Scalable High-Level Synthesis for AI Accelerator Design and Verification

Hanchen Ye

Oct. 18, 2023

Thesis Committee: Prof. Vikram Adve, Prof. Deming Chen (Chair), Prof. Jian Huang, Prof. Kai Li, Dr. Stephen Neuendorffer *(Alphabetical Order)*

AI is everywhere, changing everything

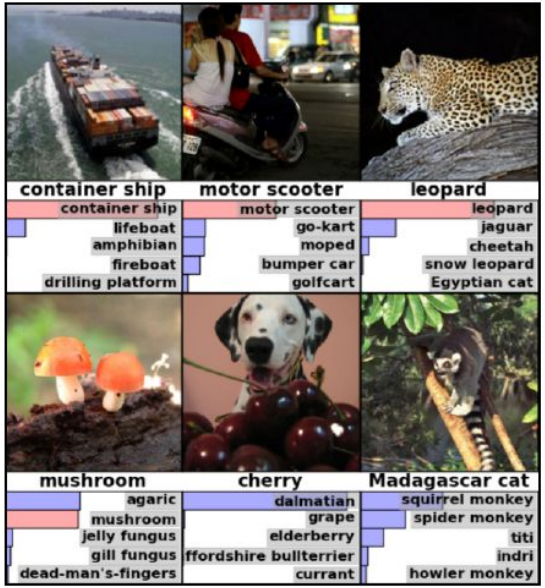
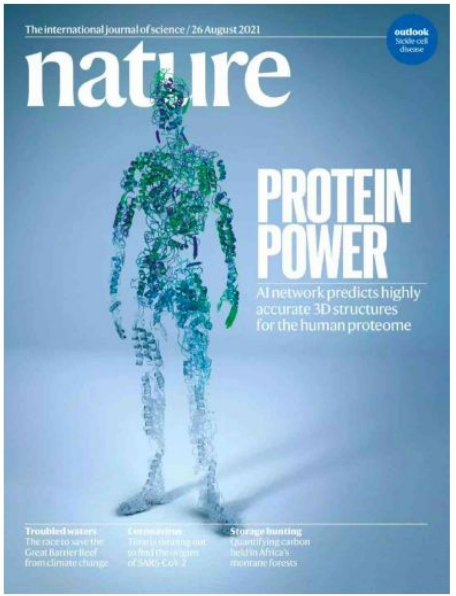


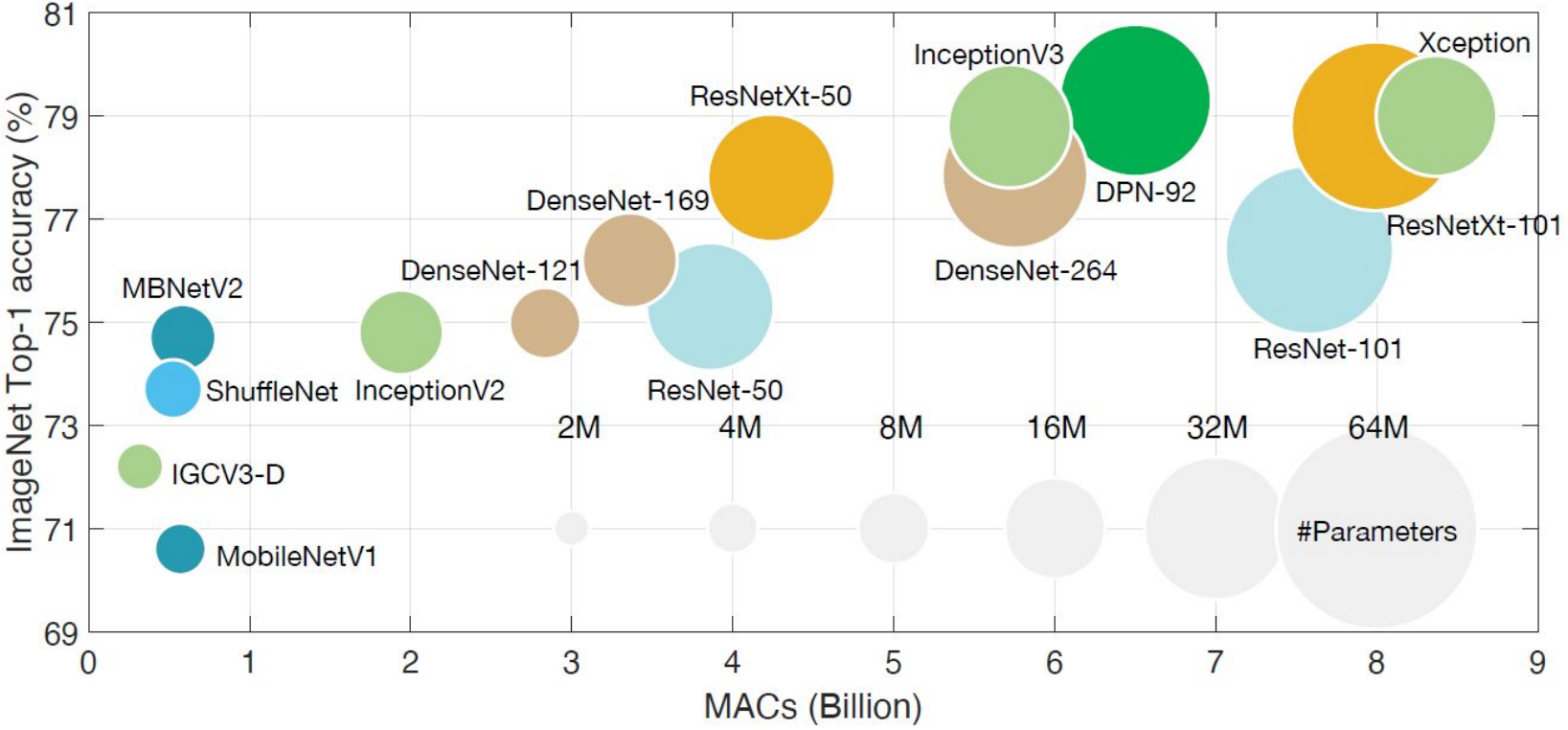
Image Classification

AlphaGo (Nature 2016)



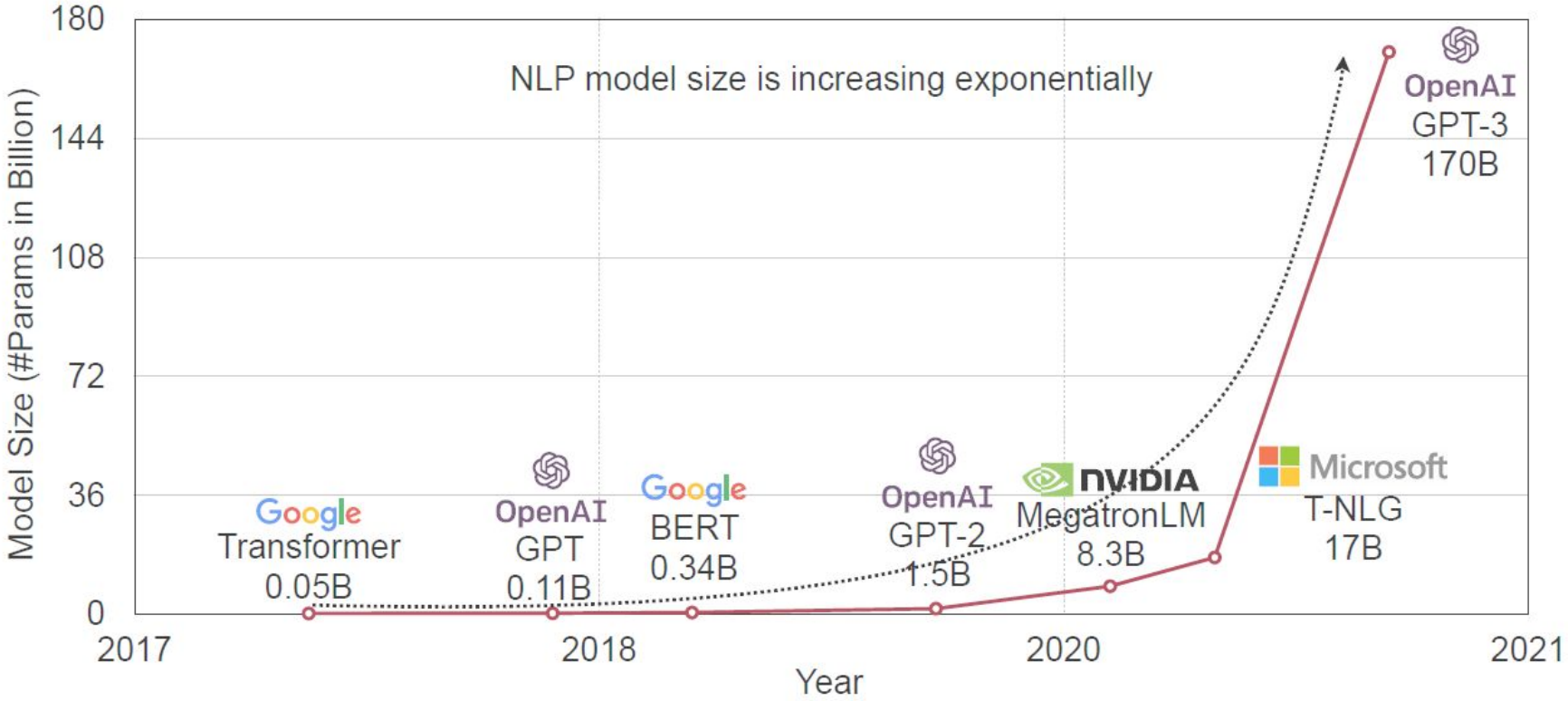
AlphaFold (Nature 2021)

Computational cost of DNNs is growing



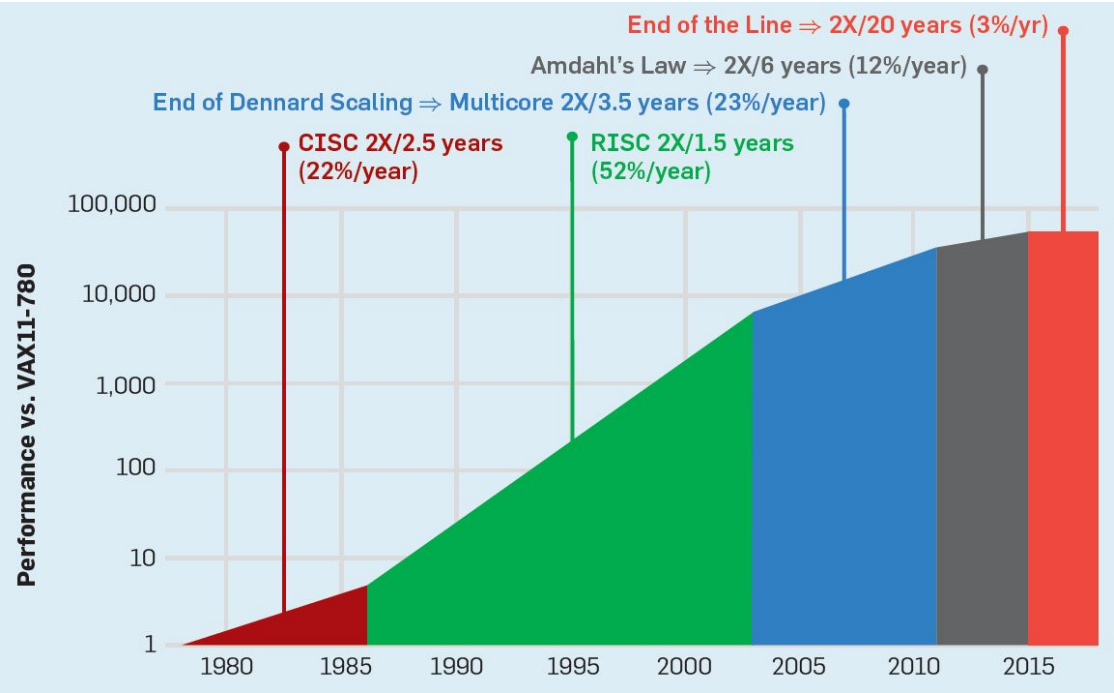
- Model Compression and Hardware Acceleration for Neural Networks: A Comprehensive Survey [Deng et al., IEEE 2020]

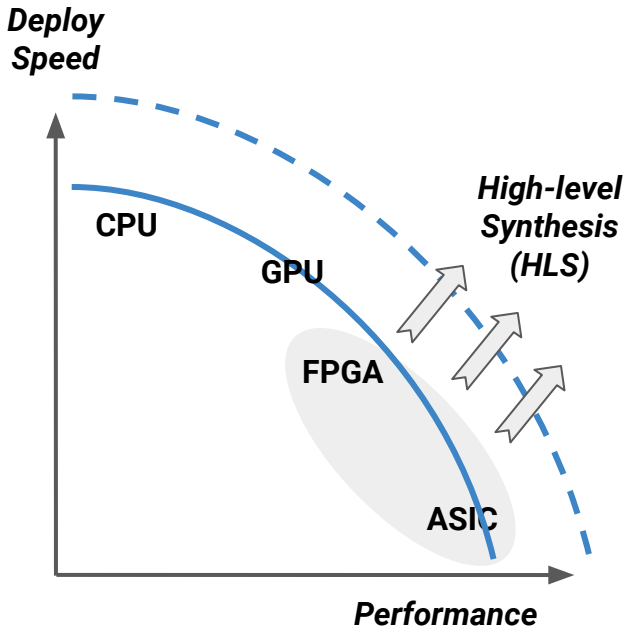
Model size of language models is growing exponentially



- TinyML and Efficient Deep Learning Computing [Han, 2023]

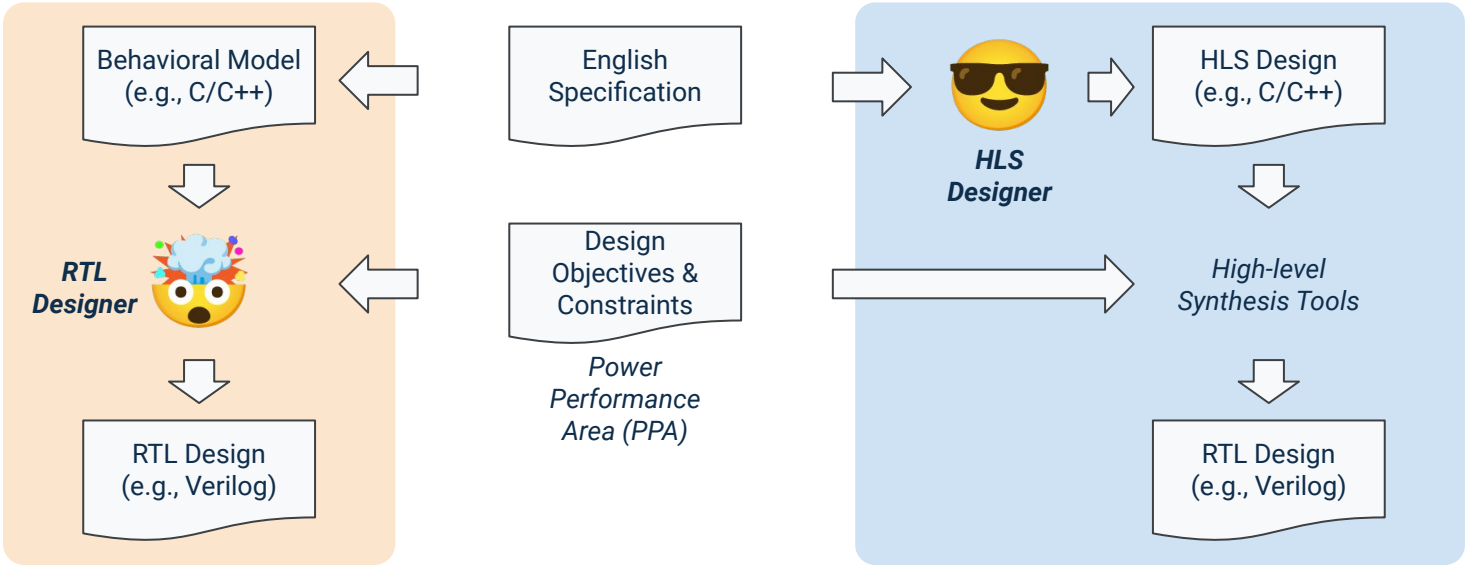
Need for domain-specific accelerators





- A New Golden Age for Computer Architecture [Hennessy et al., IEEE 2020]

High-level Synthesis (HLS)



RTL Design Flow

- **Manual** optimization and scheduling
- **Long** design cycle
- **Low** portability against different PDK or PPA requirements

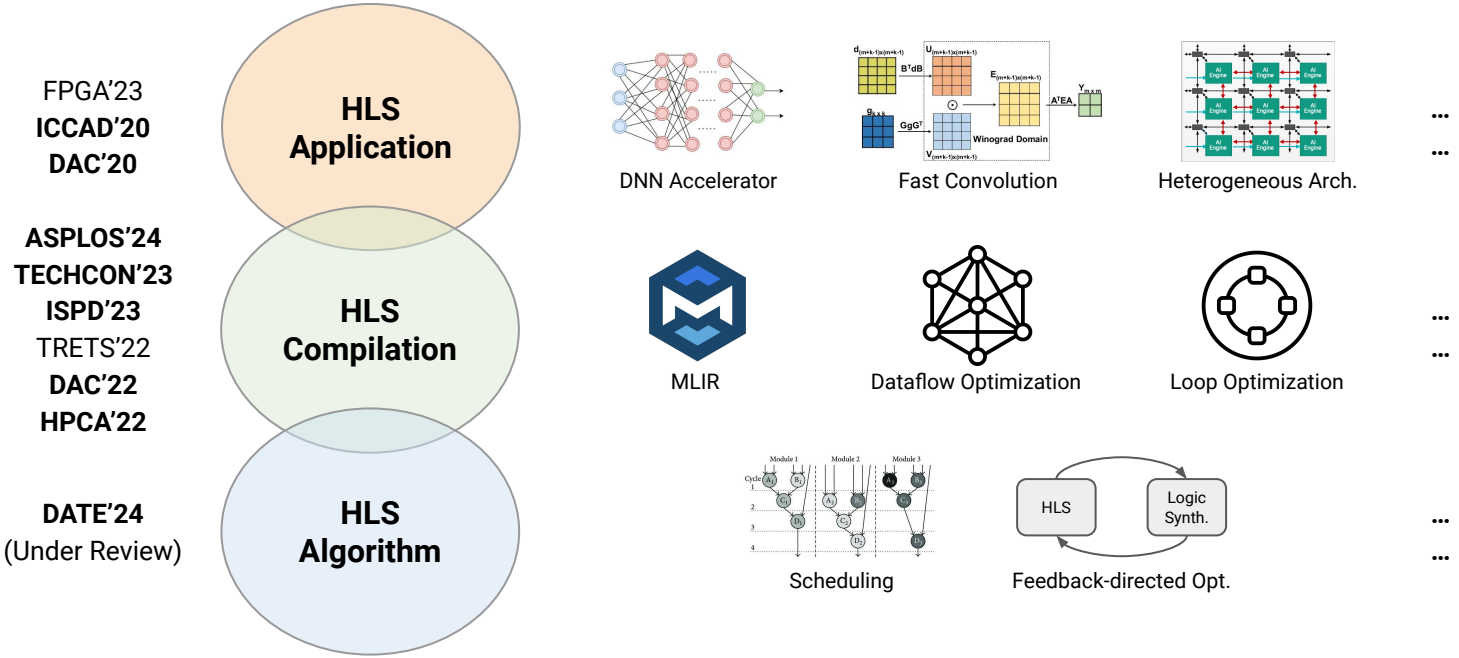
HLS Design Flow

- **Automated** optimization and scheduling
- **Short** design cycle
- **High** portability against different PDK or PPA requirements

- PDK: Process design kit.

Research Overview

Toward **Scalable** and **Efficient** HLS Solution for AI Acceleration



- Bolded papers are first or co-first authored.

