ECE527 Lecture 14
MLIR, ScaleHLS, and HIDA

Hanchen Ye, Oct. 5

UNIVERSITY OF
ILLINOIS
URBANA-CHAMPAIGN

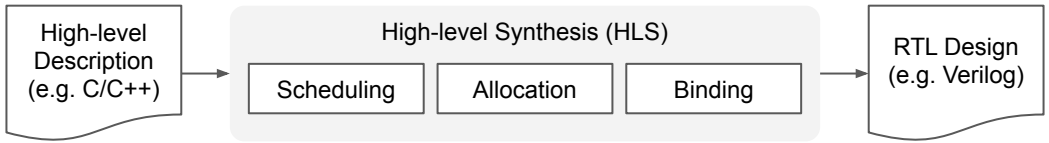
Outline

- Motivations
- Background: MLIR
- ScaleHLS Framework
- ScaleHLS Optimizations
- Design Space Exploration
- Evaluation Results
- HIDA (ScaleHLS 2.0)
- Conclusion

Outline

- **Motivations**
- Background: MLIR
- ScaleHLS Framework
- ScaleHLS Optimizations
- Design Space Exploration
- Evaluation Results
- HIDA (ScaleHLS 2.0)
- 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]

[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];
} } }
```

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.

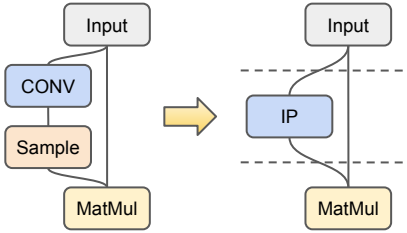
```
for (int k = 0; k < 32; k++) {
  for (int i = 0; i < 32; i++) {
    for (int j = 0; j < 32; j++) {
#pragma HLS pipeline
      if (k == 0)
        C[i][j] *= beta;
      C[i][j] += alpha * A[i][k] * B[k][j];
} } }
```

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:

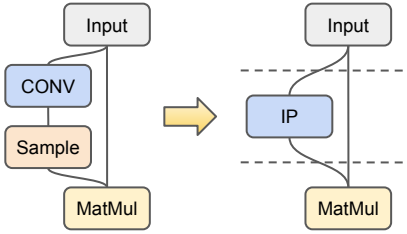
- Low-productive and error-proning
- Hard to enable automated design space exploration (DSE)
- NOT scalable!

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

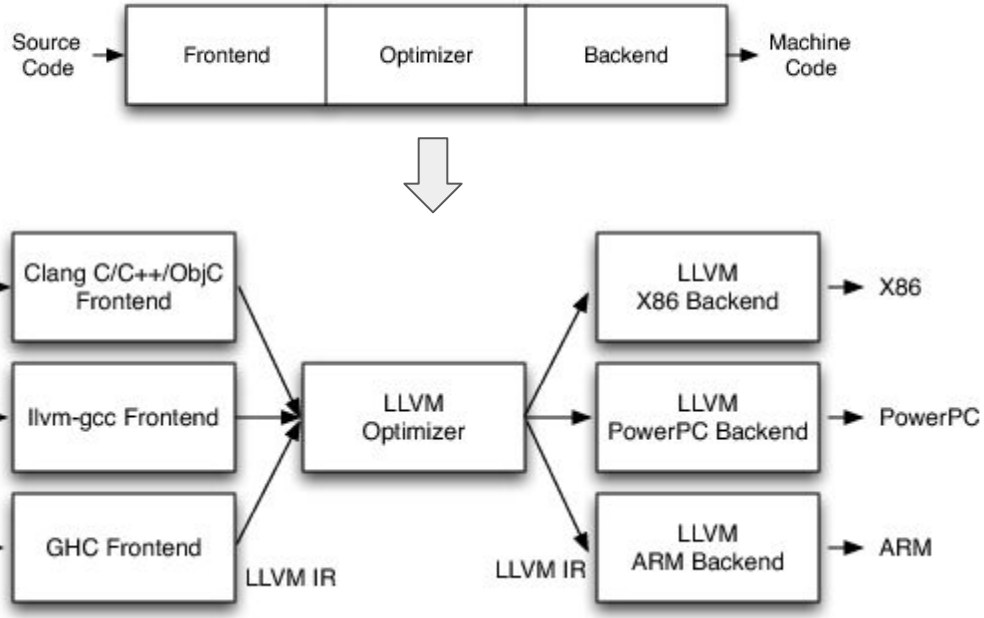
Generate RTL with XILINX VITIS and etc.

