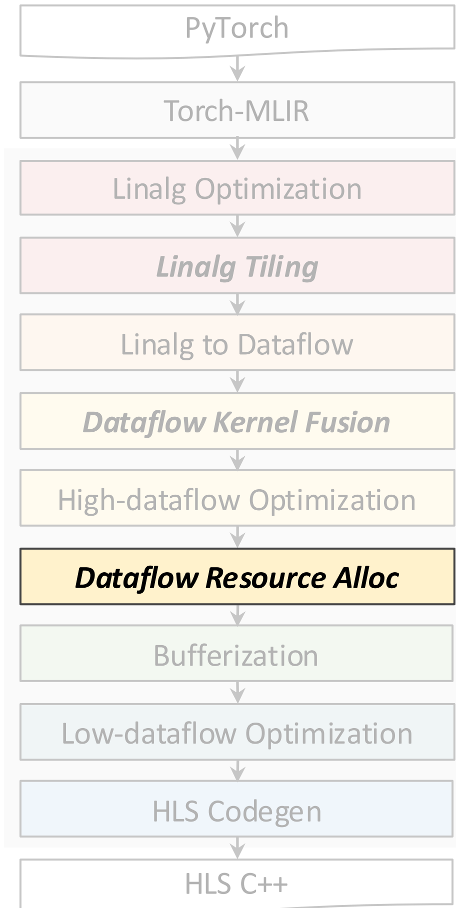
Algorithms

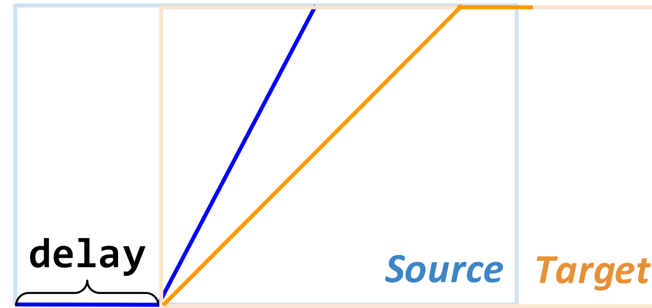
- Naive tiling
- Intensity-driven unroll and vectorization
- Heuristic permutation

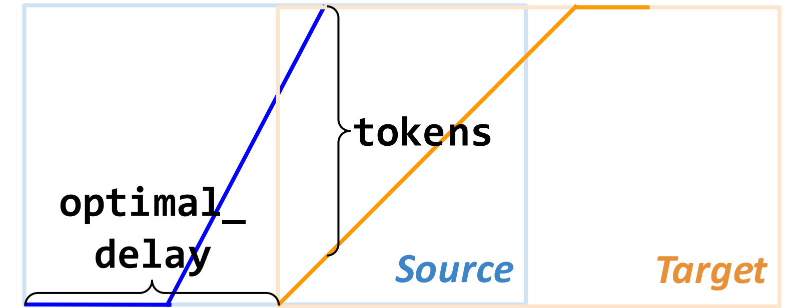
- Heuristic kernel fusion

(Profiling-based)
- LP FIFO sizing
- Heuristic memory alloc
- ILP graph partition

Linear Programming (LP) Formulation for Resource Alloc







(1) Calculate token production/consumption curves based on kernel profiling results

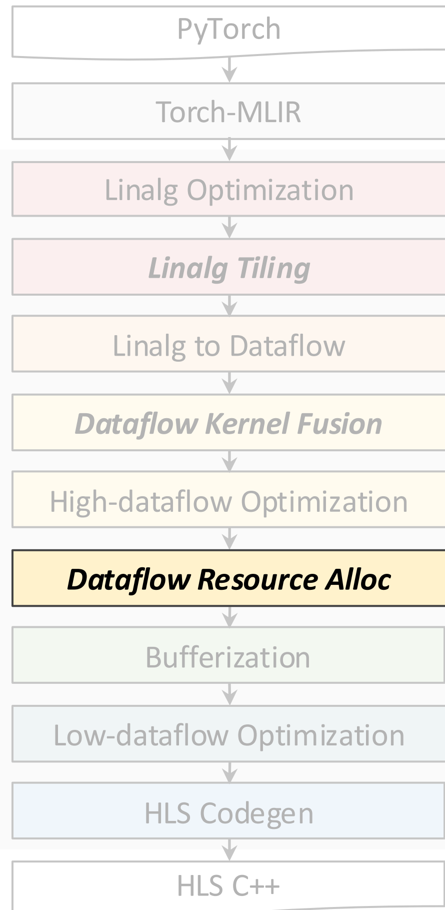
(2) Calculate minimum `delay` given a strategy

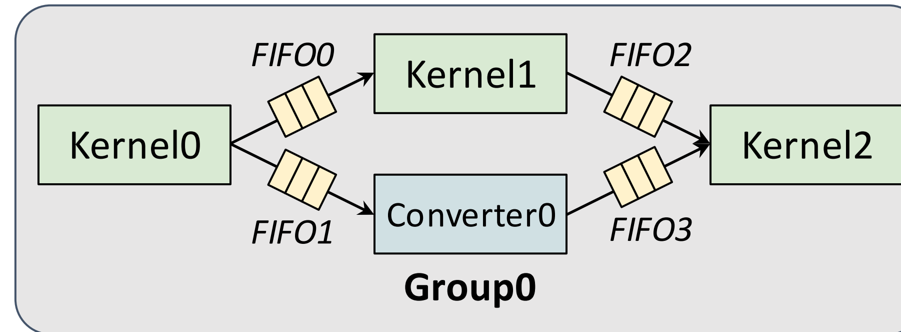
(3) Solve `optimal_delay` with LP

(4) Calculate `tokens` based on `optimal_delay` given a strategy

Transform FIFO sizing into a scheduling problem

Linear Programming (LP) Formulation (Cont.)





Delay Variables

- var["fifo0"]
- var["fifo1"]
- var["fifo2"]
- var["fifo3"]

Objective

- obj = var["fifo0"] + var["fifo1"] + var["fifo2"] + var["fifo3"]
- minimize obj

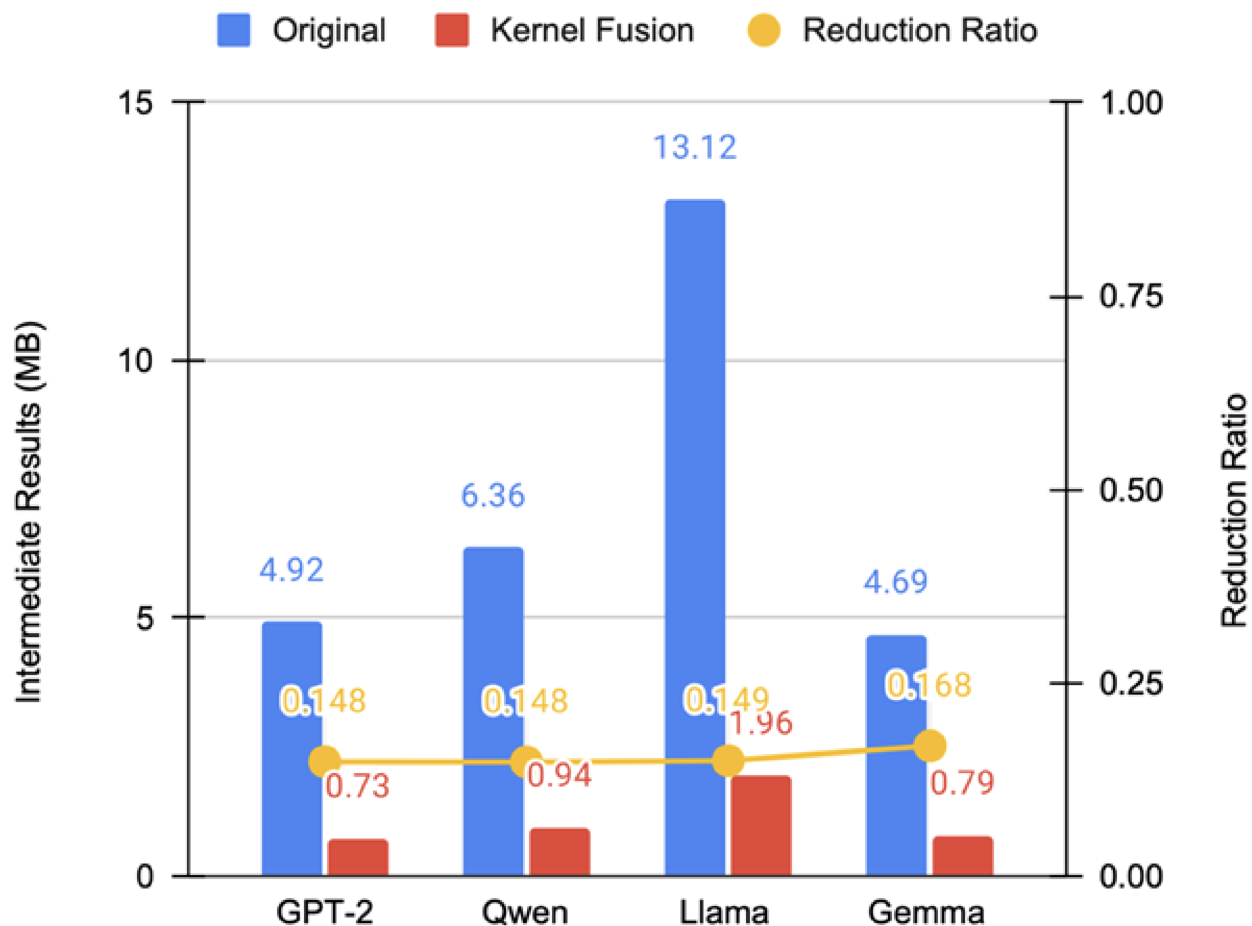
Minimum delay Constraints

- var["fifo0"] >= delay["kernel0"]
- var["fifo1"] >= delay["kernel0"]
- var["fifo2"] >= delay["kernel1"]
- var["fifo3"] >= delay["Converter0"]