Outline

- **HLS Application**
  - *HybridDNN: A Framework for High-Performance Hybrid DNN Accelerator Design and Implementation [DAC'20]*
  - *DNNExplorer: a framework for modeling and exploring a novel paradigm of FPGA-based DNN accelerator (HybridDNN 2.0) [ICCAD'20]*
- **HLS Compilation**
  - *ScaleHLS: A New Scalable High-Level Synthesis Framework on Multi-Level Intermediate Representation [HPCA'22]*
  - *HIDA: A Hierarchical Dataflow Compiler for High-Level Synthesis (ScaleHLS 2.0) [ASPLOS'24]*
- **HLS Algorithm**
  - *ISDC: Feedback-guided Iterative SDC Scheduling for High-level Synthesis [DATE'24, under review]*

HLS Application

HybridDNN: A Framework for High-Performance Hybrid DNN Accelerator Design and Implementation

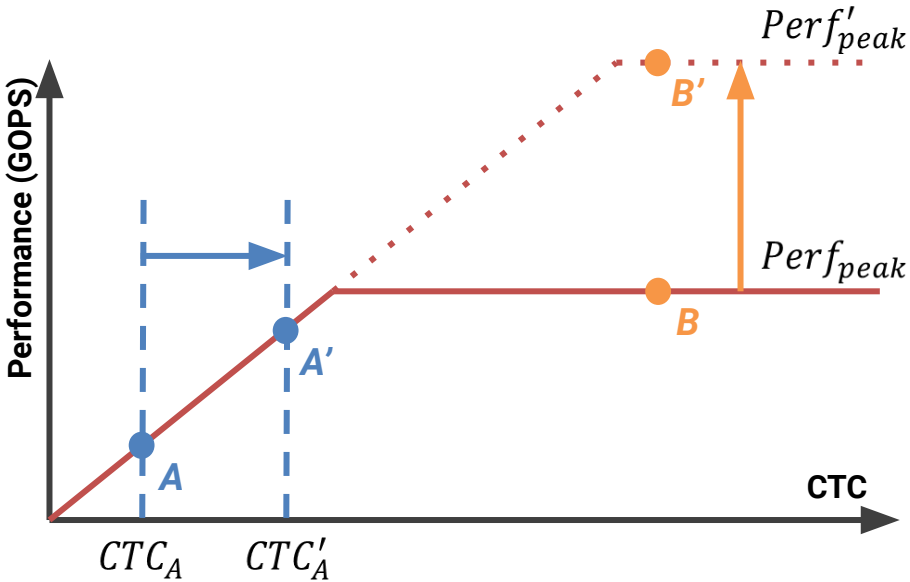
DNNExplorer: a framework for modeling and exploring a novel paradigm of FPGA-based DNN accelerator (HybridDNN 2.0)

HLS Application Outline

- HybridDNN framework
- DNNExplorer framework

DNN Acceleration on FPGA

Roofline model: $Perf_{achievable} = \min(CTC \times Bandwidth, Perf_{peak})$
CTC denotes computation-to-communication ratio, $Bandwidth$ denotes external memory bandwidth.

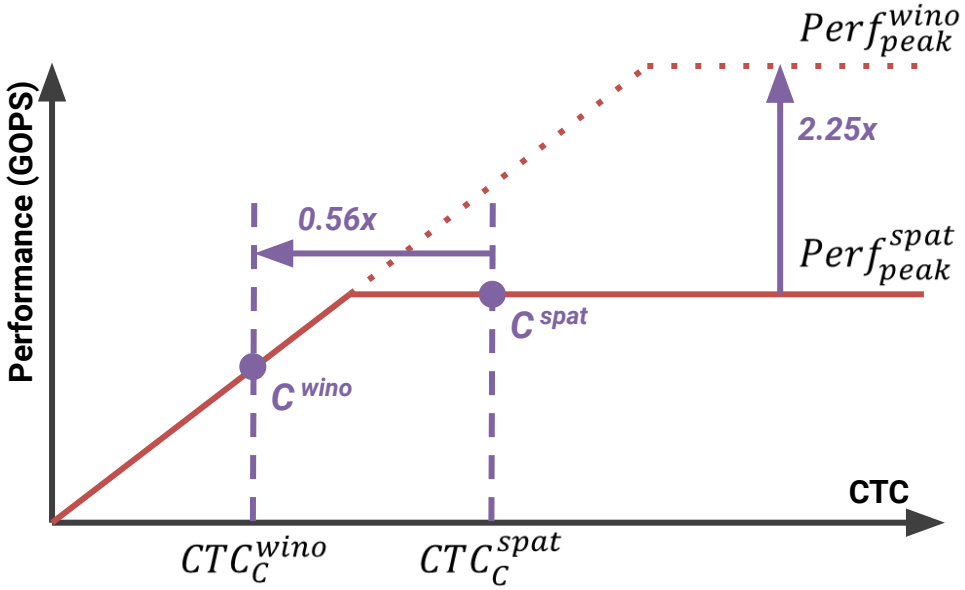


Layer *A* is memory-bounded
Layer *B* is computation-bounded

- **Data reuse optimization**
  - $CTC \uparrow$
- **DNN model compression**
  - Including quantization, pruning, etc.
  - $Perf_{peak} \uparrow, CTC \uparrow$
- **Fast convolution algorithms***
  - Including Winograd, FFT, etc.
  - Reduce the arithmetic complexity of CONV by **2.25x** (for Winograd)
  - Increase weight by 1.78x, reducing overall CTC by **0.56x** (for Winograd)
  - $Perf_{peak} \uparrow, CTC \downarrow$

- Source: L. Andrew et al. Fast algorithms for convolutional neural networks, in CVPR 2016.

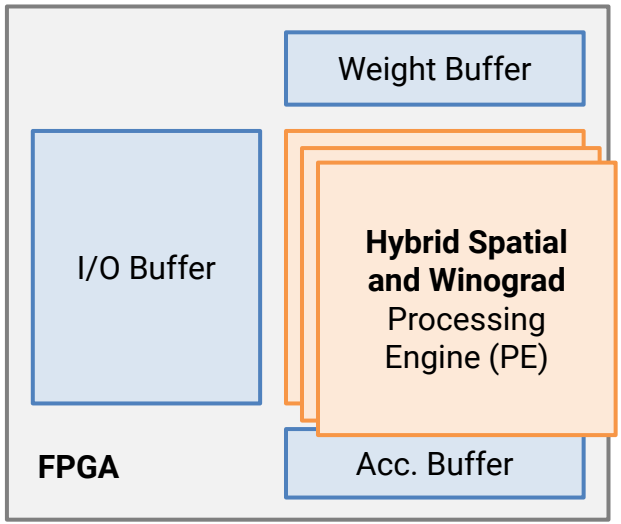
Problems of Winograd-based DNN Accelerators



Winograd is inefficient in some cases

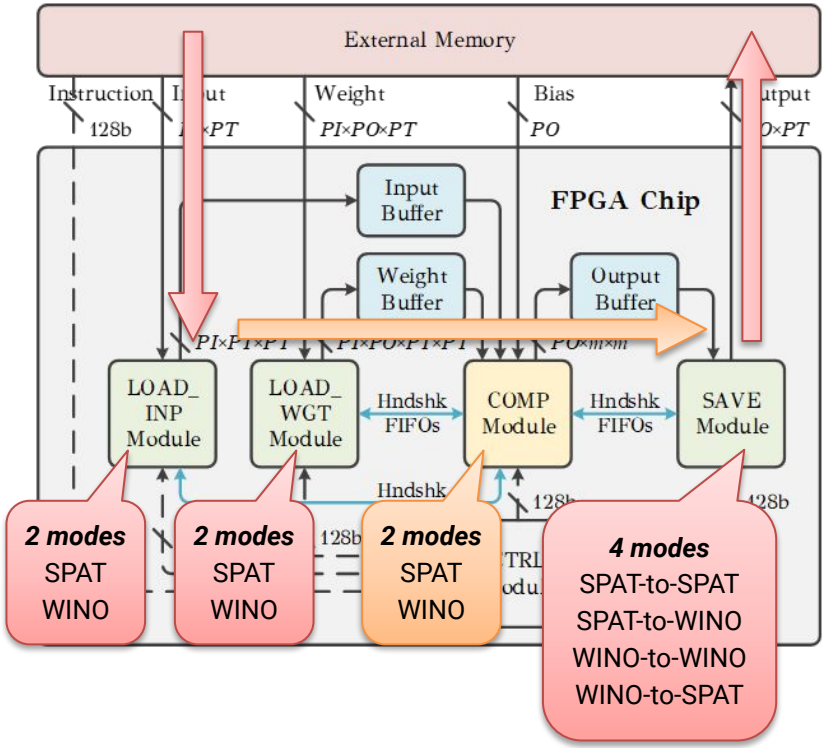
Problem

Low efficiency when Spatial CONV outperforms Winograd CONV



- Source: L. Liqiang et al. Evaluating fast algorithms for convolutional neural networks on FPGAs, in FCCM 2017.

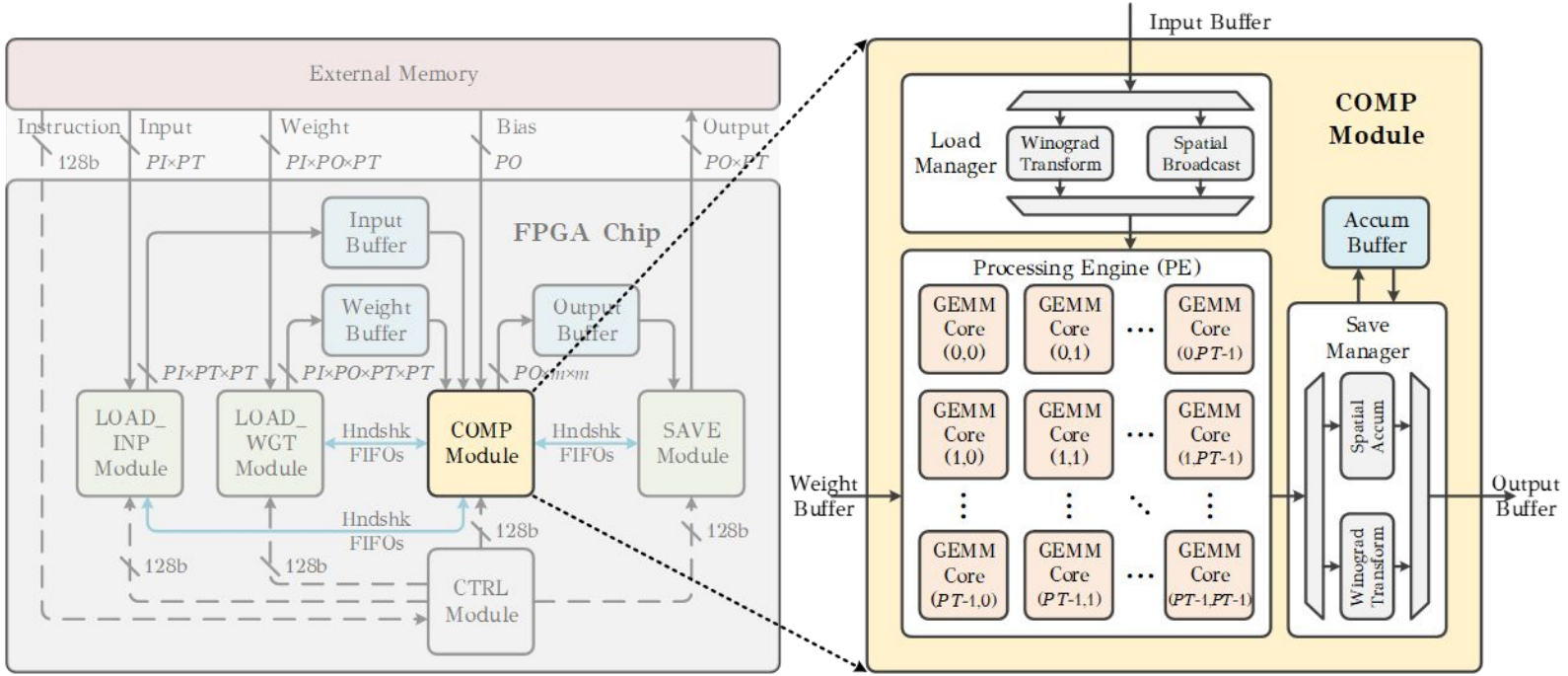
HybridDNN Functional Modules



- ***LOAD_WGT & LOAD_INP Module***
  - Load weights and input feature maps from external memory
- ***COMP Module***
  - Carry out the computation in either Spatial or Winograd CONV mode
- ***SAVE Module***
  - Write back output feature maps
  - Reorder output feature maps to ensure the data pattern in **external memory** matches the CONV mode of the successive layer

Module-level Pipeline

Compute Module Architecture



HybridDNN VGG16 Case Study

	[26]	[4]	[6]	Ours	
Device	Xilinx VU9P	Arria10 GX1150	Xilinx VU9P	Xilinx VU9P	PYNQ Z1
Model	VGG16	VGG16	VGG16	VGG16	VGG16
Precision	16-bit	16-bit	16-bit	12-bit*	12-bit*
Freq.(MHz)	210	385	214	167	100
DSPs	4096	2756	5349	5163	220
CNN Perf.(GOPS)	1510	1790	1828.6	**3375.7**	83.3
Power(W)	NA	37.5	49.3	45.9	2.6
DSP Effi. (GOPS/DSP)	0.37	0.65	0.34	**0.65**	0.38
Energy Effi. (GOPS/W)	NA	47.78	37.1	**73.5**	32.0

*DNN parameters are quantized to 8-bit; input feature maps are set to 12-bit in PE due to the Winograd matrix transformation

- Xilinx VU9P Configuration:
  - PI=4, PO=4, PT=6, NI=6
- PYNQ-Z1 Configuration:
  - PI=4, PO=4, PT=4, NI=1

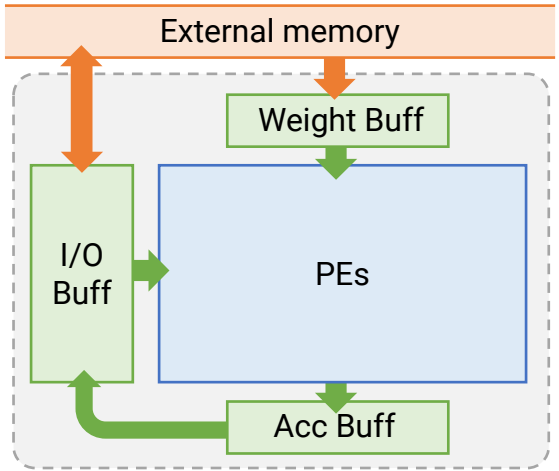
	LUTs	DSPs	18Kb BRAMs
VU9P	706353 (59.8%)	5163 (75.5%)	3169 (73.4%)
PYNQ-Z1	37034 (69.61%)	220 (100%)	277 (98.93%)

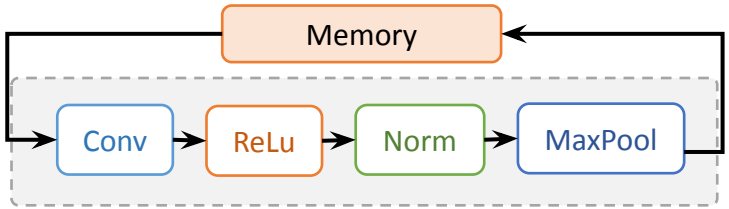
HLS Application Outline

- HybridDNN framework
- DNNExplorer framework

HybridDNN - DNN Accelerator Paradigm #1

Generic reusable architecture



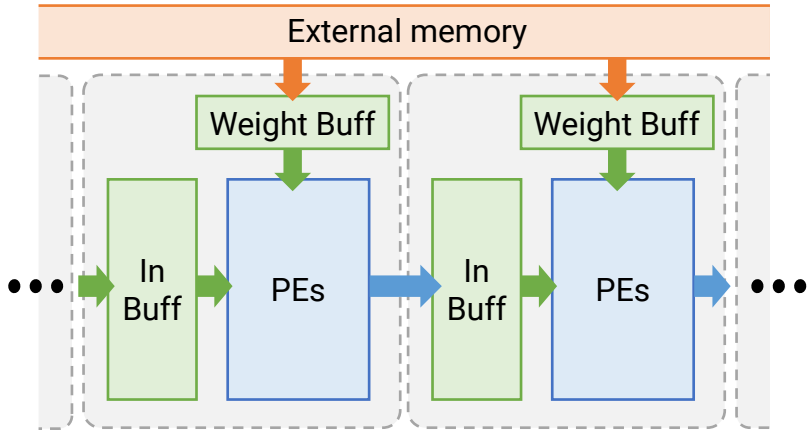


- Same PEs reused by all layers; Process one after another layers
- Major buffers for weights, inputs, and partial outputs

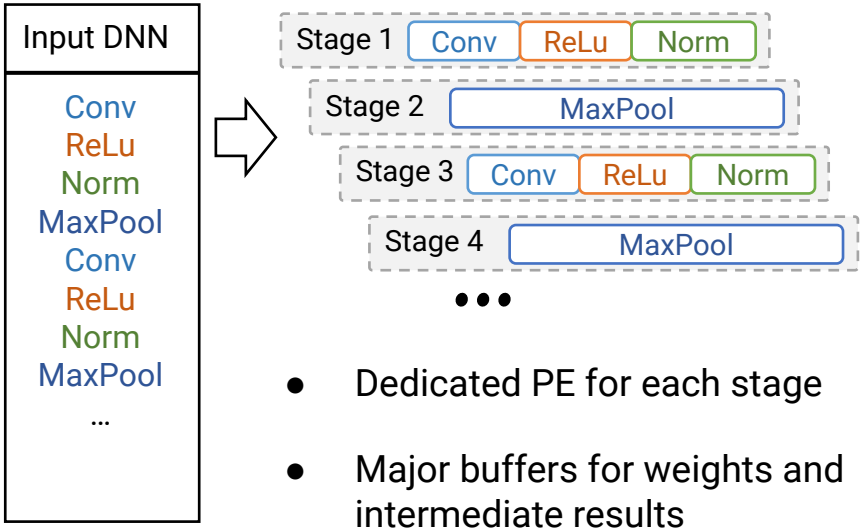
- Easy to scale up/down with given resources
- Easy to adapt regular DNNs

DNN Accelerator Paradigm #2

Layer-wise pipeline architecture



- Better configuration for each layer
- Better throughput performance
- Better external memory BW utilization

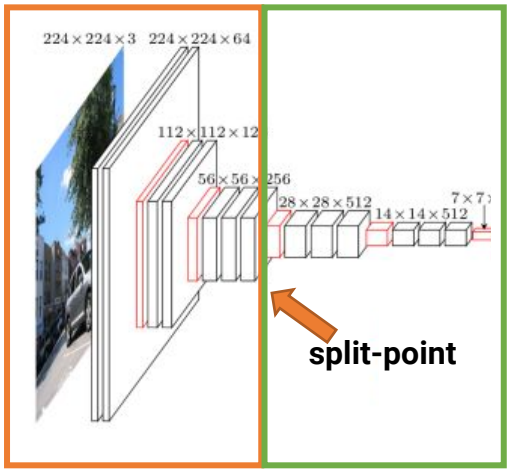


- Dedicated PE for each stage
- Major buffers for weights and intermediate results

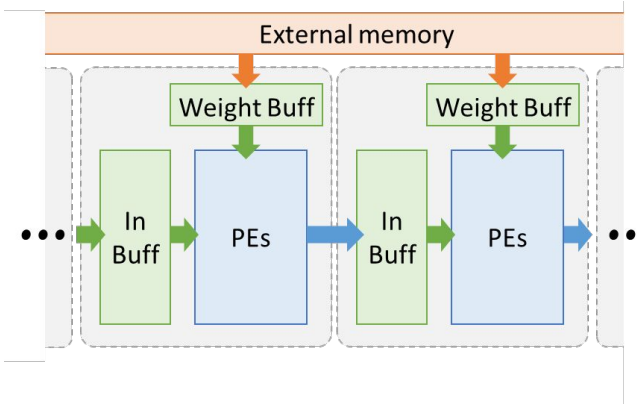
DNNExplorer (HybridDNN 2.0) Architecture

Larger FMs (H, W)
More diversity

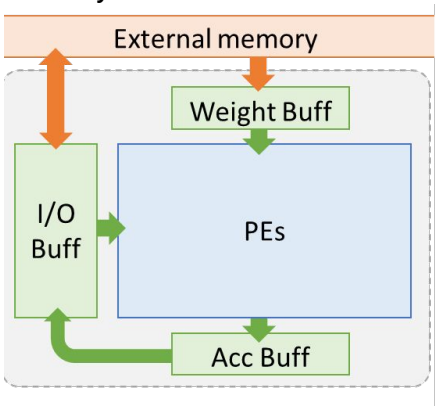
Smaller FMs (H, W)
Less diversity



Paradigm #2
DNNBuilder



Paradigm #1
HybridDNN



Dedicated design for more fine-grained adjustments

Better adaptability to larger inputs

Better scalability to deeper networks

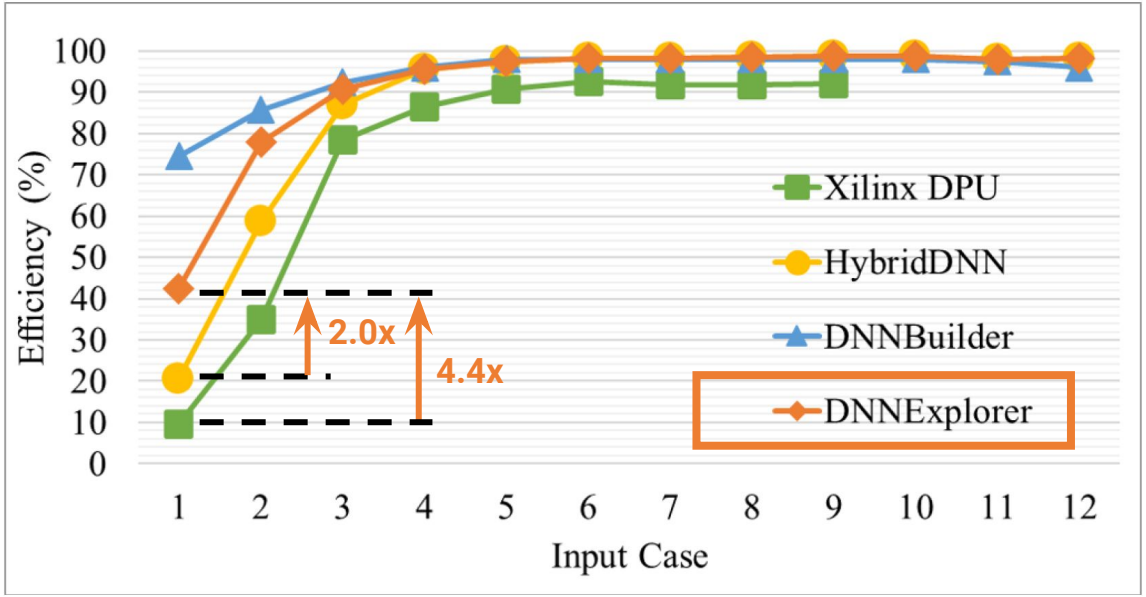
- DNNBuilder: An automated tool for building high-performance DNN hardware accelerators for FPGAs. Zhang et al. ICCAD'18.