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

Outline

- Motivations
- **Background: MLIR**
- ScaleHLS Framework
- ScaleHLS Optimizations
- Design Space Exploration
- Evaluation Results
- HIDA (ScaleHLS 2.0)
- Conclusion

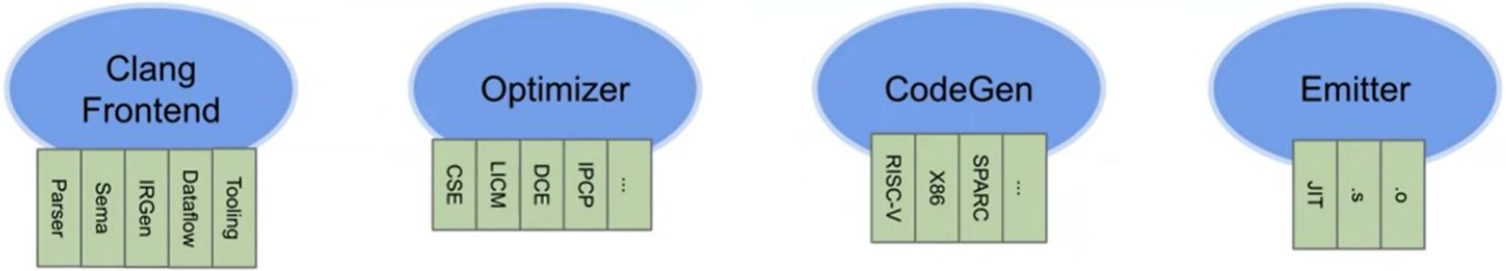
LLVM: Compiler Infrastructure



- LLVM uses the same **intermediate representation (IR)** to represent ALL programs.
- All program optimizations are based on the LLVM IR.
- LLVM dispatches the front-ends, optimizations, and back-ends. O(m*n) -> O(1)

Source: The architecture of open-source applications, C. Lattner.

LLVM: Compiler Infrastructure (Cont'd)



Key insight: Compilers as libraries, not an app!

- Enable embedding in other applications
- Mix and match components
- No hard coded lowering pipeline

Source: The Golden Age of Compiler Design in an Era of HW/SW Co-design, C. Lattner.

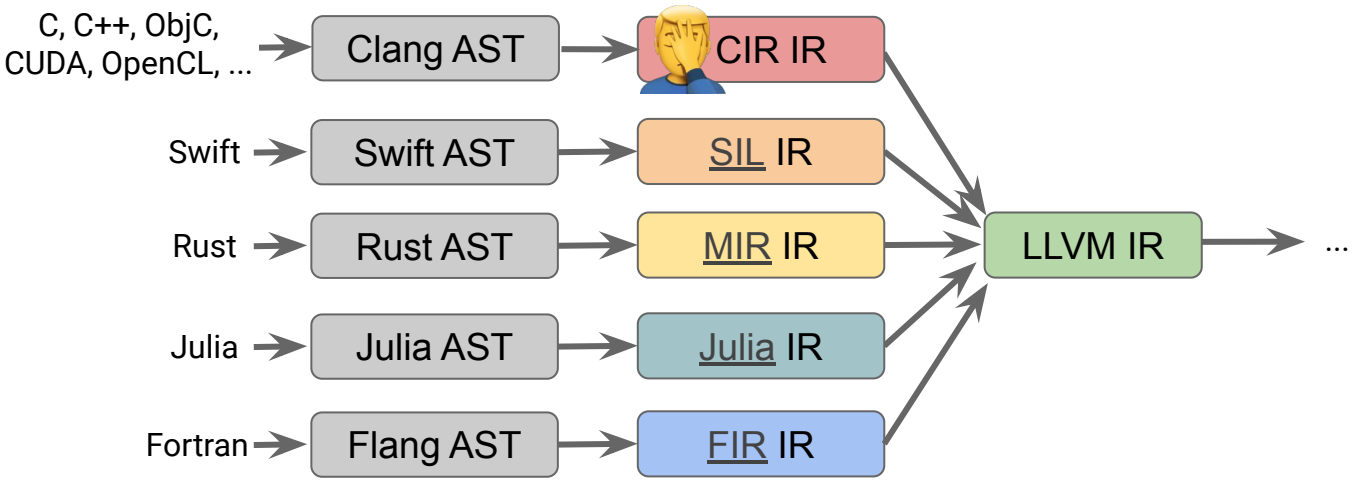
LLVM: Compiler Infrastructure (Cont'd)

What is LLVM?