Path Balance Constraints

- var["fifo0"] + var["fifo2"] >= critical_delay["kernel0"]["kernel2"]
- var["fifo1"] + var["fifo3"] >= critical_delay["kernel0"]["kernel2"]

On-chip Memory Reduction through Kernel Fusion



- Only consider intermediate results, i.e., activations, in this study.
- Parameters are always stored in external memory.
- On-chip memory are reduced to **0.15x - 0.17x** through stream-based kernel fusion.
- Without kernel fusion, the on-chip memory resources are not enough to store all intermediate results

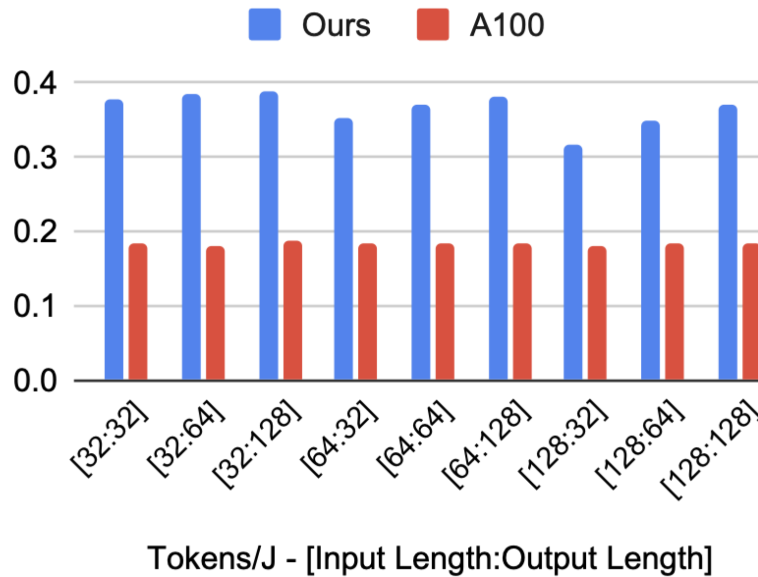
FPGA On-board Results on GPT-2

[Input Len: Output Len]	Ours			Allo [15] (Ratio of $\frac{Ours}{Allo}$)			DFX [29] (Ratio of $\frac{Ours}{DFX}$)		
	Latency (ms)	TTFT (ms)	Speed (token/s)	Latency (ms)	TTFT (ms)	Speed (token/s)	Latency (ms)	TTFT (ms)	Speed (token/s)
[32:32]	194.99	34.59	199.51	238.32 (0.82x)	81.50 (0.42x)	204.05 (0.98x)	350.00 (0.56x)	177.20 (0.20x)	185.19 (1.08x)
[64:64]	358.24	61.27	215.51	476.64 (0.75x)	162.99 (0.38x)	204.05 (1.06x)	694.70 (0.52x)	349.10 (0.18x)	185.19 (1.16x)
[128:128]	696.65	125.35	224.05	953.28 (0.73x)	325.98 (0.38x)	204.05 (1.10x)	1384.00 (0.50x)	692.80 (0.18x)	185.19 (1.21x)
[256:256]	1387.76	272.85	229.61	1906.56 (0.73x)	651.96 (0.42x)	204.05 (1.13x)	2800.00 (0.50x)	1417.60 (0.19x)	185.19 (1.24x)
Geo. Mean	-	-	-	**0.76x**	**0.40x**	**1.06x**	**0.52x**	**0.19x**	**1.17x**

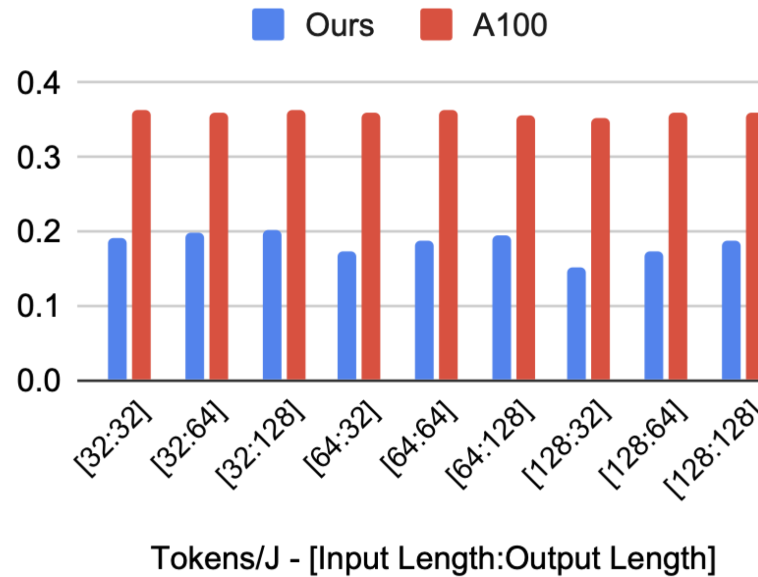
[Input Len: Output Len]	Ours			A100 (Ratio of $\frac{Ours}{A100}$)			2080Ti (Ratio of $\frac{Ours}{2080Ti}$)		
	Latency (ms)	TTFT (ms)	Speed (token/s)	Latency (ms)	TTFT (ms)	Speed (token/s)	Latency (ms)	TTFT (ms)	Speed (token/s)
[32:32]	194.99	34.59	199.51	291.16 (0.67x)	8.72 (3.97x)	113.30 (1.76x)	518.46 (0.38x)	24.98 (1.38x)	64.85 (3.08x)
[64:64]	358.24	61.27	215.51	567.41 (0.63x)	8.76 (6.99x)	114.56 (1.88x)	1010.81 (0.35x)	25.23 (2.43x)	64.94 (3.32x)
[128:128]	696.65	125.35	224.05	1118.28 (0.62x)	8.65 (14.49x)	115.35 (1.94x)	3969.76 (0.18x)	25.26 (4.96x)	32.45 (6.90x)
[256:256]	1387.76	272.85	229.61	2227.79 (0.62x)	8.53 (31.99x)	115.35 (1.99x)	7914.23 (0.18x)	25.23 (10.81x)	32.45 (7.08x)
Geo. Mean	-	-	-	**0.64x**	10.65x	**1.89x**	**0.25x**	3.67x	**4.73x**

	Ours	**Allo [15]**	**DFX [29]**	**A100**	**2080**
Platform	AMD U55C	AMD U280	AMD U280	NVIDIA A100	NVID 2080
Process Node	16nm	16nm	16nm	7nm	12n
Freq. (MHz)	250	250	200	1065	135
Quantization	W4A8	W4A8	FP16	W8A8	W8
Thermal Design Power	150W	225W	225W	300W	250
Peak Perf. (INT8 TOPS)	24.5	24.5	24.5	624	215.
Off-chip Memory	16GB HBM 460GB/s	8GB HBM 460GB/s	8GB HBM 460GB/s	80GB HBM 1935GB/s	11G DD 616G
On-chip Memory	41MB	41MB	41MB	40MB	5.5M

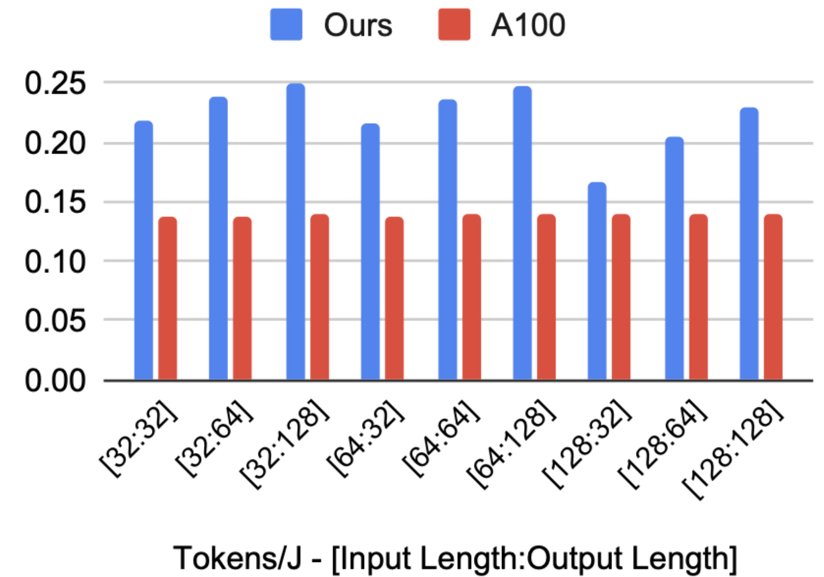
FPGA On-board Results on Other LLMs



(a) Qwen.