DNNExplorer (HybridDNN 2.0) DSP Efficiency Study

Target 12 VGG-16 models (batch=1, w/ different input sizes, w/o FC layers)

Compare to Xilinx DPU (ZCU102), HybridDNN, DNNBuilder (KU115)

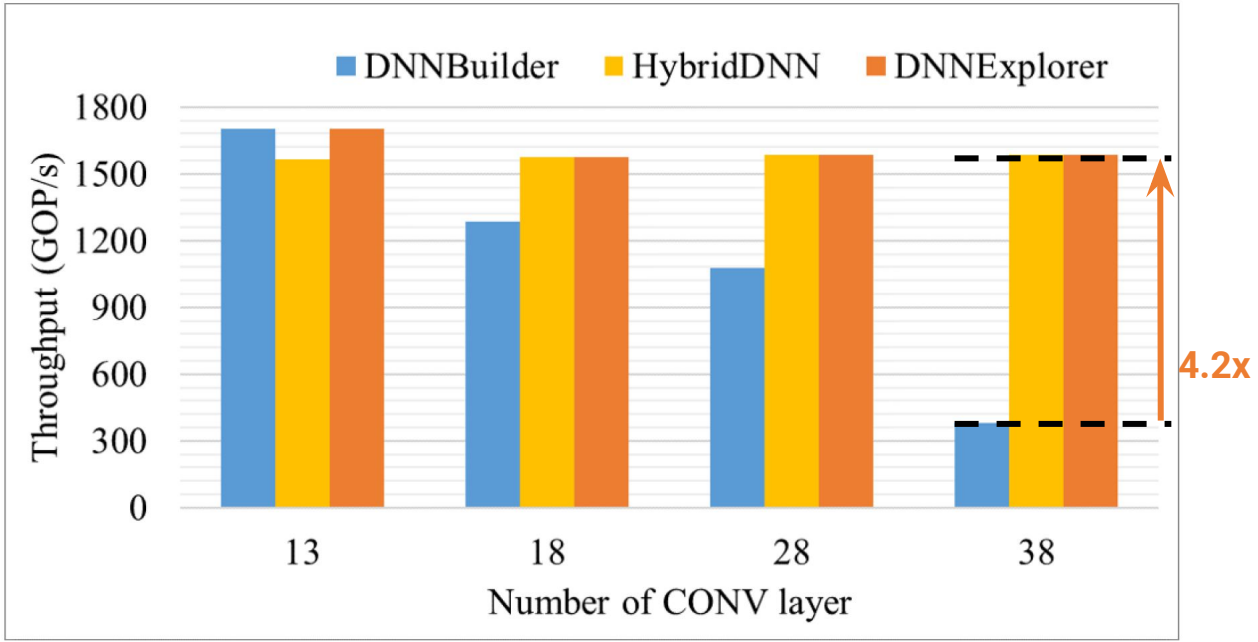
Case	Input Size
1	3x32x32
2	3x64x64
3	3x128x128
4	3x224x224
5	3x320x320
6	3x384x384
7	3x320x480
8	3x448x448
9	3x512x512
10	3x480x800
11	3x512x1382
12	3x720x1280



- DNNBuilder: An automated tool for building high-performance DNN hardware accelerators for FPGAs. Zhang et al. ICCAD'18.

DNNExplorer (HybridDNN 2.0) Throughput Study

Target deeper DNN models with 13, 18, 28, 38 CONV layers



- DNNBuilder: An automated tool for building high-performance DNN hardware accelerators for FPGAs. Zhang et al. ICCAD'18.

Challenges of Manual HLS Accelerator Design

- **Time-consuming:** Manual architecture and microarchitecture design, manual C/C++ code rewriting
- **Suboptimal:** Empirical parameter tuning, like parallel factors, buffer sizes, tiling sizes, etc.
- **Low flexibility:** Only support a small set of models

Automated HLS Optimization

HLS Compilation

ScaleHLS: A New Scalable High-Level Synthesis Framework on Multi-Level Intermediate Representation

HIDA: A Hierarchical Dataflow Compiler for High-Level Synthesis (ScaleHLS 2.0)

HLS Compilation Outline

- Motivation
- ScaleHLS framework
- ScaleHLS optimizations
- ScaleHLS design space exploration
- HIDA design space exploration

Motivations - Directive Optimizations

How do we do HLS designs?

```
for (int i = 0; i < 32; i++) {
  for (int j = 0; j < 32; j++) {
    C[i][j] *= beta;
    for (int k = 0; k < 32; k++) {
      C[i][j] += alpha * A[i][k] * B[k][j];
} } }
```

Directive Optimizations

Loop pipeline, unroll
Function pipeline, inline
Array partition, etc.

```
for (int i = 0; i < 32; i++) {
  for (int j = 0; j < 32; j++) {
    C[i][j] *= beta;
    for (int k = 0; k < 32; k++) {
#pragma HLS pipeline
      C[i][j] += alpha * A[i][k] * B[k][j];
} } }
```

Generate RTL with XILINX VITIS and etc.

Pipeline II is **5** and overall latency is **183,296**

Motivations (Cont.) - Loop Optimizations

How do we do HLS designs?

```
for (int i = 0; i < 32; i++) {
  for (int j = 0; j < 32; j++) {
    C[i][j] *= beta;
    for (int k = 0; k < 32; k++) {
      C[i][j] += alpha * A[i][k] * B[k][j];
} } }
```

Loop Optimizations

- Loop interchange
- Loop perfectization
- Loop tile, skew, etc.

```
for (int k = 0; k < 32; k++) {
  for (int i = 0; i < 32; i++) {
    for (int j = 0; j < 32; j++) {
      if (k == 0)
        C[i][j] *= beta;
      C[i][j] += alpha * A[i][k] * B[k][j];
} } }
```

Directive Optimizations

- Loop pipeline, unroll
- Function pipeline, inline
- Array partition, etc.

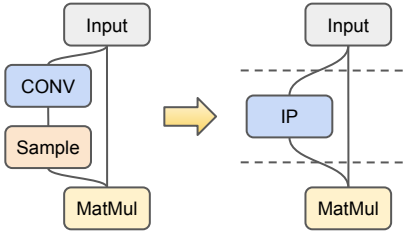
```
for (int k = 0; k < 32; k++) {
  for (int i = 0; i < 32; i++) {
    for (int j = 0; j < 32; j++) {
#pragma HLS pipeline
      if (k == 0)
        C[i][j] *= beta;
      C[i][j] += alpha * A[i][k] * B[k][j];
} } }
```

Generate RTL with XILINX VITIS and etc.

Pipeline II is **2** and overall latency is **65,552**

Motivations (Cont.) - Graph Optimizations

How do we do HLS designs?



Graph Optimizations

Node fusion
IP integration
Task-level pipeline, etc.

```
for (int i = 0; i < 32; i++) {
  for (int j = 0; j < 32; j++) {
    C[i][j] *= beta;
    for (int k = 0; k < 32; k++) {
      C[i][j] += alpha * A[i][k] * B[k][j];
} } }
```

Loop Optimizations

Loop interchange
Loop perfectization
Loop tile, skew, etc.

```
for (int k = 0; k < 32; k++) {
  for (int i = 0; i < 32; i++) {
    for (int j = 0; j < 32; j++) {
      if (k == 0)
        C[i][j] *= beta;
      C[i][j] += alpha * A[i][k] * B[k][j];
} } }
```

Directive Optimizations

Loop pipeline, unroll
Function pipeline, inline
Array partition, etc.

```
for (int k = 0; k < 32; k++) {
  for (int i = 0; i < 32; i++) {
    for (int j = 0; j < 32; j++) {
#pragma HLS pipeline
      if (k == 0)
        C[i][j] *= beta;
      C[i][j] += alpha * A[i][k] * B[k][j];
} } }
```

Generate RTL with XILINX VITIS and etc.

Pipeline II is **2** and overall latency is **65,552**

Motivations (Cont.) - Overall

Difficulties:

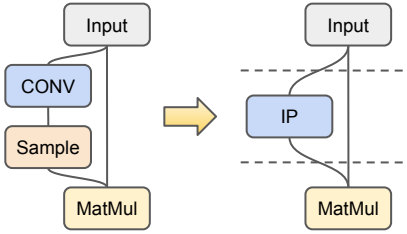
- Low-productive and error-proning
- Hard to enable automated design space exploration (DSE)
- NOT scalable!

Solve problems at the 'correct' level AND automate it

MLIR

Approaches of *ScaleHLS*:

- Represent HLS designs at multiple levels of abstractions
- Make the *multi-level* optimizations automated and parameterized
- Enable an automated DSE
- End-to-end high-level analysis and optimization flow




```
for (int i = 0; i < 32; i++) {
  for (int j = 0; j < 32; j++) {
    C[i][j] *= beta;
    for (int k = 0; k < 32; k++) {
      C[i][j] += alpha * A[i][k] * B[k][j];
} } }
```

```
for (int k = 0; k < 32; k++) {
  for (int i = 0; i < 32; i++) {
    for (int j = 0; j < 32; j++) {
      if (k == 0)
        C[i][j] *= beta;
      C[i][j] += alpha * A[i][k] * B[k][j];
} } }
```

```
for (int k = 0; k < 32; k++) {
  for (int i = 0; i < 32; i++) {
    for (int j = 0; j < 32; j++) {
#pragma HLS pipeline
      if (k == 0)
        C[i][j] *= beta;
      C[i][j] += alpha * A[i][k] * B[k][j];
} } }
```

How do we do HLS designs?

Graph Optimizations
Node fusion
IP integration
Task-level pipeline, etc.

Manual Code Rewriting

Loop Optimizations
Loop interchange
Loop perfectization
Loop tile, skew, etc.

Manual Code Rewriting

Directive Optimizations
Loop pipeline, unroll
Function pipeline, inline
Array partition, etc.

Manual Code Rewriting

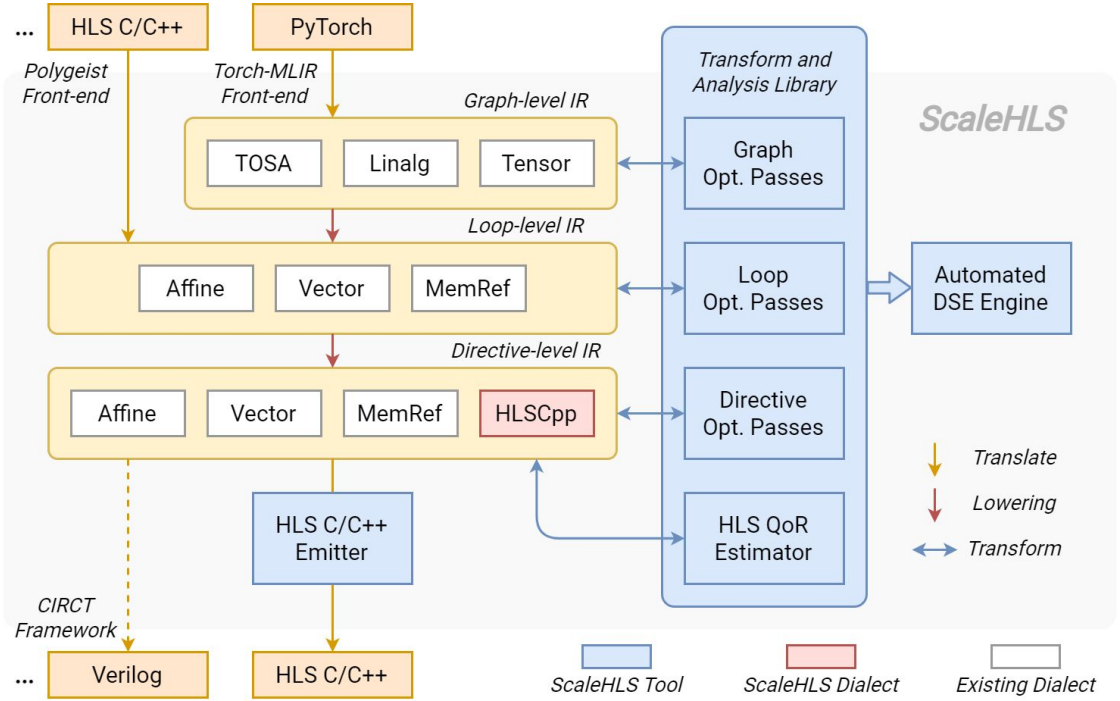
Generate RTL with XILINX VITIS and etc.

Pipeline II is **2** and overall latency is **65,552**

HLS Compilation Outline

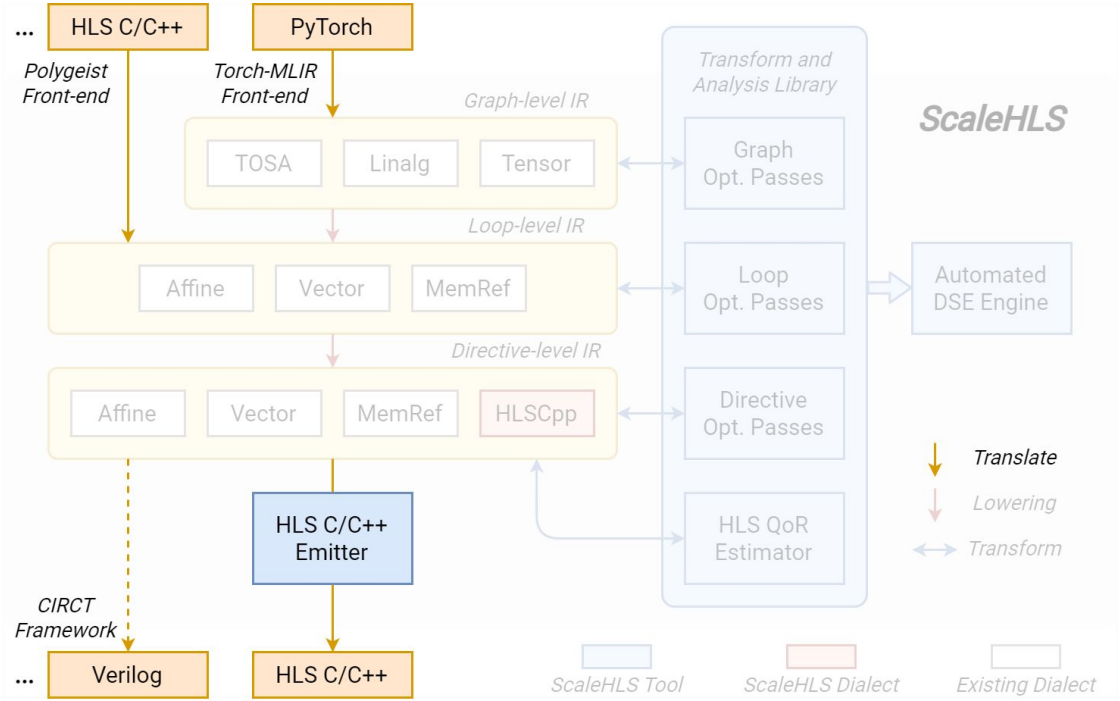
- Motivation
- ScaleHLS framework
- ScaleHLS optimizations
- ScaleHLS design space exploration
- HIDA design space exploration

ScaleHLS Framework: Integration



[1] Polygeist: https://github.com/wsmoses/Polygeist [2] Torch-MLIR: https://github.com/llvm/torch-mlir [3] CIRCT: https://github.com/llvm/circt

ScaleHLS Framework: Integration (Cont'd)



Inputs

C/C++ Polygeist [1]

PyTorch Torch-MLIR [2]

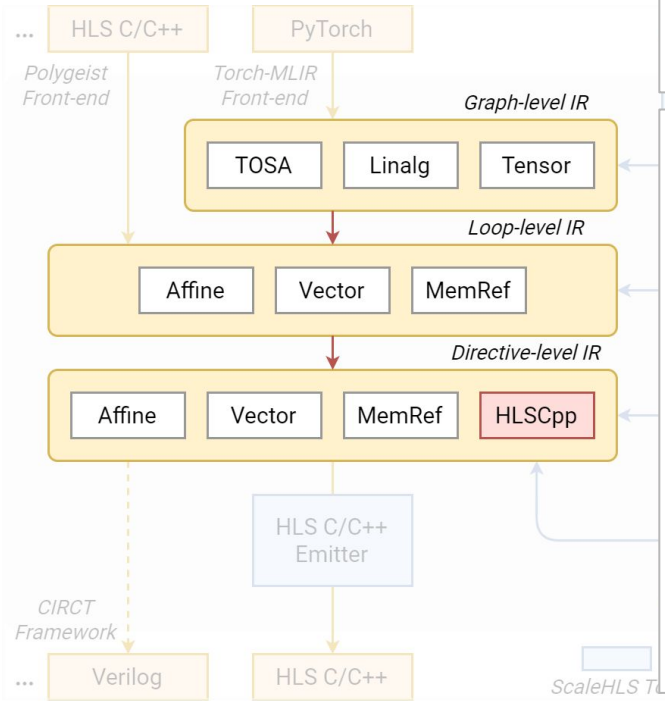
Outputs

C/C++ C/C++ Emitter

Verilog CIRCT [3] (work-in-progress)

[1] Polygeist: https://github.com/wsmoses/Polygeist [2] Torch-MLIR: https://github.com/llvm/torch-mlir [3] CIRCT: https://github.com/llvm/circt

ScaleHLS Framework: Representation



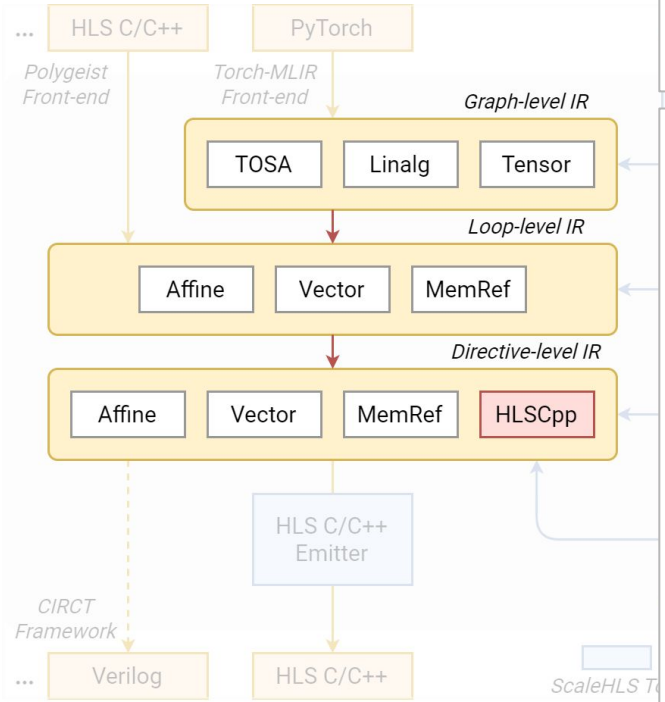

```
%O = "tosa.conv2d"(%I, %W, ...) {...} : (tensor<1x32x32x3xi8>, tensor<64x3x3x3xi8>, ...) ->
                                        tensor<1x30x30x64xi8>
```

Conv2D Graph-level MLIR

```
affine.for %x = 0 to 30 {
  affine.for %y = 0 to 30 {
    affine.for %k = 0 to 64 {
      affine.for %r = 0 to 3 {
        affine.for %s = 0 to 3 {
          affine.for %c = 0 to 3 {
            %i = affine.load %I[0, %x + %r, %y + %s, %c] : memref<1x32x32x3xi8>
            %w = affine.load %W[%r, %s, %c, %k] : memref<3x3x3x64xi8>
            %o = affine.load %O[0, %x, %y, %k] : memref<1x30x30x64xi8>
            %mul = arith.muli %i, %w : i8
            %add = arith.addi %o, %mul : i8
            affine.store %add, %O[0, %x, %y, %k] : memref<1x30x30x64xi8>
          }
        }
      }
    }
  }
}
```

Conv2D Loop-level MLIR

ScaleHLS Framework: Representation (Cont'd)



