

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

Hanchen Ye¹, Cong Hao², Jianyi Cheng³, Hyunmin Jeong¹, Jack Huang¹,
Stephen Neuendorffer⁴, Deming Chen¹

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



UNIVERSITY OF
ILLINOIS
URBANA-CHAMPAIGN



Imperial College
London

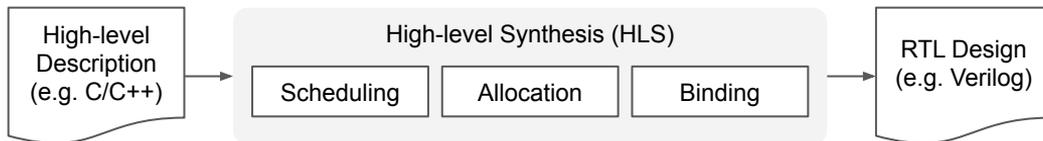


Outline

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

Motivations

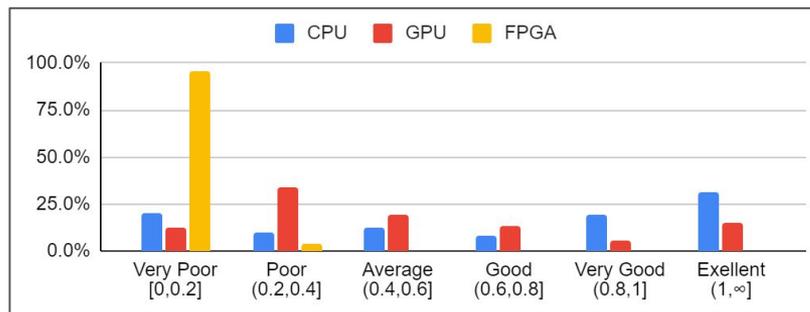
High-level Synthesis (HLS) is wonderful!



- **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 (Y axis) in each performance range (X axis). 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 (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];  
    } }  
}
```

```
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];  
    } }  
}
```

Directive Optimizations

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

Generate RTL with  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];  
    } }  
}
```

```
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];  
    } }  
}
```

Loop Optimizations

Loop interchange
Loop perfectization
Loop tile, skew, etc.

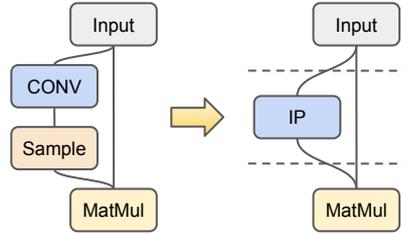
Directive Optimizations

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

Generate RTL with  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  and etc.
Pipeline II is 2 and overall latency is **65,552**

Motivations (cont.) - Overall

Difficulties:

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

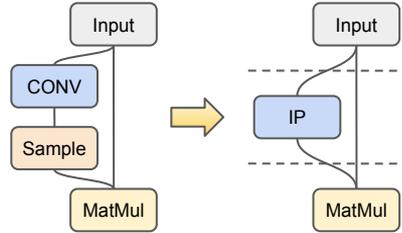


Solve problems at the 'correct' level AND automate it



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  and etc.
Pipeline II is 2 and overall latency is 65,552

Background: MLIR

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



- **M**ulti-**L**evel Intermediate **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 = 🇪🇺🇪🇺



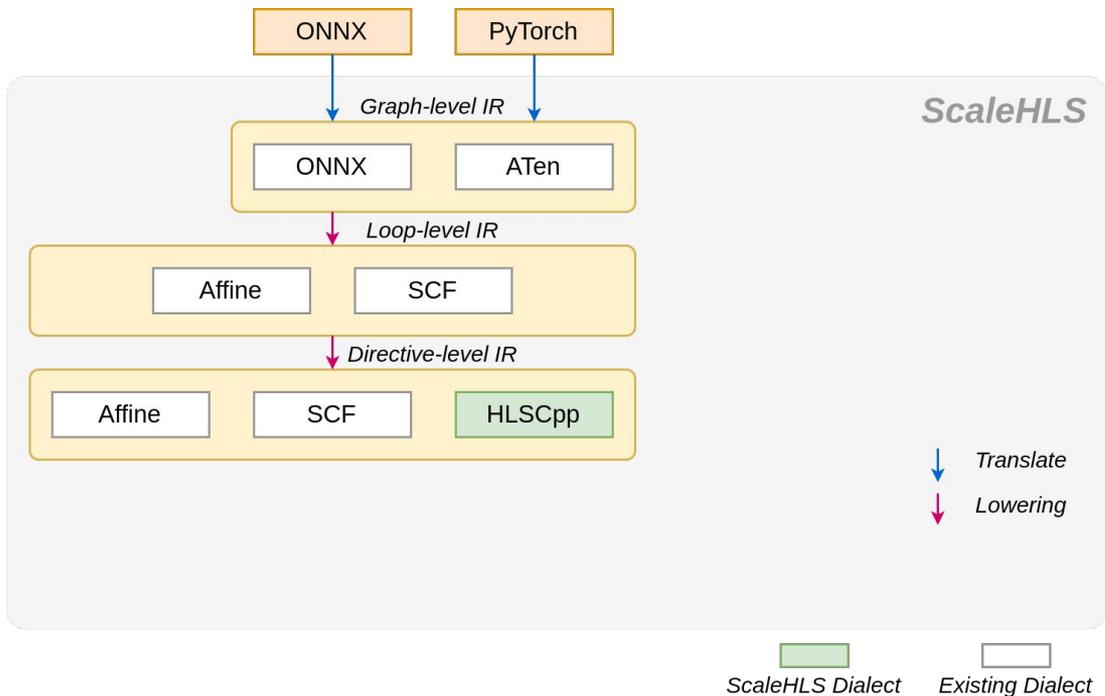
<https://mlir.llvm.org>

See more (e.g.):

[2020 CGO Keynote Talk Slides](#)

[2021 CGO Paper](#)

ScaleHLS Framework



- [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.

