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

Hanchen Ye[1], Cong Hao[2], Jianyi Cheng[3], Hyunmin Jeong[1], Jack Huang[1], Stephen Neuendorffer[4], Deming Chen[1]

[1]University of Illinois at Urbana-Champaign, [2]Georgia Institute of Technology, [3]Imperial College London, [4]Xilinx Inc.

UNIVERSITY OF ILLINOIS URBANA-CHAMPAIGN

Imperial College London

Background: MLIR

MLIR: Compiler Infra at the End of Moore's Law

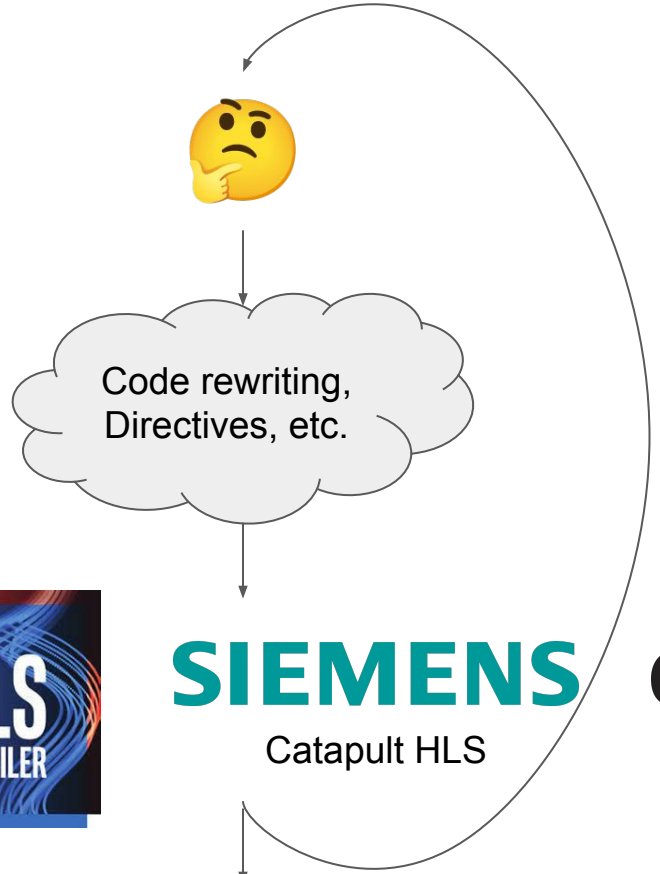
- **M**ulti-**L**evel **I**ntermediate **R**epresentation
- Joined LLVM, follows open library-based philosophy
- 🧩 ***Modular***, extensible, general to many domains
  - Being used for CPU, GPU, TPU, FPGA, HW, quantum, ...
- Easy to learn, great for research
- MLIR + LLVM IR + RISC-V CodeGen = 💝💝

https://mlir.llvm.org

See more (e.g.):
2020 CGO Keynote Talk Slides
2021 CGO Paper

Source: The Golden Age of Compiler Design in an Era of HW/SW Co-design by Chris Lattner

Motivations



XILINX VITIS™

HLS COMPILER

SIEMENS
Catapult HLS

cadence®
Stratus HLS

aws

Motivations (Cont.)

ScaleHLS

Automated
design space
exploration

XILINX VITIS™

HLS COMPILER

SIEMENS
Catapult HLS

cadence®
Stratus HLS

aws

Motivations (Cont.) - 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.

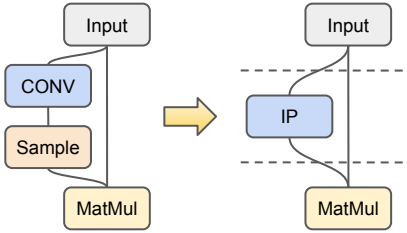
```
for (int k = 0; k < 32; k++) {
  for (int i = 0; i < 32; i++) {
    for (int j = 0; j < 32; j++) {
#pragma HLS pipeline
      if (k == 0)
        C[i][j] *= beta;
      C[i][j] += alpha * A[i][k] * B[k][j];
} } }
```

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:

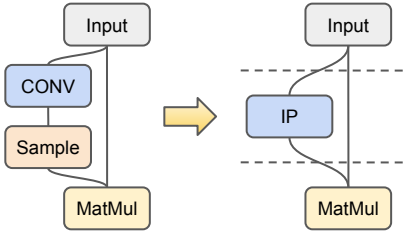
- Low-productive and error-proning
- Hard to enable automated design space exploration (DSE)
- NOT scalable!

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

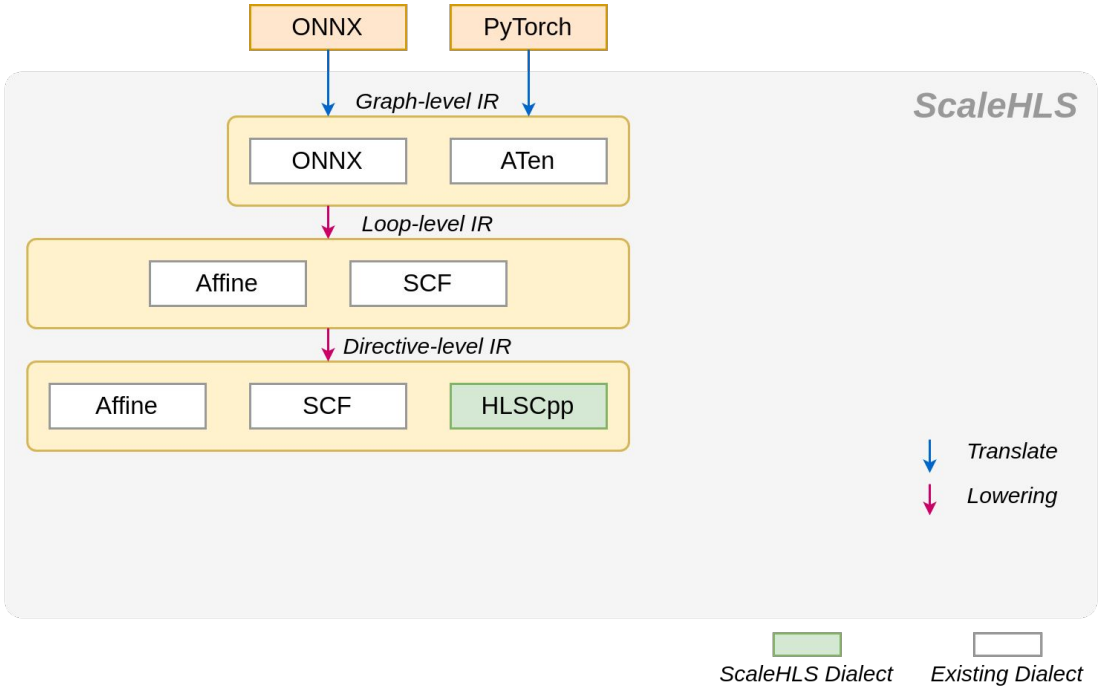
Directive Optimizations
Loop pipeline, unroll
Function pipeline, inline
Array partition, etc.

Manual Code Rewriting

Generate RTL with XILINX VITIS and etc.

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

ScaleHLS Framework



Represent It!

Graph-level IR: ONNX [1] and ATen [2] dialect.

Loop-level IR: Affine [3] and SCF (structured control flow) [3] dialect. Can leverage the transformation and analysis libraries applicable in MLIR.

Directive-level IR: HLSCpp, Affine, and SCF dialect.