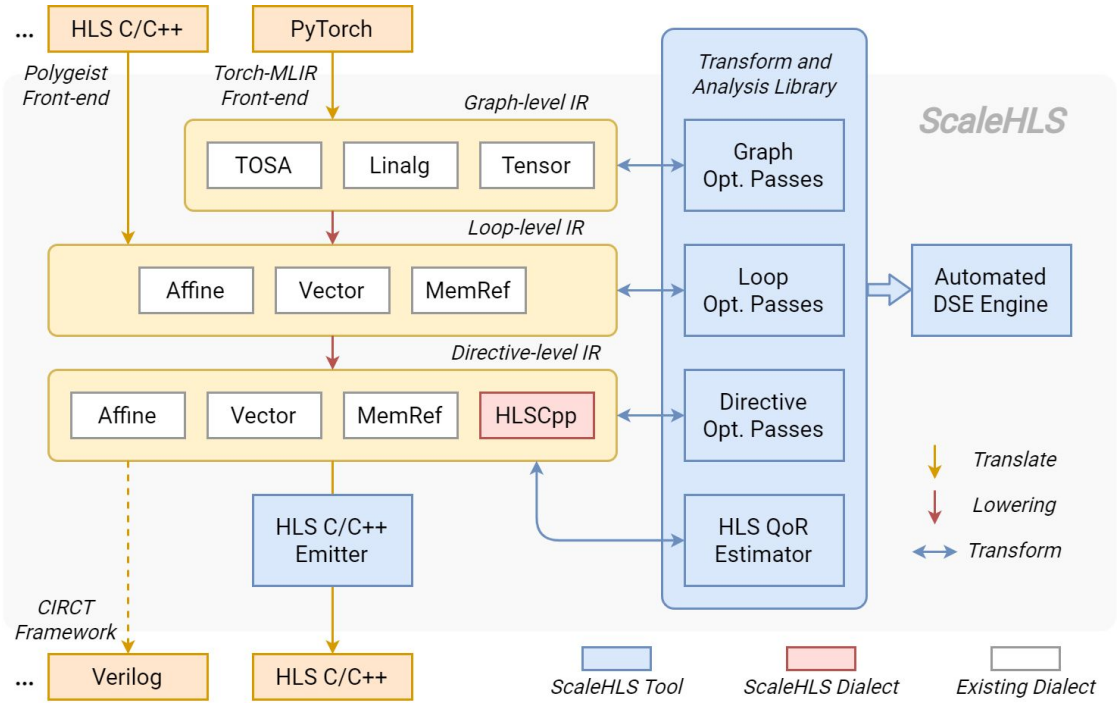

```
%O = "tosa.conv2d"(%I, %W, ...) {...} : (tensor<1x32x32x3xi8>, tensor<64x3x3x3xi8>, ...) ->
                                          tensor<1x30x30x64xi8>
```

Conv2D Graph-level MLIR

```
affine.for %x = 0 to 30 {
  affine.for %y = 0 to 30 {
    affine.for %k = 0 to 64 step 2 {
      affine.for %r = 0 to 3 {
        affine.for %s = 0 to 3 {
          affine.for %c = 0 to 3 {
            %i = affine.load %I[0, %x + %r, %y + %s, %c] : memref<1x32x32x3xi8>
            %w = vector.transfer_read %W[%r, %s, %c, %k], : memref<3x3x3x64xi8>, vector<2xi8>
            ... ...
            %mul = "hlscpp.mul_prim"(%i, %w) : (i8, vector<2xi8>) -> vector<2xi16>
            ... ...
            %mul32 = "hlscpp.cast_prim"(%mul) : (vector<2xi16>) -> vector<2xi32>
            ... ...
          } {loop_directive = #hlscpp.ld<pipeline=true, targetII=1, ...}
        }
      }
    }
  }
}
```

Conv2D Directive-level MLIR

ScaleHLS Framework



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.

HLS Compilation Outline

- Motivation
- ScaleHLS framework
- ScaleHLS optimizations
- ScaleHLS design space exploration
- HIDA design space exploration

ScaleHLS Intra-node Transformations

	Passes	Target	Parameters
Graph	**-legalize-dataflow**	function	insert-copy
	-split-function	function	min-gran
Loop	**-affine-loop-perfectization**	loop band	-
	-affine-loop-order-opt	loop band	perm-map
	-remove-variable-bound	loop band	-
	-affine-loop-tile	loop	tile-size
	-affine-loop-unroll	loop	unroll-factor
Direct.	**-loop-pipelining**	loop	target-ii
	-func-pipelining	function	target-ii
	-array-partition	function	part-factors
Misc.	**-simplify-affine-if**	function	-
	-affine-store-forward	function	-
	-simplify-memref-access	function	-
	-canonicalize -cse	function	-

Boldface ones are new passes provided by us, while others are MLIR built-in passes.

```
void syrk(int alpha, int beta, int C[32][32], int A[32][32]) {
  for (int i = 0; i < 32; i++) {
    for (int j = 0; j <= i; j++) {
      C[i][j] *= beta;
      for (int k = 0; k < 32; k++) {
        C[i][j] += alpha * A[i][k] * A[j][k];
} } } }
```

Baseline C

Loop and Directive Opt in MLIR

```
void syrk(int alpha, int beta, int C[32][32], int A[32][32]) {
#pragma HLS interface s_axilite port=return bundle=ctrl
#pragma HLS interface s_axilite port=alpha bundle=ctrl
#pragma HLS interface s_axilite port=beta bundle=ctrl
#pragma HLS interface bram port=C
#pragma HLS interface bram port=A

#pragma HLS resource variable=C core=ram_s2p_bram

#pragma HLS array_partition variable=A cyclic factor=2 dim=2
#pragma HLS resource variable=A core=ram_s2p_bram

  for (int k = 0; k < 32; k += 2) {
    for (int i = 0; i < 32; i += 1) {
      for (int j = 0; j < 32; j += 1) {
#pragma HLS pipeline II = 3
        if ((i - j) >= 0) {
          int v7 = C[i][j];
          int v8 = beta * v7;
          int v9 = A[i][k];
          int v10 = A[j][k];
          int v11 = (k == 0) ? v8 : v7;
          int v12 = alpha * v9;
          int v13 = v12 * v10;
          int v14 = v11 + v13;
          int v15 = A[i][(k + 1)];
          int v16 = A[j][(k + 1)];
          int v17 = alpha * v15;
          int v18 = v17 * v16;
          int v19 = v14 + v18;
          C[i][j] = v19;
} } } } }
```

Optimized C emitted by the C/C++ emitter

ScaleHLS Intra-node Transformations (Cont.)

Loop Order Permutation

- The minimum II (Initiation Interval) of a loop pipeline can be calculated as:

$$II_{min} = \max_{d} \left(\left\lceil \frac{Delay_d}{Distance_d} \right\rceil \right)$$

- $Delay_d$ and $Distance_d$ are the scheduling delay and distance (calculated from the dependency vector) of each loop-carried dependency d.
- To achieve a smaller II, the loop order permutation pass performs affine analysis and attempt to permute loops associated with loop-carried dependencies in order to maximize the $Distance$.

```
void syrk(int alpha, int beta, int C[32][32], int A[32][32]) {
  for (int i = 0; i < 32; i++) {
    for (int j = 0; j <= i; j++) {
      C[i][j] *= beta;
      for (int k = 0; k < 32; k++) {
        C[i][j] += alpha * A[i][k] * A[j][k];
} } } }
```

Loop perfectization

Baseline C

Loop and Directive Opt in MLIR

```
void syrk(int alpha, int beta, int C[32][32], int A[32][32]) {
#pragma HLS interface s_axilite port=return bundle=ctrl
#pragma HLS interface s_axilite port=alpha bundle=ctrl
#pragma HLS interface s_axilite port=beta bundle=ctrl
#pragma HLS interface bram port=C
#pragma HLS interface bram port=A

#pragma HLS resource variable=C core=ram_s2p_bram

#pragma HLS array_partition variable=A cyclic factor=2 dim=2
#pragma HLS resource variable=A core=ram_s2p_bram

  for (int k = 0; k < 32; k += 2) {
    for (int i = 0; i < 32; i += 1) {
      for (int j = 0; j < 32; j += 1) {
#pragma HLS pipeline II = 3
        if ((i - j) >= 0) {
          int v7 = C[i][j];
          int v8 = beta * v7;
          int v9 = A[i][k];
          int v10 = A[j][k];
          int v11 = (k == 0) ? v8 : v7;
          int v12 = alpha * v9;
          int v13 = v12 * v10;
          int v14 = v11 + v13;
          int v15 = A[i][(k + 1)];
          int v16 = A[j][(k + 1)];
          int v17 = alpha * v15;
          int v18 = v17 * v16;
          int v19 = v14 + v18;
          C[i][j] = v19;
} } } } }
```

Loop order permutation; Loop unroll

Remove variable loop bound

Optimized C emitted by the C/C++ emitter

ScaleHLS Intra-node Transformations (Cont.)

Loop Pipelining

- Apply loop pipelining directives to a loop and set a targeted initiation interval.
- In the IR of ScaleHLS, directives are represented using the HLSCpp dialect. In the example, the pipelined %j loop is represented as:

```
affine.for %j = 0 to 32 {
  … …
} attributes {loop_directive = #hlscpp.ld<pipeline=1,
targetII=3, dataflow=0, flatten=0, … … >}
```

```
void syrk(int alpha, int beta, int C[32][32], int A[32][32]) {
  for (int i = 0; i < 32; i++) {
    for (int j = 0; j <= i; j++) {
      C[i][j] *= beta;
      for (int k = 0; k < 32; k++) {
        C[i][j] += alpha * A[i][k] * A[j][k];
} } } }
```

Loop perfectization

Baseline C

Loop and Directive Opt in MLIR

```
void syrk(int alpha, int beta, int C[32][32], int A[32][32]) {
#pragma HLS interface s_axilite port=return bundle=ctrl
#pragma HLS interface s_axilite port=alpha bundle=ctrl
#pragma HLS interface s_axilite port=beta bundle=ctrl
#pragma HLS interface bram port=C
#pragma HLS interface bram port=A

#pragma HLS resource variable=C core=ram_s2p_bram

#pragma HLS array_partition variable=A cyclic factor=2 dim=2
#pragma HLS resource variable=A core=ram_s2p_bram

  for (int k = 0; k < 32; k += 2) {
    for (int i = 0; i < 32; i += 1) {
      for (int j = 0; j < 32; j += 1) {
#pragma HLS pipeline II = 3
        if ((i - j) >= 0) {
          int v7 = C[i][j];
          int v8 = beta * v7;
          int v9 = A[i][k];
          int v10 = A[j][k];
          int v11 = (k == 0) ? v8 : v7;
          int v12 = alpha * v9;
          int v13 = v12 * v10;
          int v14 = v11 + v13;
          int v15 = A[i][(k + 1)];
          int v16 = A[j][(k + 1)];
          int v17 = alpha * v15;
          int v18 = v17 * v16;
          int v19 = v14 + v18;
          C[i][j] = v19;
} } } } }
```

Loop order permutation; Loop unroll

Remove variable loop bound

Loop pipeline

Optimized C emitted by the C/C++ emitter

ScaleHLS Intra-node Transformations (Cont.)

Array Partition

- Array partition is one of the most important directives because the memories requires enough bandwidth to comply with the computation parallelism.
- The array partition pass analyzes the accessing pattern of each array and automatically select suitable partition fashion and factor.
- In the example, the %A array is accessed at address `[i,k]` and `[i,k+1]` simultaneously after pipelined, thus %A array is cyclically partitioned with two.

```
void syrk(int alpha, int beta, int C[32][32], int A[32][32]) {
  for (int i = 0; i < 32; i++) {
    for (int j = 0; j <= i; j++) {
      C[i][j] *= beta;
      for (int k = 0; k < 32; k++) {
        C[i][j] += alpha * A[i][k] * A[j][k];
} } } }
```

Loop perfectization

Baseline C

Loop and Directive Opt in MLIR

```
void syrk(int alpha, int beta, int C[32][32], int A[32][32]) {
#pragma HLS interface s_axilite port=return bundle=ctrl
#pragma HLS interface s_axilite port=alpha bundle=ctrl
#pragma HLS interface s_axilite port=beta bundle=ctrl
#pragma HLS interface bram port=C
#pragma HLS interface bram port=A

#pragma HLS resource variable=C core=ram_s2p_bram

#pragma HLS array_partition variable=A cyclic factor=2 dim=2
#pragma HLS resource variable=A core=ram_s2p_bram

  for (int k = 0; k < 32; k += 2) {
    for (int i = 0; i < 32; i += 1) {
      for (int j = 0; j < 32; j += 1) {
#pragma HLS pipeline II = 3
        if ((i - j) >= 0) {
          int v7 = C[i][j];
          int v8 = beta * v7;
          int v9 = A[i][k];
          int v10 = A[j][k];
          int v11 = (k == 0) ? v8 : v7;
          int v12 = alpha * v9;
          int v13 = v12 * v10;
          int v14 = v11 + v13;
          int v15 = A[i][(k + 1)];
          int v16 = A[j][(k + 1)];
          int v17 = alpha * v15;
          int v18 = v17 * v16;
          int v19 = v14 + v18;
          C[i][j] = v19;
} } } } }
```

Array partition

Loop order permutation; Loop unroll

Remove variable loop bound

Loop pipeline

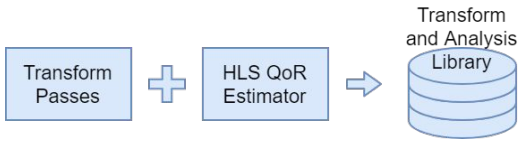
Simplify if ops;
Store ops forward;
Simplify memref ops

Optimized C emitted by the C/C++ emitter

ScaleHLS Intra-node Transformations (Cont.)

Transform and Analysis Library

- Apart from the optimizations, ScaleHLS provides a QoR estimator based on an ALAP scheduling algorithm. The memory ports are considered as non-shareable resources and constrained in the scheduling.
- The interfaces of all optimization passes and the QoR estimator are packaged into a library, which can be called by the DSE engine to generate and evaluate design points.




```
void syrk(int alpha, int beta, int C[32][32], int A[32][32]) {
  for (int i = 0; i < 32; i++) {
    for (int j = 0; j <= i; j++) {
      C[i][j] *= beta;
      for (int k = 0; k < 32; k++) {
        C[i][j] += alpha * A[i][k] * A[j][k];
} } } }
```

Loop perfectization

Baseline C

Loop and Directive Opt in MLIR

```
void syrk(int alpha, int beta, int C[32][32], int A[32][32]) {
#pragma HLS interface s_axilite port=return bundle=ctrl
#pragma HLS interface s_axilite port=alpha bundle=ctrl
#pragma HLS interface s_axilite port=beta bundle=ctrl
#pragma HLS interface bram port=C
#pragma HLS interface bram port=A

#pragma HLS resource variable=C core=ram_s2p_bram

#pragma HLS array_partition variable=A cyclic factor=2 dim=2
#pragma HLS resource variable=A core=ram_s2p_bram

  for (int k = 0; k < 32; k += 2) {
    for (int i = 0; i < 32; i += 1) {
      for (int j = 0; j < 32; j += 1) {
#pragma HLS pipeline II = 3
        if ((i - j) >= 0) {
          int v7 = C[i][j];
          int v8 = beta * v7;
          int v9 = A[i][k];
          int v10 = A[j][k];
          int v11 = (k == 0) ? v8 : v7;
          int v12 = alpha * v9;
          int v13 = v12 * v10;
          int v14 = v11 + v13;
          int v15 = A[i][(k + 1)];
          int v16 = A[j][(k + 1)];
          int v17 = alpha * v15;
          int v18 = v17 * v16;
          int v19 = v14 + v18;
          C[i][j] = v19;
} } } } }
```

Array partition

Loop order permutation; Loop unroll

Remove variable loop bound

Loop pipeline

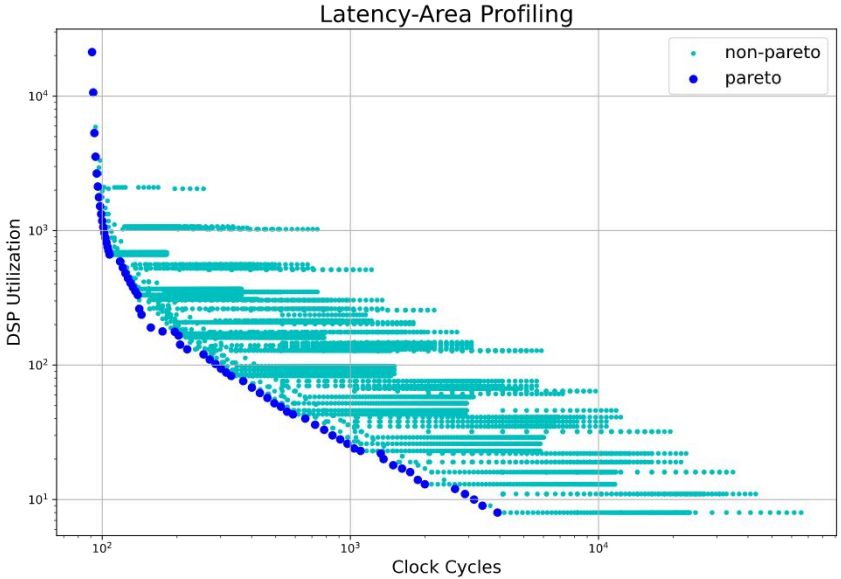
Simplify if ops;
Store ops forward;
Simplify memref ops

Optimized C emitted by the C/C++ emitter

HLS Compilation Outline

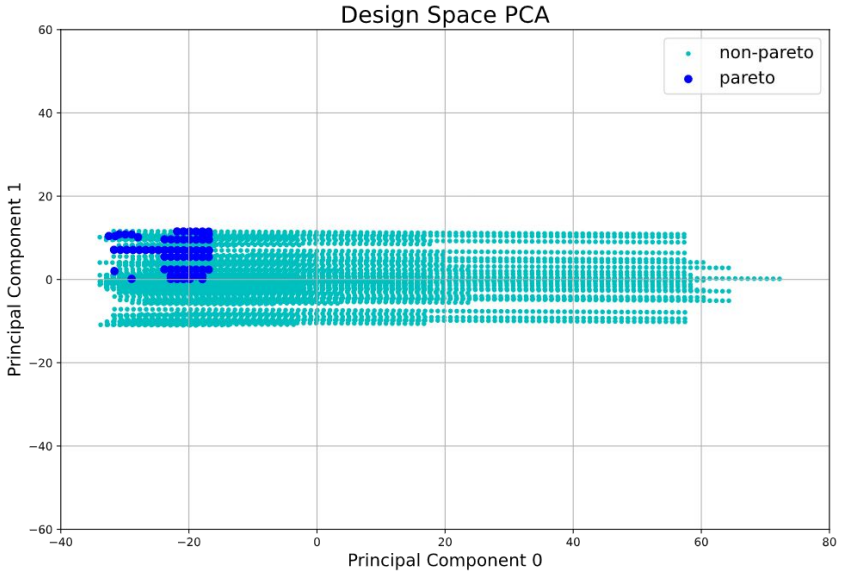
- Motivation
- ScaleHLS framework
- ScaleHLS optimizations
- ScaleHLS design space exploration
- HIDA design space exploration

Intra-node Design Space Exploration - Observation



Pareto frontier of a GEMM kernel

- Latency and area are profiled for each design point
- Dark blue points are Pareto points
- Loop perfectization, loop order permutation, loop tiling, loop pipelining, and array partition passes are involved

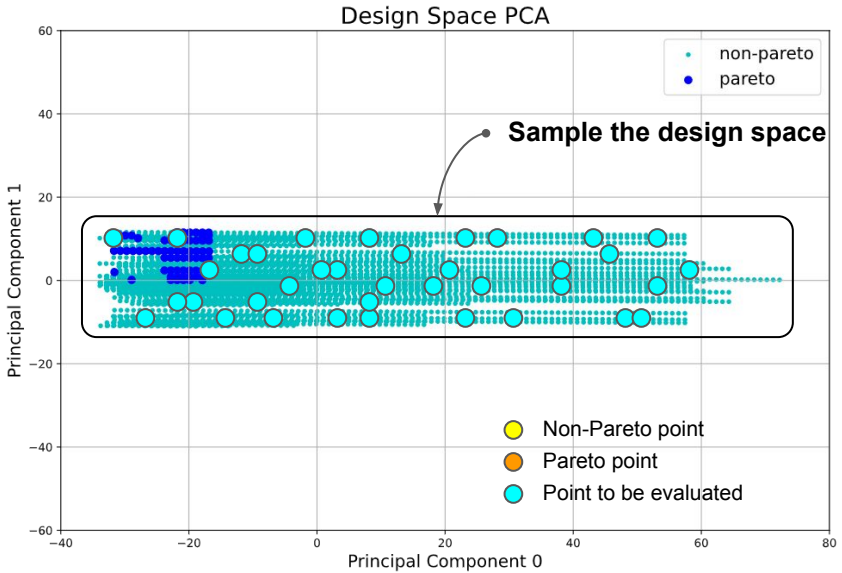


- Each parameter of a pass becomes one dimension, the original 4-dimensional design space is reduced to two dimensions through PCA
- Pareto points are located at a corner of the design space, the variance of Pareto points is much smaller than the overall variance

Intra-node Design Space Exploration (Cont.)