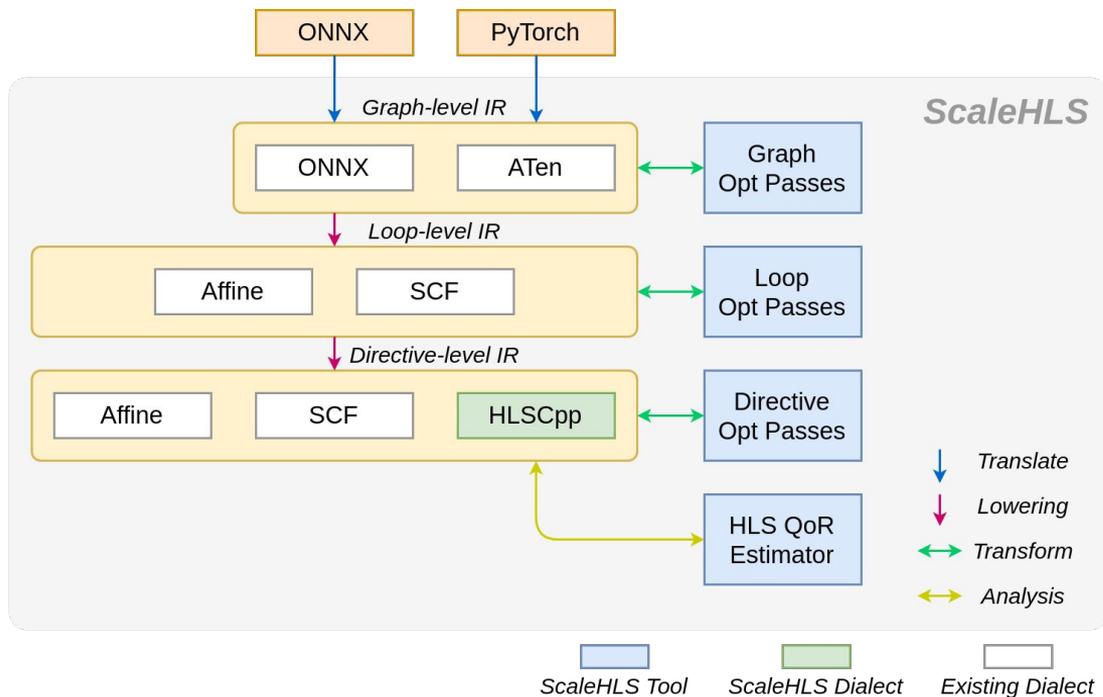
Directive-level IR: HLSCpp, Affine, and SCF dialect.

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

```
affine.for %i = 0 to 1 { Loop-level IR  
  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 %0[%i, %j] : memref<1x10xf32>  
      %4 = mulf %2, %3 : f32  
      %5 = addf %4, %5 : f32  
      affine.store %5, %0[%i, %j] : memref<1x10xf32>  
    } }  
  } }
```

```
affine.for %i = 0 to 1 { Directive-level IR  
  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, ...>
```

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.

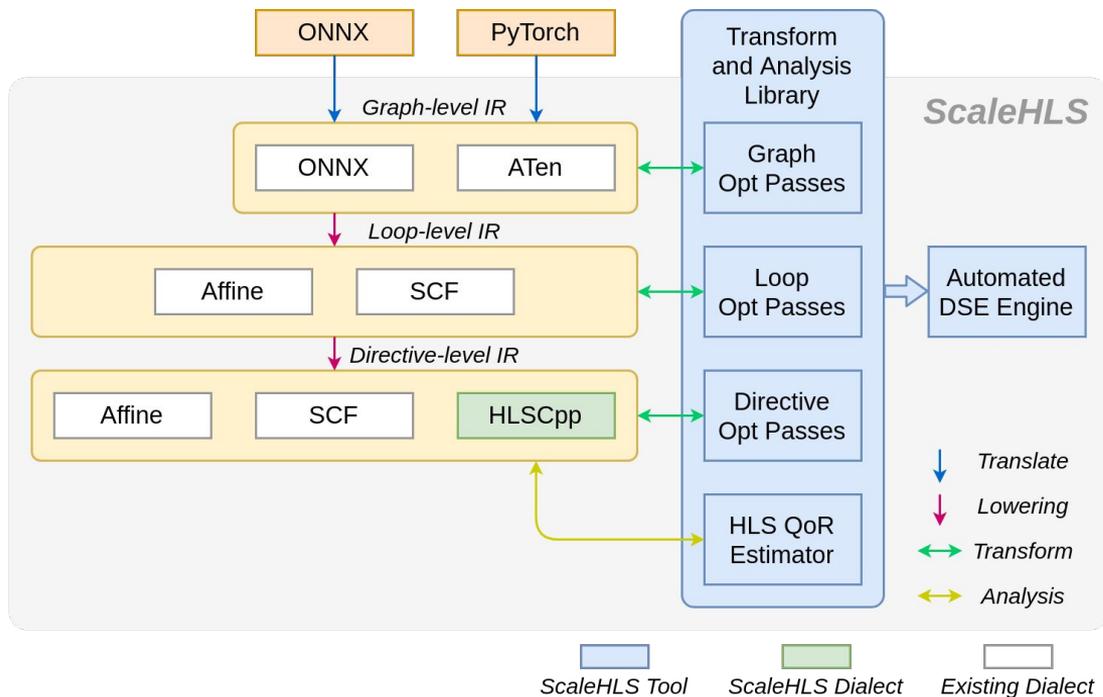
[1] ONNX-MLIR: Compiling ONNX Neural Network Models Using MLIR. <https://github.com/onnx/onnx-mlir>

[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.

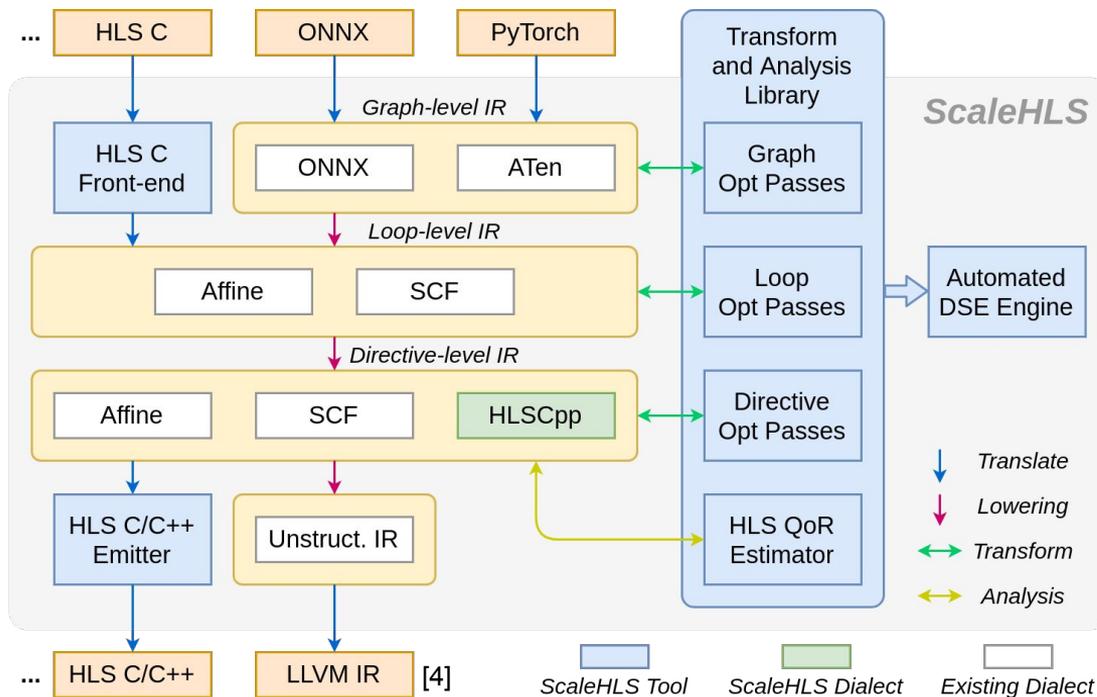
[1] ONNX-MLIR: Compiling ONNX Neural Network Models Using MLIR. <https://github.com/onnx/onnx-mlir>

