ScaleHLS: Scalable High-Level Synthesis through MLIR

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.

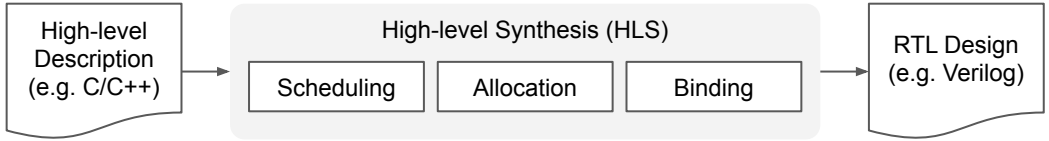
UNIVERSITY OF ILLINOIS URBANA-CHAMPAIGN

Imperial College London

Outline

- Motivations
- Background: MLIR
- ScaleHLS Framework
- ScaleHLS Optimizations
- Design Space Exploration
- Evaluation Results
- Conclusion

Motivations (1)

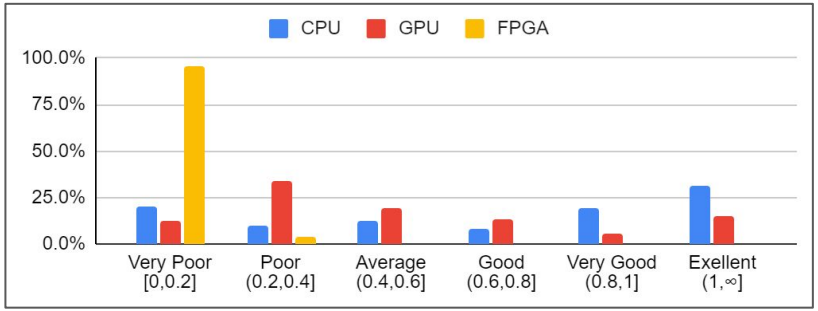


High-level Synthesis (HLS) is wonderful!

- **Reduce design complexity:** Code density can be reduced by 7X - 8X moving from RTL to C/C++ [1]
- **Improve design productivity:** Get to working designs faster and reduce time-to-market [2]
- **Identify performance-area trade-offs:** Implement design choices quickly and avoid premature optimization [3]

Design HLS accelerator is challenging 😈

- **Friendly to experts:** Rely on the designers writing 'good' code to achieve high design quality [4]
- **Large design space:** Different combinations of applicable optimizations for large-scale designs [3]
- **Correlation of design factors:** It is difficult for human to discover the complicated correlations [5]



Students are requested to accelerate a CNN model using CPU, GPU, and FPGA. The figure shows the percentage of students' submissions in each performance range. The performances are normalized with respect to 75% of expert design's performance [4].

[1] P. Coussy, et al. High-Level Synthesis: from Algorithm to Digital Circuit. 2008. Springer.
[2] J. Cong, et al. High-Level Synthesis for FPGAs: From Prototyping to Deployment. 2011. TCAD.
[3] B. C. Schafer, et al. High-Level Synthesis Design Space Exploration: Past, Present, and Future. 2020. TCAD.
[4] A. Sohrabizadeh, et al. AutoDSE: Enabling Software Programmers Design Efficient FPGA Accelerators. 2010. ArXiv.
[5] M. Yu. Chimera: An Efficient Design Space Exploration Tool for FPGA High-level Synthesis. 2021. Master thesis.

Motivations (2)

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 

, 

, etc.

Motivations (3)