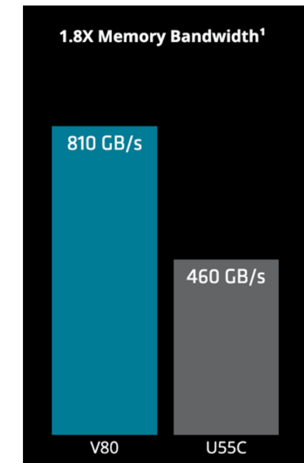


(b) Llama.



(c) Gemma.

- Achieved higher energy efficiency on Qwen and Gemma.
- Energy efficiency of Llama is lower than GPU because the intermediate results of Llama is larger than other models, limiting the design space explorations and reducing the execution overlap between dataflow kernels.
- Currently working to support the V80 board – more memory both on-chip and off-chip.

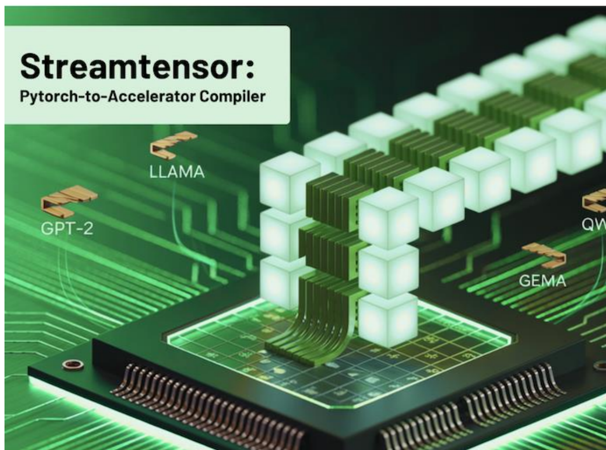


StreamTensor: Make Tensors Stream in Dataflow Accelerators for LLMs [MICRO'25]

Technology | Artificial Intelligence | Editors Pick | Hardware | Staff

StreamTensor: A PyTorch-to-Accelerator Compiler that Streams LLM Intermediates Across FPGA Dataflows

By Michal Sutter - October 5, 2025



Why treat LLM inference as batched kernels to DRAM when a dataflow compiler can pipe tiles through on-chip FIFOs and stream converters?StreamTensor is a compiler that lowers PyTorch LLM graphs (GPT-2, Llama, Qwen, Gemma) into stream-scheduled dataflow accelerators on AMD's Alveo U55C FPGA. The system introduces an *iterative tensor* ("itensor") type to encode tile/order of streams, enabling provably correct inter-kernel streaming

Ali Irturk · 1st
Chief Technology Officer (CTO) at Rithum | AI & ML Leader | Angel Invest...
1mo ·

Forget faster math, the next revolution in AI hardware is all about smoother flow.

I came across a fascinating paper from UIUC called "StreamTensor: Make Tensors Stream in Dataflow Accelerators for LLMs."

It tackles a problem most of us overlook when we think about model performance: not compute, but movement. Every layer in a large model writes its results to memory, then waits for the next layer to fetch them back. It's like cooking each dish, putting it in the fridge, and reheating it before the next step.

StreamTensor asks: What if we just kept the data flowing?

It turns the entire model into a streaming pipeline, where tensors flow directly from one operation to the next, never stopping to rest in memory.

The parts that really stood out:
* A clever new "itensor" typing system that describes how data moves, not just what it is.
* A compiler that automatically figures out the best way to tile, fuse, and allocate hardware, so the whole model behaves like a well-choreographed factory line.
* And even a mathematical model that right-sizes the tiny buffers between layers to keep everything moving smoothly.

The results are impressive: roughly 2× better energy efficiency and ~40% faster inference on transformer workloads compared to GPUs.

But what I found most interesting is the shift in mindset...

For years, AI hardware has been compute-centric, measured in FLOPs. StreamTensor makes it flow-centric, measured in how efficiently we can move

Rayudu Sunilvikas · 3rd+ + Follow
Attended Kakinada Institute of Engineering and Technology
1w ·

PyTorch models run everywhere except efficiently on FPGAs. StreamTensor transforms the code into streaming dataflow that matches how the hardware actually works.

This breakthrough changes everything for LLM deployment.

Traditional FPGA approaches use batched kernels writing to memory. StreamTensor does something different. It streams data through on-chip FIFOs instead.

The results speak for themselves:

•0.64× lower latency vs GPUs
•1.99× better energy efficiency vs NVIDIA A100
•0.76× faster than previous FPGA accelerators

The magic happens in three key areas:

⚡ Stream-scheduled dataflow architecture
⚡ Hierarchical design space exploration
⚡ Linear programming for optimal FIFO sizing

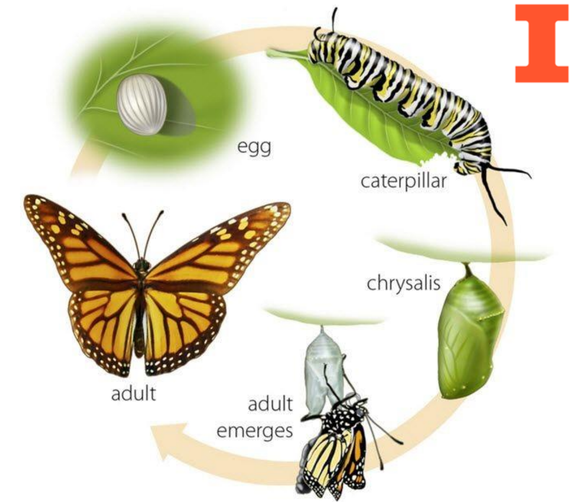
This isn't just academic research. StreamTensor works with real models like GPT-2, Llama, Qwen, and Gemma on AMD's Alveo U55C platform.

The compiler handles the complex translation from PyTorch to hardware automatically. No more manual optimization headaches.

Memory usage drops to just 15% of original designs through smart kernel fusion. That's massive.

One Promising Direction: LLM-Aided Design (LAD)

LLM-Aided Design (LAD)



- **LLM-Aided Debugging for HLS and RTL Code**:
  - **Automation**: Developing automated flows for bug injection through LLMs for high-quality buggy code generation. E.g., *Chrysalis* dataset [LAD'24].
  - **DebugGPT**: Fine-tuning LLMs on HLS & RTL codes on buggy version and bug-free version.
- **LLM-Aided Formal Verification**:
  - **Assertion Generation**: Enhancing formal verification by generating precise assertions for proof of correctness.
  - **Iterative Refinement**: Using feedback loops between theorem provers and LLMs to refine and verify assertions, increasing productivity and reliability.
- **LLM-Aided Design Automation**:
  - **Engineer Oversight**: Engineers provide specifications/feedback and review LLM outputs to ensure alignment with design intentions – an iterative process with human in the loop.
  - **Multi-Agent Systems**: Deploying specialized LLM agents for managing design, verification, and debugging phases.
  - **Efficiency and Optimization**: Creating an efficient pipeline for high-performance designs and comprehensive verification results.
- **Many Others:**
  - Benchmarks & Datasets for LAD
  - Data Privacy-Preserving LLM Frameworks
  - …

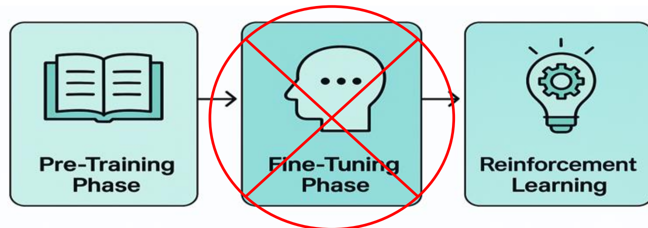
IEEE International Conference on LLM-Aided Design, 2026

Time: July 30-31, 2026
Location: Stanford University, Stanford, CA

LLM for Hardware Design Can Be Expensive!

Motivation:

- **LLM Limitations:** Despite breakthroughs (ChatGPT-4/5, Qwen, DeepSeek), LLMs often generate code that fails formal verification.
- **Cost of Fine-Tuning:** Traditional fine-tuning is resource-intensive and may lead to catastrophic forgetting.
- **Domain-Specific Needs:** Critical domains (hardware design, safety-critical systems) require provable correctness.