[2] NPComp: MLIR based compiler toolkit for numerical python programs. <https://github.com/lvm/mlir-npcomp>

[3] MLIR: Multi-Level Intermediate Representation. <https://github.com/lvm/lvm-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.

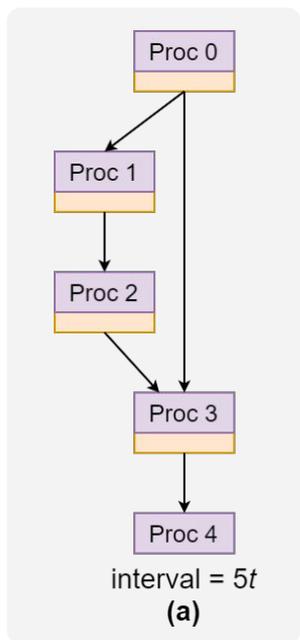
Enable End-to-end Flow!

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

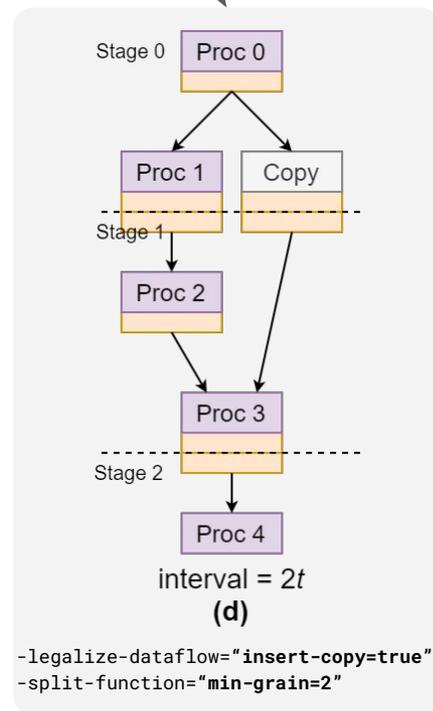
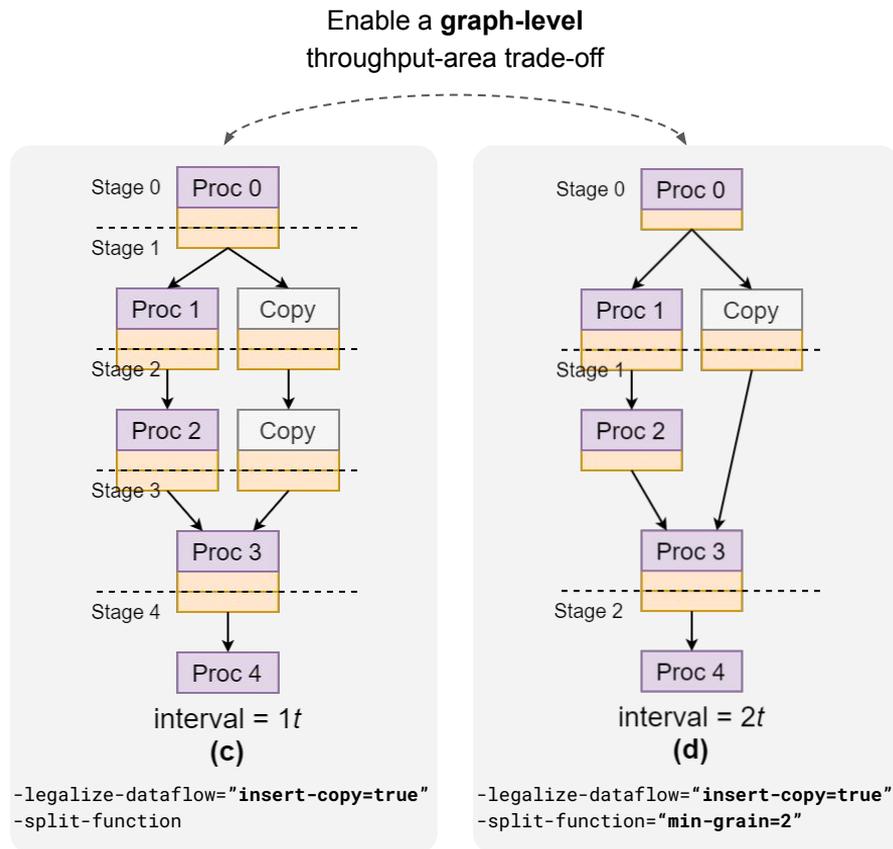
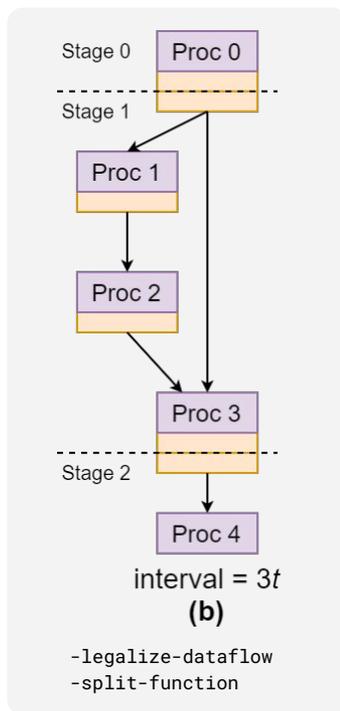
HLS C/C++ Emitter: Generate synthesizable HLS designs for downstream tools, such as Vivado HLS.