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];
} } }
```

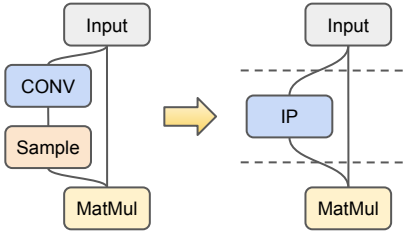
Generate RTL with 

, 

, etc.

Motivations (4)

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 Vivado™ HLS, XILINX VITIS™, etc.

Motivations (5)

Difficulties:

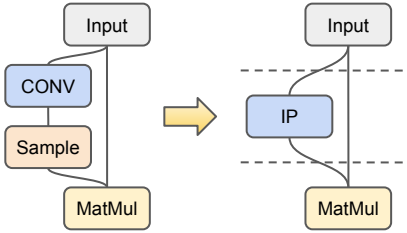
- Low-productive and error-proning
- Hard to enable automated design space exploration (DSE)
- NOT scalable!

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 Vivado™ HLS, XILINX VITIS™, etc.

Background: MLIR

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

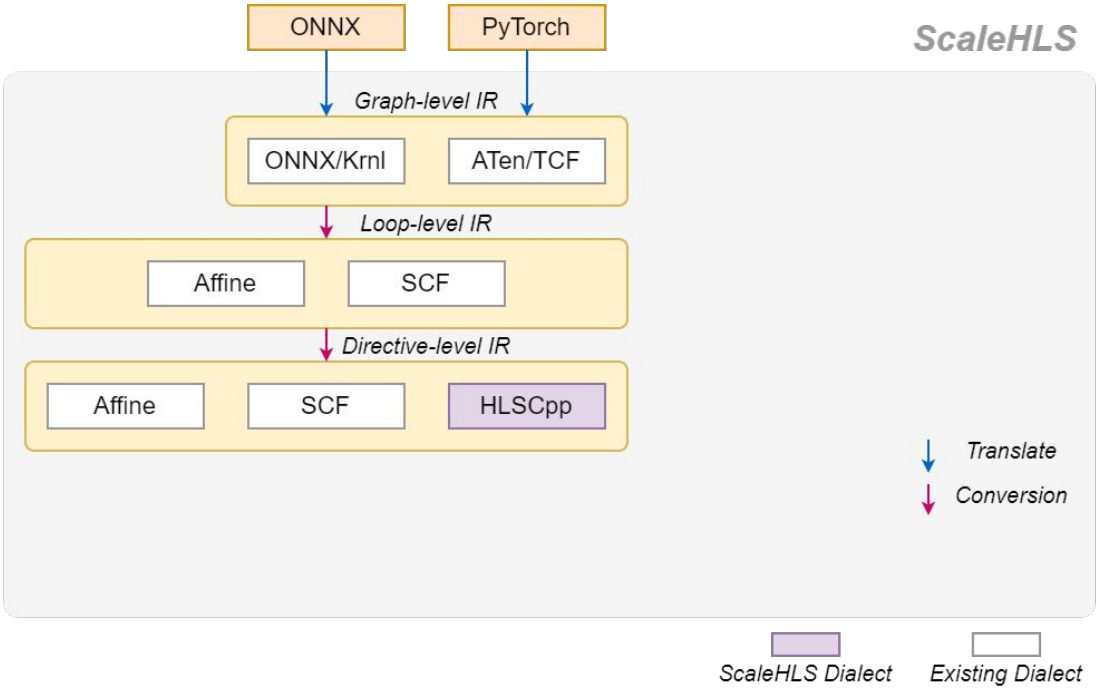
- **M**ulti-**L**evel **I**ntermediate **R**epresentation
- Joined LLVM, follows open library-based philosophy
- 🧩 ***Modular***, extensible, general to many domains
  - Being used for CPU, GPU, TPU, FPGA, HW, quantum, ...
- Easy to learn, great for research
- MLIR + LLVM IR + RISC-V CodeGen = 💝💝

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

ScaleHLS Framework (1)



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

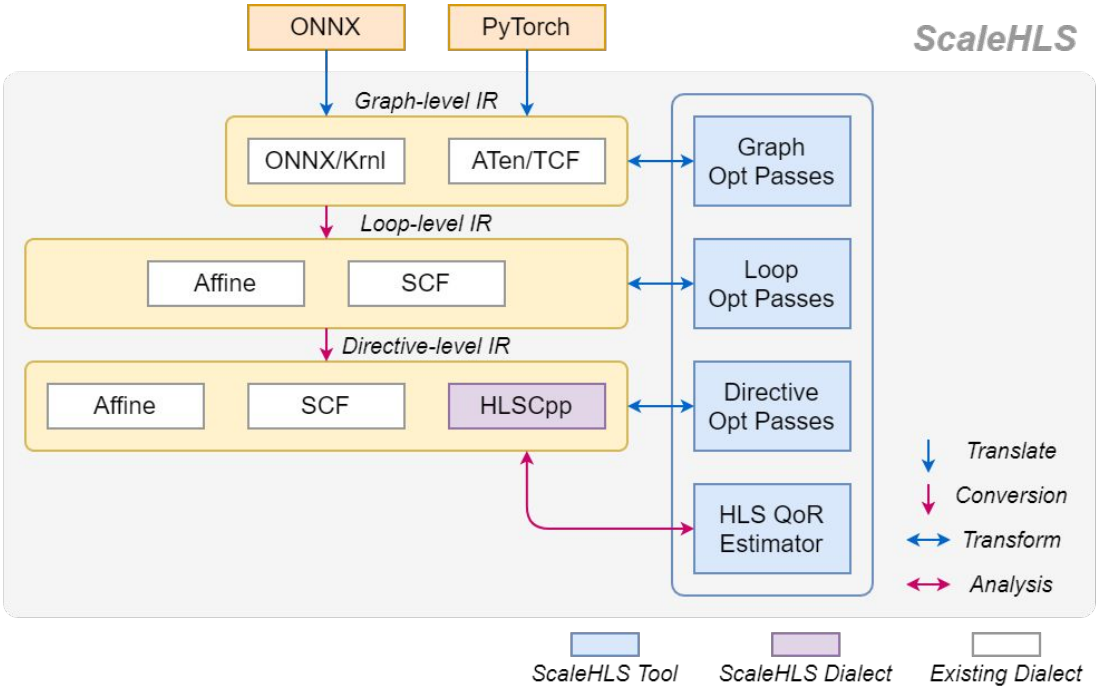
```
affine.for %i = 0 to 1 {
  affine.for %j = 0 to 10 {
    ... ...
    affine.for %k = 0 to 512 {
      %1 = affine.load %I[%i, %k] : memref<1x512xf32>
      %2 = affine.load %W[%j, %k] : memref<10x512xf32>
      %3 = affine.load %O[%i, %j] : memref<1x10xf32>
      %4 = mulf %2, %3 : f32
      %5 =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



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.