- An open source framework for building tools
  - Tools are created by linking together various libraries provided by the LLVM project and your own
- An extensible, strongly typed intermediate representation, i.e. LLVM IR
  - https://llvm.org/docs/LangRef.html
- An industrial strength C/C++ optimizing compiler
  - Which you might know as clang/clang++ but these are really just drivers that invoke different parts (libraries) of LLVM

Source: Introduction to LLVM, UG3 Compiling Techniques.

From LLVM to MLIR



- More and more programming languages demand customized IR for optimization.
- The IR for different languages have different abstraction level.
- Language-specific IR can be lowered to LLVM for back-end code generation.

Source: MLIR: Multi-Level Intermediate Representation Compiler Infrastructure, C. Lattner.

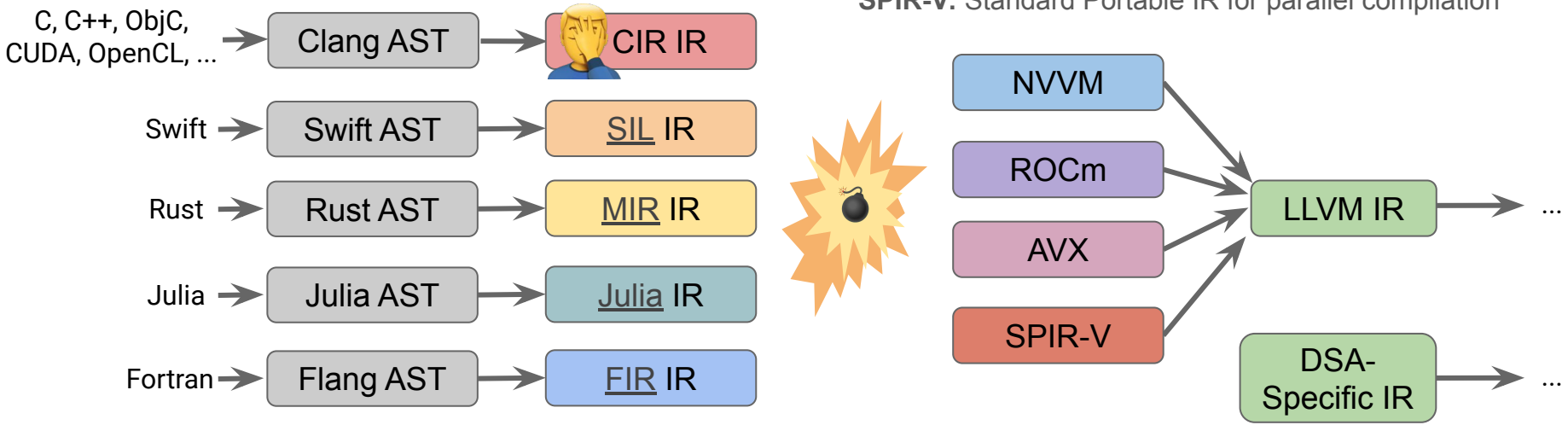
From LLVM to MLIR (Cont'd)

NVVM: IR for Nvidia GPU

ROCm: IR for AMD GPU

AVX: IR for Intel vector extension

SPIR-V: Standard Portable IR for parallel compilation



- Different back-ends demand customized IR for optimization
- DSAs (Domain-Specific Accelerator) even cannot use LLVM for generating back-end codes and demand their own IR for code generation