ScaleHLS Optimizations

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



**Coarse-grained
Pipelining**
(dataflow pragma)



ScaleHLS Optimizations (Cont.)

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

Baseline C

Loop perfectization

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 Optimizations (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 Optimizations (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];  
            }  
        }  
    }  
}
```

Baseline C

Loop perfectization

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

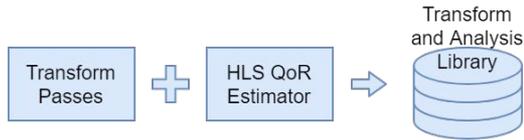
Simplify if ops;
Store ops forward;
Simplify memref ops

**Optimized C
emitted by the
C/C++ emitter**

ScaleHLS Optimizations (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];  
      }  
    }  
  }  
}
```

Baseline C

Loop perfectization

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=ctrl1  
  #pragma HLS interface s_axilite port=alpha bundle=ctrl1  
  #pragma HLS interface s_axilite port=beta bundle=ctrl1  
  #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

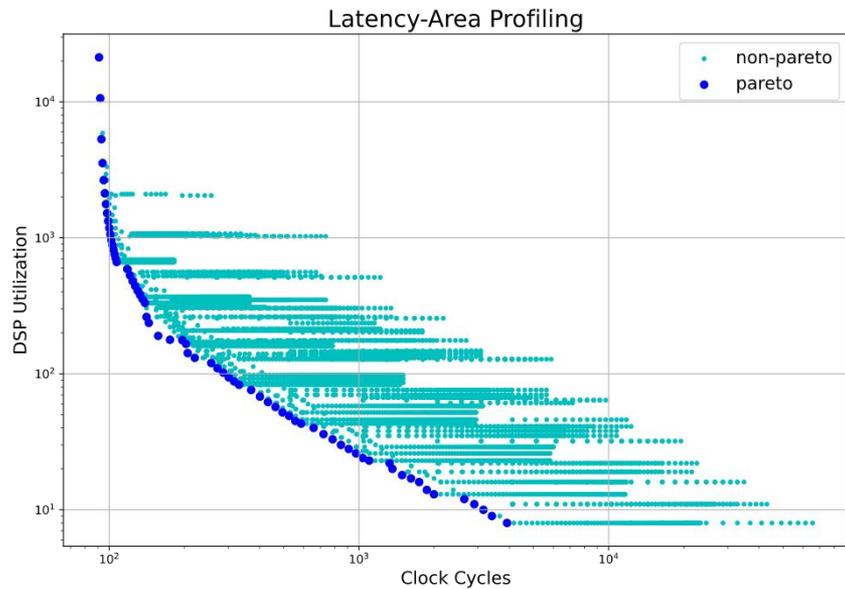
Remove variable loop bound

Loop pipeline

Simplify if ops;
Store ops forward;
Simplify memref ops

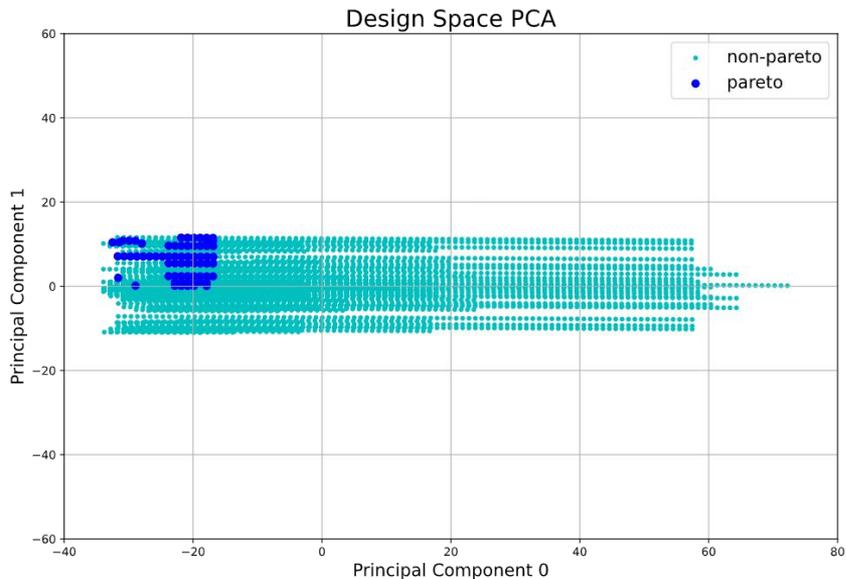
Optimized C
emitted by the
C/C++ emitter

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

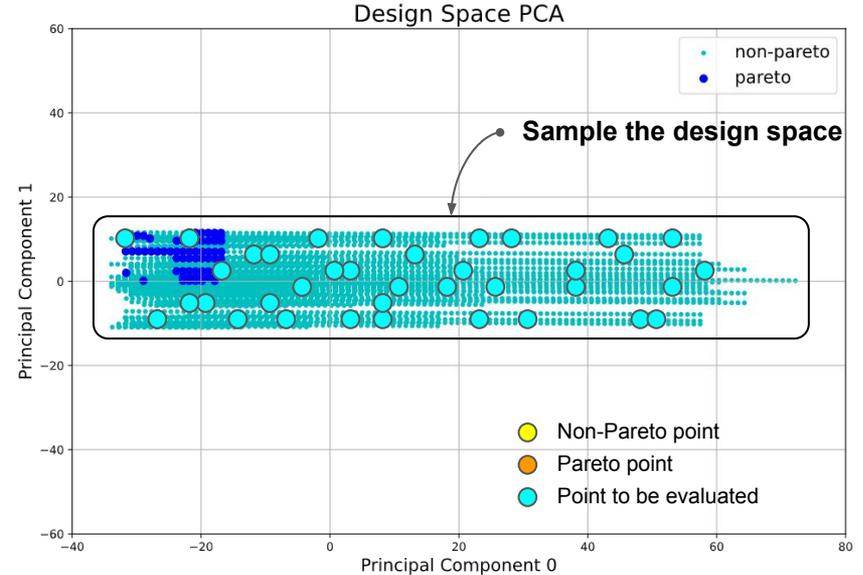


- Each parameter of a pass becomes one dimension, the original 4-dimensional design space is reduced to two dimensions through PCA
- Pareto points are located at a corner of the design space, the variance of Pareto points is much smaller than the overall variance

Design Space Exploration (Cont.)