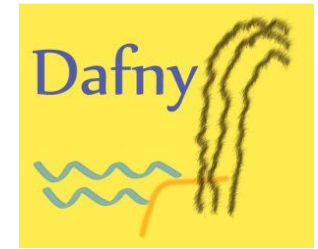
Model Size	Strategy	GPU Hours
~100M	Full Fine-Tuning	~5–10 hours
~ 100M	PEFT (LoRA)	~1–3 hours
~1B	Full Fine-Tuning	~20–40 hours
~1B	PEFT	~5–10 hours
~7B	Full Fine-Tuning	~50–100 GPU hours
~7B	PEFT (LoRA/Adapters)	~10–20 GPU hours
~175B	Full Fine-Tuning	Hundreds to thousands of GPU hours
~175B	PEFT	~10–20% of full fine-tuning cost

Model	#Param Count
Chat-GPT 4	1.76 T*
Llama	8B; 70B; 450B
DeepSeek-R1	671B
Qwen	0.5B – 72B
Gemini Pro	500B*

Food for thought:

Are we over-fitting the models? Can we reduce the resource complexity needed for fine-tuning?

Our Hypothesis and Correctness by Construction!



Our Hypothesis:

- Existing LLMs already contain rich latent reasoning. With the right prompt repair, guided by verifier feedback, we can steer them toward generating verified code without full-scale fine-tuning.

Problem Statement:

- How do we generate correct-by-construction code efficiently without fine-tuning a model?

Key Insight:

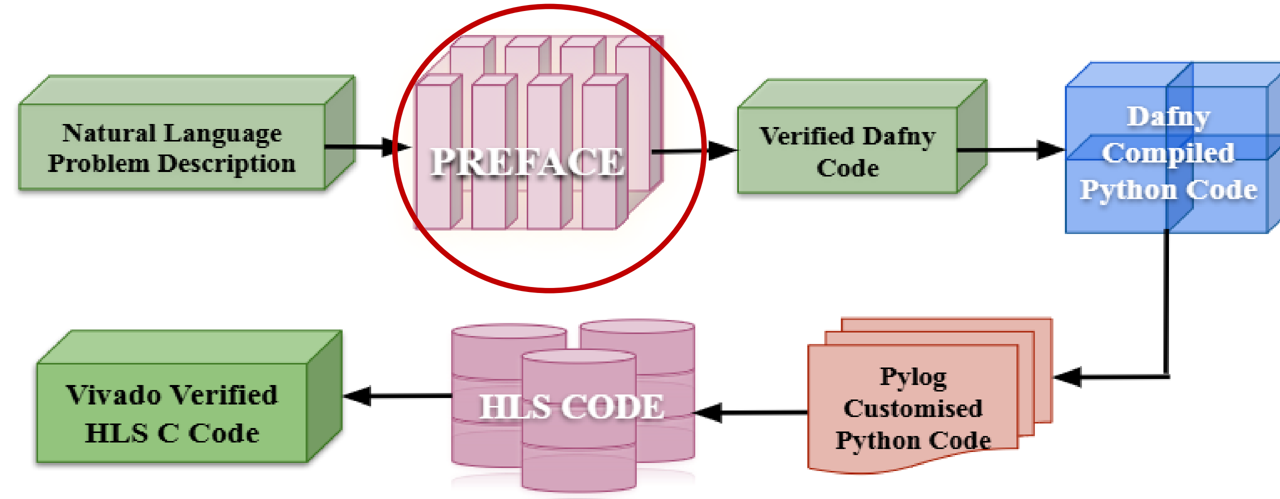
- Exploit LLMs' latent reasoning using a reinforcement learning (RL) agent that optimizes prompts based on verifier feedback rather than fine-tuning the entire model.

What will be the input spec?

- **Dafny** is a programming language with built-in support for **formal verification** through preconditions, postconditions, and loop invariants.
- It allows **specification and implementation in the same language**, making it ideal for correctness-by-construction.
- Dafny integrates with **Z3** for proof automation, removing the need for external proof engineering.
- Outputs detailed error messages and counts, which is ideal for reward generation.
- It has an inbuilt compiler to convert to languages like Python.

[SRC TECHCON'25]

Proof2Silicon: The Complete Flow



- **Natural Language Spec → PREFACE** (RL-based Prompt Repair)
  - Small Language Model (SLM) optimizes prompts for a frozen LLM
  - Verifier feedback used as RL reward → generates verified Dafny code
- **Dafny → Clean Python**
  - Dafny compiler converts verified code into Python
  - Strip Dafny-specific wrappers & replace math ops with NumPy equivalents
- **Python → HLS C via PyLog [1]**
  - Use PyLog decorators for hardware-aware optimizations: Loop unrolling | Memory partitioning | Resource balancing
- **HLS C → RTL (Vivado HLS)**
  - Synthesizes FPGA-ready RTL on Zynq-7000 SoC

[1] Sitao Huang, et.al., "PyLog: An Algorithm-Centric Python-Based FPGA Programming and Synthesis Flow," *IEEE TC*, 2021.

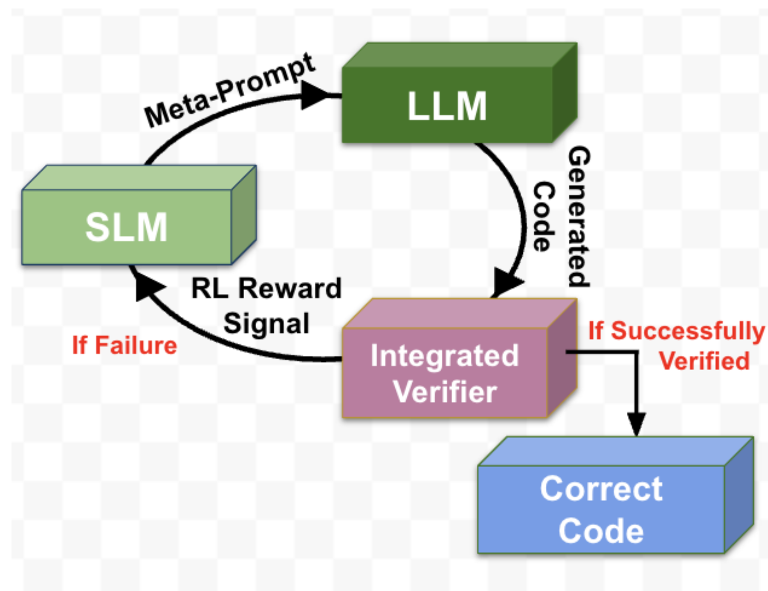
PREFACE Engine: SLM-LLM Collaborative Framework

➢ **Separation of Responsibilities:**

- **Small Language Model (SLM):** (we use DeepSeek-R1-Distill-Qwen-1.5B)
  - Acts as the reasoning and prompt optimization agent.
  - Learns to interpret verifier errors and decide on prompt modifications.
  - Trained via reinforcement learning.
- **Frozen LLM:**
  - Performs the heavy lifting of code generation.
  - Remains unchanged during training to retain general-purpose coding capabilities.

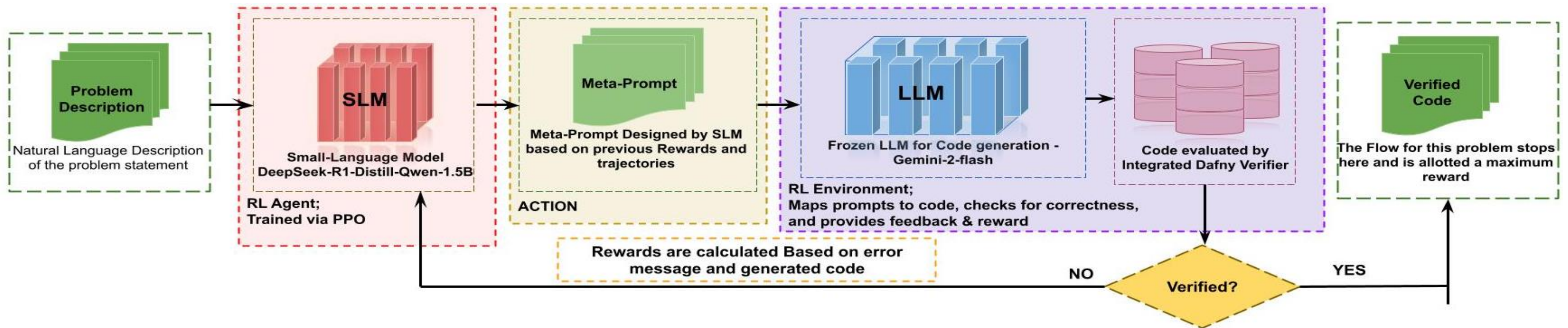
➢ **Advantages:**

- Modularity: Upgrading the code-generation LLM does not require retraining the SLM.
- Efficiency: The SLM is lightweight, making the RL process computationally tractable.
- Data Efficiency: Learns prompt adaptation online using verifier outcomes as feedback.



Overview of the RL prompt-repair loop. A frozen LLM generates Dafny code, which is verified. Based on verifier feedback, the RL agent proposes prompt edits. This loop continues until verification is successful or there are max iterations.

PREFACE Engine: System Pipeline Overview



➢ Initial Prompt & Code Generation:

- Start with a natural language task specification.
- Construct an initial prompt and feed it into a frozen LLM to generate candidate Dafny code.