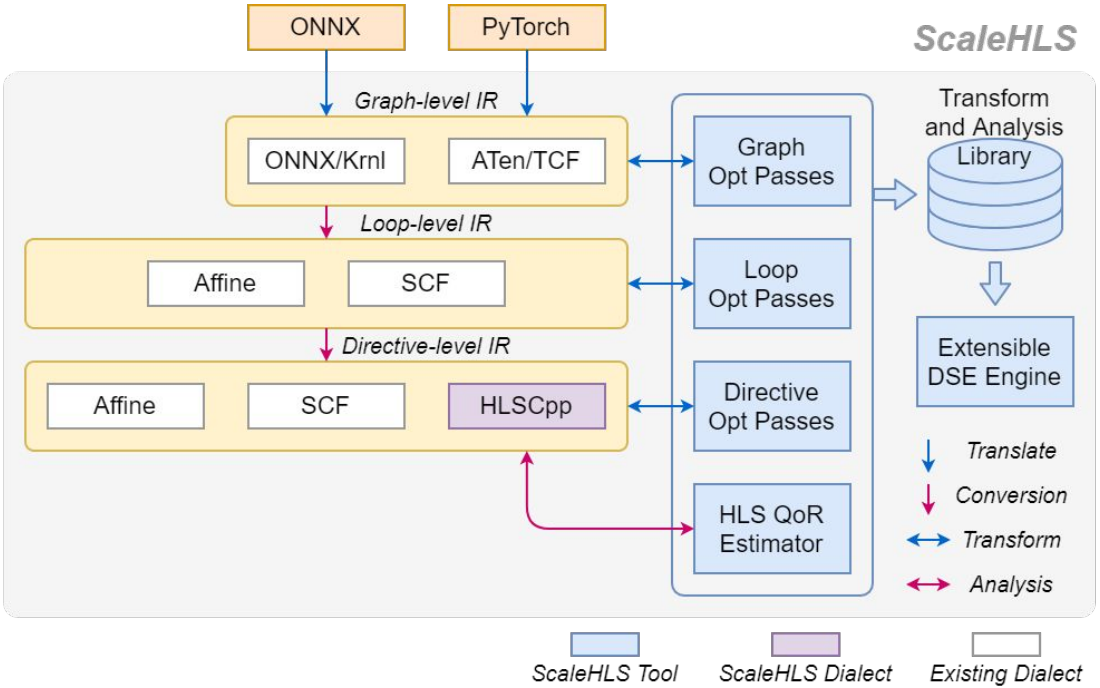
Optimize It!

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 (3)



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.

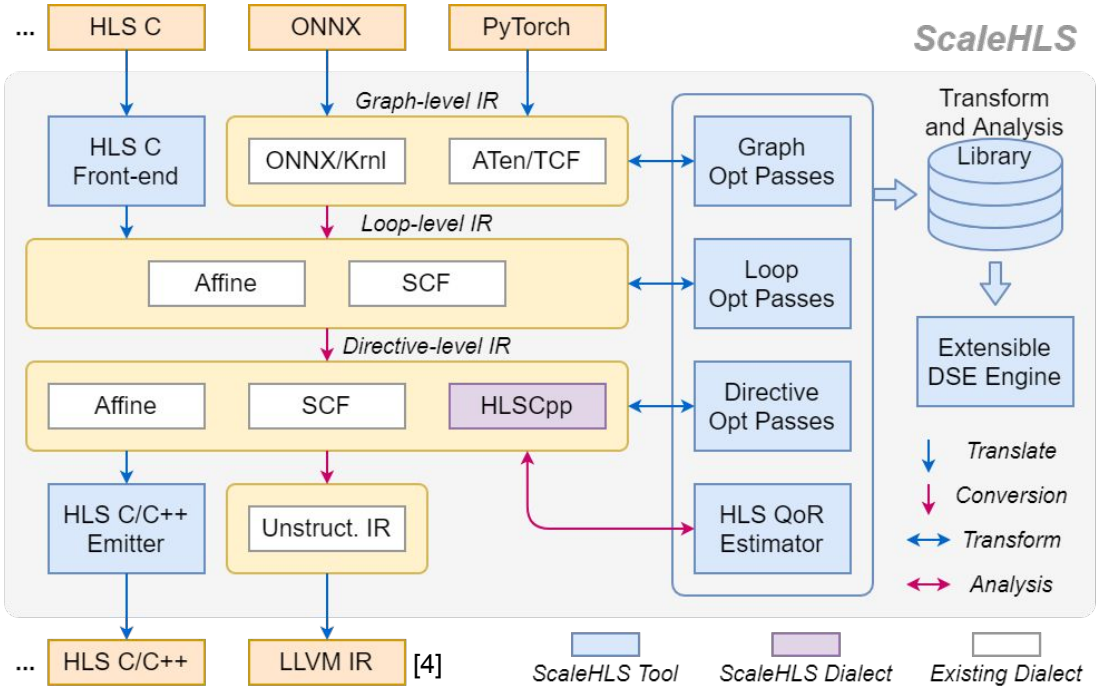
Explore It!