Severe Fragmentation: IRs have different implementations and "frameworks"

MLIR: Compiler Infrastructure for the End of Moore's Law

- **M**ulti-**L**evel **I**ntermediate **R**epresentation
- State of the art compiler technology
- Built on top of LLVM's open and library-based philosophy
- **Modular and extensible**
- Originally created within Google for compiling TensorFlow
- **Sufficiently general** to compile lots of domains

https://mlir.llvm.org

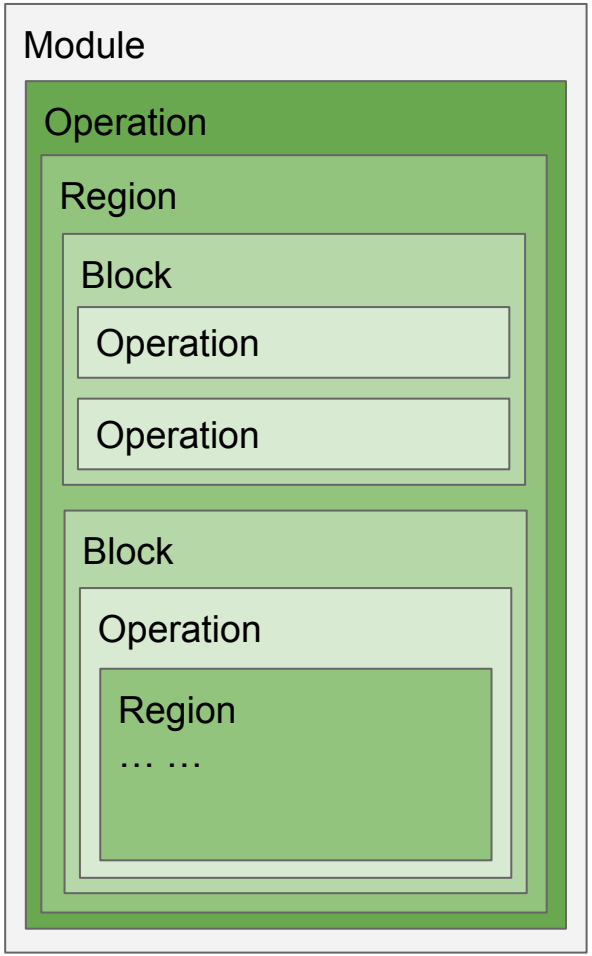
Source: Applying Circuit IR Compilers and Tools (CIRCT) to ML Applications, M. Urbach.

Syntax of MLIR

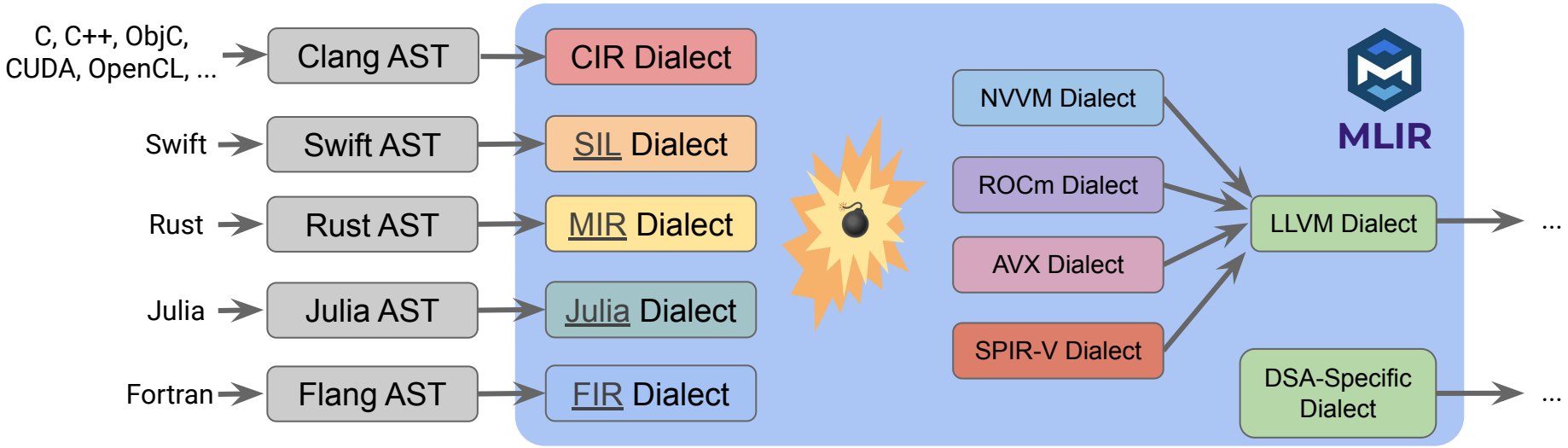
- SSA-based IR design, explicit typing system
- Module/Operation/Region/Block/Operation hierarchy
- Operation can contain multiple Regions

```
func.func @testFunction(%arg0: i32) -> i32 {
  %a = func.call @thingToCall(%arg0) : (i32) -> i32
  cf.br ^bb1
^bb1:
  %c = affine.for %i = 0 to 10 iter_args(%b = %a) -> i32 {
    %i_i32 = arith.index_cast %i : index to i32
    %b_new = arith.addi %i_i32, %b : i32
    affine.yield %b_new : i32
  }
  func.return %c : i32
}
```

Dialect A C++ namespace that contains customized operations, types, and attributes. Implement the “correct” abstraction for your domain.