Graph-level IR

```
%O = "onnx.Gemm"(%I, %W, %B) {...} :
(tensor<1x512xf32>, tensor<10x512xf32>, tensor<10xf32>)
-> tensor<1x10xf32>
```

Loop-level IR

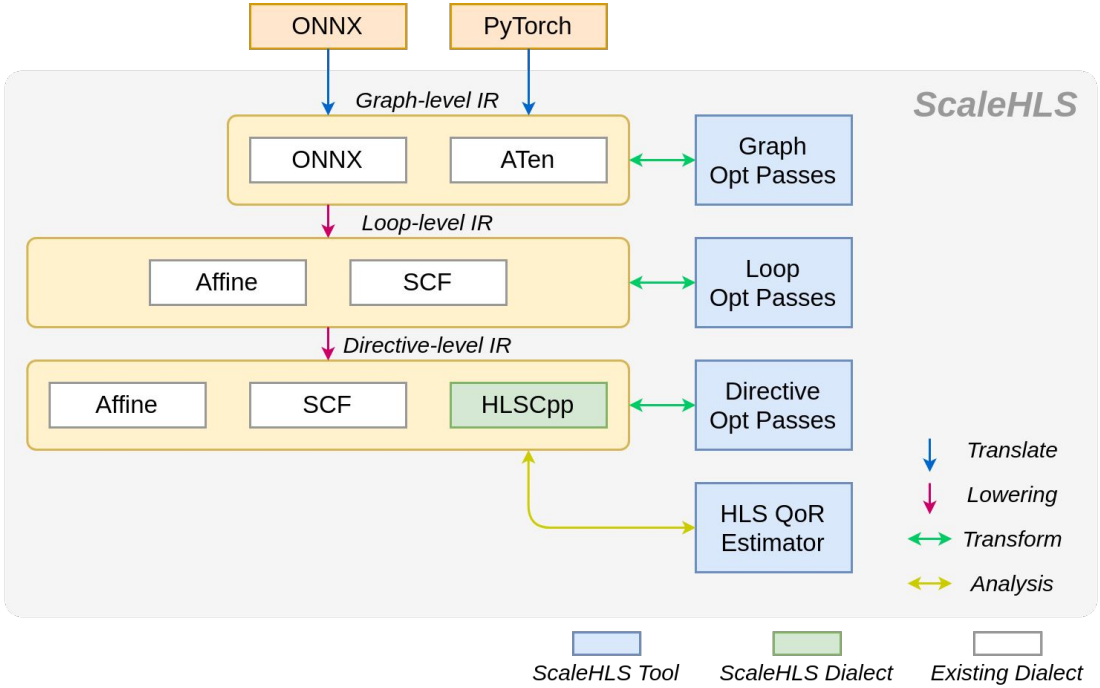
```
affine.for %i = 0 to 1 {
  affine.for %j = 0 to 10 {
    ... ...
    affine.for %k = 0 to 512 {
      %1 = affine.load %I[%i, %k] : memref<1x512xf32>
      %2 = affine.load %W[%j, %k] : memref<10x512xf32>
      %3 = affine.load %O[%i, %j] : memref<1x10xf32>
      %4 = mulf %2, %3 : f32
      %5 = addf %4, %5 : f32
      affine.store %5, %O[%i, %j] : memref<1x10xf32>
} } }
```

Directive-level IR

```
affine.for %i = 0 to 1 {
  affine.for %j = 0 to 10 {
    ... ...
    affine.for %k = 0 to 512 {
      ... ...
    } {loop_directive = #hlscpp.ld<pipeline=1, ...>}
  } {loop_directive = #hlscpp.ld<pipeline=0, ...>}
} {loop_directive = #hlscpp.ld<pipeline=0, ...>}
```


[1] ONNX-MLIR: Compiling ONNX Neural Network Models Using MLIR. https://github.com/onnx/onnx-mlir
[2] NPComp: MLIR based compiler toolkit for numerical python programs. https://github.com/llvm/mlir-npcomp
[3] MLIR: Multi-Level Intermediate Representation. https://github.com/llvm/llvm-project/tree/main/mlir
[4] Vitis HLS Front-end: https://github.com/Xilinx/HLS

ScaleHLS Framework (Cont.)



Represent It!

Graph-level IR: ONNX [1] and ATen [2] dialect.

Loop-level IR: Affine [3] and SCF (structured control flow) [3] dialect. Can leverage the transformation and analysis libraries applicable in MLIR.

Directive-level IR: HLSCpp, Affine, and SCF dialect.

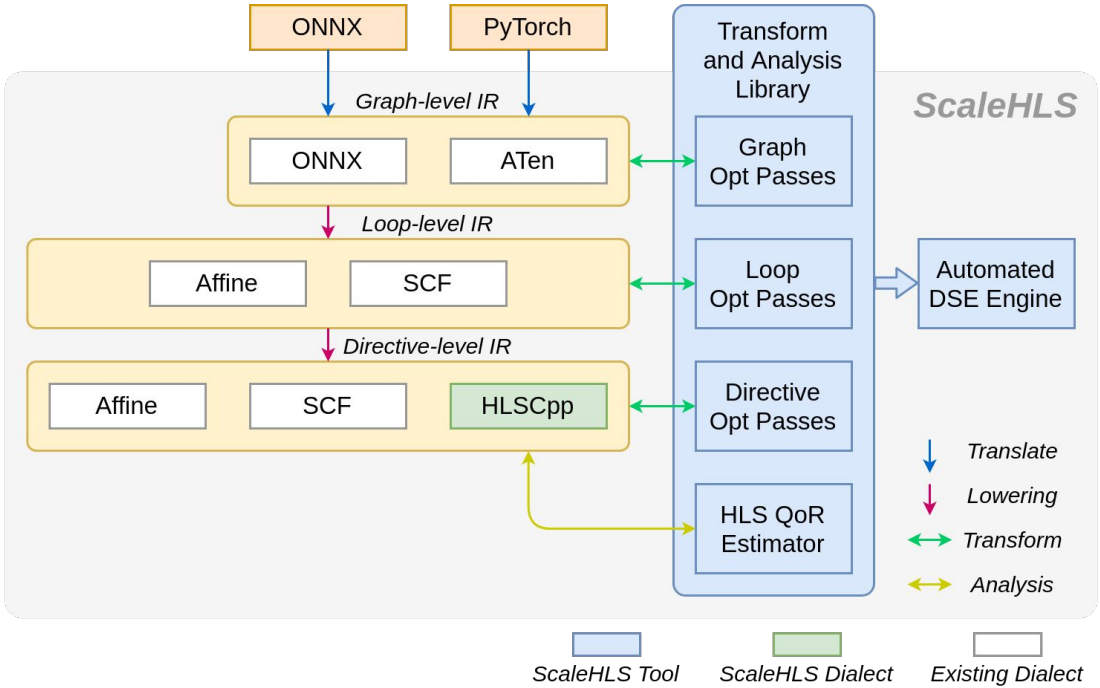
Optimize It!

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.

[1] ONNX-MLIR: Compiling ONNX Neural Network Models Using MLIR. https://github.com/onnx/onnx-mlir
[2] NPComp: MLIR based compiler toolkit for numerical python programs. https://github.com/llvm/mlir-npcomp
[3] MLIR: Multi-Level Intermediate Representation. https://github.com/llvm/llvm-project/tree/main/mlir
[4] Vitis HLS Front-end: https://github.com/Xilinx/HLS

ScaleHLS Framework (Cont.)



Represent It!

Graph-level IR: ONNX [1] and ATen [2] dialect.

Loop-level IR: Affine [3] and SCF (structured control flow) [3] dialect. Can leverage the transformation and analysis libraries applicable in MLIR.

Directive-level IR: HLSCpp, Affine, and SCF dialect.

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.