DSE algorithm:

1. Sample the whole design space and evaluate each sampled design point with the QoR estimator

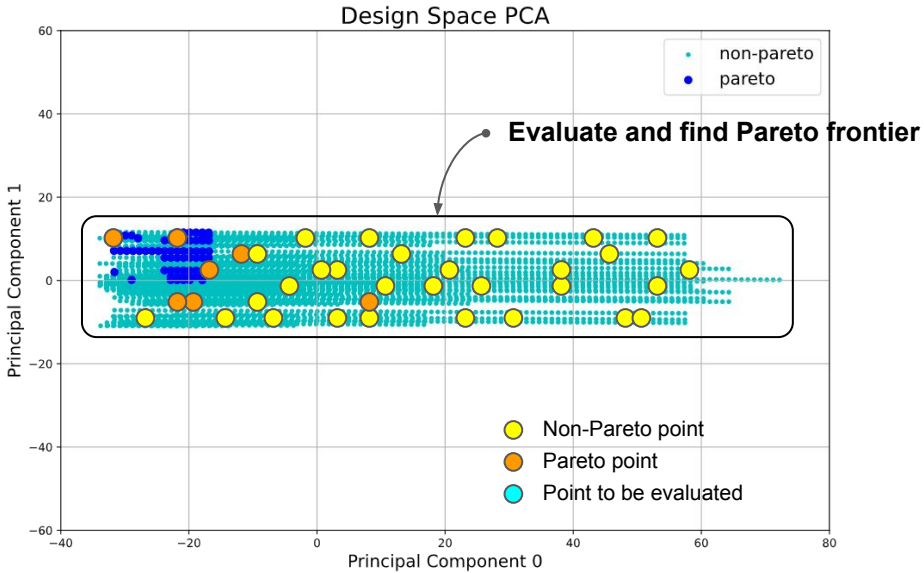


- Each parameter of a pass becomes one dimension, the original 4-dimensional design space is reduced to two dimensions through PCA
- Pareto points are located at a corner of the design space, the variance of Pareto points is much smaller than the overall variance

Intra-node Design Space Exploration (Cont.)

DSE algorithm:

1. Sample the whole design space and evaluate each sampled design point with the QoR estimator
2. Extract the Pareto frontier from all evaluated design points

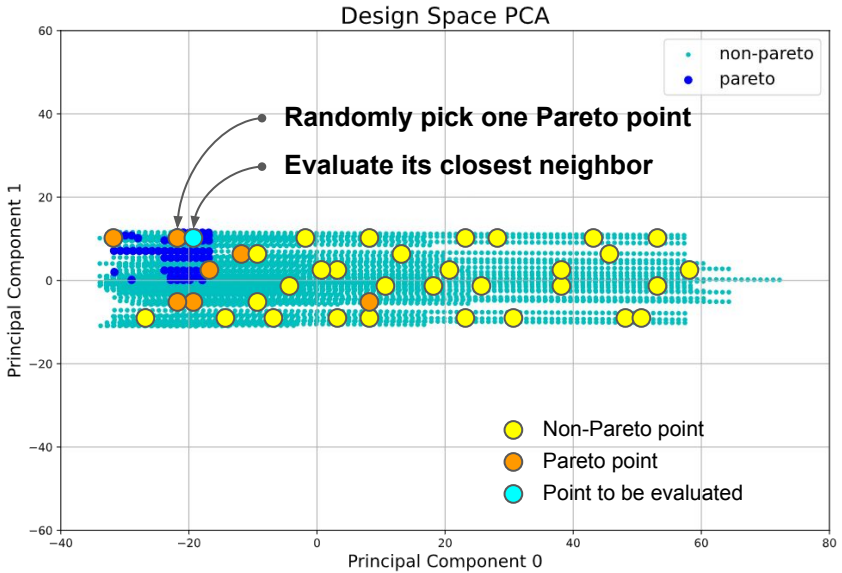


- Each parameter of a pass becomes one dimension, the original 4-dimensional design space is reduced to two dimensions through PCA
- Pareto points are located at a corner of the design space, the variance of Pareto points is much smaller than the overall variance

Intra-node Design Space Exploration (Cont.)

DSE algorithm:

1. Sample the whole design space and evaluate each sampled design point with the QoR estimator
2. Extract the Pareto frontier from all evaluated design points
3. Evaluate the closest neighbor of a randomly selected design point in the current Pareto frontier

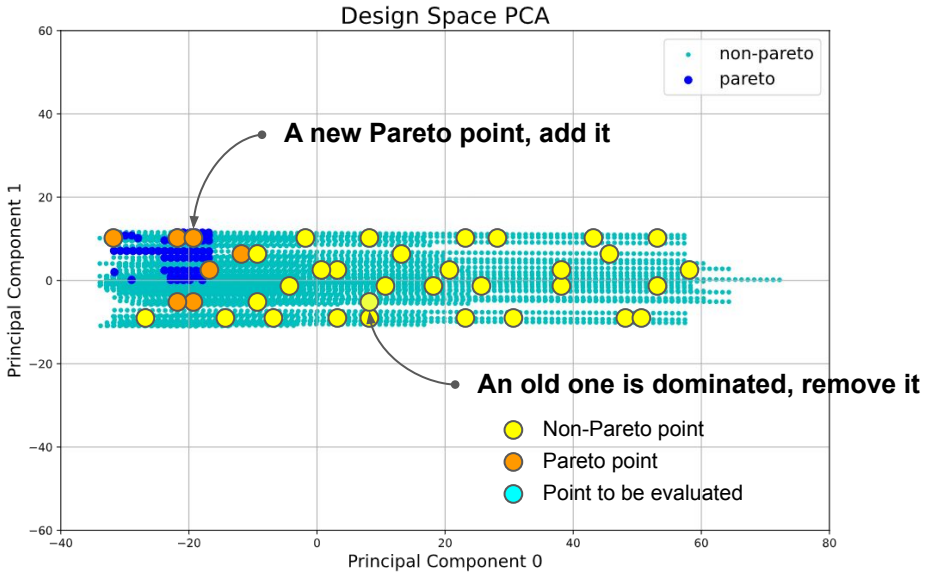


- Each parameter of a pass becomes one dimension, the original 4-dimensional design space is reduced to two dimensions through PCA
- Pareto points are located at a corner of the design space, the variance of Pareto points is much smaller than the overall variance

Intra-node Design Space Exploration (Cont.)

DSE algorithm:

1. Sample the whole design space and evaluate each sampled design point with the QoR estimator
2. Extract the Pareto frontier from all evaluated design points
3. Evaluate the closest neighbor of a randomly selected design point in the current Pareto frontier
4. Repeat step (2) and (3) to update the discovered Pareto frontier



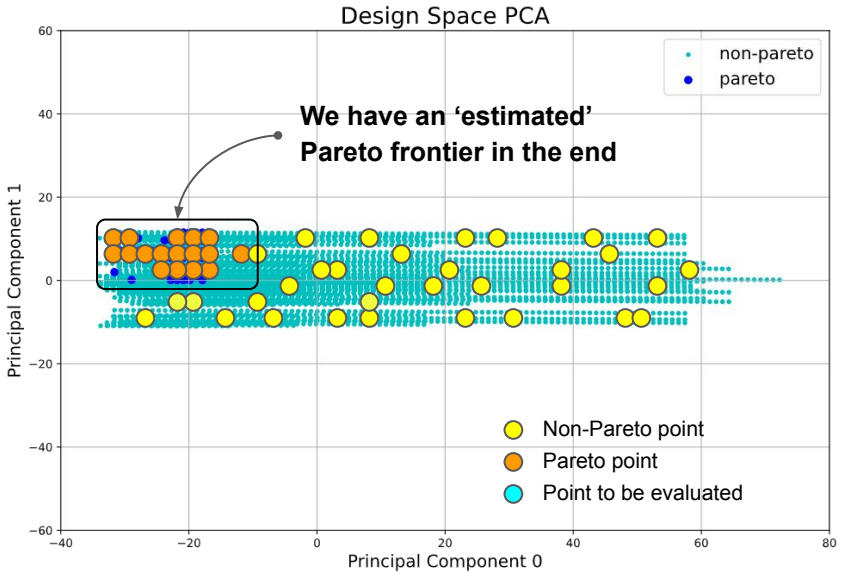
- Each parameter of a pass becomes one dimension, the original 4-dimensional design space is reduced to two dimensions through PCA
- Pareto points are located at a corner of the design space, the variance of Pareto points is much smaller than the overall variance

Intra-node Design Space Exploration (Cont.)

DSE algorithm:

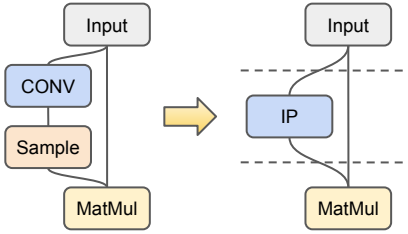
1. Sample the whole design space and evaluate each sampled design point with the QoR estimator
2. Extract the Pareto frontier from all evaluated design points
3. Evaluate the closest neighbor of a randomly selected design point in the current Pareto frontier
4. Repeat step (2) and (3) to update the discovered Pareto frontier
5. Stop when no eligible neighbor can be found or meeting the early-termination criteria

Given the **Transform and Analysis Library** provided by ScaleHLS, the DSE engine can be extended to support other optimization algorithms in the future.

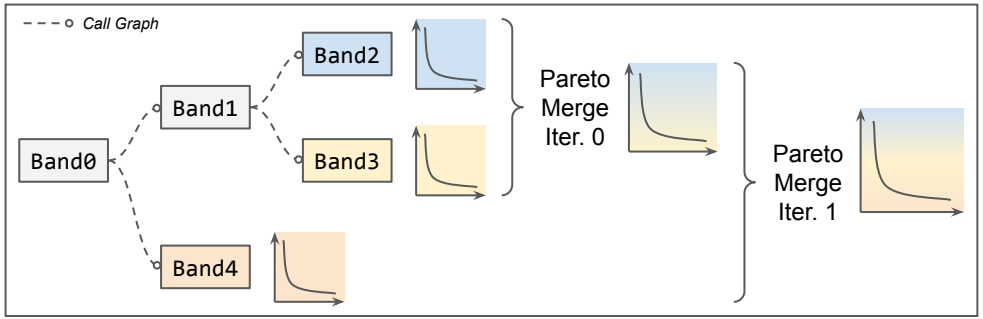


- Each parameter of a pass becomes one dimension, the original 4-dimensional design space is reduced to two dimensions through PCA
- Pareto points are located at a corner of the design space, the variance of Pareto points is much smaller than the overall variance

ScaleHLS Global Design Space Exploration



Graph Optimizations



Step (2) Global multi-kernel Pareto curving merging

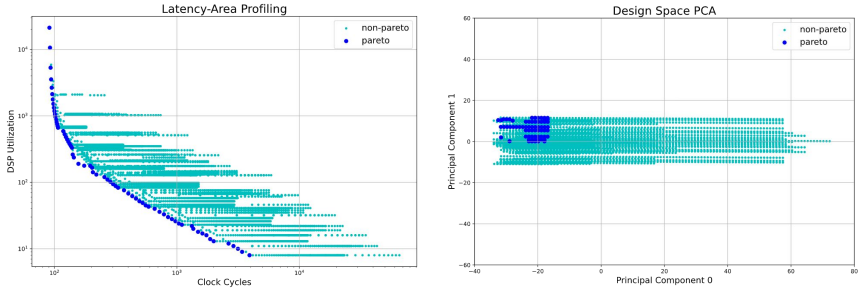
Loop Optimizations

Directive Optimizations

```
for (int i = 0; i < 32; i++) {
  for (int j = 0; j < 32; j++) {
    C[i][j] *= beta;
    for (int k = 0; k < 32; k++) {
      C[i][j] += alpha * A[i][k] * B[k][j];
} } }
```

```
for (int k = 0; k < 32; k++) {
  for (int i = 0; i < 32; i++) {
    for (int j = 0; j < 32; j++) {
      if (k == 0)
        C[i][j] *= beta;
      C[i][j] += alpha * A[i][k] * B[k][j];
} } }
```

```
for (int k = 0; k < 32; k++) {
  for (int i = 0; i < 32; i++) {
    for (int j = 0; j < 32; j++) {
#pragma HLS pipeline
      if (k == 0)
        C[i][j] *= beta;
      C[i][j] += alpha * A[i][k] * B[k][j];
} } }
```



Step (1) Local single-kernel *loop* and *directive* DSE

ScaleHLS DSE Results of C/C++ Kernel

Kernel	Prob. Size	Speedup	LP	RVB	Perm. Map	Tiling Sizes	Pipeline II	Array Partition
BICG	4096	41.7×	No	No	[1, 0]	[16, 8]	43	A:[8, 16], s:[16], q:[8], p:[16], r:[8]
GEMM	4096	768.1×	Yes	No	[1, 2, 0]	[8, 1, 16]	3	C:[1, 16], A:[1, 8], B:[8, 16]
GESUMMV	4096	199.1×	Yes	No	[1, 0]	[8, 16]	9	A:[16, 8], B:[16, 8], tmp:[16], x:[8], y:[16]
SYR2K	4096	384.0×	Yes	Yes	[1, 2, 0]	[8, 4, 4]	8	C:[4, 4], A:[4, 8], B:[4, 8]
SYRK	4096	384.1×	Yes	Yes	[1, 2, 0]	[64, 1, 1]	3	C:[1, 1], A:[1, 64]
TRMM	4096	590.9×	Yes	Yes	[1, 2, 0]	[4, 4, 32]	13	A:[4, 4], B:[4, 32]

DSE results of PolyBench-C computation kernels

1. The target platform is Xilinx XC7Z020 FPGA, which is an edge FPGA with 4.9 Mb memories, 220 DSPs, and 53,200 LUTs. The data types of all kernels are single-precision floating-points.
2. Among all six benchmarks, a **speedup** ranging from 41.7× to 768.1× is obtained compared to the baseline design, which is the original computation kernel from PolyBench-C without the optimization of DSE.
3. **LP** and **RVB** denote Loop Perfectization and Remove Variable Bound, respectively.
4. In the Loop Order Optimization (**Perm. Map**), the i-th loop in the loop nest is permuted to location $PermMap$ [i], where locations are from the outermost loop to inner.

ScaleHLS Results of DNN Models

Model	Speedup	Runtime (seconds)	Memory (SLR Util. %)	DSP (SLR Util. %)	LUT (SLR Util. %)	FF (SLR Util. %)	Our DSP Effi. (OPs/Cycle/DSP)	DSP Effi. of TVM-VTA [26]
ResNet-18	3825.0×	60.8	91.7Mb (79.5%)	1326 (58.2%)	157902 (40.1%)	54766 (6.9%)	1.343	0.344
VGG-16	1505.3×	37.3	46.7Mb (40.5%)	878 (38.5%)	88108 (22.4%)	31358 (4.0%)	0.744	0.296
MobileNet	1509.0×	38.1	79.4Mb (68.9%)	1774 (77.8%)	138060 (35.0%)	56680 (7.2%)	0.791	0.468

Optimization results of representative DNN models

1. The target platform is one SLR (super logic region) of Xilinx VU9P FPGA which is a large FPGA containing 115.3 Mb memories, 2280 DSPs and 394,080 LUTs on each SLR.
2. The PyTorch implementations are parsed into ScaleHLS and optimized using the proposed multi-level optimization methodology.
3. By combining the graph, loop, and directive levels of optimization, a **speedup** ranging from 1505.3× to 3825.0× is obtained compared to the baseline designs, which are compiled from PyTorch to HLS C/C++ through ScaleHLS but without the multi-level optimization applied.

HLS Compilation Outline

- Motivation
- ScaleHLS framework
- ScaleHLS optimizations
- ScaleHLS design space exploration
- HIDA design space exploration

Limitation of ScaleHLS

```
1  float A[32][16];
2  NODE0_I: for (int i=0; i<32; i++)
3    NODE0_K: for (int k=0; k<16; k++)
4      A[i][k] = ...; // Load array A.
5
6  float B[16][16];
7  NODE1_K: for (int k=0; k<16; k++)
8    NODE1_J: for (int j=0; j<16; j++)
9      B[k][j] = ...; // Load array B.
10
11 float C[16][16];
12 NODE2_I: for (int i=0; i<16; i++)
13   NODE2_J: for (int j=0; j<16; j++)
14     NODE2_K: for (int k=0; k<16; k++)
15       C[i][j] = A[i*2][k] * B[k][j];
```

Inter-kernel Correlation

- Node0 is connected to Node2 through buffer A
  - If buffer A is on-chip, the partition strategy of A is HIGHLY correlated with the parallel strategies of both Node0 and Node2
- Node1 is connected to Node2 through buffer B
  - Same as above

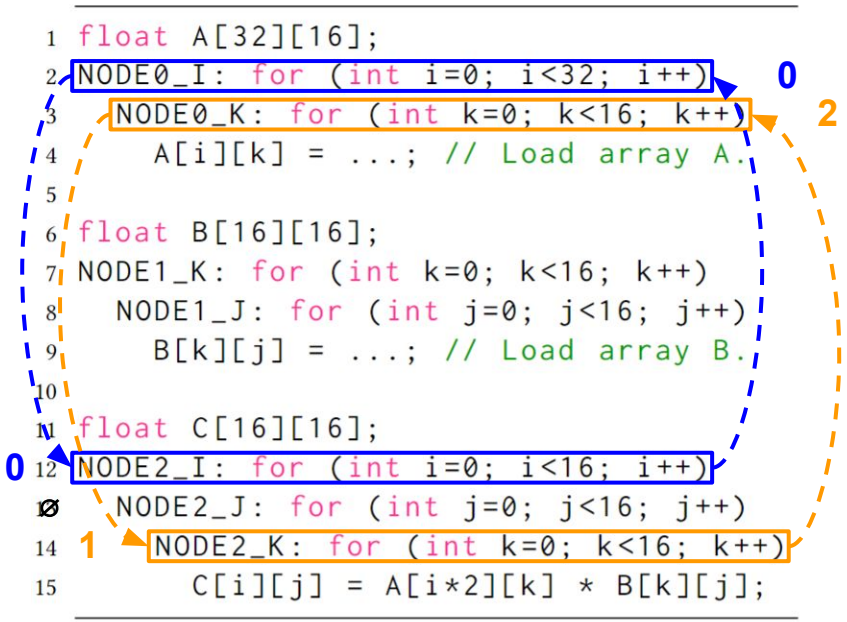
Connectedness

- Node0, 1, and 2 have different trip count: 32*16, 16*16, and 16*16*16
  - To enable efficient pipeline execution of Node0, 1, and 2, their latencies after parallelization should be similar

Intensity

Simply merging the local Pareto curves will not work well!