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 (4)



[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.

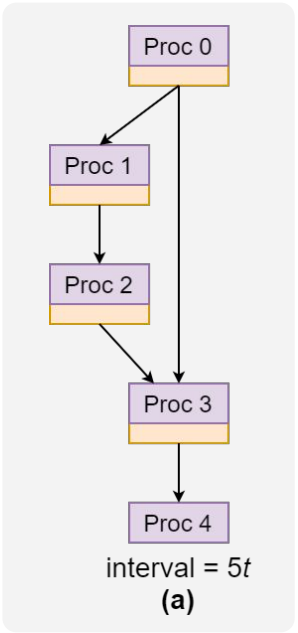
Enable End-to-end Flow!

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

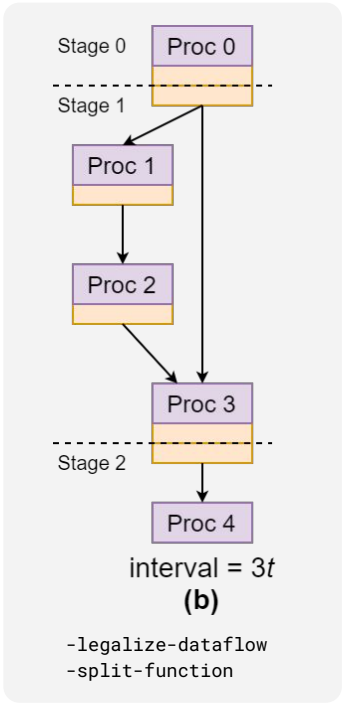
HLS C/C++ Emitter: Generate synthesizable HLS designs for downstream tools, such as Vivado HLS.

ScaleHLS Optimizations (1)

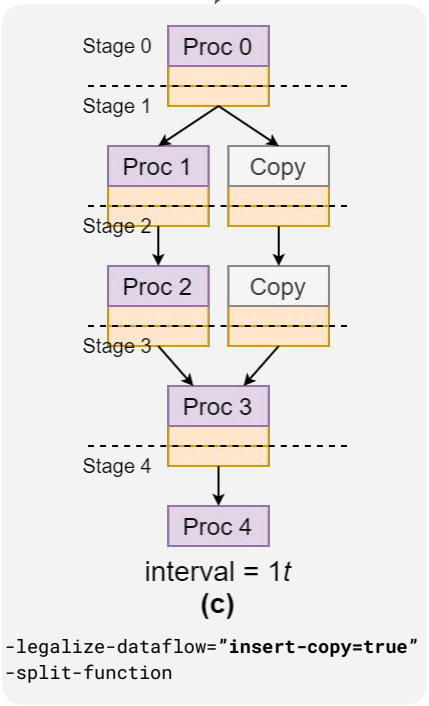
	Passes	Target	Parameters
Graph	-legalize-dataflow	function	insert-copy
	-split-function	function	min-gran

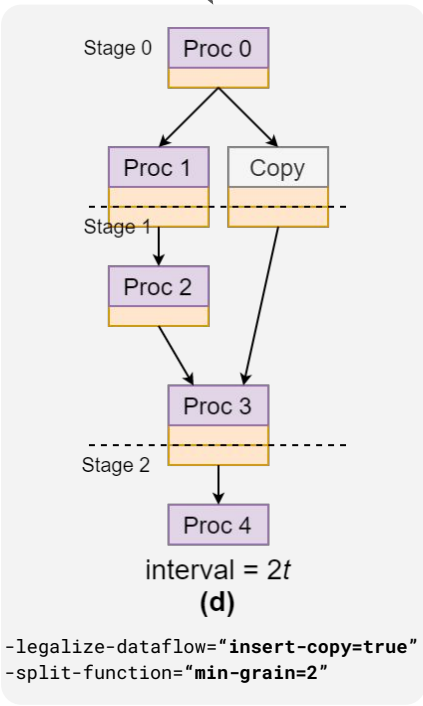


Coarse-grained Pipelining
(dataflow pragma)



Enable a **graph-level** throughput-area trade-off



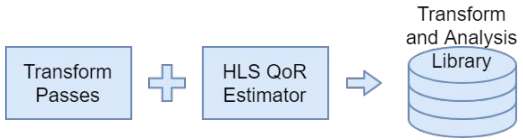


ScaleHLS Optimizations (2)

	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.