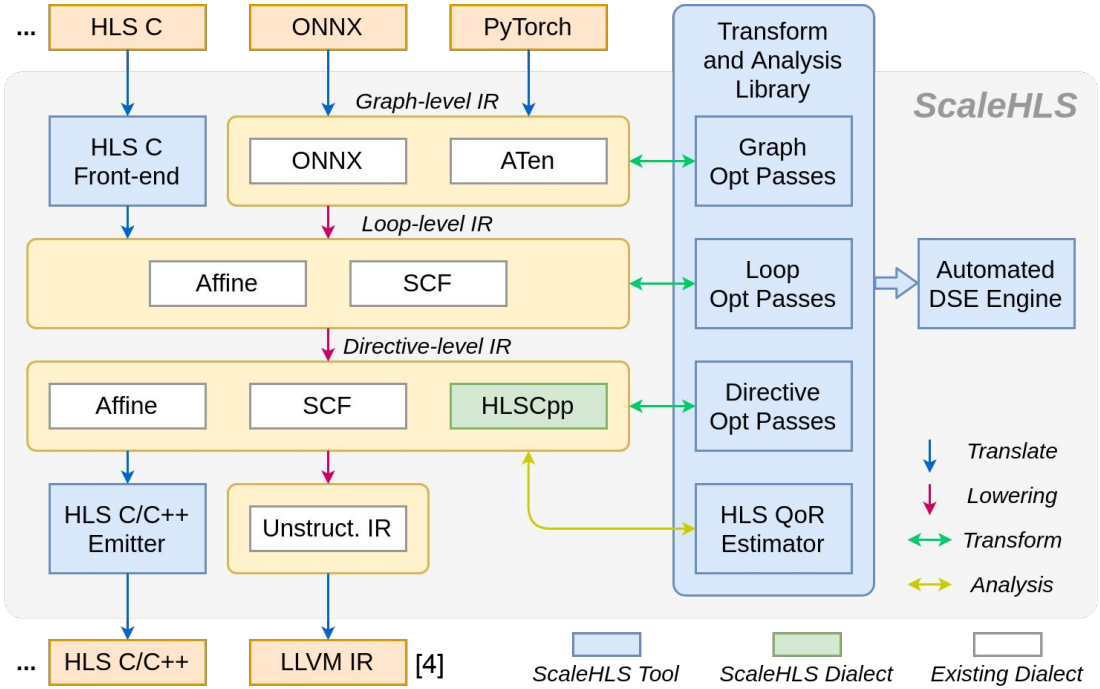
Explore It!

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.

[1] ONNX-MLIR: Compiling ONNX Neural Network Models Using MLIR. https://github.com/onnx/onnx-mlir
[2] NPComp: MLIR based compiler toolkit for numerical python programs. https://github.com/llvm/mlir-npcomp
[3] MLIR: Multi-Level Intermediate Representation. https://github.com/llvm/llvm-project/tree/main/mlir
[4] Vitis HLS Front-end: https://github.com/Xilinx/HLS

ScaleHLS Framework (Cont.)



[1] ONNX-MLIR: Compiling ONNX Neural Network Models Using MLIR. https://github.com/onnx/onnx-mlir
[2] NPComp: MLIR based compiler toolkit for numerical python programs. https://github.com/llvm/mlir-npcomp
[3] MLIR: Multi-Level Intermediate Representation. https://github.com/llvm/llvm-project/tree/main/mlir
[4] Vitis HLS Front-end: https://github.com/Xilinx/HLS

Represent It!

Graph-level IR: ONNX [1] and ATen [2] dialect.

Loop-level IR: Affine [3] and SCF (structured control flow) [3] dialect. Can leverage the transformation and analysis libraries applicable in MLIR.

Directive-level IR: HLSCpp, Affine, and SCF dialect.

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. 🚀

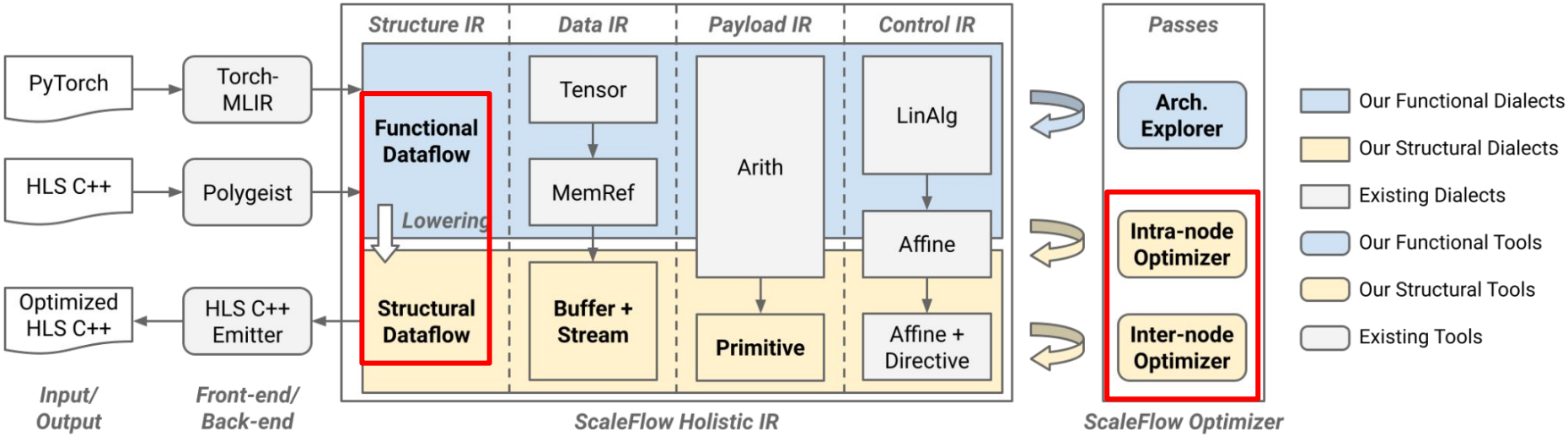
Automated DSE Engine: Find the Pareto-frontier of the throughput-area trade-off design space.

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.

ScaleHLS Framework v2 - ScaleFlow



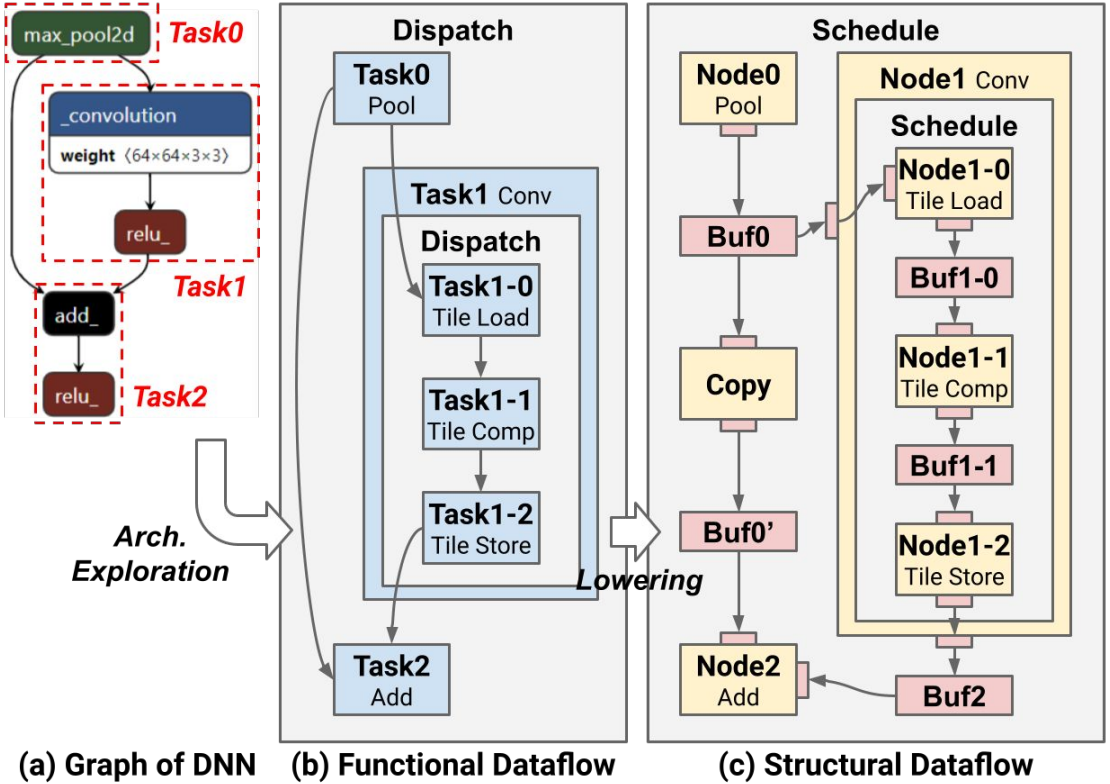
ScaleFlow Holistic IR

Two-level dataflow representation - *Functional* and *Structural* dataflow, for high-level task partition and low-level u-arch optimization, respectively.