What we did in HIDA

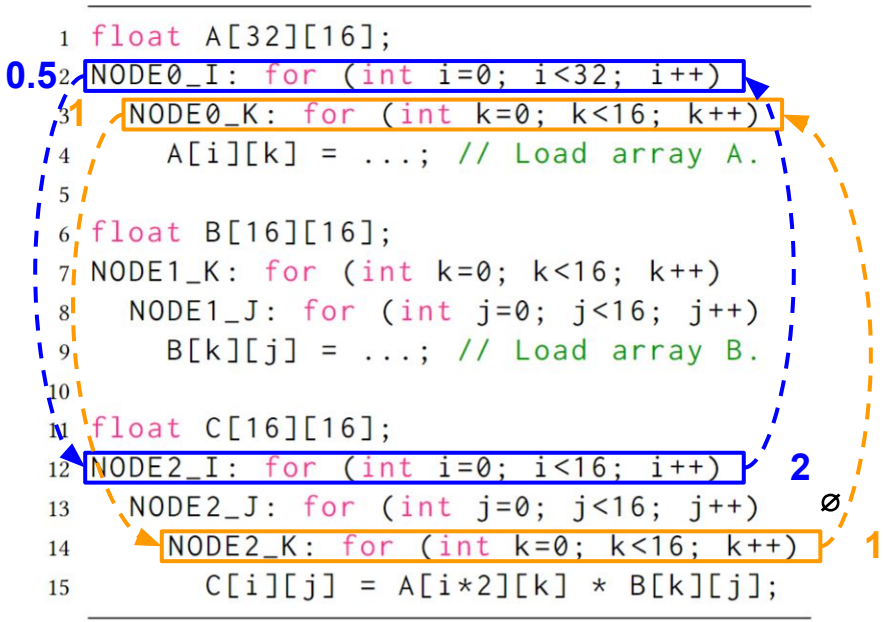


Step (1) Connectedness Analysis

Source	Target	Buffer	Permutation Map		Scaling Map	
			S-to-T	T-to-S	S-to-T	T-to-S
Node0	Node2	A	[0, ∅, 1]	[0, 2]	[0.5, 1]	[2, ∅, 1]
Node1	Node2	B	[∅, 1, 0]	[2, 1]	[1, 1]	[∅, 1, 1]

- **Permutation Map**
  - Record the alignment between loops

What we did in HIDA (Cont'd)



Step (1) Connectedness Analysis

Source	Target	Buffer	Permutation Map		Scaling Map	
			S-to-T	T-to-S	S-to-T	T-to-S
Node0	Node2	A	[0, ∅, 1]	[0, 2]	[0.5, 1]	[2, ∅, 1]
Node1	Node2	B	[∅, 1, 0]	[2, 1]	[1, 1]	[∅, 1, 1]

- **Permutation Map**
  - Record the alignment between loops
- **Scaling Map**
  - Record the alignment between strides
- **Affine Analysis-based**
  - Demand preprocessing: Loop normalize and perfectize, memory canonicalize

What we did in HIDA (Cont'd)

```
1  float A[32][16];
2  NODE0_I: for (int i=0; i<32; i++)
3    NODE0_K: for (int k=0; k<16; k++)
4      A[i][k] = ...; // Load array A.
5
6  float B[16][16];
7  NODE1_K: for (int k=0; k<16; k++)
8    NODE1_J: for (int j=0; j<16; j++)
9      B[k][j] = ...; // Load array B.
10
11 float C[16][16];
12 NODE2_I: for (int i=0; i<16; i++)
13   NODE2_J: for (int j=0; j<16; j++)
14     NODE2_K: for (int k=0; k<16; k++)
15       C[i][j] = A[i*2][k] * B[k][j];
```

Step (2) Node Sorting

Node	Connectedness	Intensity
Node0	1	512
Node1	1	256
Node2	2	4096

- **Descending Order of Connectedness**
  - Higher-connectedness node will affect more nodes
- **Intensity as Tie-breaker**
  - Higher-intensity nodes are more computationally complex, being more sensitive to optimization
- **Order: Node2 -> Node0 -> Node1**

What we did in HIDA (Cont'd)

```
1  float A[32][16];
2  NODE0_I: for (int i=0; i<32; i++)
3    NODE0_K: for (int k=0; k<16; k++)
4      A[i][k] = ...; // Load array A.
5
6  float B[16][16];
7  NODE1_K: for (int k=0; k<16; k++)
8    NODE1_J: for (int j=0; j<16; j++)
9      B[k][j] = ...; // Load array B.
10
11 float C[16][16];
12 NODE2_I: for (int i=0; i<16; i++)
13   NODE2_J: for (int j=0; j<16; j++)
14     NODE2_K: for (int k=0; k<16; k++)
15       C[i][j] = A[i*2][k] * B[k][j];
```

Step (3) Node Parallelization

- **Assuming maximum parallel factor is 32**
- **Node2 Parallelization: [4, 8, 1]**
  - Overall parallel factor is 32
  - ScaleHLS DSE without constraints
  - Solution unroll factors: [4, 8, 1]

What we did in HIDA (Cont'd)

```
1  float A[32][16];
2  NODE0_I: for (int i=0; i<32; i++)
3    NODE0_K: for (int k=0; k<16; k++)
4      A[i][k] = ...; // Load array A.
5
6  float B[16][16];
7  NODE1_K: for (int k=0; k<16; k++)
8    NODE1_J: for (int j=0; j<16; j++)
9      B[k][j] = ...; // Load array B.
10
11 float C[16][16];
12 NODE2_I: for (int i=0; i<16; i++)
13   NODE2_J: for (int j=0; j<16; j++)
14     NODE2_K: for (int k=0; k<16; k++)
15       C[i][j] = A[i*2][k] * B[k][j];
```

Source	Target	Buffer	Permutation Map		Scaling Map	
			S-to-T	T-to-S	S-to-T	T-to-S
Node0	Node2	A	[0, ∅, 1]	[0, 2]	[0.5, 1]	[2, ∅, 1]
Node1	Node2	B	[∅, 1, 0]	[2, 1]	[1, 1]	[∅, 1, 1]

Step (3) Node Parallelization

- **Assuming maximum parallel factor is 32**
- **Node2 Parallelization: [4, 8, 1]**
- **Node0 Parallelization: [4, 1]**
  - Overall parallel factor is 4, calculated from intensities of Node0 and 2 (32*512/4096)
  - ScaleHLS DSE with connectedness constraints, the unroll factors must NOT be mutually indivisible with constraints
    - Multiply with scaling map:
    - [4, 8, 1] ⊙ [2, ∅, 1] = [8, ∅, 1]
    - Permute with permutation map:
    - permute([8, ∅, 1], [0, 2]) = [8, 1]
  - Solution unroll factors: [4, 1]

What we did in HIDA (Cont'd)

```
1  float A[32][16];
2  NODE0_I: for (int i=0; i<32; i++)
3    NODE0_K: for (int k=0; k<16; k++)
4      A[i][k] = ...; // Load array A.
5
6  float B[16][16];
7  NODE1_K: for (int k=0; k<16; k++)
8    NODE1_J: for (int j=0; j<16; j++)
9      B[k][j] = ...; // Load array B.
10
11 float C[16][16];
12 NODE2_I: for (int i=0; i<16; i++)
13   NODE2_J: for (int j=0; j<16; j++)
14     NODE2_K: for (int k=0; k<16; k++)
15       C[i][j] = A[i*2][k] * B[k][j];
```

Source	Target	Buffer	Permutation Map		Scaling Map	
			S-to-T	T-to-S	S-to-T	T-to-S
Node0	Node2	A	[0, ∅, 1]	[0, 2]	[0.5, 1]	[2, ∅, 1]
Node1	Node2	B	[∅, 1, 0]	[2, 1]	[1, 1]	[∅, 1, 1]

Step (3) Node Parallelization

- **Assuming maximum parallel factor is 32**
- **Node2 Parallelization: [4, 8, 1]**
- **Node0 Parallelization: [4, 1]**
- **Node1 Parallelization: [1, 2]**
  - Overall parallel factor is 2, calculated from intensities of Node0 and 1 (32*256/4096)
  - ScaleHLS DSE with connectedness constraints
  - Solution unroll factors: [1, 2]

What we did in HIDA (Cont'd)

```
1  float A[32][16];
2  NODE0_I: for (int i=0; i<32; i++)
3    NODE0_K: for (int k=0; k<16; k++)
4      A[i][k] = ...; // Load array A.
5
6  float B[16][16];
7  NODE1_K: for (int k=0; k<16; k++)
8    NODE1_J: for (int j=0; j<16; j++)
9      B[k][j] = ...; // Load array B.
10
11 float C[16][16];
12 NODE2_I: for (int i=0; i<16; i++)
13   NODE2_J: for (int j=0; j<16; j++)
14     NODE2_K: for (int k=0; k<16; k++)
15       C[i][j] = A[i*2][k] * B[k][j];
```

Step (3) Node Parallelization

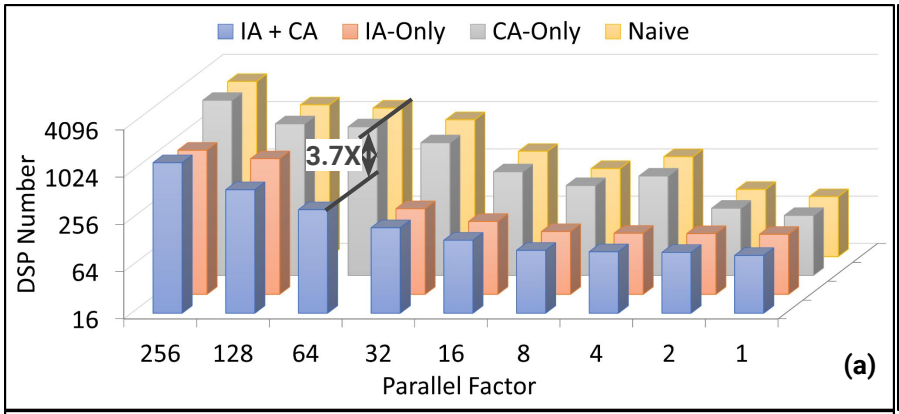
Node	Intensity	Parallel Factor		Loop Unroll Factors			
		w/o IA	w/ IA	IA+CA	IA	CA	Naive
Node0	512	32	4	[4, 1]	[2, 2]	[8, 4]	[4, 8]
Node1	256	32	2	[1, 2]	[1, 2]	[4, 8]	[4, 8]
Node2	4,096	32	32	[4, 8, 1]	[4, 8, 1]	[4, 8, 1]	[4, 8, 1]

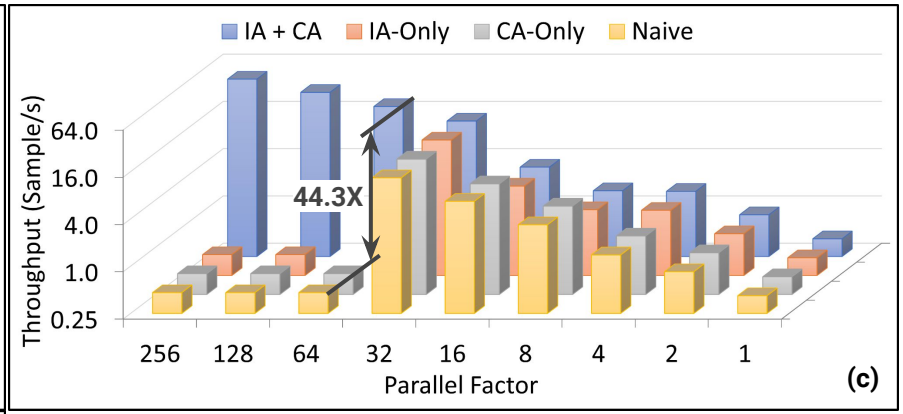
Intensity-aware (IA) Connectedness-aware (CA) HIDA DSE

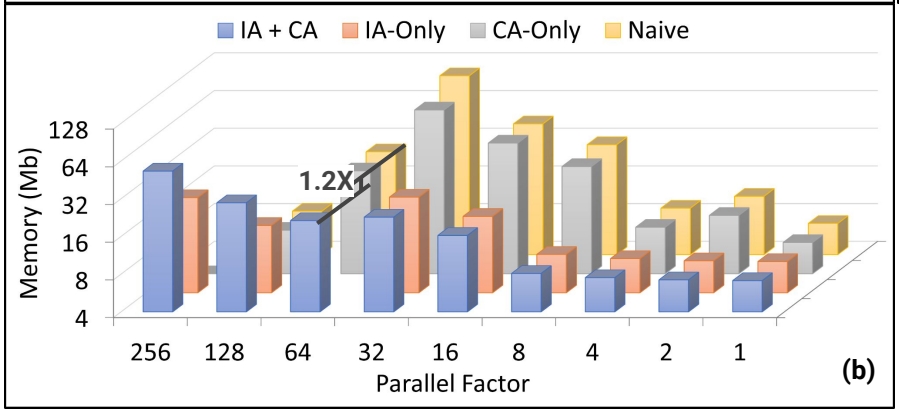
Naive ScaleHLS DSE

Array	Array Partition Factors				Bank Number				
	IA+CA	IA	CA	Naive	IA+CA	IA	CA	Naive	
A	[8, 1]	[8, 2]	[8, 4]	[8, 8]	8	16	32	64	***8x***
B	[1, 8]	[2, 8]	[4, 8]	[8, 8]	8	16	32	64	***8x***
C	[4, 8]	[4, 8]	[4, 8]	[4, 8]	32	32	32	32	***1x***

ResNet-18 Ablation Study on HIDA

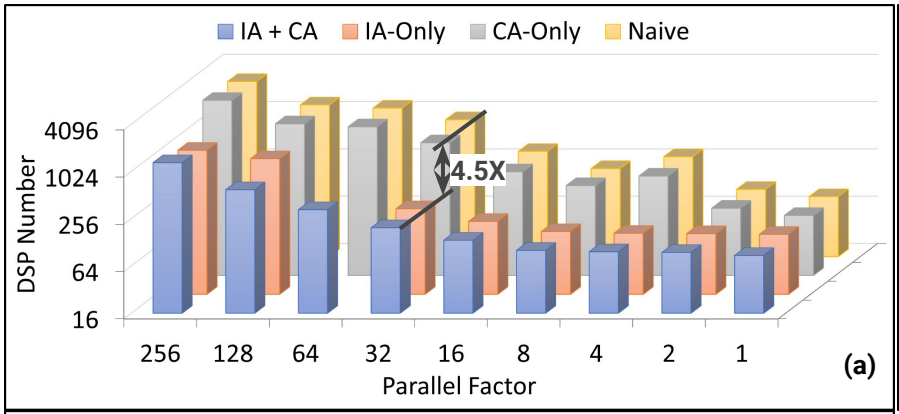


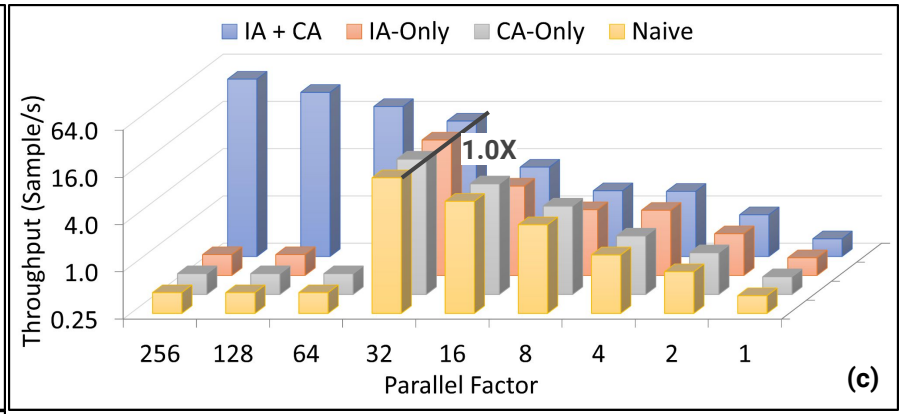


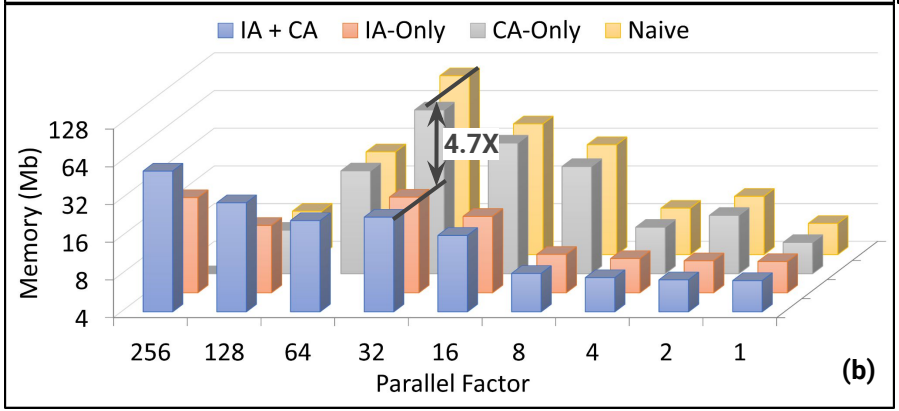


- IA+CA parallelization can determine whether the solution is scalable

ResNet-18 Ablation Study on HIDA (Cont'd)







- IA+CA parallelization can determine whether the solution is scalable
- IA+CA parallelization can significantly reduce resource utilization

HIDA (ScaleHLS 2.0) DSE Results of C/C++ Kernel

Kernel	HIDA Compile Time (s)	LUT Number	FF Number	DSP Number	Throughput (Samples/s)*			
					HIDA	ScaleHLS [68]	SOFF [37]	Vitis [34]
2mm	0.65	38.8k	27.4k	269	239.22	122.39 (1.95×)	30.67 (7.80×)	1.23 (194.88×)
3mm	0.79	38.7k	27.8k	243	175.43	92.33 (1.90×)	-	1.04 (167.99×)
atax	2.06	44.6k	34.6k	260	1,021.39	932.26 (1.10×)	2,173.17 (0.47×)	103.18 (9.90×)
bicg	0.72	16.0k	15.1k	61	2,869.69	2,869.61 (1.00×)	2,295.75 (1.25×)	104.19 (27.54×)
correlation	0.91	14.5k	12.3k	66	67.33	59.77 (1.13×)	3.96 (16.99×)	1.32 (50.97×)
gesummv	0.60	34.2k	22.8k	232	31,685.68	31,685.68 (1.00×)	3,466.70 (9.14×)	266.65 (118.83×)
jacobi-2d	1.98	91.4k	56.6k	352	257.27	128.63 (2.00×)	-	2.71 (94.95×)
mvt	0.42	23.8k	16.5k	162	9,979.04	4,989.02 (2.00×)	870.01 (11.47×)	62.13 (160.62×)
seidel-2d	3.59	5.5k	2.5k	4	0.14	0.14 (1.00×)	-	0.11 (1.28×)
symm	1.05	14.9k	9.5k	74	2.62	2.62 (1.00×)	-	2.02 (1.29×)
syr2k	0.69	14.3k	12.8k	78	27.68	27.67 (1.00×)	-	1.44 (19.23×)
Geo. Mean	**0.99**					**1.29×**	**4.49×**	**31.08×**

* Numbers in () show throughput improvements of HIDA over others.

HIDA (ScaleHLS 2.0) Results of DNN Models

Model	HIDA Compile Time (s)	LUT Number	DSP Number	Throughput (Samples/s)*			DSP Efficiency*		
				HIDA	DNNBuilder [75]	ScaleHLS [68]	HIDA	DNNBuilder [75]	ScaleHLS [68]
ResNet-18	83.1	142.1k	667	45.4	-	3.3 (13.88×)	73.8%	-	5.2% (14.24×)
MobileNet	110.8	132.9k	518	137.4	-	15.4 (8.90×)	75.5%	-	9.6% (7.88×)
ZFNet	116.2	103.8k	639	90.4	112.2 (0.81×)	-	82.8%	79.7% (1.04×)	-
VGG-16	199.9	266.2k	1118	48.3	27.7 (1.74×)	6.9 (6.99×)	102.1%	96.2% (1.06×)	18.6% (5.49×)
YOLO	188.2	202.8k	904	33.7	22.1 (1.52×)	-	94.3%	86.0% (1.10×)	-
MLP	40.9	21.0k	164	938.9	-	152.6 (6.15×)	90.0%	-	17.6% (5.10×)
Geo. Mean	**108.7**				**1.29×**	**8.54×**		**1.07×**	**7.49×**

* Numbers in () show throughput/DSP efficiency improvements of HIDA over others.

Improve HLS core algorithms

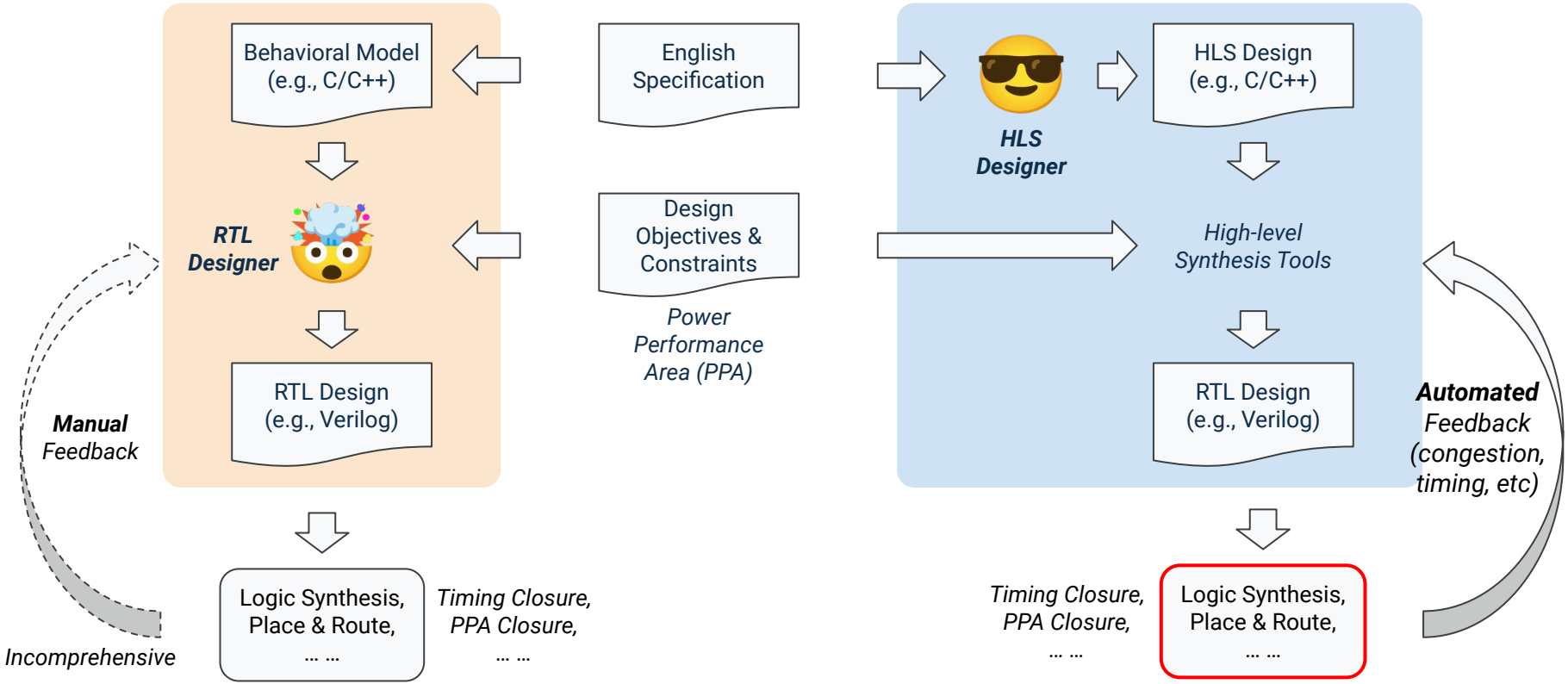
- ScaleHLS and HIDA explore the **functional** and **architectural** design space of HLS-based accelerators
- ScaleHLS and HIDA treat HLS tool, such as AMD Vitis HLS, as a black box

Any opportunity to improve the HLS core algorithms?