Each parameter of an optimization is a **tunable knob** in DSE




```
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

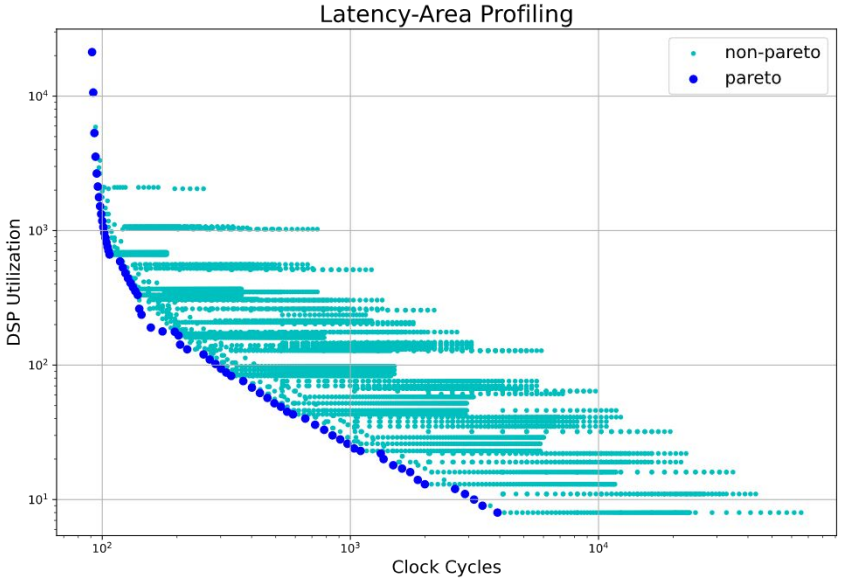
Remove variable loop bound

Loop pipeline

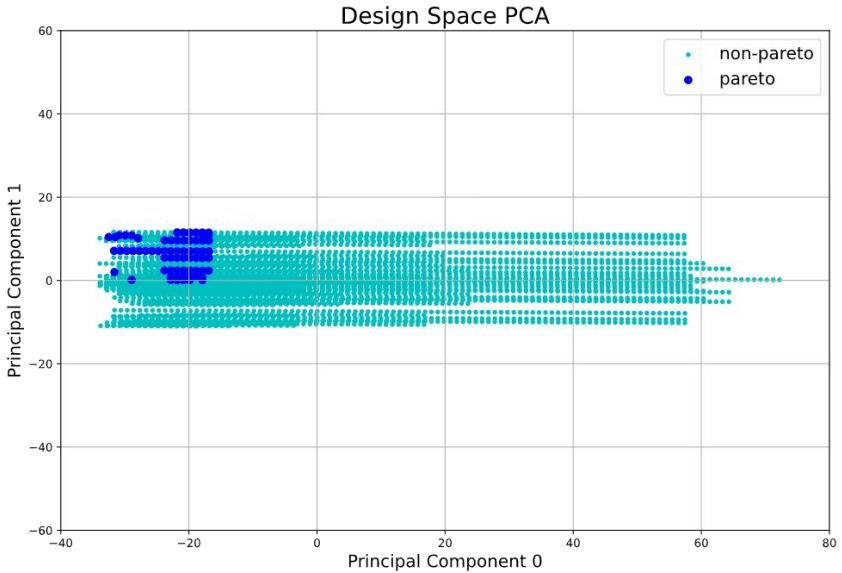
Simplify if ops;
Store ops forward;
Simplify memref ops

Optimized C emitted by the C/C++ emitter

Design Space Exploration (1)



Pareto frontier of a GEMM kernel

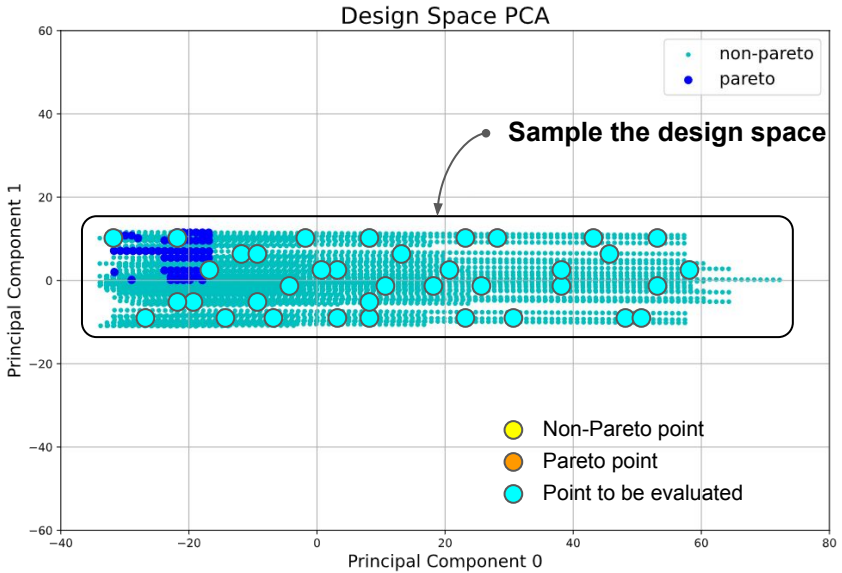


- Each parameter of a transform pass is one dimension, the multi-dimensional design space is reduced to two dimensions through PCA
- Pareto points are located at a corner of the design space, the variance of Pareto points is much smaller than the overall variance

Design Space Exploration (2)

DSE algorithm:

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

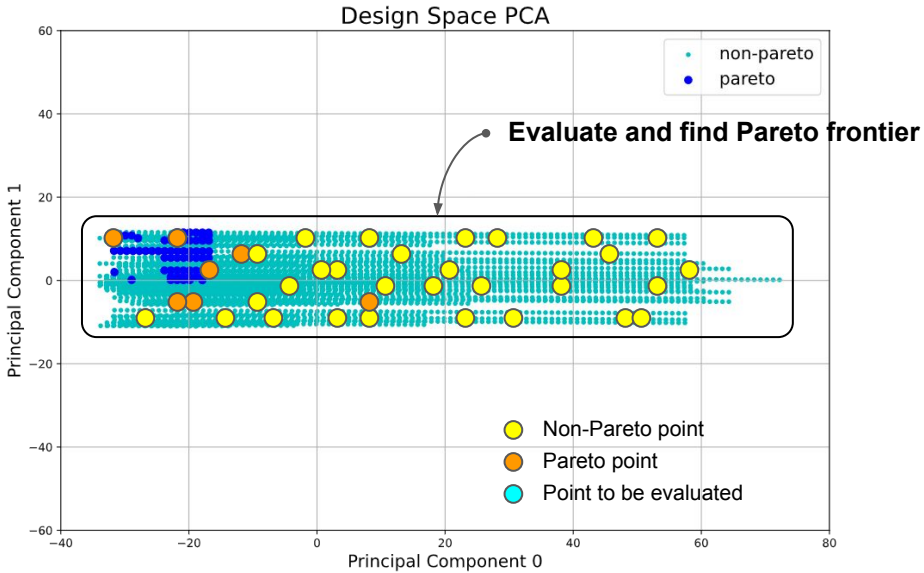


- Each parameter of a transform pass is one dimension, the multi-dimensional design space is reduced to two dimensions through PCA
- Pareto points are located at a corner of the design space, the variance of Pareto points is much smaller than the overall variance

Design Space Exploration (3)

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

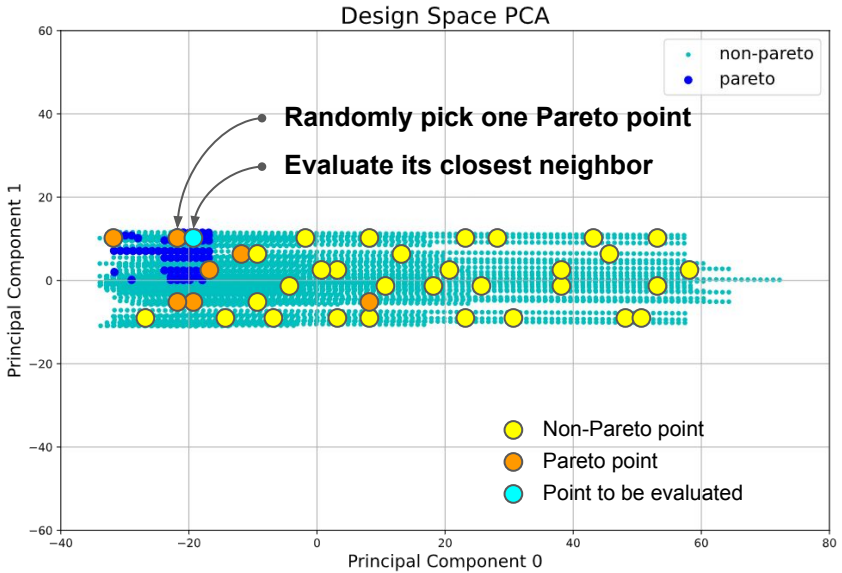


- Each parameter of a transform pass is one dimension, the multi-dimensional design space is reduced to two dimensions through PCA