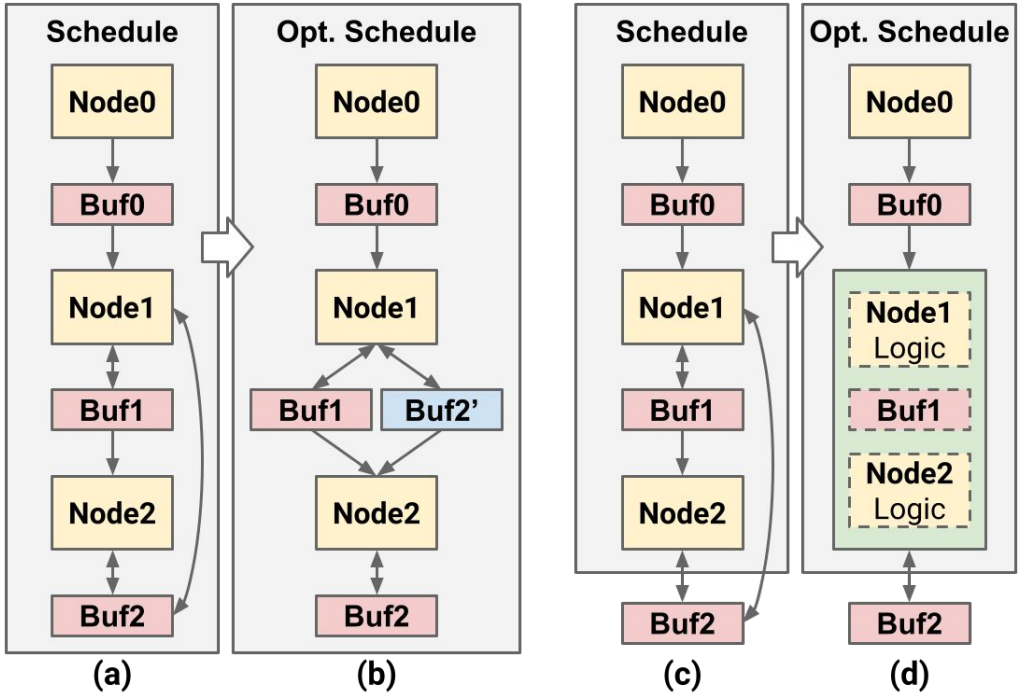
ScaleFlow Optimizer

An architecture explorer at the *Functional* level to explore the dataflow construction. A decoupled intra-node optimizer and inter-node optimizer at the Structural level to optimize low-level u-arch.

ScaleHLS Framework v2 - ScaleFlow (Cont.)

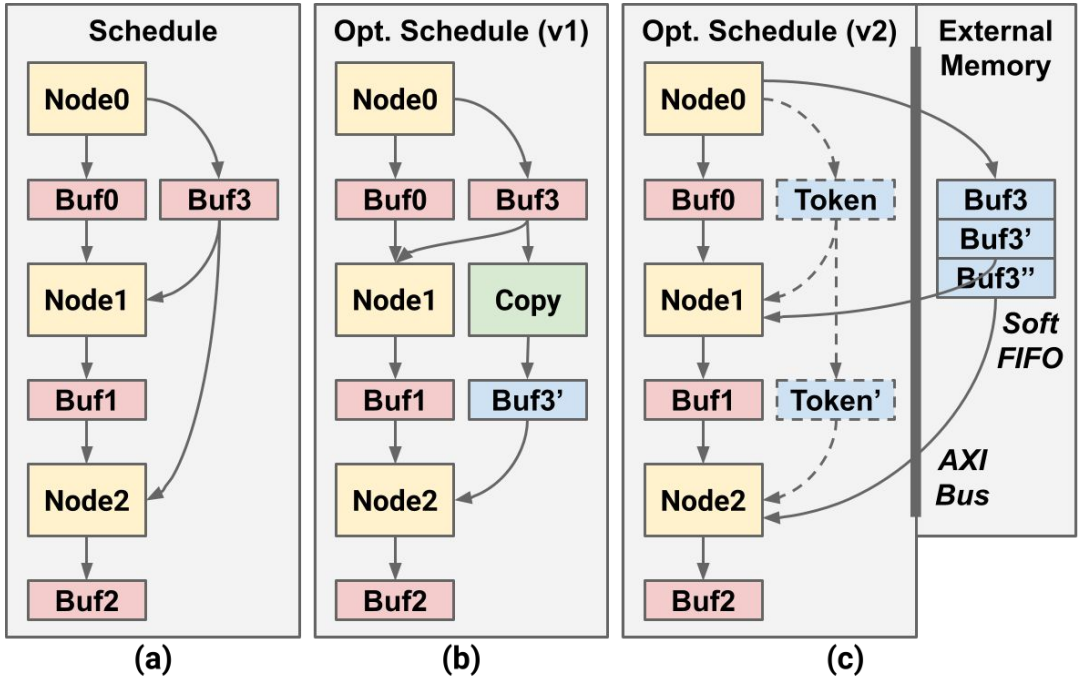


ScaleHLS Inter-Node Transformations



Eliminate Multi-Producer

ScaleHLS Inter-Node Transformations (Cont.)



Balance Datapaths

ScaleHLS Inter-Node Transformations (Cont.)

Listing 3: A dataflow example in C++.

```
1  float A[32][16];
2  NODE0_LOOP_I: for (int i = 0; i < 32; i++)
3    NODE0_LOOP_K: for (int k = 0; k < 16; k++)
4      A[i][k] = ...;
5
6  float B[16][16];
7  NODE1_LOOP_K: for (int k = 0; k < 16; k++)
8    NODE1_LOOP_J: for (int j = 0; j < 16; j++)
9      B[k][j] = ...;
10
11 float C[16][16];
12 NODE2_LOOP_I: for (int i = 0; i < 16; i++)
13   NODE2_LOOP_J: for (int j = 0; j < 16; j++)
14     NODE2_LOOP_K: for (int k = 0; k < 16; k++)
15       C[i][j] = A[i * 2][k] * B[k][j];
```

Table 2: Node connections appear in Listing 3.

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]

Table 3: Node parallelization results of Listing 3 assuming a maximum parallel factor of 32. IA and CA represent intensity-aware and connection-aware.