➢ Verification Feedback Loop:

- Dafny Verifier: Checks code correctness, returning detailed error messages (failed invariants, precondition violations).
- Error Parsing: Error messages are parsed and stored in metadata (each data point contains the error code, message, reward, prompt, and depth).

➢ RL-Guided Prompt Adaptation:

- An SLM (RL policy agent) analyzes verifier feedback. The agent proposes corrective prompt modifications based on the RL policy.
- States are current prompts/code, actions are prompt updates, and rewards are derived from verifier outcomes.

PREFACE Engine: RL Formulation (Stage 1)

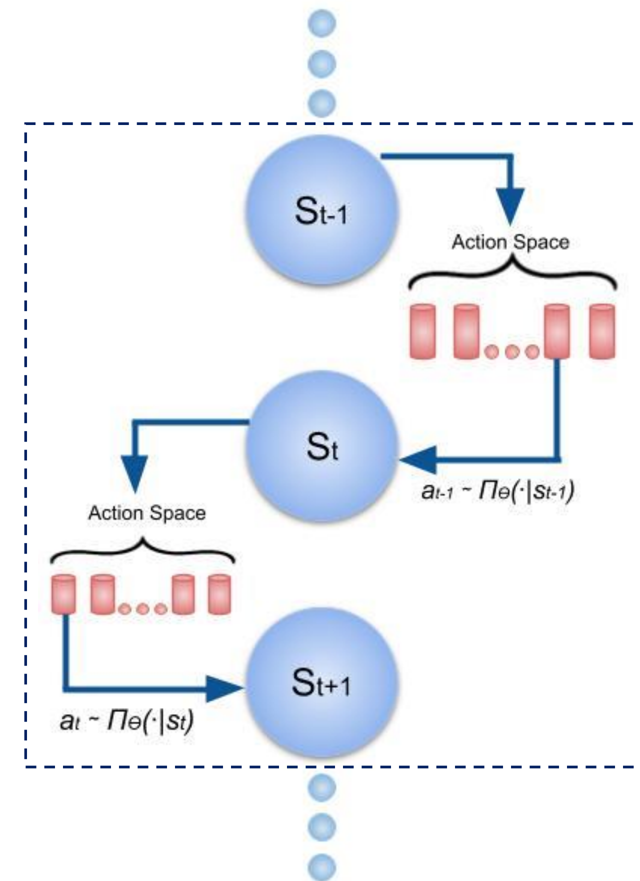
➢ Markov Decision Process (MDP) formulation for Prompt Adaptation:

$$G = (S, A, T, R, y)$$

- **State *S*:**
  - Each state $s_t \epsilon S$ encodes three pieces of information.
  - The current prompt p_t, generated code c_t, and verifier log $\{e_t, o_t\}$ (error message and error count).
- **Action *A*:**
  - Each action $a_t \epsilon A$ represents a proposed prompt modification (natural language).
  - The policy network $\pi_\Theta(a_t|s_t)$, where Θ are learnable parameters.
  - The policy is a conditional distribution over strings given the state :

$$\pi_\Theta(a_t|s_t) = \prod_{i=1}^{|a_t|} \pi_\theta \left(a_t^{(i)} \mid s_t, a_t^{(<i)} \right),$$

  where $a_t^{(i)}$ is the i-th token and $a_t^{(<i)}$ are preceding tokens.
- **Transition Dynamics *T*:**
  - The transition dynamics are governed by the interaction between the edited prompt and the Dafny code generator (LLM).



Policy-conditioned action sampling in an MDP. At each step, the agent samples an action from the policy's distribution over the action space given the current state

PREFACE Engine: RL Formulation (Stage 1)

- **Markov Decision Process (MDP) formulation for Prompt Adaptation:**

$$G = (S, A, T, R, y)$$

- **Reward *R*:**

$$R(s_t, a_t) = \begin{cases} +R_{\text{succ}}, & \text{if } e_{t+1} = 0, \\ -\alpha\, e_{t+1} - \beta, & \text{if } e_{t+1} > 0, \end{cases}$$

  - Large positive reward if the generated code verifies.
  - Large negative reward if an empty file is generated to prevent reward hacking.
  - Shaped penalties based on the number and severity of failed proof obligations and step penalties: $-\alpha \cdot e_t - \beta$

    e_t denotes the failed proof obligations at step t, α is the per-error penalty, and β is the per-iteration penalty.

- **Discount Factor *y* :**
  - After the reward is generated, the model learns the policy Π_θ to maximize the expected discounted cumulative reward.

$$\mathbb{E}\left[\sum_{t=0}^{\infty} \gamma^t R(s_t, a_t)\right].$$

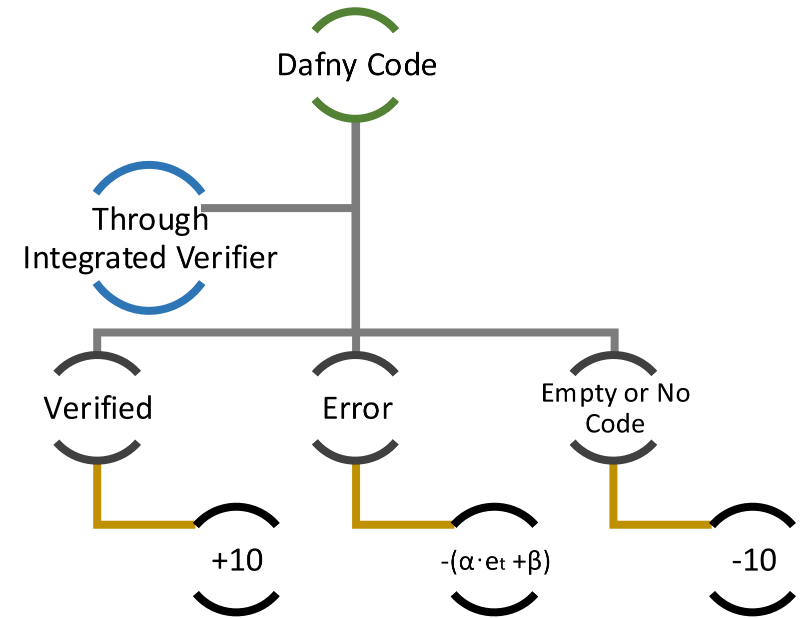


Figure 5: PREFACE RL-agent reward workflow.

Proof2Silicon: The Complete Flow with Examples

DAFNY CODE

```
method Abs(x: int) returns (y: int)
    ensures x >= 0 ==> y == x
    ensures x < 0 ==> y + x == 0
{
    if (x >= 0) {
        y := x;
    } else {
        y := -x;
    }
}
```

Generated by Frozen LLM (ChatGPT-o1-mini) via the Prompt Generated by trained SLM and verified by the integrated verifier.

```
import sys
from typing import Callable, Any, TypeVar, NamedTuple
from math import floor
from itertools import count

import module_ as module_
import _dafny as _dafny
import System_ as System_

# Module: module_

class default__:
    def __init__(self):
        pass

    @staticmethod
    def Abs(x):
        y: int = int(0)
        if (x) >= (0):
            y = x
        elif True:
            y = (0) - (x)
        return y
```

PYTHON Via Dafny Compiler

```
import sys
sys.path.append("/mnt/shared/gpfs/home/manvij2")
from pylog import *
import numpy as np

@pylog(mode='cgen')

def Abs(x):
    y = 0
    if x >= 0:
        y = x
    else:
        y = 0 - x
    return y

if __name__ == "__main__":
    Abs(np.int32(10))
```

Customized PYTHON with Test Case

```
#include "ap_int.h"
#include "ap_fixed.h"
#include "hls_math.h"

int Absss(int x)
{
    #pragma HLS INTERFACE s_axilite register port=x bundle=ctrl
    #pragma HLS INTERFACE s_axilite register port=return bundle=ctrl
    int y = 0;
    if (x >= 0)
    {
        y = x;
    }
    else
    {
        y = 0 - x;
    }

    return y;
}
```

HLS C via PyLog

```
================================================================
== Synthesis Summary Report of 'Absss'
================================================================
+ General Information:
    * Date:           Tue Jul 15 23:55:45 2025
    * Version:        2024.2 (Build 5238294 on Nov  8 2024)
    * Project:        Absss
    * Solution:       sol1 (Vivado IP Flow Target)
    * Product family: zynq
    * Target device:  xc7z020-clg484-1


+ Performance & Resource Estimates:

    PS: '+' for module; 'o' for loop; '*' for dataflow
```

Modules & Loops	Issue Type	Slack	Latency (cycles)	Latency (ns)	Iteration Latency	Interval	Trip Count	Pipelined	BRAM	DSP	FF	LUT	URAM
+ Absss	-	3.05	1	10.000	-	2	-	no	-	-	140 (~0%)	253 (~0%)	-

HLS Synthesis Report via Vivado HLS

Proof2Silicon end-to-end workflow: an RL-driven Small Language Model (SLM) iteratively refines meta-prompts to a frozen LLM under guidance from the integrated Dafny verifier, producing formally verified Dafny code. Verified code is then compiled to Python, transformed into hardware-aware C via the PyLog backend, and synthesized to RTL with Vivado HLS on a Zynq-7000 SoC.