- Pareto points are located at a corner of the design space, the variance of Pareto points is much smaller than the overall variance

Design Space Exploration (4)

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 random selected design point in the current Pareto frontier

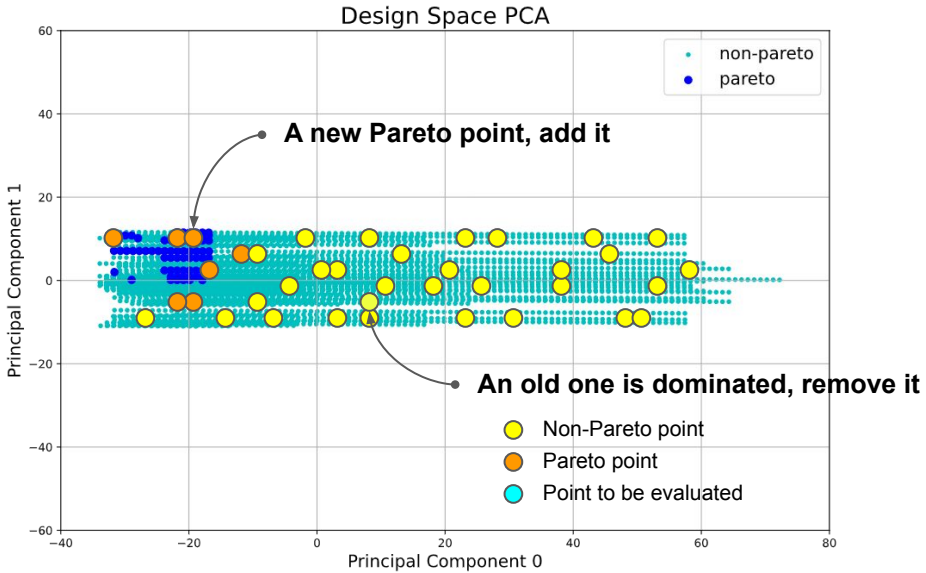


- Each parameter of a transform pass is one dimension, the multi-dimensional design space is reduced to two dimensions through PCA
- Pareto points are located at a corner of the design space, the variance of Pareto points is much smaller than the overall variance

Design Space Exploration (5)

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 random selected design point in the current Pareto frontier
4. Repeat step (2) and (3) to update the discovered Pareto frontier



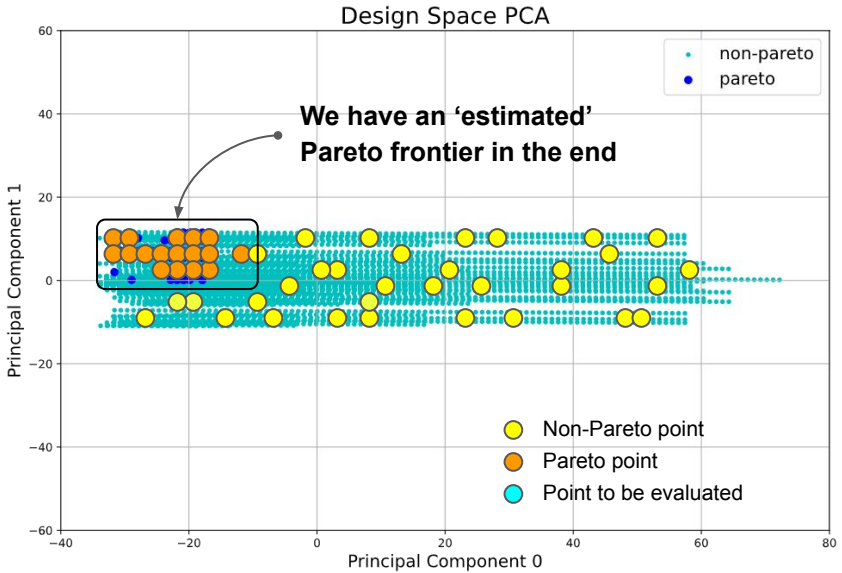
- Each parameter of a transform pass is one dimension, the multi-dimensional design space is reduced to two dimensions through PCA
- Pareto points are located at a corner of the design space, the variance of Pareto points is much smaller than the overall variance

Design Space Exploration (6)

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 random 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.

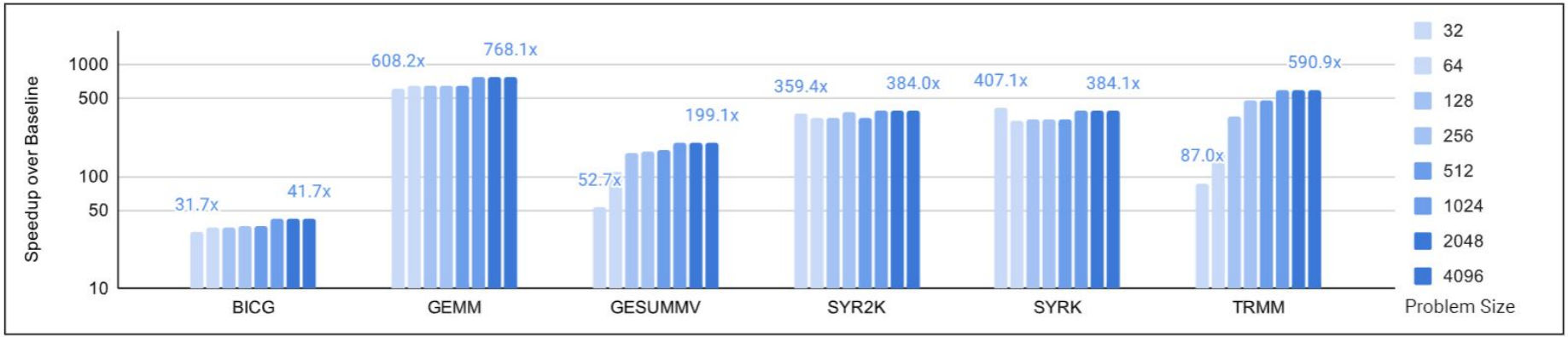


- Each parameter of a transform pass is one dimension, the multi-dimensional design space is reduced to two dimensions through PCA