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

Dataflow-aware Parallelization

- **Intensity-aware:** The latencies of dataflow nodes are balanced when the parallel factor is proportional to the computation intensity
- **Connection-aware:** The unroll factors should be compatible with connected dataflow nodes to access the shared buffer (either external or on-chip) efficiently

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

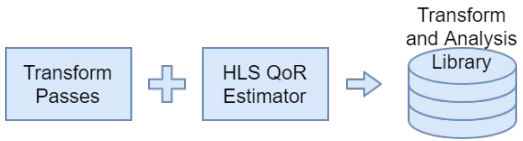
Simplify if ops;
Store ops forward;
Simplify memref ops

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

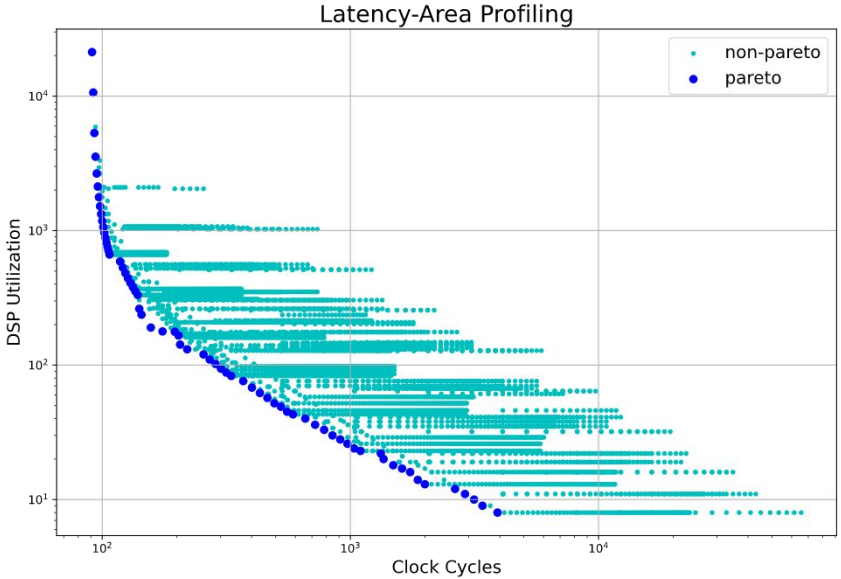
Remove variable loop bound

Loop pipeline

Simplify if ops;
Store ops forward;
Simplify memref ops

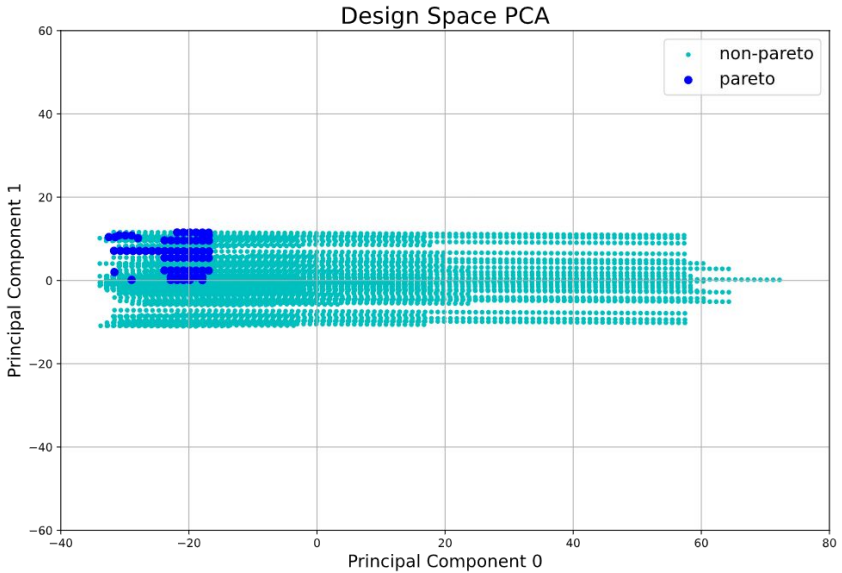
Optimized C emitted by the C/C++ emitter

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

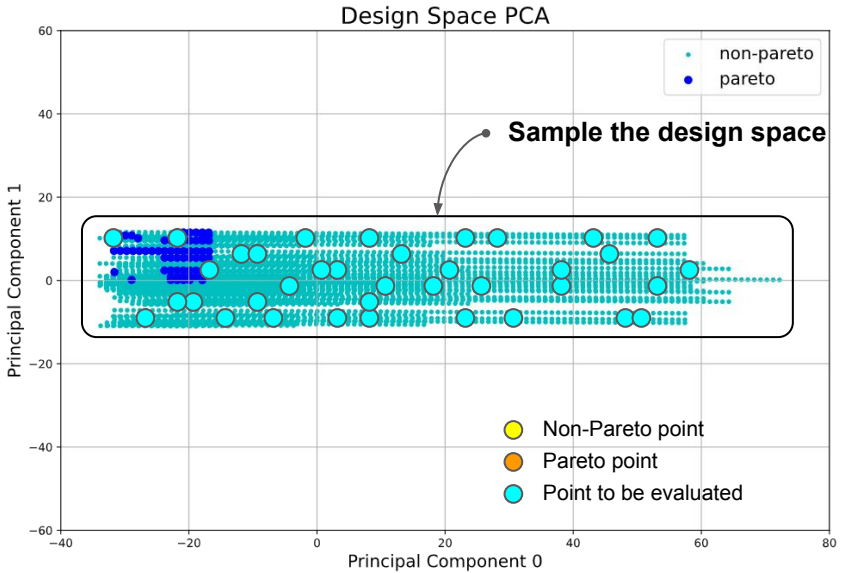


- Each parameter of a pass becomes one dimension, the original 4-dimensional design space is reduced to two dimensions through PCA
- Pareto points are located at a corner of the design space, the variance of Pareto points is much smaller than the overall variance

Design Space Exploration (Cont.)

DSE algorithm:

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

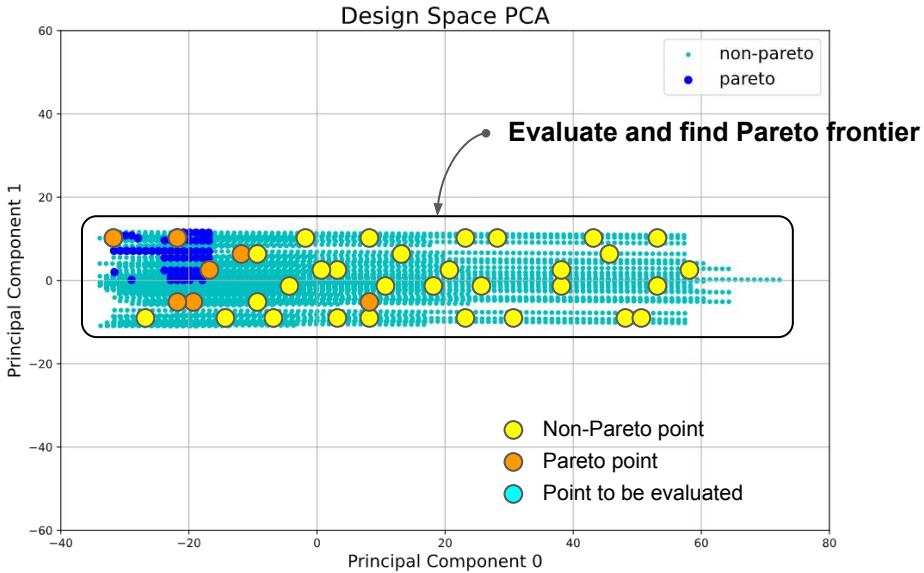


- Each parameter of a pass becomes one dimension, the original 4-dimensional design space is reduced to two dimensions through PCA
- Pareto points are located at a corner of the design space, the variance of Pareto points is much smaller than the overall variance