DSE algorithm:

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

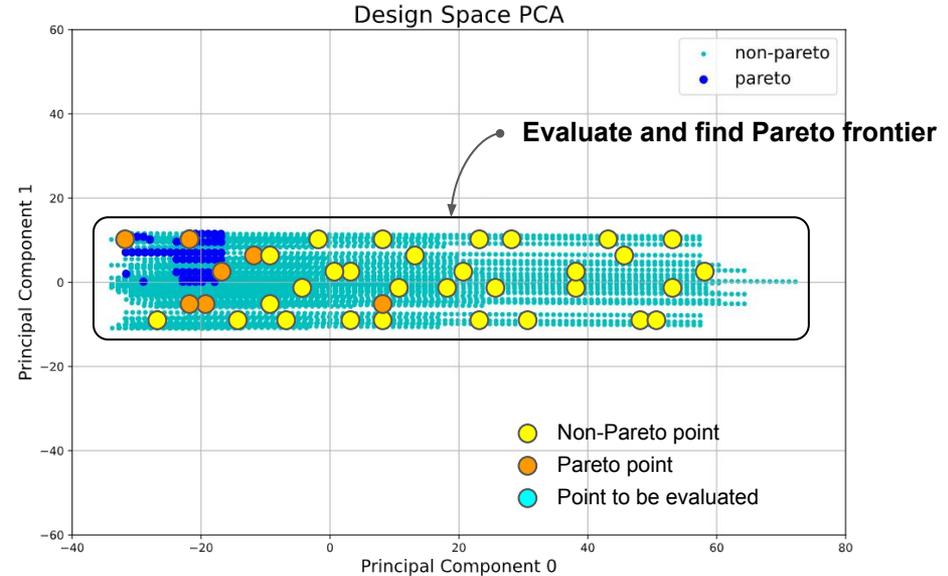


- Each parameter of a pass becomes one dimension, the original 4-dimensional design space is reduced to two dimensions through PCA
- Pareto points are located at a corner of the design space, the variance of Pareto points is much smaller than the overall variance

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

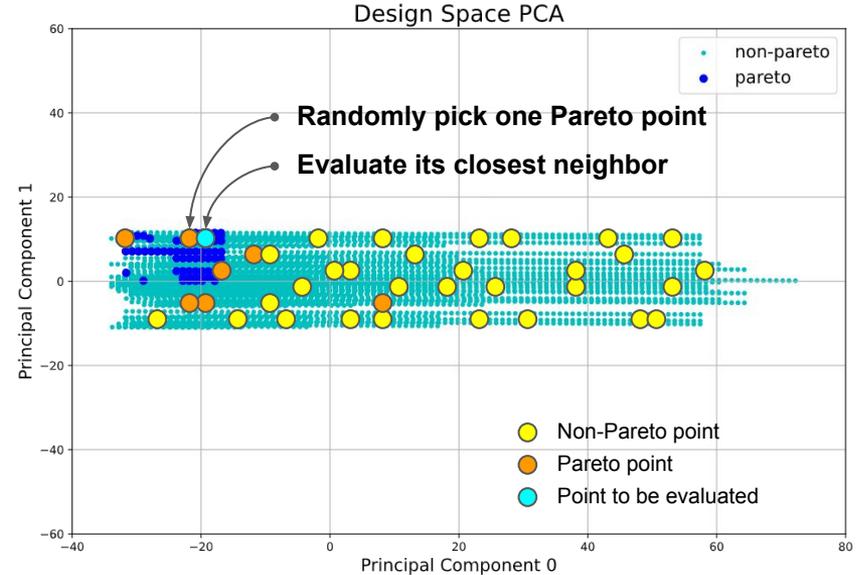


- Each parameter of a pass becomes one dimension, the original 4-dimensional design space is reduced to two dimensions through PCA
- Pareto points are located at a corner of the design space, the variance of Pareto points is much smaller than the overall variance

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

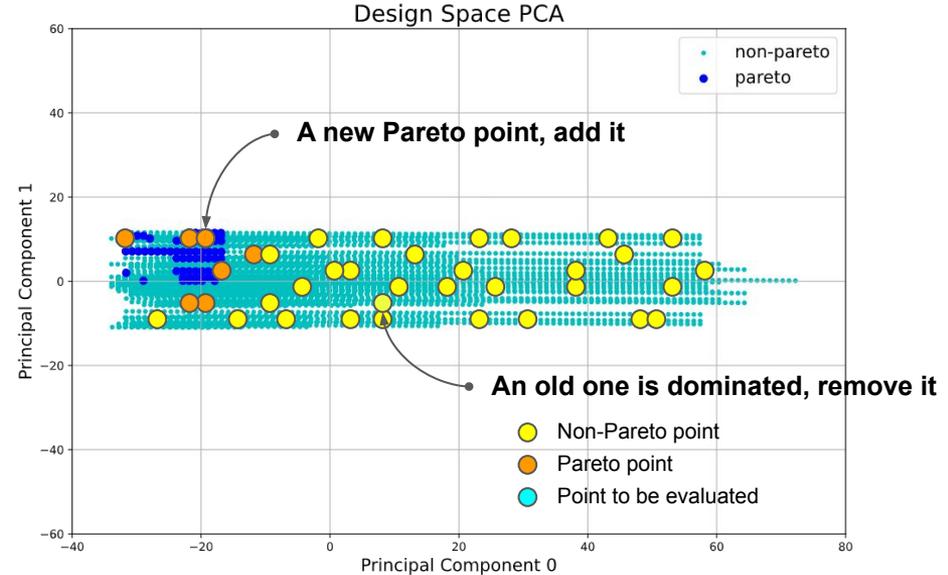


- Each parameter of a pass becomes one dimension, the original 4-dimensional design space is reduced to two dimensions through PCA
- Pareto points are located at a corner of the design space, the variance of Pareto points is much smaller than the overall variance