HLS Algorithm

ISDC: Feedback-guided Iterative SDC Scheduling for High-level Synthesis

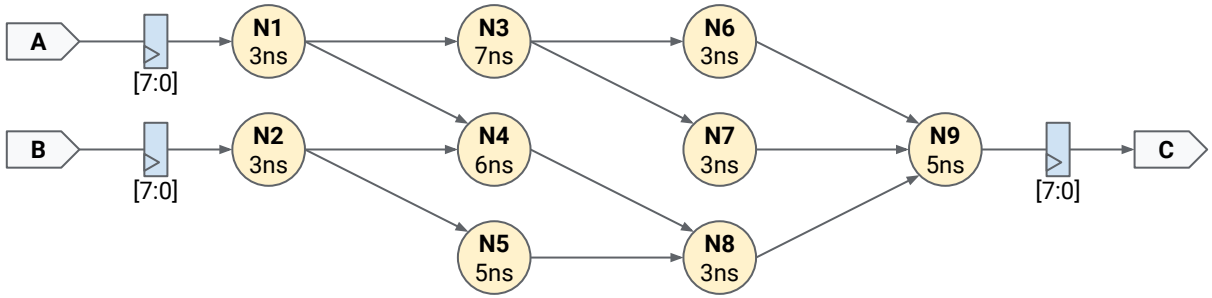
Automated feedback-directed optimization (FDO)



- PDK: Process design kit.

What is pipeline scheduling?

Unscheduled XLS Graph

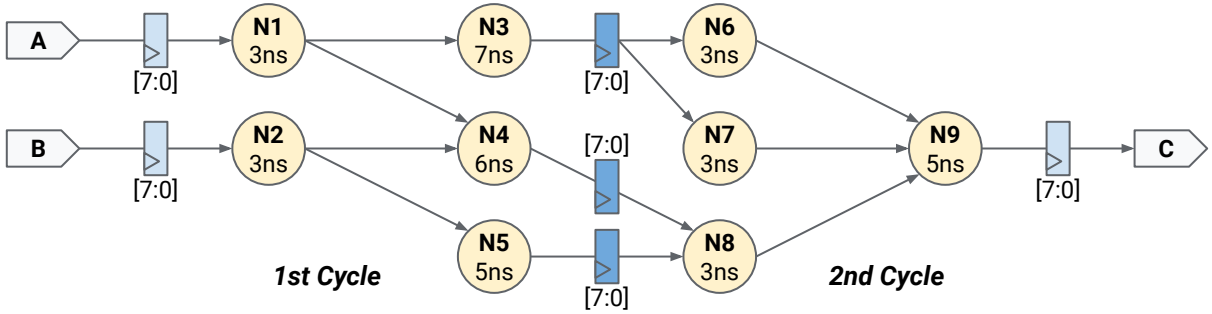


Pipeline Scheduling in XLS

Target Clock Period: 10ns

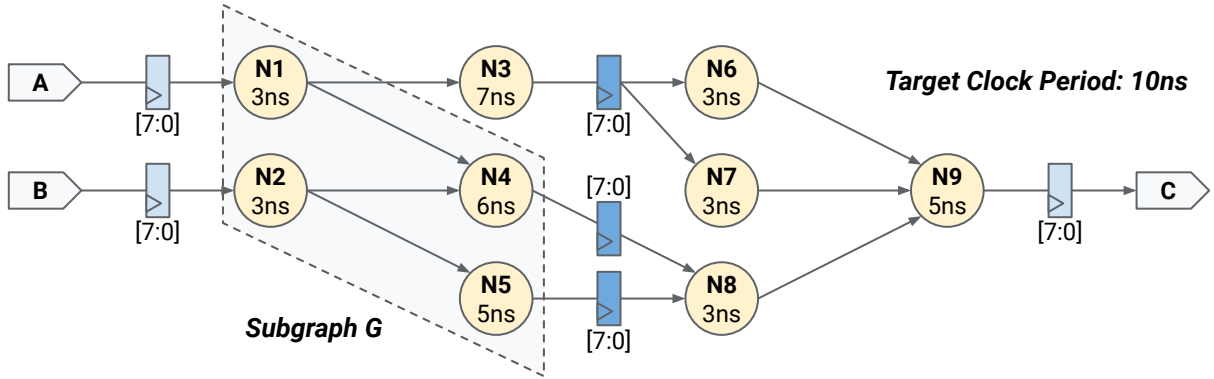
Objective: Minimize register number

Scheduled XLS Graph



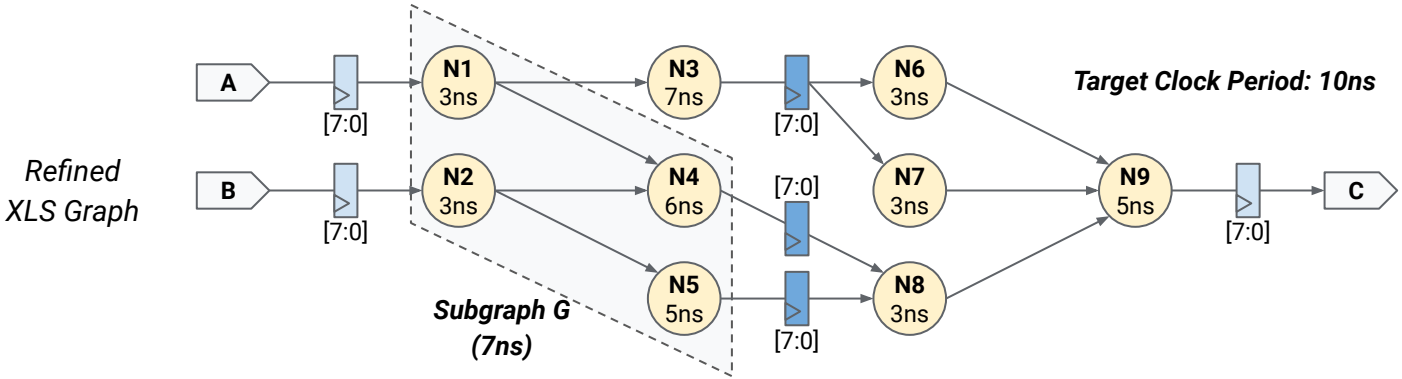
Intuition behind feedback-guided scheduling

Scheduled XLS Graph



Without feedback: $\boldsymbol{Delay_G}$ **= 9ns**

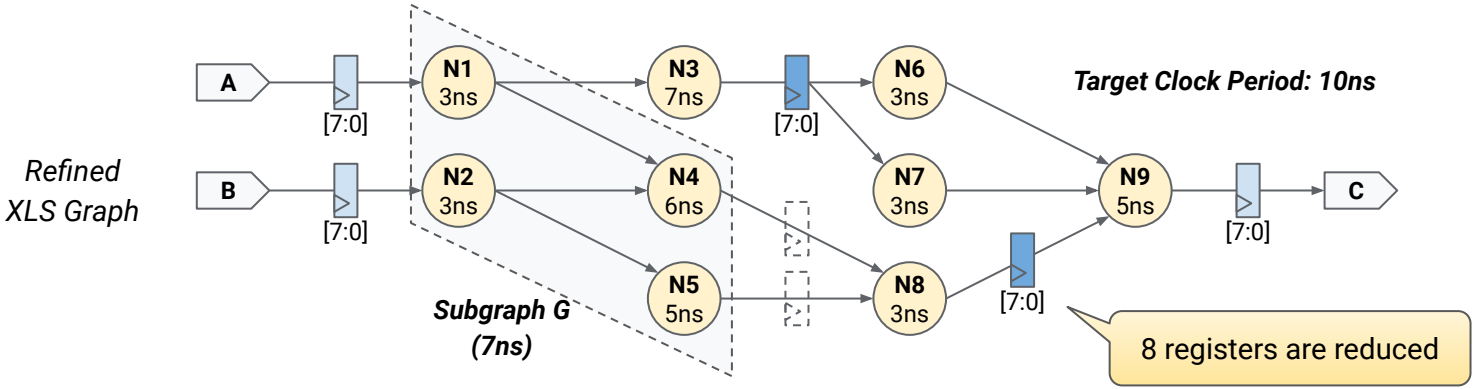
Intuition behind feedback-guided scheduling (Cont'd)



~~Without feedback: $Delay_G$ = 9ns~~

With feedback (e.g., OpenROAD): $Delay_G$ = 7ns

Intuition behind feedback-guided scheduling (Cont'd)



~~Without feedback: $Delay_G$ = 9ns~~

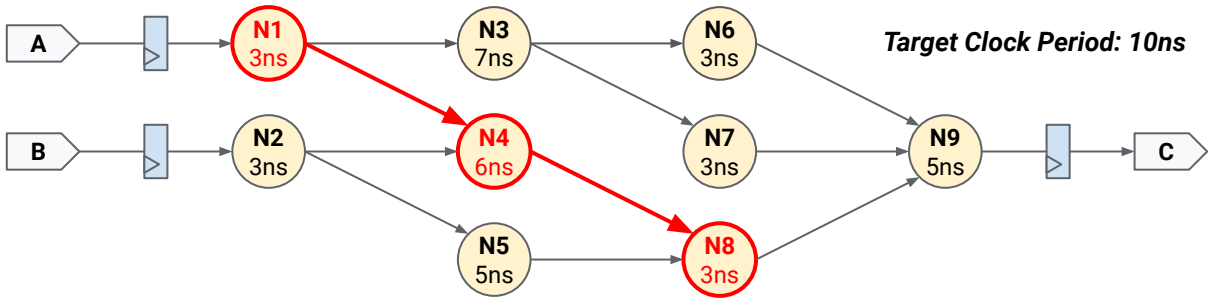
With feedback (e.g., OpenROAD): $Delay_G$ = 7ns

Q: Where does the difference come from?

A: Mainly comes from inter-node optimizations in downstream tools, such as logic synthesis.

Original pipeline scheduling in XLS

Unscheduled XLS Graph



SDC (System of Difference Constraints) Scheduling [1]

Variables:

`cycle_1`
`cycle_2`
… …
`cycle_9`

Timing Constraints:

Delay_1_8 = 12ns > 10ns ⇒ `cycle_8 - cycle_1 >= 1`
Delay_2_8 = 12ns > 10ns ⇒ `cycle_8 - cycle_2 >= 1`
… …
(for each path longer than 10ns)

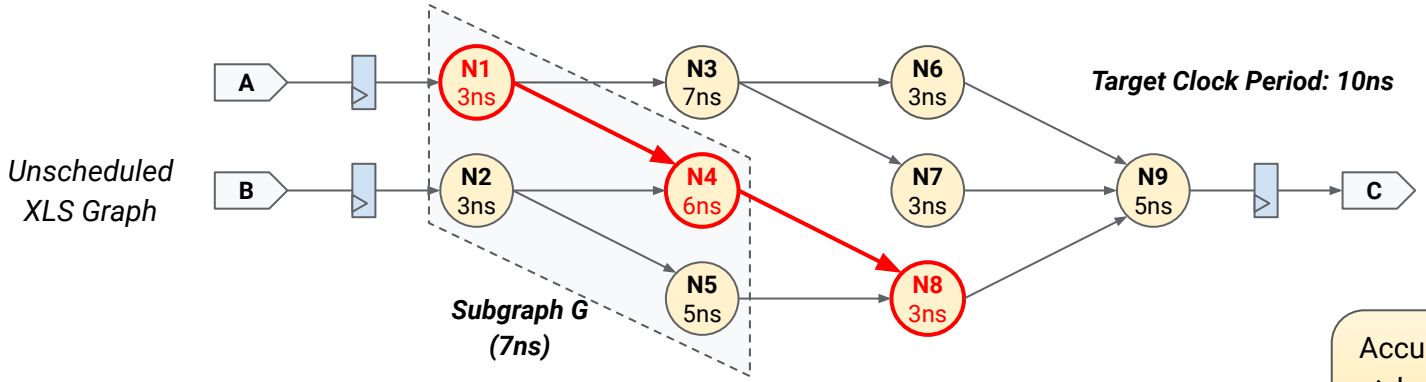
Def-use Constraints, Resource Constraints, etc.

Minimize Register Number

Linear Programming Problem

1. An efficient and versatile scheduling algorithm based on SDC formulation (paper)

SDC reformulation with feedbacks



SDC (System of Difference Constraints) Scheduling [1]

Accurate feedbacks
⇒ Less constraints
⇒ Larger search space
⇒ Better results

Variables:

```
cycle_1
cycle_2
… …
cycle_9
```

Timing Constraints:

Delay_1_8 <= 7ns + 3ns ⇒ ~~cycle_8 - cycle_1 >= 1~~

Delay_2_8 <= 7ns + 3ns ⇒ ~~cycle_8 - cycle_2 >= 1~~

… …

(for each path longer than 10ns)

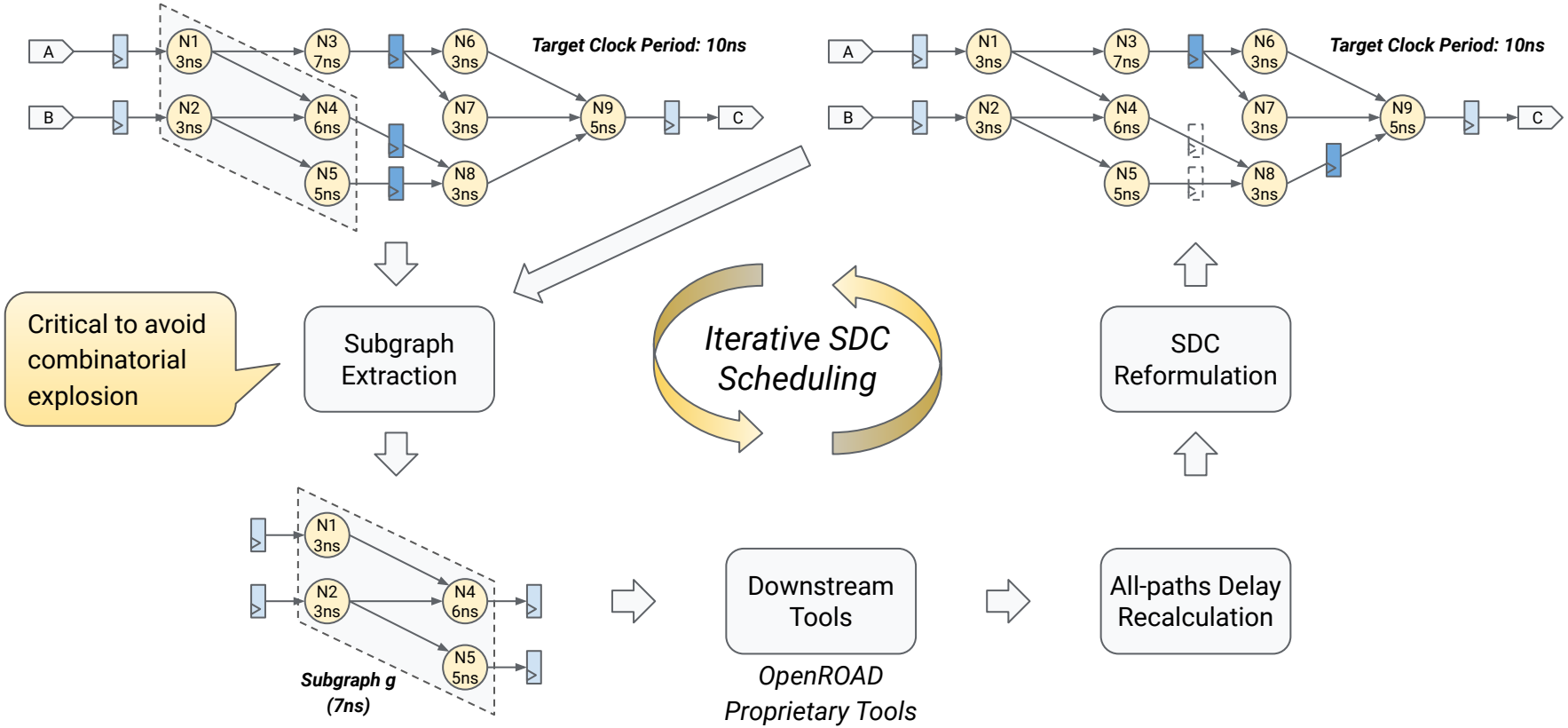
Def-use Constraints, Resource Constraints, etc.

Minimize Register Number

Linear Programming Problem

1. An efficient and versatile scheduling algorithm based on SDC formulation (paper)

Automated iterative SDC scheduling

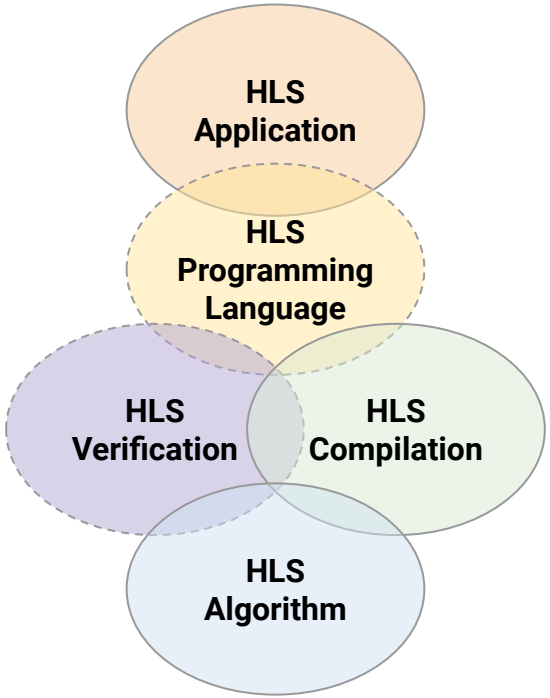


Full results

Benchmarks	Clock Period (ps)	Original XLS (SDC)					Ours (Iterative SDC)				
		STA Delay (ps)	Stage #	Register #	Schedule Runtime (s)	Minimum Iteration #	STA Delay (ps)	Stage #	Register #	Schedule Runtime (s)	Converge Iteration #
Design 1	2500	1338.35	2	99	0.14	1	1770.28	1	50	6.73	3
Design 2	5000	4056.07	2	109	0.11	1	4056.07	2	109	0.10	1
Design 3	2500	1633.77	2	192	0.08	1	2000.67	1	96	2.98	2
Design 4	5000	3559.35	3	138	0.13	1	4227.13	2	101	23.90	6
Design 5	2500	1981.34	3	71	0.12	1	2063.82	3	70	7.56	4
Design 6	5000	3549.27	3	134	0.11	1	3850.27	2	102	10.64	3
Design 7	5000	3859.1	3	162	0.12	1	3837.34	2	108	19.26	4
Design 8	2500	755.65	3	75	0.11	1	813.51	1	38	4.76	3
Design 9	5000	3764.42	5	298	0.15	1	3480.8	4	234	21.28	4
Design 10	5000	3668.75	6	480	0.44	1	3969.27	3	209	94.30	14
Design 11	5000	3165.32	8	1214	1.62	1	4048.76	5	729	101.61	13
Design 12	2500	2279.86	10	819	0.43	1	2463.29	6	474	27.62	9
Design 13	5000	3797.98	10	1055	1.79	3	4855.09	8	797	118.47	14
Design 14	2500	2473.14	12	1756	24.28	4	2333.69	12	1732	316.62	11
Design 15	2500	2128.78	26	3095	13.73	1	2439.58	25	2976	167.04	10
Design 16	2500	2267.34	112	85545	284.47	1	2425.89	97	73990	3280.88	11
Design 17	5000	4557.25	121	30569	240.90	1	4763.03	114	29242	3441.08	13
Geo. Mean	-	-	**6.93**	**569.86**	**0.84**	-	-	**4.85**	**407.19**	**34.46**	-
Ratio	-	-	**100.0%**	**100.0%**	**100.0%**	-	-	**70.0%**	**71.5%**	**4080.5%**	-