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

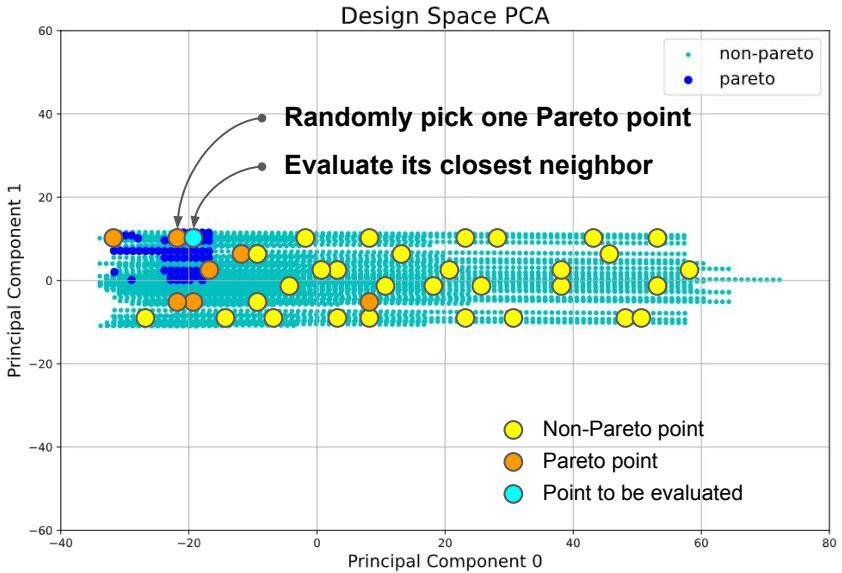


- Each parameter of a pass becomes one dimension, the original 4-dimensional design space is reduced to two dimensions through PCA
- Pareto points are located at a corner of the design space, the variance of Pareto points is much smaller than the overall variance

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

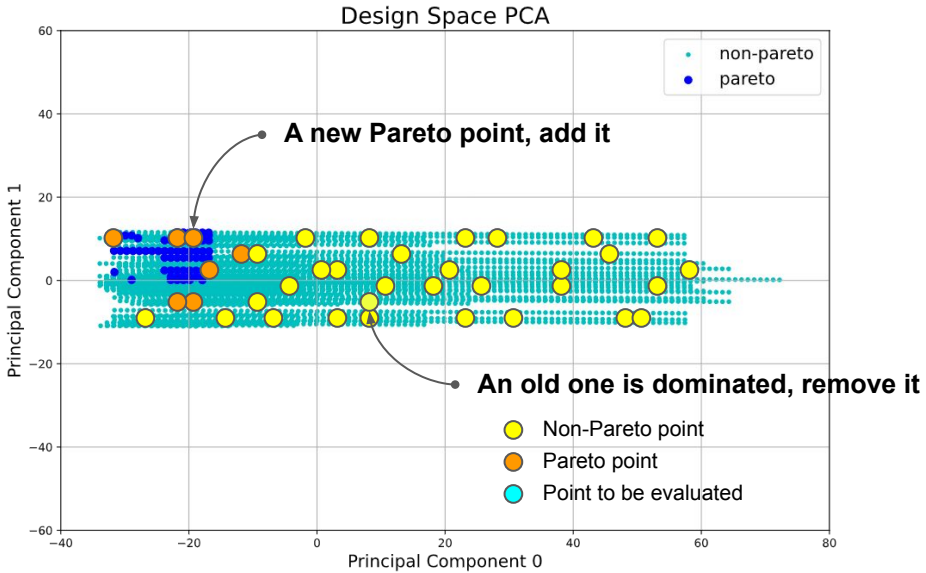


- Each parameter of a pass becomes one dimension, the original 4-dimensional design space is reduced to two dimensions through PCA
- Pareto points are located at a corner of the design space, the variance of Pareto points is much smaller than the overall variance

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



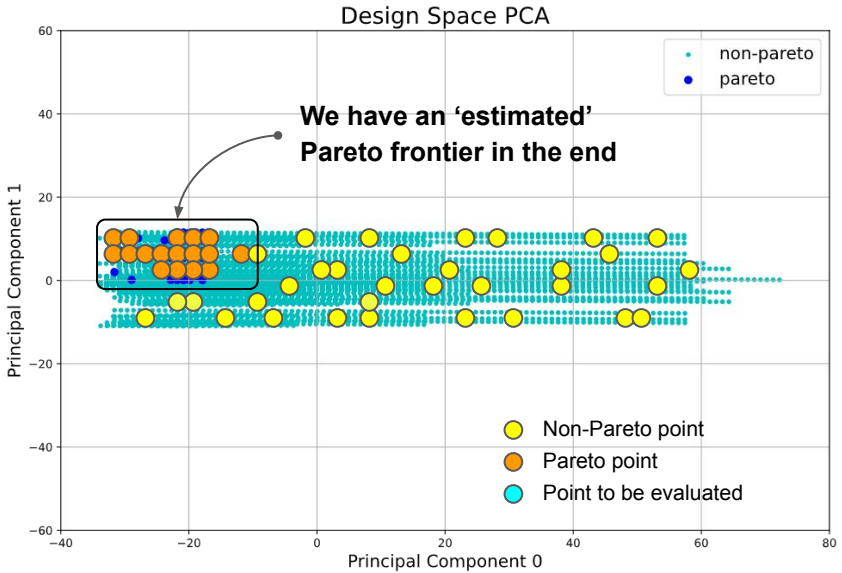
- Each parameter of a pass becomes one dimension, the original 4-dimensional design space is reduced to two dimensions through PCA
- Pareto points are located at a corner of the design space, the variance of Pareto points is much smaller than the overall variance

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

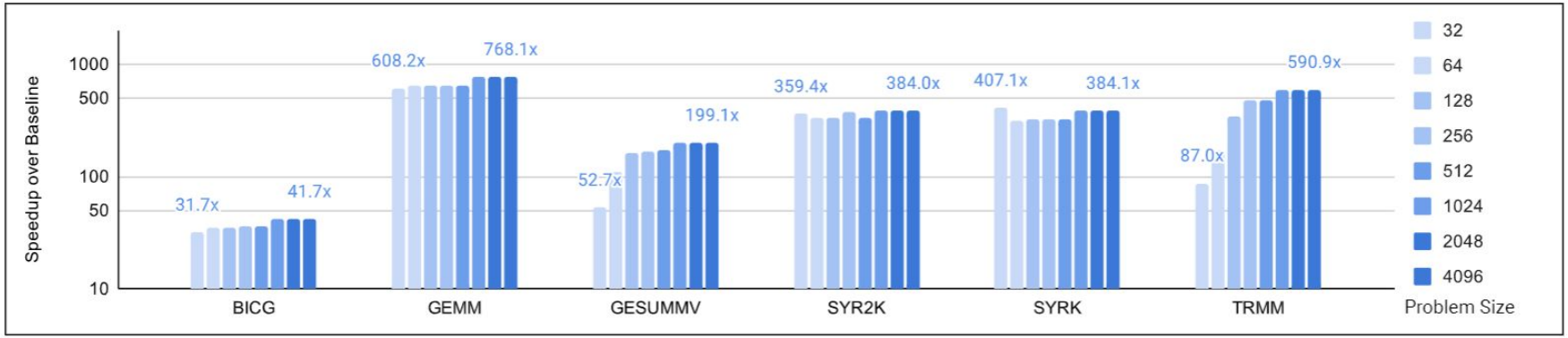
DSE Results of Computation Kernel

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.

DSE Results of Computation Kernel (Cont.)



Scalability study of computation kernels

1. The problem sizes of computation kernels are scaled from 32 to 4096 and the DSE engine is launched to search for the optimal solutions under each problem size.
2. For BICG, GEMM, SYR2K, and SYRK benchmarks, the DSE engine can achieve stable speedup under all problem sizes.
3. For GESUMMV and TRMM, the speedups are limited by the small problem sizes.

DSE Results of Computation Kernel on ScaleFlow

Table 5: Optimization results of C++ kernels from PolyBench dataset. The data types of all kernels are single-precision floating-point. The *ScaleHLS* designs are automatically generated by [37] and then optimized by Vitis HLS [18]. The *Vitis* designs are only optimized by Vitis HLS.