- Pareto points are located at a corner of the design space, the variance of Pareto points is much smaller than the overall variance

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]

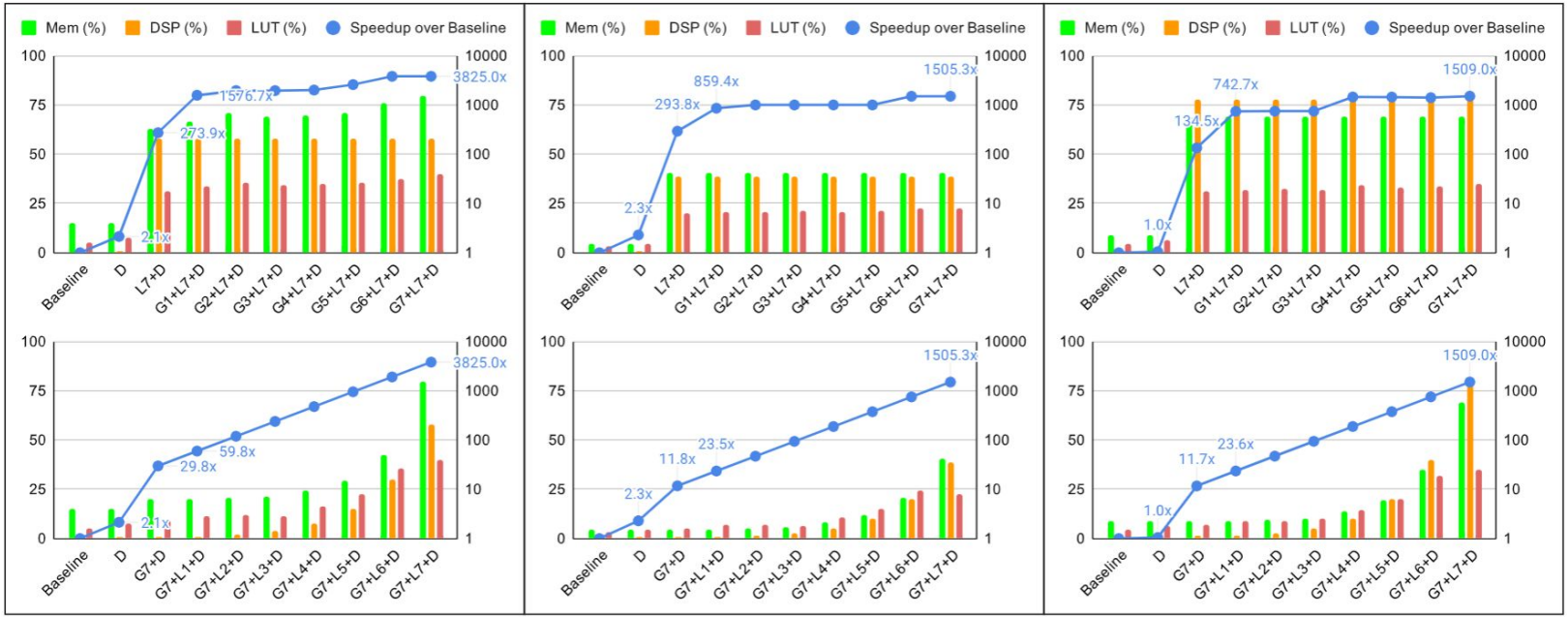
Speedup is with respect to the baseline designs only optimized by Xilinx Vivado HLS. LP and RVB denote Loop Perfectization and Remove Variable Bound, respectively.
In the Loop Order Optimization,the i-th loop in the loop nest is permuted to location $PermMap[i]$, where locations are from the outermost loop to inner.



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

Speedup is over the baseline design only optimized by Vivado HLS.



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

D, $L\{n\}$, and $G\{n\}$ denote directive, loop, and graph optimizations, respectively. Larger n indicates stronger optimizations are applied.

Conclusion

- We presented ScaleHLS, a new MLIR-based HLS compilation flow, which features multi-level representation and optimization of HLS designs and supports a transform and analysis library dedicated for HLS.
- ScaleHLS enables an end-to-end compilation pipeline by providing an HLS C front-end and a synthesizable C/C++ emitter.
- An automated and extensible DSE engine is developed to search for optimal solutions in the multi-dimensional design spaces.
- Experimental results demonstrate that ScaleHLS has a strong scalability to optimize large-scale and sophisticated HLS designs and achieves significant performance and productivity improvements on a set of benchmarks.