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



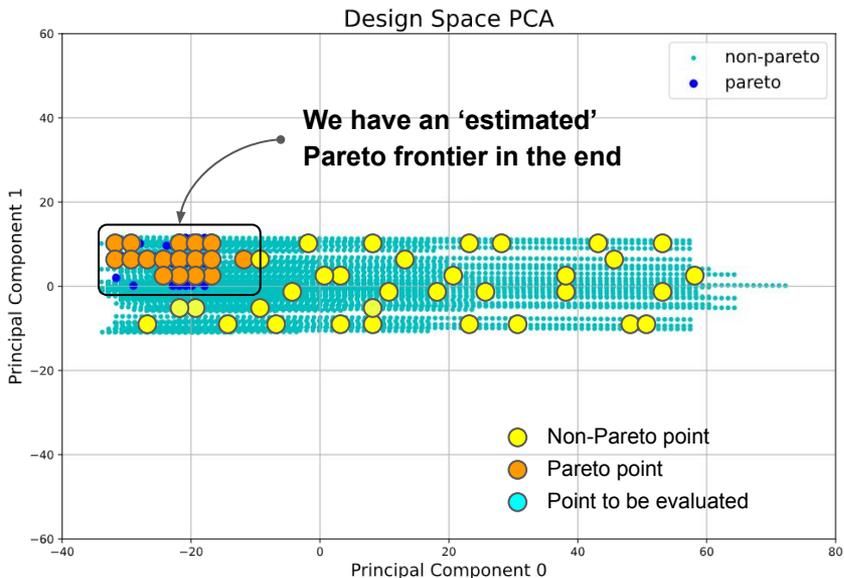
- Each parameter of a pass becomes one dimension, the original 4-dimensional design space is reduced to two dimensions through PCA
- Pareto points are located at a corner of the design space, the variance of Pareto points is much smaller than the overall variance

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

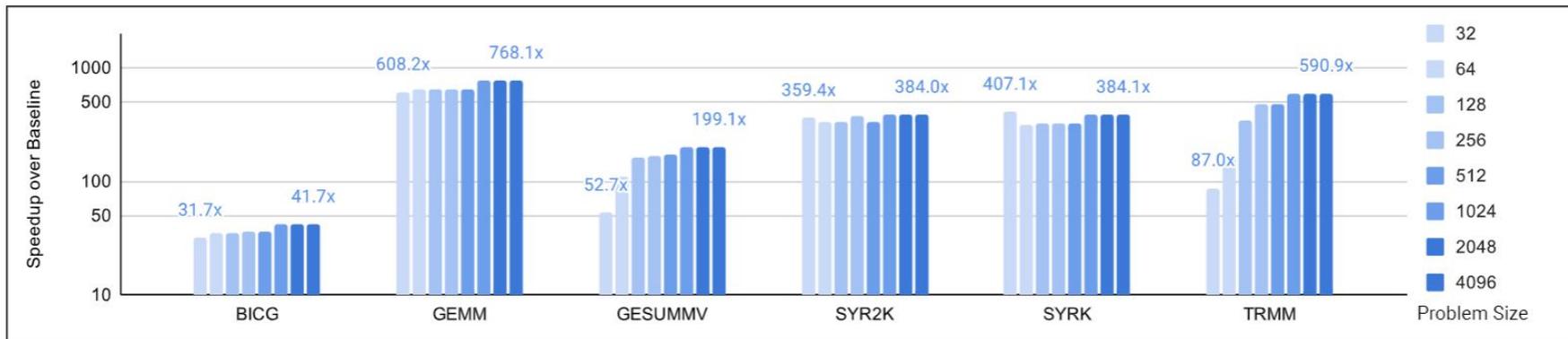
DSE Results of Computation Kernel

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.

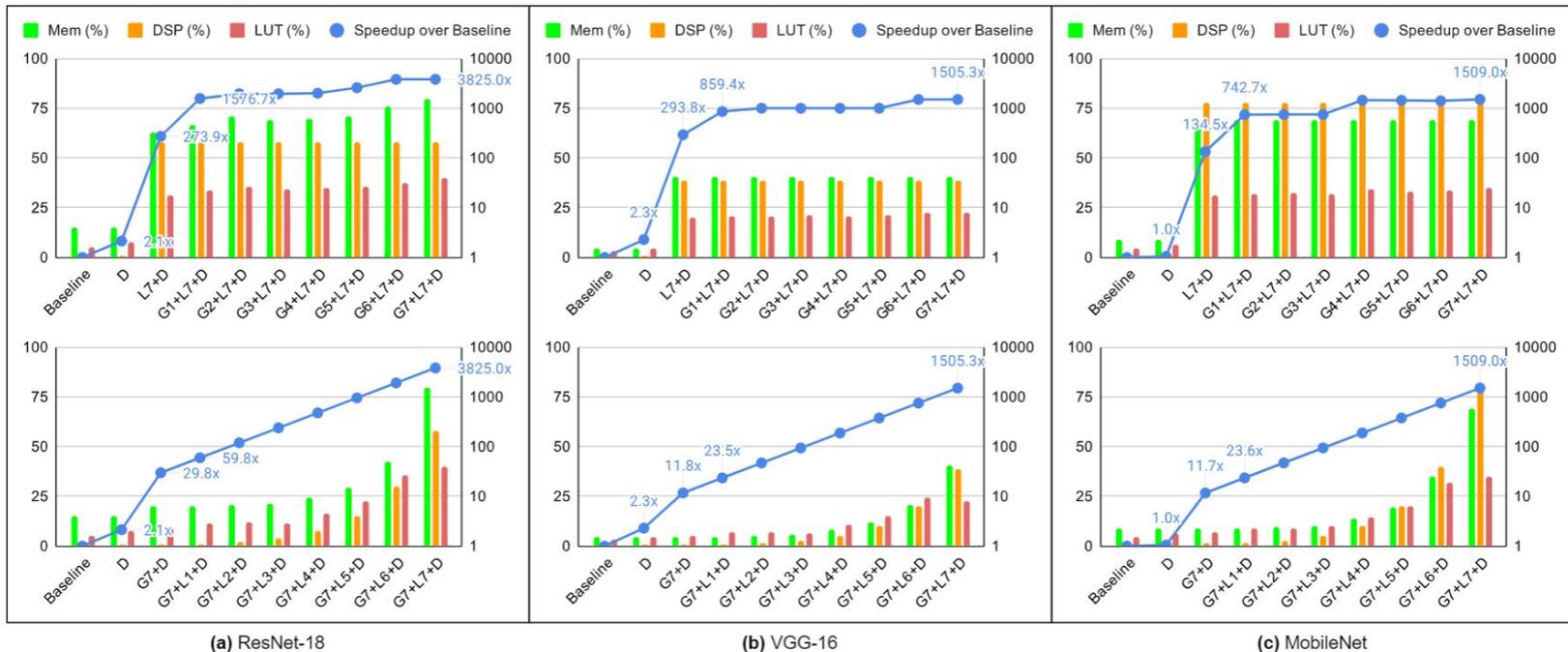
Optimization Results of DNN Models

Model	Speedup	Runtime (seconds)	Memory (SLR Util. %)	DSP (SLR Util. %)	LUT (SLR Util. %)	FF (SLR Util. %)	Our DSP Eff. (OPs/Cycle/DSP)	DSP Eff. 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.