Kernel	Compile Time (s)	LUT Number	FF Number	DSP Number					Throughput (Sample/s)				
				Ours	ScaleHLS	Vitis	Ours v.s. ScaleHLS	Ours v.s. Vitis	Ours	ScaleHLS	Vitis	Ours v.s. ScaleHLS	Ours v.s. Vitis
2mm	0.65	38.8k	27.4k	269	152	9	1.77×	29.89×	239.22	122.39	1.23	1.95×	194.88×
3mm	0.79	38.7k	27.8k	243	173	19	1.40×	12.79×	175.43	92.33	1.04	1.90×	167.99×
atax	2.06	44.6k	34.6k	260	205	5	1.27×	52.00×	1021.39	932.26	103.18	1.10×	9.90×
bicg	0.72	16.0k	15.1k	61	61	6	1.00×	10.17×	2869.69	2,869.61	104.19	1.00×	27.54×
correlation	0.91	14.5k	12.3k	66	55	5	1.20×	13.20×	67.33	59.77	1.32	1.13×	50.97×
gesummv	0.60	34.2k	22.8k	232	232	6	1.00×	38.67×	31685.68	31,685.68	266.65	1.00×	118.83×
jacobi-2d	1.98	91.4k	56.6k	352	176	11	2.00×	32.00×	257.27	128.63	2.71	2.00×	94.95×
mvt	0.42	23.8k	16.5k	162	82	5	1.98×	32.40×	9979.04	4,989.02	62.13	2.00×	160.62×
seidel-2d	3.59	5.5k	2.5k	4	4	2	1.00×	2.00×	0.14	0.14	0.11	1.00×	1.28×
symm	1.05	14.9k	9.5k	74	74	8	1.00×	9.25×	2.62	2.62	2.02	1.00×	1.29×
syr2k	0.69	14.3k	12.8k	78	78	6	1.00×	13.00×	27.68	27.67	1.44	1.00×	19.23×
syrk	0.87	12.2k	10.8k	72	72	6	1.00×	12.00×	27.69	27.69	1.44	1.00×	19.24×
Geo. Mean	**0.98**						**1.25×**	**16.12×**				**1.27×**	**29.88×**

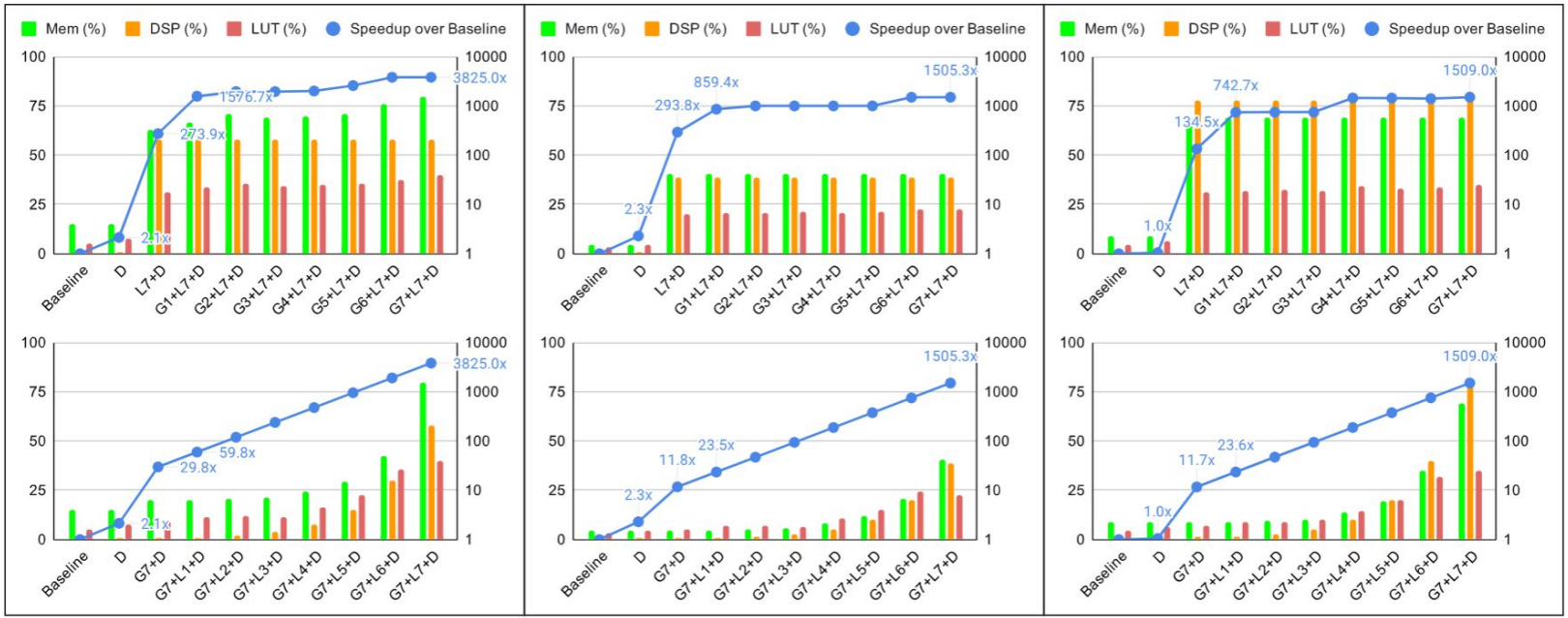
Optimization Results of DNN Models

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.

Optimization Results of DNN Models (Cont.)



(a) ResNet-18 **(b)** VGG-16 **(c)** MobileNet

Ablation study of DNN models

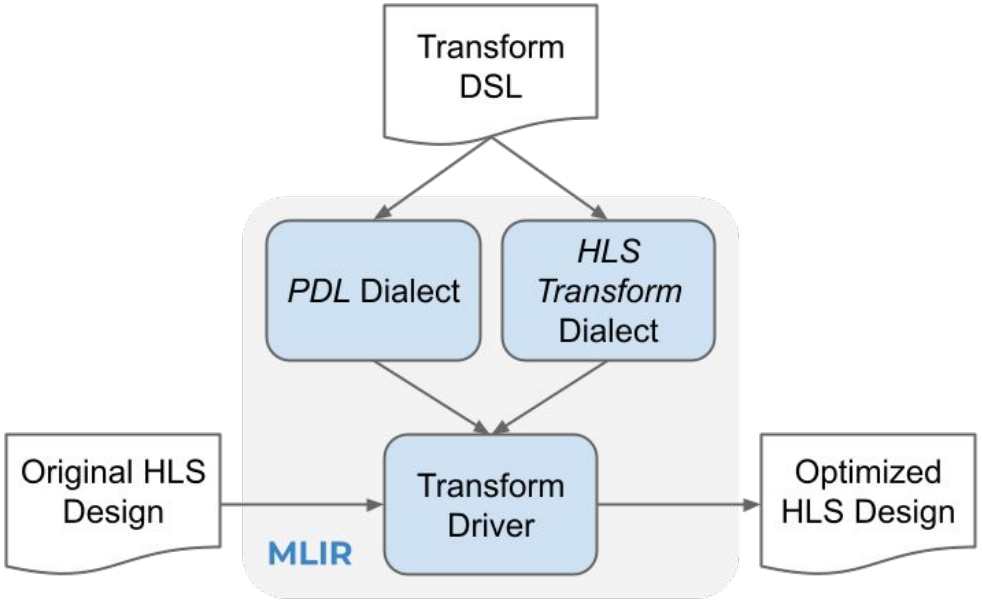
1. D, $L\{n\}$, and $G\{n\}$ denote directive, loop, and graph optimizations, respectively. Larger n indicates larger loop unrolling factor and finer dataflow granularity for loop and graph optimizations, respectively.
2. We can observe that the directive (D), loop ($L7$), and graph ($G7$) optimizations contribute 1.8×, 130.9×, and 10.3 × average speedups on the three DNN benchmarks, respectively.