Github: https://github.com/hanchenye/scalehls

Paper: https://arxiv.org/abs/2107.11673

Future Directions

- **IP Integration.** The graph-level representation of ScaleHLS enables the ability to integrate existing HLS IPs, such as Vitis Accelerated Libraries [1], into the compilation flow. Meanwhile, new IPs can be generated through launching the DSE engine of ScaleHLS.
- **DSE algorithms.** The parameterized interfaces provided by the analysis and transform libraries of ScaleHLS enable a large opportunity to investigate the optimization algorithms for the multi-dimensional DSE problem of HLS.
- **Machine-learning based QoR estimation.** Machine-learning methods can potentially capture more features from the hierarchical IR of ScaleHLS, thereby generating better estimation results than the analytical model-based methods.
- **Generate RTL code within MLIR.** Currently ScaleHLS leverages external back-ends for generating the RTL code. However, a direct RTL code generation can keep more information from the higher level IR and exploit the RTL-level representation and optimization (CIRCT [2]) to further improve the quality of the accelerator designs.

[1] Vitis Accelerated Libraries. https://github.com/Xilinx/Vitis_Libraries
[2] CIRCT: Circuit IR Compilers and Tools. https://github.com/llvm/circt/tree/main/

Thanks! Q&A

Hanchen Ye

hanchenye@gmail.com

Aug 5, 2021