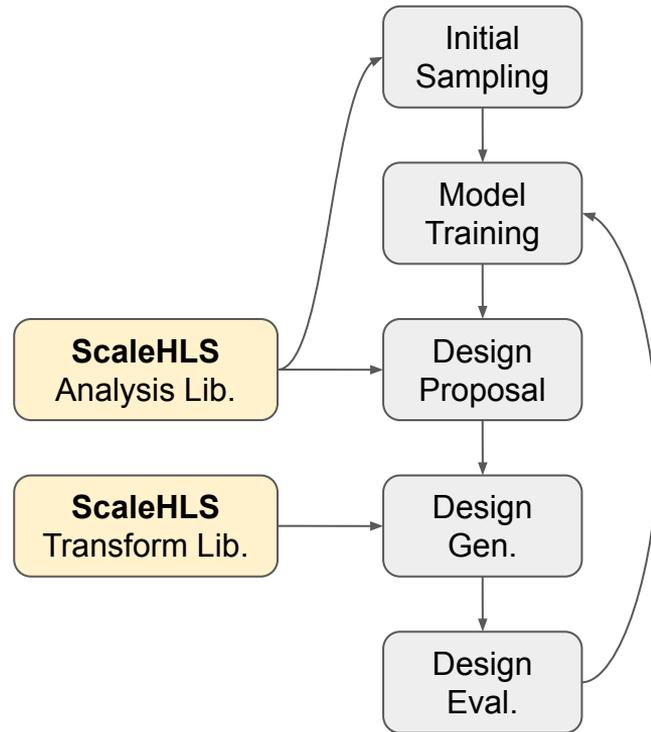
Optimization Results of DNN Models (Cont.)



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 \times , 130.9 \times , and 10.3 \times average speedups on the three DNN benchmarks, respectively.

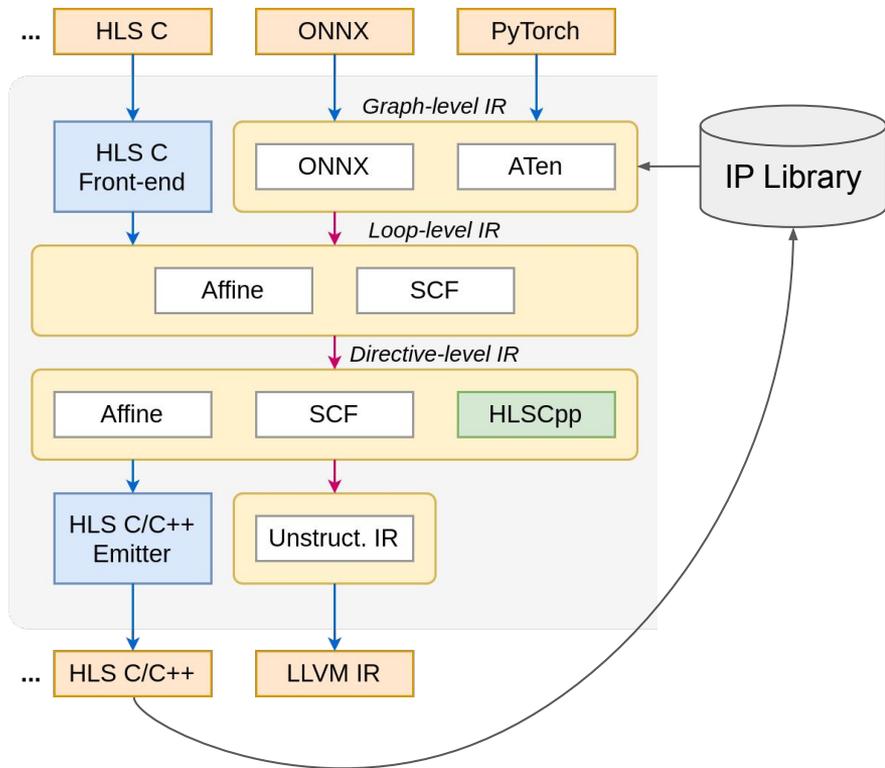
Future Directions (1) - Design Space Exploration



ML-based DSE

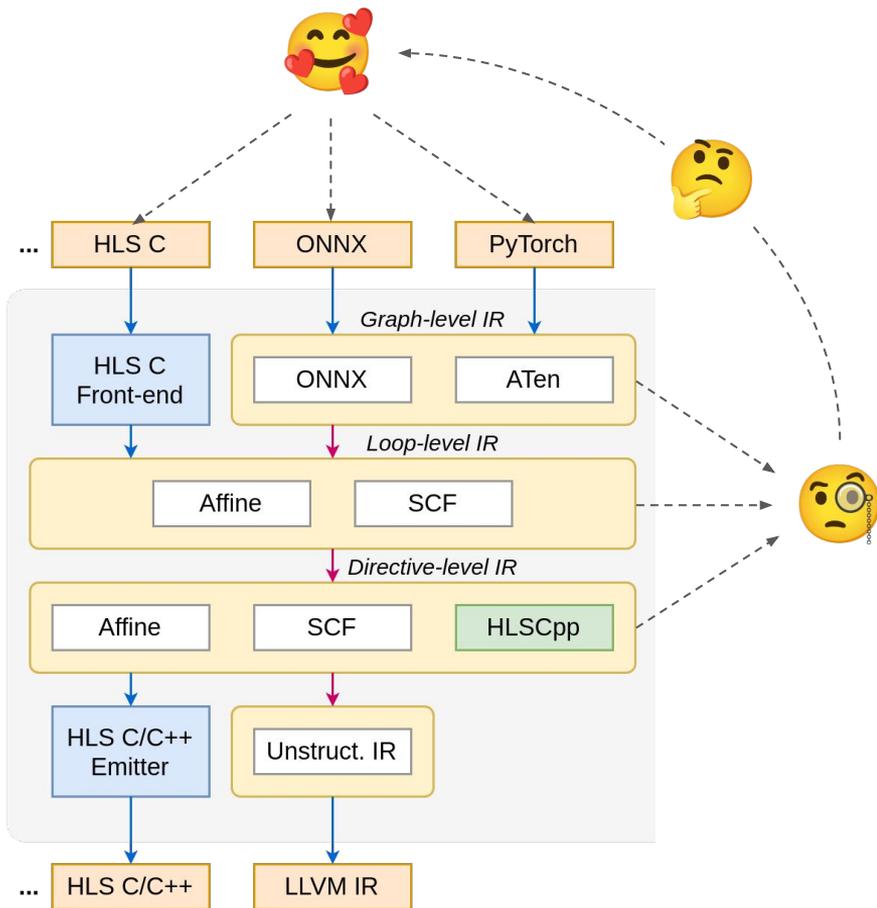
- Use ScaleHLS & MLIR analysis library to extract program information in order to initialize the design space and propose new design points.
- Predict the performance and resource utilization using machine learning model and train the model with the evaluated design points.
- Use ScaleHLS transform library to generate the proposed design.