Optimization Results of DNN Models on ScaleFlow

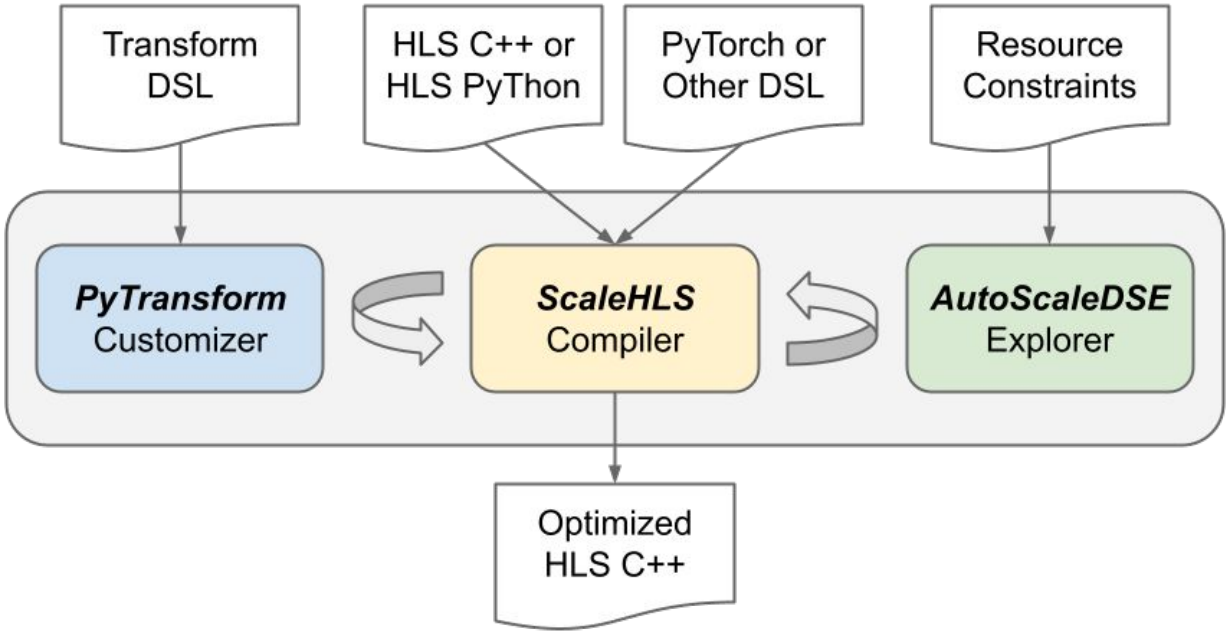
Table 6: Optimization results of PyTorch neural network models. The *DNNBuilder* results are directly from their paper [42]. The *ScaleHLS* designs are automatically generated by [37] and then optimized by Vitis HLS [18].

Model	Compile Time (s)	LUT Number	Memory (Mb)	DSP Number			Throughput (Sample/s)			DSP Efficiency		
				Ours	Ours v.s. DNNBuilder	Ours v.s. ScaleHLS	Ours	Ours v.s. DNNBuilder	Ours v.s. ScaleHLS	Ours	Ours v.s. DNNBuilder	Ours v.s. ScaleHLS
ResNet-18	83.1	142.1k	33.1	667	-	684 (0.98×)	45.4	-	3.3 (13.88×)	73.8%	-	5.2% (14.24×)
MobileNet	110.8	132.9k	27.9	518	-	459 (1.13×)	137.4	-	15.4 (8.90×)	75.5%	-	9.6% (7.88×)
ZFNet	116.2	103.8k	22.4	639	824 (0.78×)	-	90.4	112.2 (0.81×)	-	82.8%	79.7% (1.04×)	-
VGG-16	199.9	266.2k	42.1	1118	680 (1.64×)	878 (1.27×)	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	35.4	904	680 (1.33×)	-	33.7	22.1 (1.52×)	-	94.3%	86.0% (1.10×)	-
MLP	40.9	21.0k	7.7	164	-	136 (1.21×)	938.9	-	152.6 (6.15×)	90.0%	-	17.6% (5.10×)
Geo. Mean	**108.7**				**1.19×**	**1.14×**		**1.29×**	**8.54×**		**1.07×**	**7.49×**

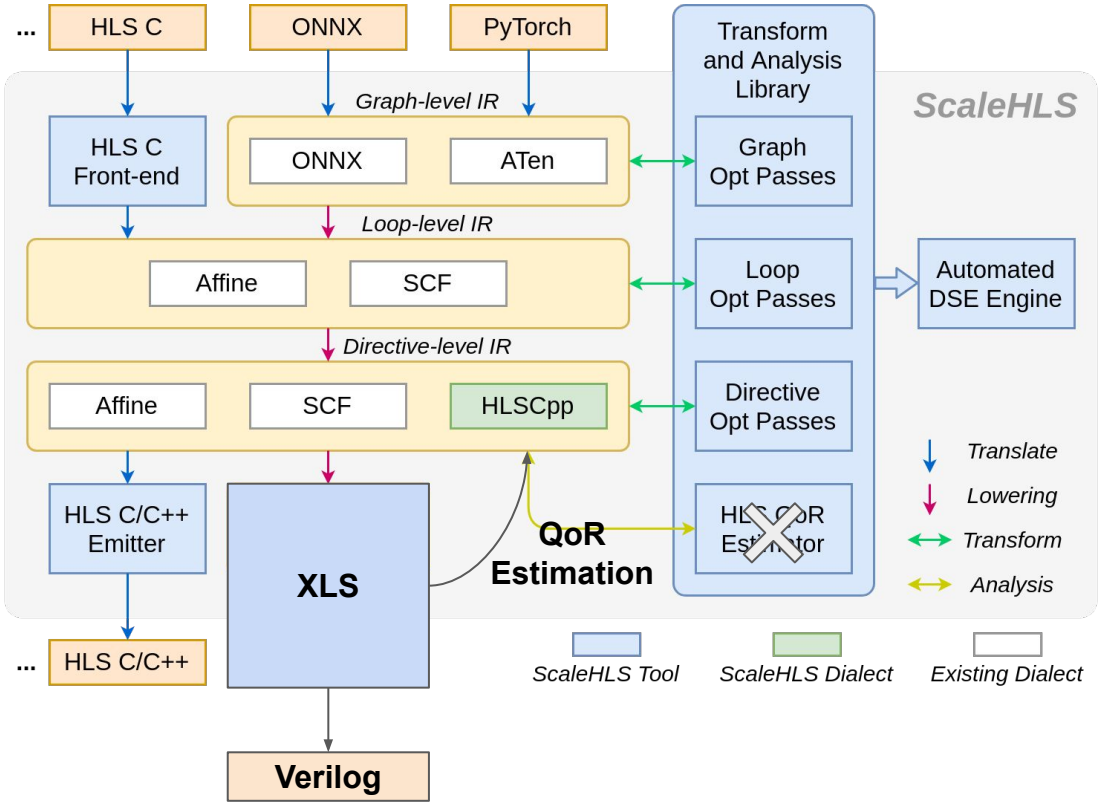
Future Directions (1) - Design Customization

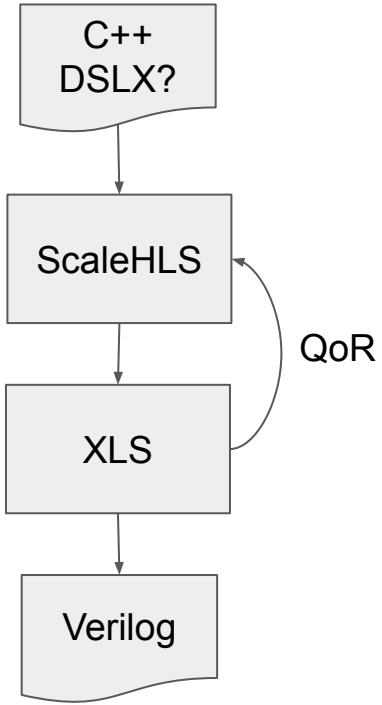


Future Directions (2) - Design Space Exploration



Future Directions (3) - Link with XLS (FE and BE)





Thanks! Q&A

Feb. 3, 2022