Initial LLM Generation Results (Stage 1)

Model	Without SLM (baseline)		Without Trained SLM		With Trained SLM	
	W/O Feedback	With Feedback	W/O Feedback	With Feedback	W/O Feedback	With Feedback
ChatGPT-4o	25%	36%	10% (-15%)	44% (+8%)	23% (-2%)	50% (+14%)
ChatGPT-o1-mini	23%	35%	8% (-14%)	42% (+7%)	23% (±0%)	52% (+17%)
Qwen2.5-Coder-14B	15%	21%	2% (-13%)	12% (-9%)	16% (+1%)	31% (+10%)
Qwen2.5-7B	3%	7%	0% (-3%)	4% (-3%)	3% (±0%)	11% (+4%)
Gemini-2-Flash	20%	33%	3% (-17%)	32% (-1%)	29% (+9%)	55% (+21%)

Verification success rates on a 100-task benchmark for various LLMs using prompts through PREFACE pipeline from both untrained and 100-hr-trained small language models, comparing single-shot (W/O Feedback) versus iterative error-feedback (With Feedback) modes. We employ the "One-line and Detailed Description" prompting mechanism; values are color-coded green for improvements and red for degradations relative to their respective baseline values.

- **W/O Feedback** indicates that only the initial prompt is used and is sufficient to produce valid code.
- **With Feedback** indicates that the initial prompt is insufficient, and error feedback is iteratively provided to the SLM to generate revised prompts, for up to seven iterations.

Initial Hardware Results (Stage 1)

Model	Feedback	Verified Count	Compiled to HLS	HLS Synthesized	Avg. Elpased Time	Avg. Peak Memory
ChatGPT-4o	No	23	17	12 (52.1%)	32.98 sec	705.41 MB
ChatGPT-4o	Yes	50	29	25 (50%)	32.96 sec	678.51 MB
ChatGPT-o1-mini	No	23	17	12 (52.1%)	32.83 sec	677.47 MB
ChatGPT-o1-mini	Yes	52	35	29 (55.8%)	32.84 sec	682.46 MB
Qwen2.5-Coder-14B	No	16	12	10 (62.5%)	33.98 sec	678.56 MB
Qwen2.5-Coder-14B	Yes	31	23	17 (54.8%)	33.47 sec	681.97 MB
Qwen2.5-7B	No	3	3	2 (66.67%)	33.51 sec	685.46 MB
Qwen2.5-7B	Yes	11	7	6 (54.54)	32.76 sec	677.746 MB
Gemini-2-Flash	No	29	24	21 (72.4%)	33.88 sec	679.98 MB
Gemini-2-Flash	Yes	55	44	38 (69.1%)	34.21 sec	683.36 MB

Proof2Silicon Hardware Synthesis Results: Compilation and synthesis statistics for verified Dafny programs across LLMs and verifier feedback settings.

- Many times, the generated Dafny codes are excluded because PyLog does not support certain Dafny-derived constructs, most notably recursion, dynamic memory usage, or while-loops
- Not all have completed RTL synthesis in Vivado HLS, which imposes further restrictions such as prohibiting recursive function calls and rejecting kernels with subtle type mismatches

Stage-1 Proof2Silicon: Limitations and Bottlenecks

- **Verified ≠ Synthesizable**
  - Dafny-verified code may still contain recursion, unbounded loops, or dynamic memory
    → Causes PyLog/Vivado synthesis failures
- **Reward Signal Is Under-Specified**
  - The current flow only considers the error count and verification success
  - Ignores varying error difficulty and resolution cost (different error categories require different weights)
- **Structural Program Effects Are Unmodeled**
  - No tracking of invariant growth, lemma insertion, ghost variables, or recursion introduction
- **No Prompt Efficiency Control**
  - Allows unnecessary prompt expansion
  - Increases inference cost and reduces training stability

Stage-2 Proof2Silicon: Why?

_________________From Feasibility to Production-Grade System Design_________________

- **Stage-1 validated the concept**
  Demonstrated end-to-end generation from natural language → verified code → RTL.
- **But exposed system-level gaps**
  Verification alone was insufficient for synthesis reliability, scalability, and training stability.
- **Need for hardware-aware intelligence**
  Prompt optimization must consider synthesizability constraints, not only logical correctness.
- **Need for structured learning signals**
  Reward design must reflect error severity, structural program changes, and efficiency trade-offs.
- **Goal of Stage-2**
  Transform Proof2Silicon from a functional prototype into a **robust, hardware-ready, and scalable AI-driven synthesis framework**.

Stage-2 Proof2Silicon: RL Reward Formulation

- **Key Design Change:**
  We retain the same MDP formulation as Stage-1
- **Redesign the reward function** to explicitly encode:
  - ✓ verification correctness
  - ✓ error-resolution progress
  - ✓ structural program properties
  - ✓ prompt efficiency
  - ✓ policy stability

○ **Overall Reward *R (at a given iteration t)*:**

$$R_t = R_{verify} + R_{progress} + R_{structure} + R_{efficiency} + R_{stability}$$

Each term captures a **distinct system objective**, enabling multi-objective optimization beyond verification success alone.

Stage-2 Proof2Silicon: RL Reward Formulation

1. Verification Reward:

$$R_{verify} = \begin{cases} +10 & if\ verified \\ -10 & if\ Empty\ File \\ -5 & if\ unverified \end{cases}$$

Description:

- Full reward is assigned only when Dafny verification succeeds.
- Empty files are heavily penalized to prevent reward hacking.
- Unverified programs receive a base penalty and are further evaluated by progress and structure terms.

2. Error- Resolution Progress Reward:

$$R_{Progress} = \sum_i w^i \left(errors_{t-1}^{(i)} - errors_t^{(i)}\right)$$

Description:

- Rewards *reduction* in verifier errors rather than absolute counts.
- Error categories are weighted by their relative difficulty and importance.
- Encourages structured, meaningful verification progress.

Where error categories i include:

Error Type	Weight
Syntax	0.2
Type Error	0.3
Missing Invariant	0.8
Postcondition	1.0
Termination	1.2

Stage-2 Proof2Silicon: RL Reward Formulation

3. Structural Program Reward:

$$R_{structure} = \sum_j w^j$$

Where j is structure categories with w^j being their weight (cap at 1.5)

Structure	Weight
New Lemma	+0.2
Recursion	-1.0
New Invariant	+0.3
New Ghost variable ($error_{t-1}$ < $error_t$)	-0.5
New Ghost variable ($error_{t-1} = error_t$)	-0.2
New Ghost variable ($error_{t-1}$ > $error_t$)	+0.4

Description

- Explicitly models how structural changes affect verification and synthesis.
- Encourages useful abstraction (lemmas, invariants).
- Penalizes recursion and ghost-variable overuse when harmful.

Stage-2 Proof2Silicon: RL Reward Formulation

4. Prompt Efficiency Reward:

$$R_{efficieny} = -0.001 \times \Delta(prompt\ tokens)$$

Description:

- Penalizes unnecessary prompt growth.
- Encourages concise, targeted prompt modifications.
- Reduces inference cost and improves RL convergence stability.

5. Policy Stability Reward

$$R_{stability} = -0.1 \times KL\ (\pi_t || \pi_{t-1})$$

Description:

- Penalizes large policy shifts between iterations.
- Prevents prompt oscillation and unstable exploration.
- Encourages smooth, incremental prompt refinement.

Stage-2 Proof2Silicon: What Does It Enable?

Practical Effects of Stage-2 Reward Shaping

- **Faster Convergence**
  - Dense progress-based feedback reduces sparse reward dependence
  - Accelerates verifier success compared to Stage-1
- **Hardware-Compatible Code Generation**
  - Structural penalties discourage recursion and unsupported constructs
  - Improves downstream PyLog and Vivado synthesis success
- **Stable Prompt Optimization**
  - KL regularization prevents oscillatory prompt behavior
  - Enables smoother policy updates
- **Cost-Efficient Prompt Refinement**
  - Token efficiency term limits prompt growth
  - Reduces inference cost and unnecessary verbosity

Stage-2 reward transforms prompt repair into a structured, hardware-aware optimization problem.

Stage-2 Proof2Silicon: What Does It Enable?

Stage-1 vs Stage-2 Reward: Key Differences

Aspect	Stage-1	Stage-2
Reward Signal	Sparse, verification-based	Dense, multi-objective
Structural Awareness	None	Explicit (recursion, invariants, lemmas)
Prompt Control	None	Token efficiency regularization
Training Stability	Unstable	KL-regularized
Hardware Awareness	Indirect	Explicitly encoded

Stage-2 reward transforms prompt repair into a structured, hardware-aware optimization problem.