Future Directions (2) - IP Integration



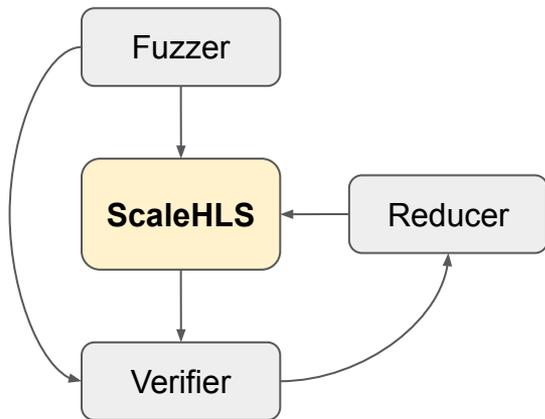
- Represent and integrate hardware IPs, such as Vitis Accelerated Libraries [1], at the graph level representation
- Parameterize hardware IPs with MLIR attributes
- Generate new hardware IPs through the ScaleHLS compilation flow
- Search optimized IP parameters in the global design space exploration

Future Directions (3) - Design Specialization



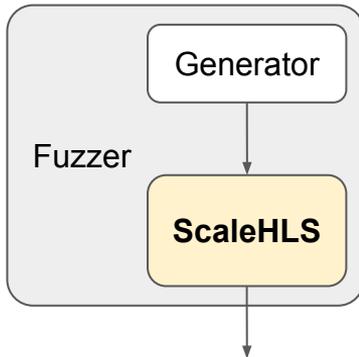
- Start with a basic design and add additional features, specializations, and optimizations step by step to approach the final design, keeping human designers in the loop
- High-level functionality in MLIR can be transformed and represented in lower-level details to show its potential for optimizing the whole design.
- Interaction can help designers to make informed decisions at the system level.

Future Directions (4) - Design Verification



Verifying ScaleHLS

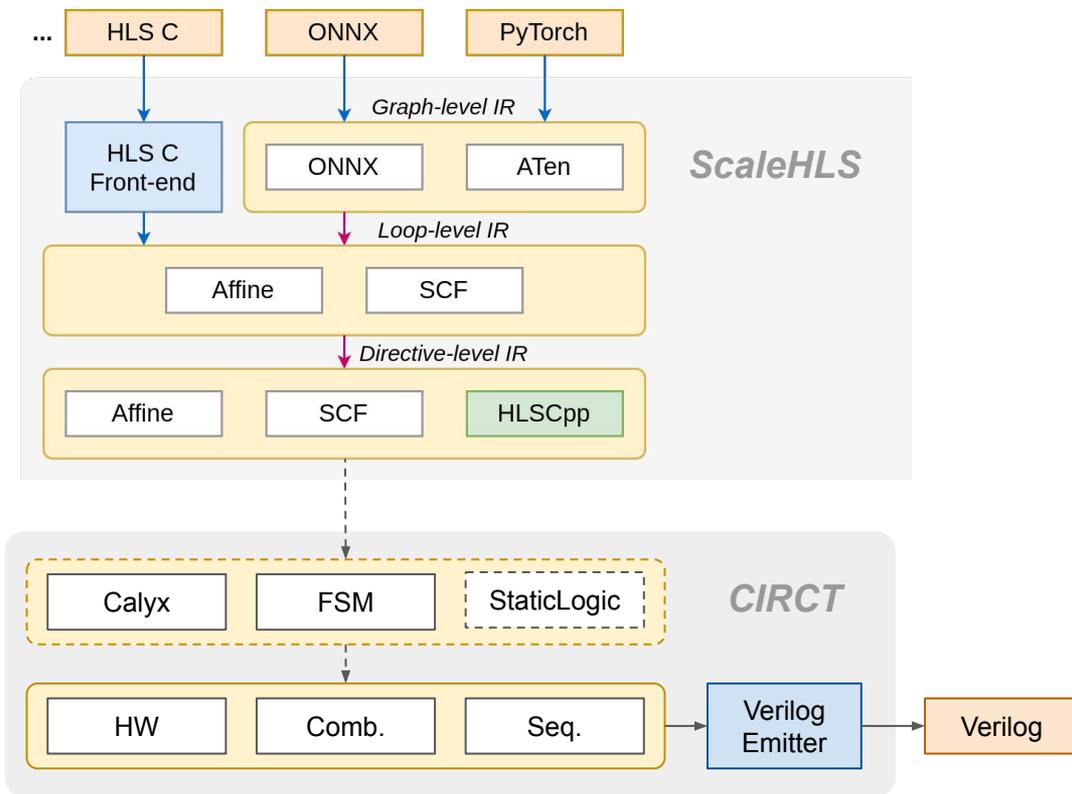
- Generate random program with C/C++ fuzzer, PyTorch fuzzer, etc.
- Verify the correctness of ScaleHLS conversions, optimizations, and code generation
- Reduce the targeted program to locate the bugs



Controlled Fuzzing with ScaleHLS

- HLS fuzzing is an emerging research topic investigating the verification of HLS tools
- We can cherry pick the transforms of ScaleHLS to improve the pertinence and quality of fuzzing

Future Directions (5) - Generate RTL within MLIR



- Currently ScaleHLS leverages external back-ends for generating the RTL code.
- A direct RTL code generation can keep more information from the higher level IR and exploit RTL-level optimizations (CIRCT [1]) to further improve the QoR of the accelerator designs.
- The directive-level IR in ScaleHLS can be converted to hardware description IRs in CIRCT and finally be emitted as SystemVerilog designs.

[1] CIRCT: Circuit IR Compilers and Tools. <https://github.com/llvm/circt/tree/main/>

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>

HPCA'22 Paper: <https://arxiv.org/abs/2107.11673>

Acknowledgement

We thank Eric Cheng of Laboratory for Physical Sciences (LPS) and Samuel Bayliss of Xilinx for insightful discussions. This work is supported in part by Xilinx Center of Excellence at UIUC, Xilinx Adaptive Compute Cluster (XACC) initiative, and BAH HT 15-1158 contract.



Thanks! Q&A

Nov. 29, 2021