Future Research Plan

Future Research Plan Overview



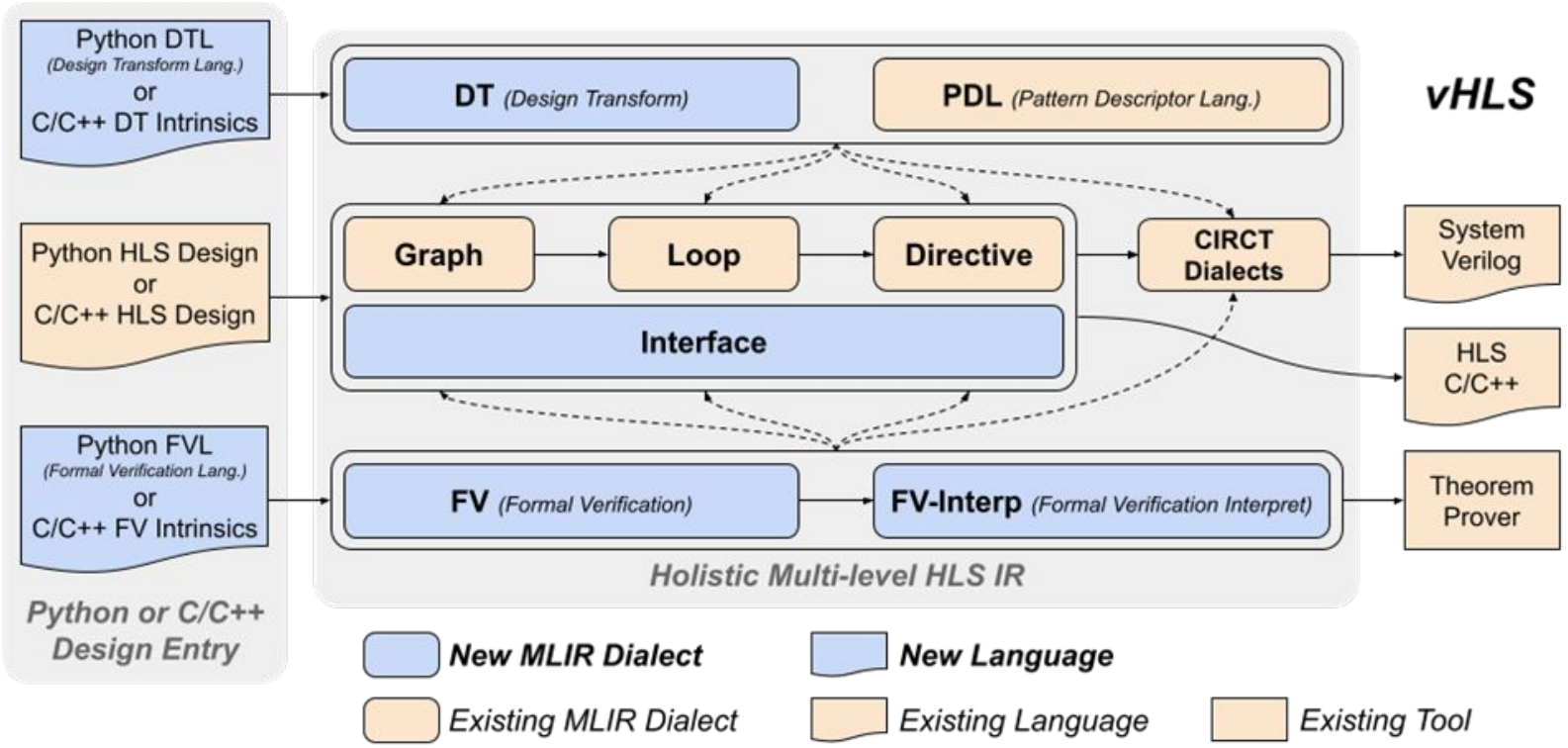
- HLS-based Fully Homomorphic Encryption (FHE) acceleration

- Pythonic HLS design and scheduling language
- Equivalence Graph (E-Graph)-based automatic HLS scheduling

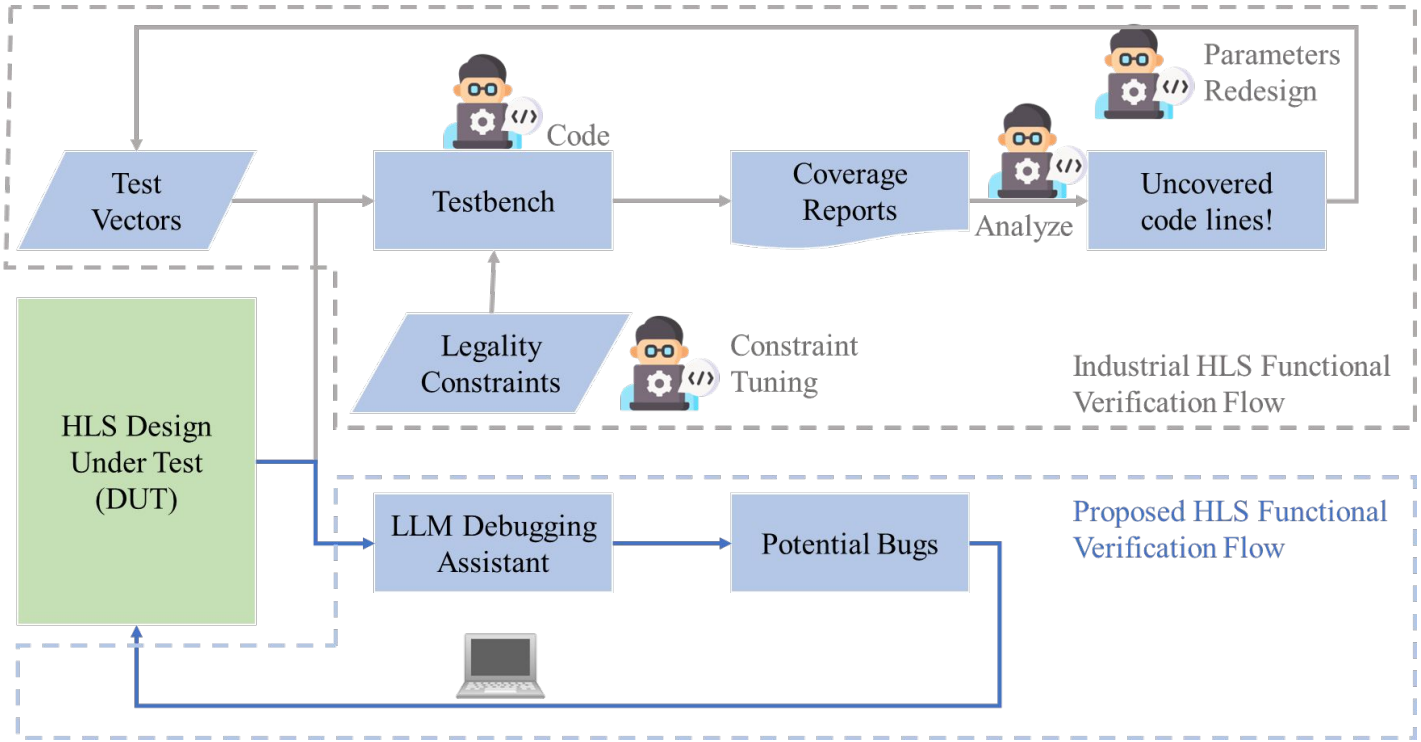
- Correct-by-construction HLS flow (WIP)
- Large Language Model (LLM)-driven HLS bug detection (WIP)
- Automatic HLS IP integration and optimization (WIP)

- Simultaneous HLS scheduling and datapath optimization (WIP)

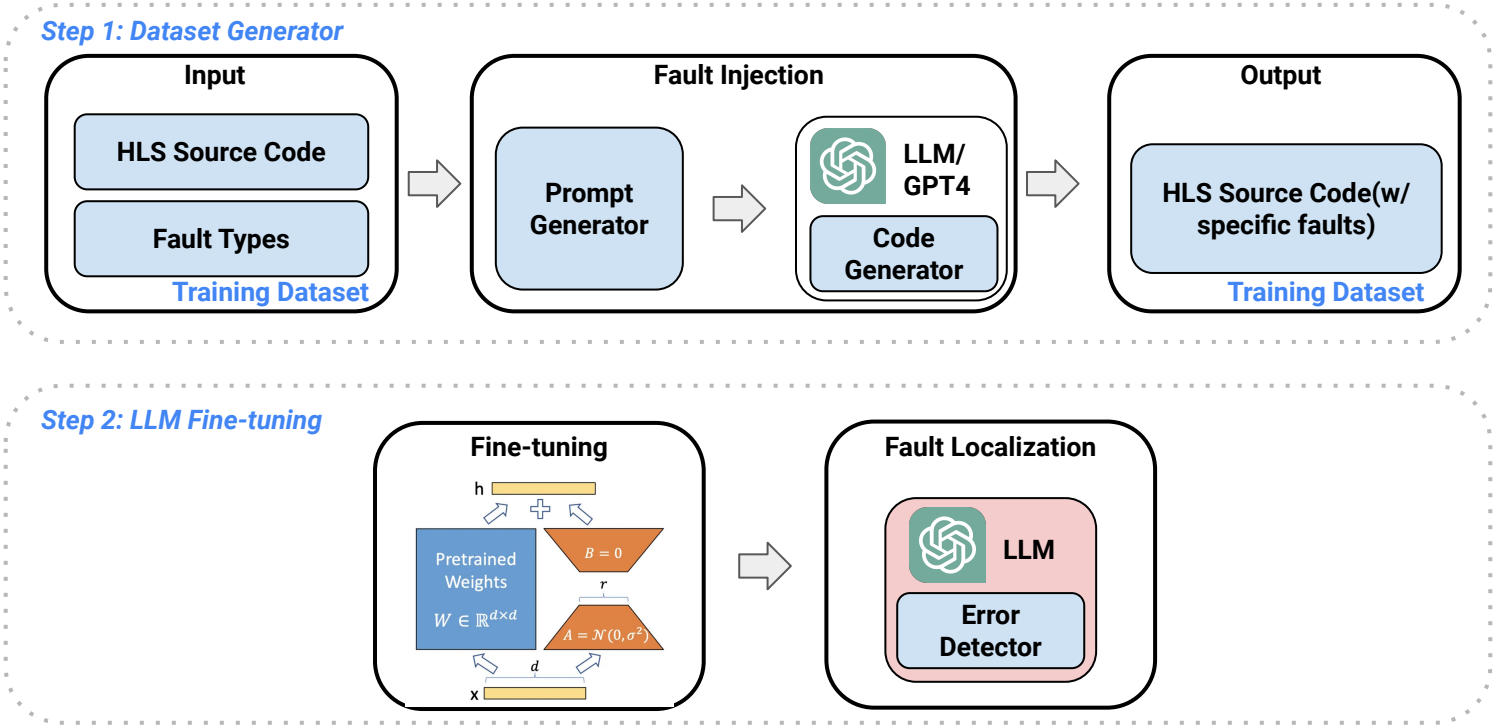
Correct-by-Construction HLS



LLM-driven HLS Bug Detection



LLM-driven HLS Bug Detection (Cont.)



UFlow: Automated HLS IP Integration

```python
@uflow.register_ip("gemv")
class Gemv(uflow.Ip):
    def __init__(self):
        super().__init__()
        self.data_type = uflow.Float()
        self.index_type = uflow.UInt(32)

        self.par_m = uflow.CompileParam(self.index_type, [2, 4, 8])
        self.dim_m = uflow.DynamicParam(self.index_type, (16, 1024))
        self.dim_n = uflow.DynamicParam(self.index_type, (16, 1024))

        self.mat_a = uflow.InputPort(uflow.DynamicTensor(
            self.data_type, [self.dim_m, self.dim_n],
            lambda m, n: [m / self.par_m, n, m % self.par_m]))
        self.vec_b = uflow.InputPort(
            uflow.DynamicTensor(self.data_type, [self.dim_n], lambda n: [n]), size=[1])
        self.vec_c = uflow.OutputPort(
            uflow.DynamicTensor(self.data_type, [self.dim_m],
            lambda m: [m / self.par_m, m % self.par_m]), size=[1])

    def semantics(self):
        for m in self.dim_m:
            for n in self.dim_n:
                self.vec_c[m] += self.mat_a[m, n] * self.vec_b[n]
```

Register a IP “gemv”

IP data types

IP compile-time params

IP runtime params

IP input port “mat_a”

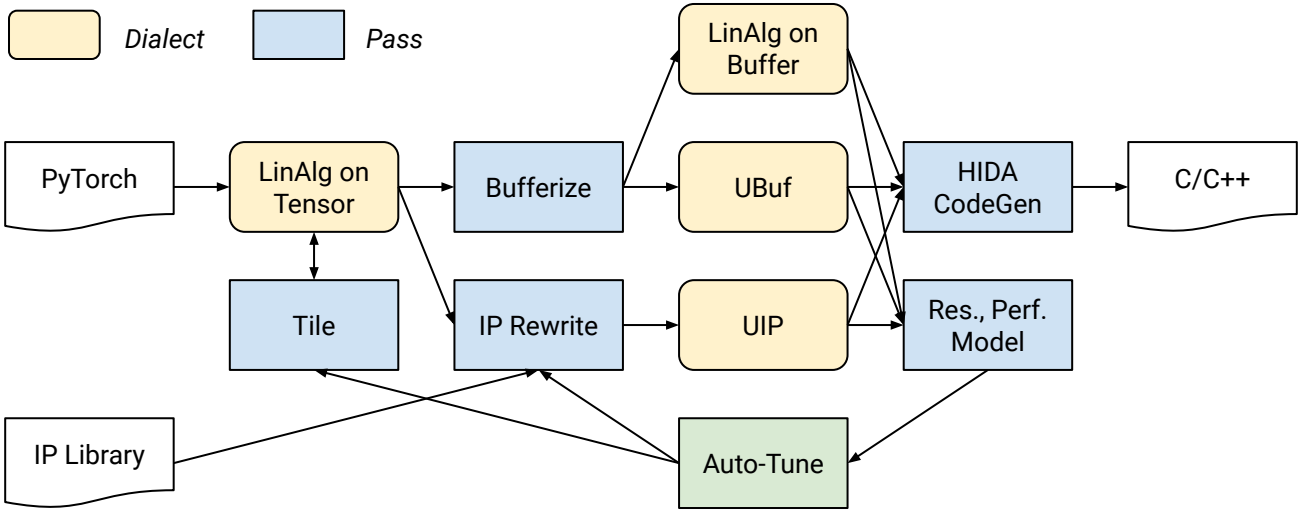
Shape and layout of “mat_a”

IP output port “vec_c”

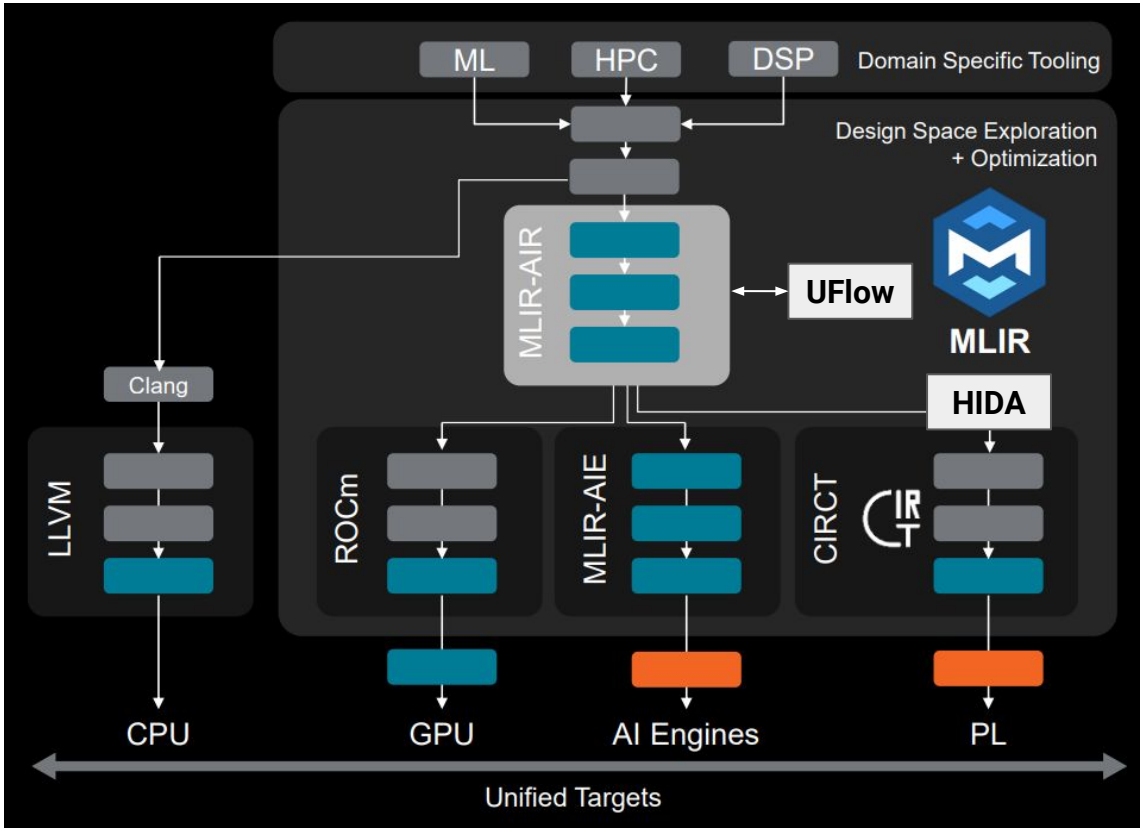
Size 1 indicates “vec_c” is FIFO

IP semantics for pattern match

UFlow: Automated HLS IP Integration (Cont.)



PyTorch Compiler for Versal



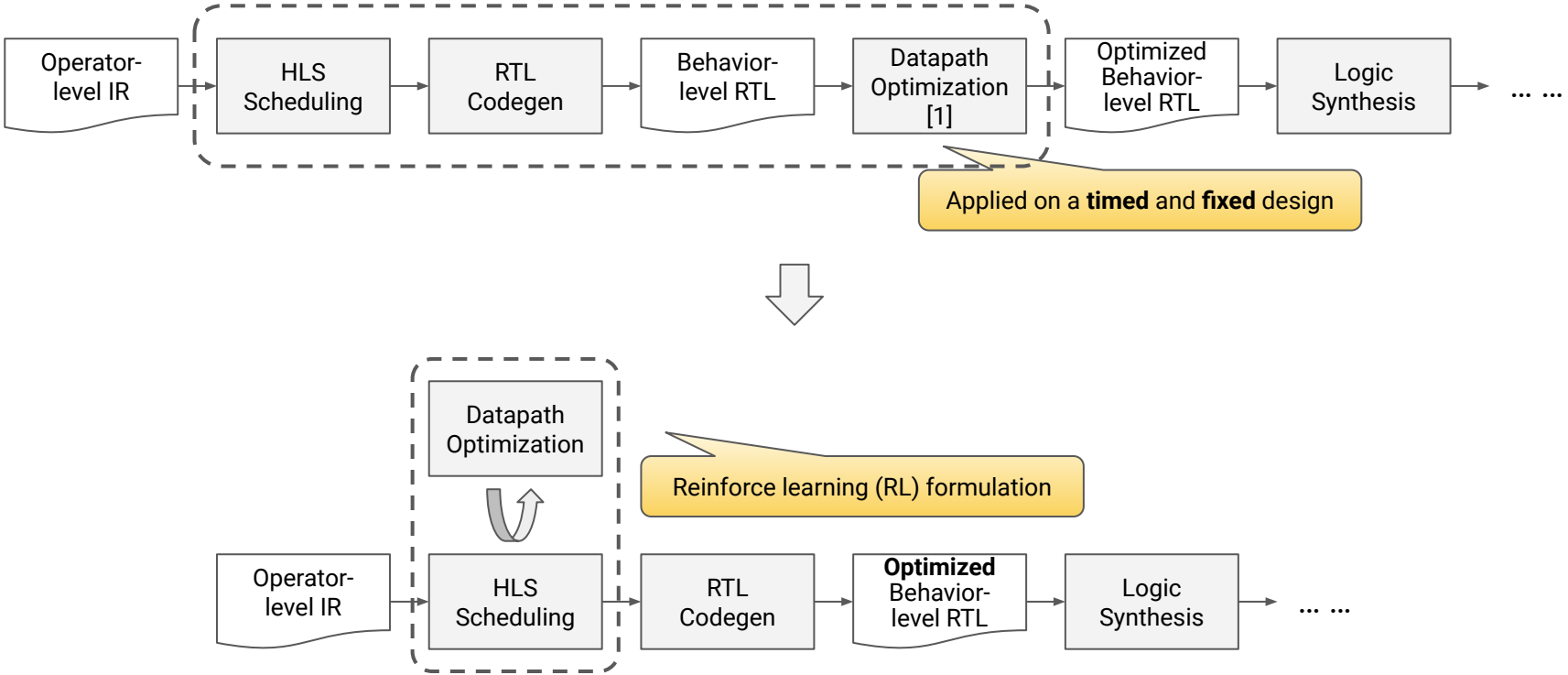
HIDA Integration

- Integrated before CIRCT
- FPGA logic optimization
- FPGA-AIE communication optimization

UFlow Integration

- Take HLS and AIE IP library registration as input
- Automatic HLS and AIE IPs integration
- Inter-IP communication optimization

Simultaneous HLS scheduling and datapath optimization



1. Automatic Datapath Optimization using E-Graphs. Coward et al. DAC'23.

Thanks for your time!

Hanchen Ye

Oct. 18, 2023