Stage-2 Proof2Silicon: Results

MODEL	Without SLM (Baseline)			Without Trained SLM			With Trained SLM (Until Now)		
	W/O Feedback	With Feedback	Total	W/O Feedback	With Feedback	Total	W/O Feedback	With Feedback	Total
ChatGPT 5.1 Codex	41	33	74	30	41	71	34	53	**87**
ChatGPT 5.3	52	16	68	34	42	68	63	25	**88**
DeepSeek-chat	46	9	55	24	33	57	65	7	**72**

Stage-2 Verification success rates on a 100-task benchmark for various LLMs using prompts through PREFACE pipeline from both untrained and trained small language models, comparing single-shot (W/O Feedback) versus iterative error-feedback (With Feedback) modes. Training time (~144hrs)

- **W/O Feedback** indicates that only the initial prompt is used and is sufficient to produce valid code.
- **With Feedback** indicates that the initial prompt is insufficient, and error feedback is iteratively provided to the SLM to generate revised prompts, for up to seven iterations.

AI Agents-Driven Synthesis!

One Example: LLM-enhanced Logic Synthesis Model

Extract quality-of-result prediction information directly from RTL code.

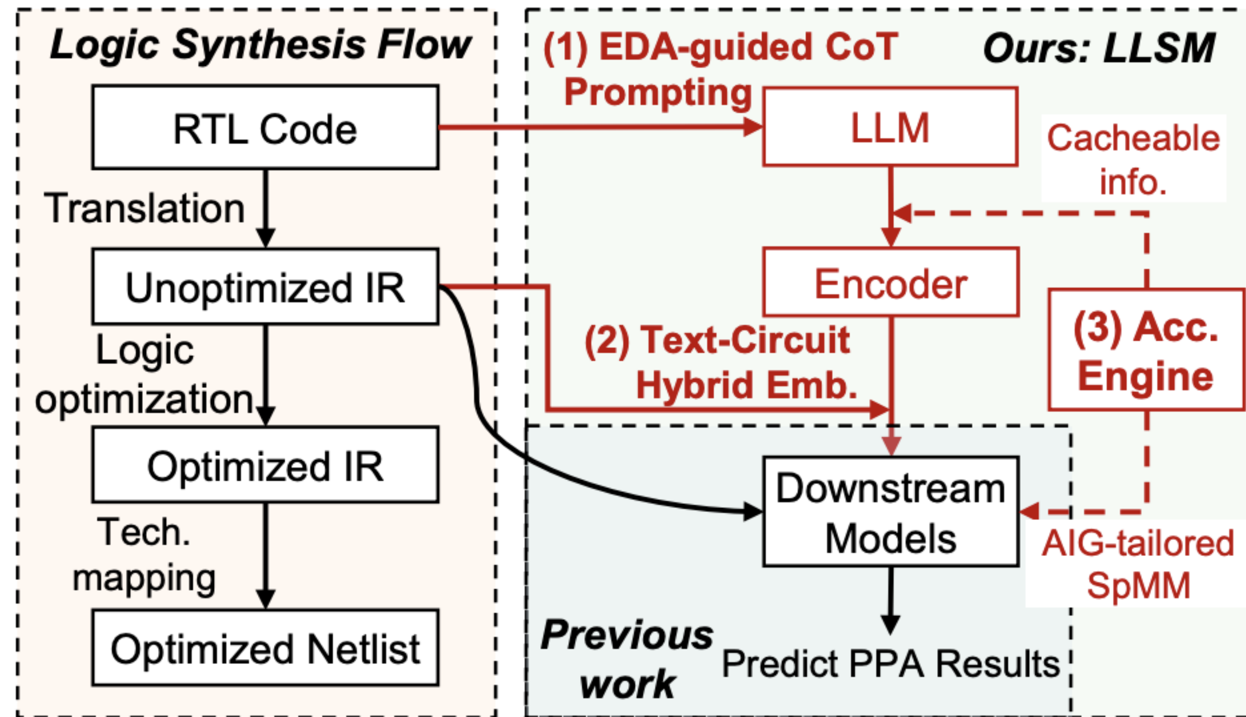


Figure 1: LLSM overall workflow for RTL-stage design PPA prediction compared to the traditional logic synthesis flow and previous works.

(Figure extracted from the paper)

Shan Huang et. al., "LLSM: LLM-enhanced Logic Synthesis Model with EDA-guided CoT Prompting, Hybrid Embedding and AIG-tailored Acceleration", ASPDAC'25

Another Example: Multi-Agent Self-Evolved ABC

(Figure extracted from the paper)

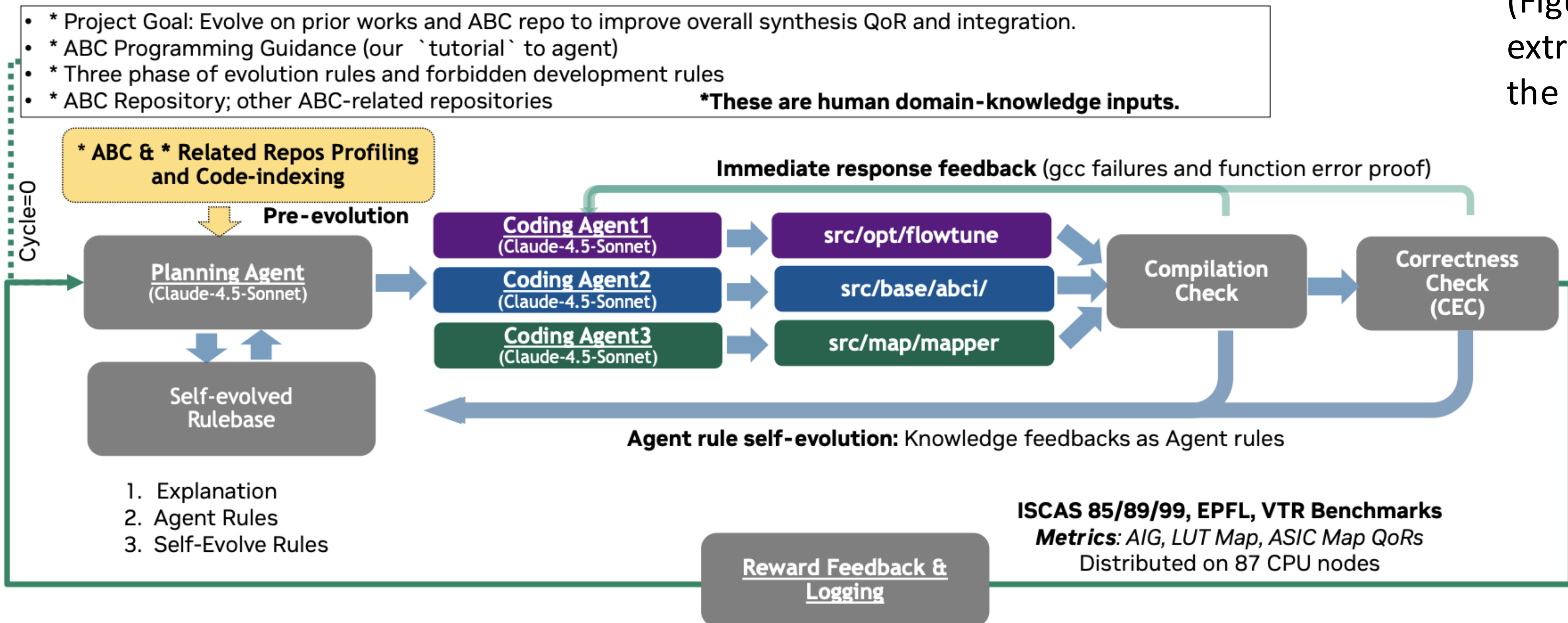


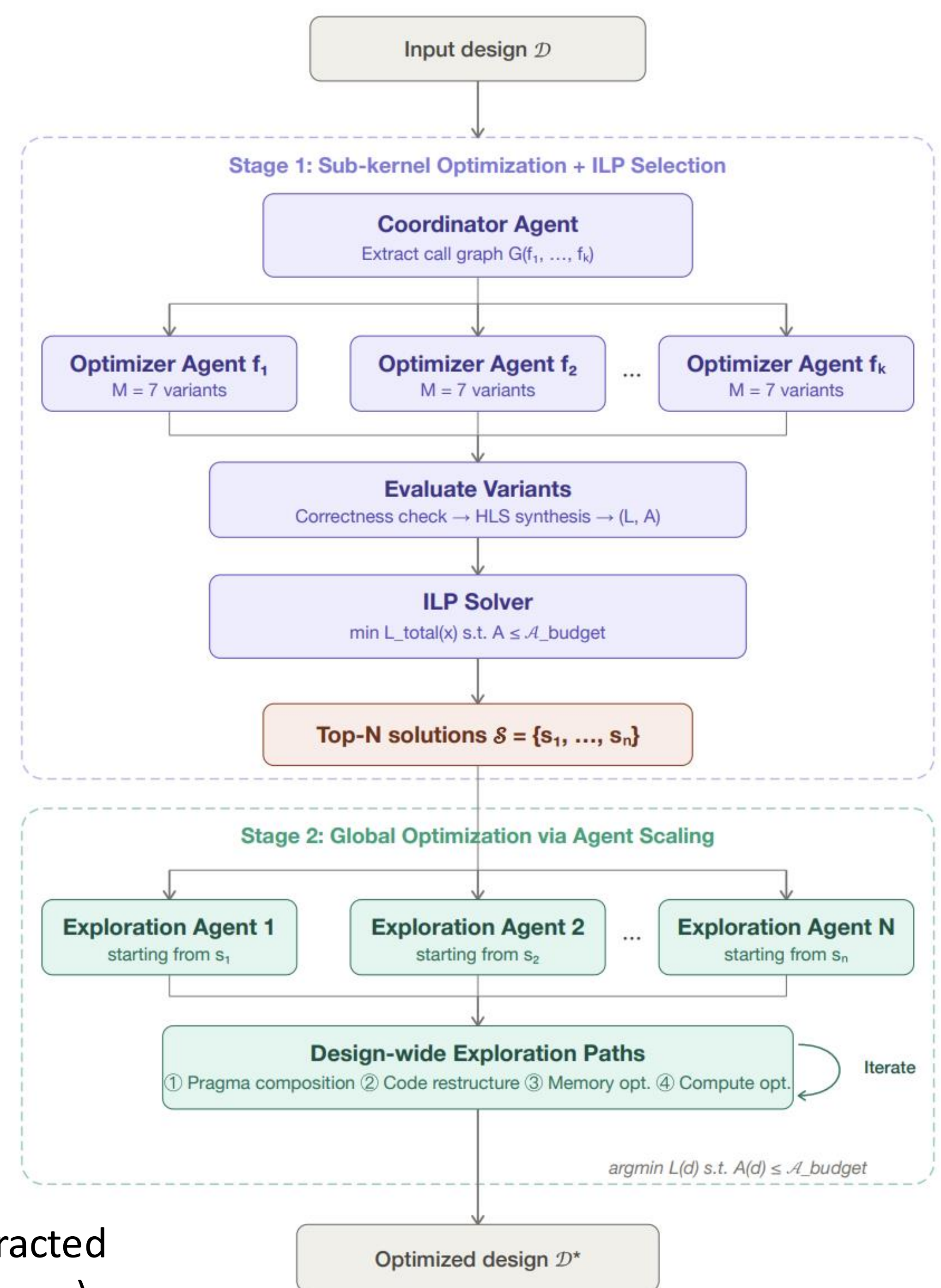
Figure 1: Overview of the multi-agent self-evolving framework for ABC. Specialized LLM agents evolve distinct subsystems (flow optimization, core algorithms, and mapping), with each iteration undergoing compilation, formal CEC verification, and full QoR evaluation. A planning agent coordinates global decisions, a coding agent implements edits, and all agents follow a shared rulebase and unified evaluation pipeline to enable coordinated, correctness-preserving improvements.

Cunxi Yu and Haoxing Ren, "Autonomous Evolution of EDA Tools: Multi-Agent Self-Evolved ABC", DAC'26

Third Time's the Charm: Agent Factory

- Two-stage agent-based pipeline for HLS design space exploration.
- Given an input design D, a coordinator agent extracts the function call graph G and spawns one optimizer agent per sub-function f1, . . . , fK. Variants are evaluated for correctness and synthesized to obtain (latency, area) pairs.
- An ILP solver then selects the top-N combinations S = {s1, . . . , sN } that minimize total latency subject to the area budget.
- In Stage 2, N exploration agents each start from a candidate solution and iteratively apply design-wide optimization paths, to produce the final optimized design D∗.

Abhishek Bhandwaldar, Mihir Choudhury, Ruchir Puri, Akash Srivastava, "Agent Factories for High Level Synthesis: How Far Can General-Purpose Coding Agents Go in Hardware Optimization?". https://arxiv.org/pdf/2603.25719.



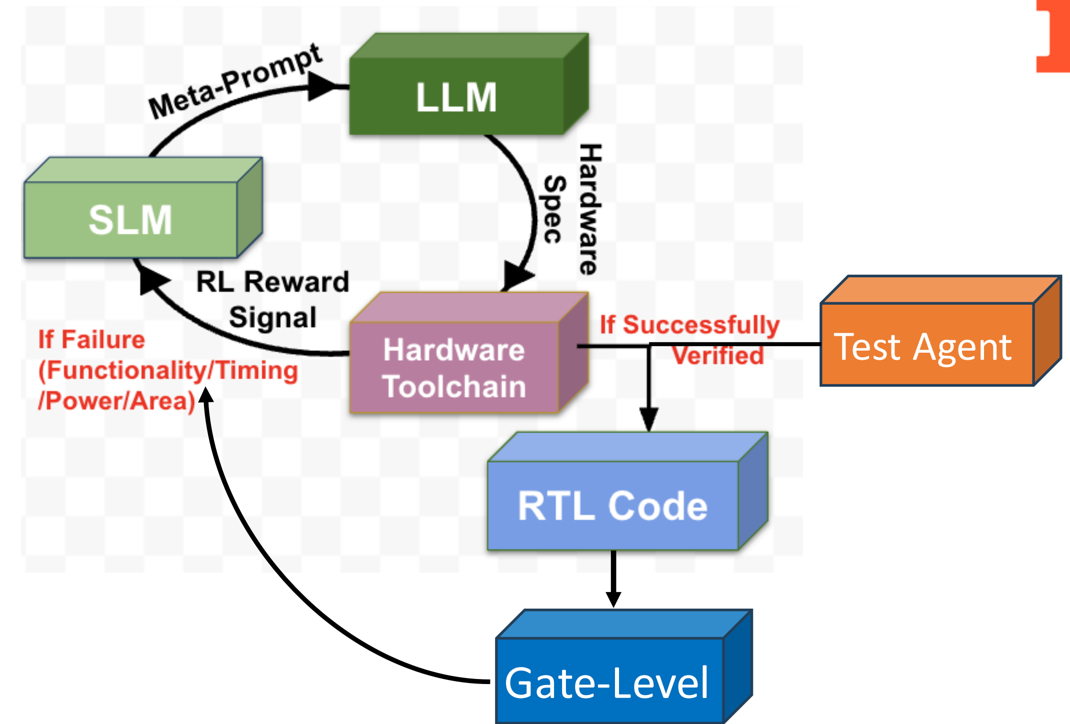
(Figure extracted from the paper)

A Couple of Future Directions

Direct RTL Generation with Correctness & Quality Guarantees

- Reinforcement learning (RL) on SLM that iteratively repairs the prompts provided to frozen LLMs
- Multi-level verification feedback loop
- Structure-aware refinement
- PPA-aware optimization during generation
- Large-scale synthetic data generation and self-improving training
- Leverage OpenRTLSet dataset
  - Large, open-source Verilog dataset (131K → 470K designs) for training and benchmarking
- **LLM fine-tuning (orthogonal enhancement)**

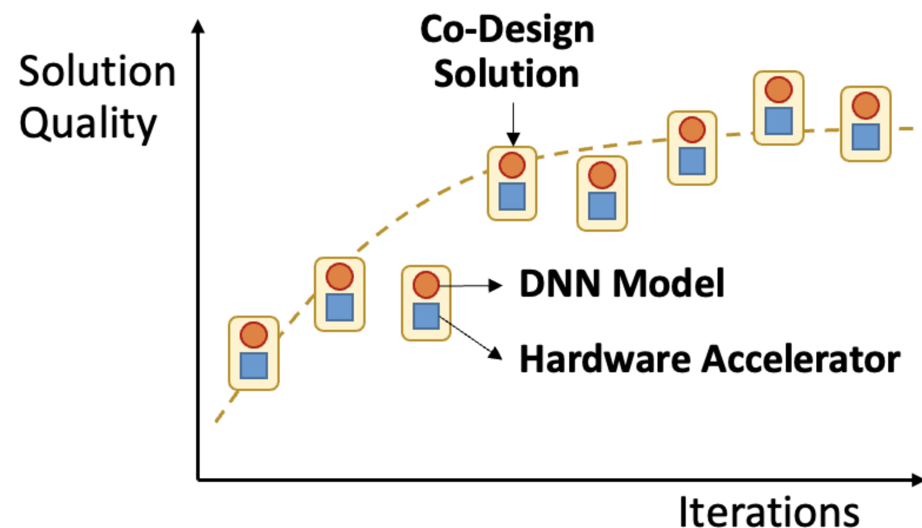
Jinghua Wang, et.al. "OpenRTLSet: A Fully Open-Source Dataset for Large Language Model-based Verilog Module Design," *IEEE International Conference on LLM-Aided Design*, June 2025.



- Autonomous, trustworthy RTL generation framework
- Higher reliability and robustness through multi-level verification and coverage-driven testing
- Scalable generation of complex systems, extending from modules to full SoC architectures
- Creation of large-scale open datasets for hardware-aware LLM training
- **Logic-level PPA-optimized designs**

Food for Thought for New AI+HW Synthesis

- The LLM jointly generates the ML model and its corresponding hardware, enabling coordinated co-optimization.
  - *HW could be generated by the framework introduced in the previous slide*
    - *And the spec is the ML model itself!*
  - **It requires a substantial amount of compute.**



The Journey Has Just Begun!

Acknowledgement

All the Collaborators:

Co-authors listed in the various publications.

Sponsors:

The Grainger College of Engineering
IBM-Illinois Discovery Accelerator Institute

Booz | Allen | Hamilton®

Thank you