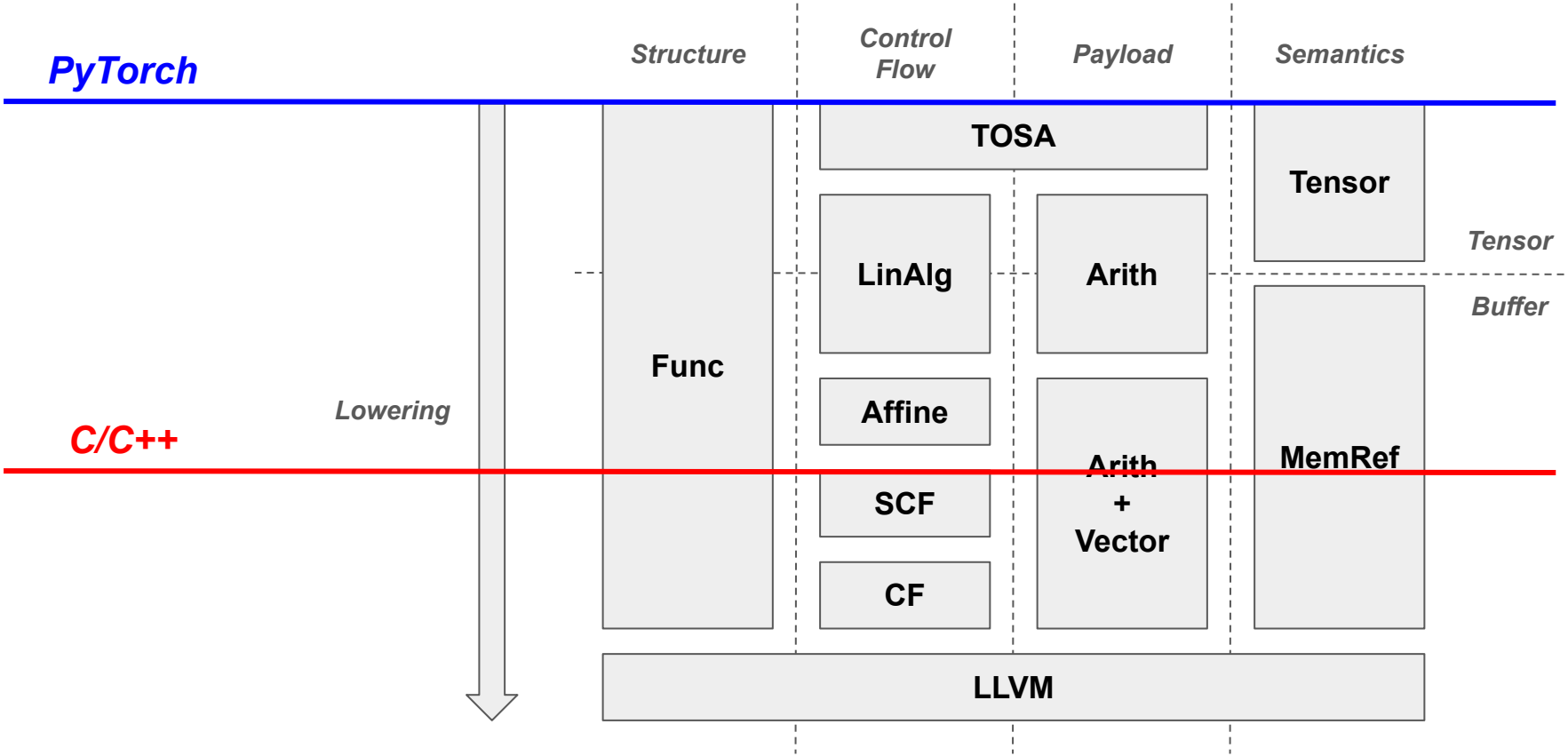
MLIR: “Meta IR” and Compiler Infrastructure



MLIR is a **“Meta IR”** and **compiler infrastructure** for:

- Design and implement **dialect**
- Optimization and transform inside of a **dialect**
- Conversion between different **dialects**
- Code generation of **dialect**

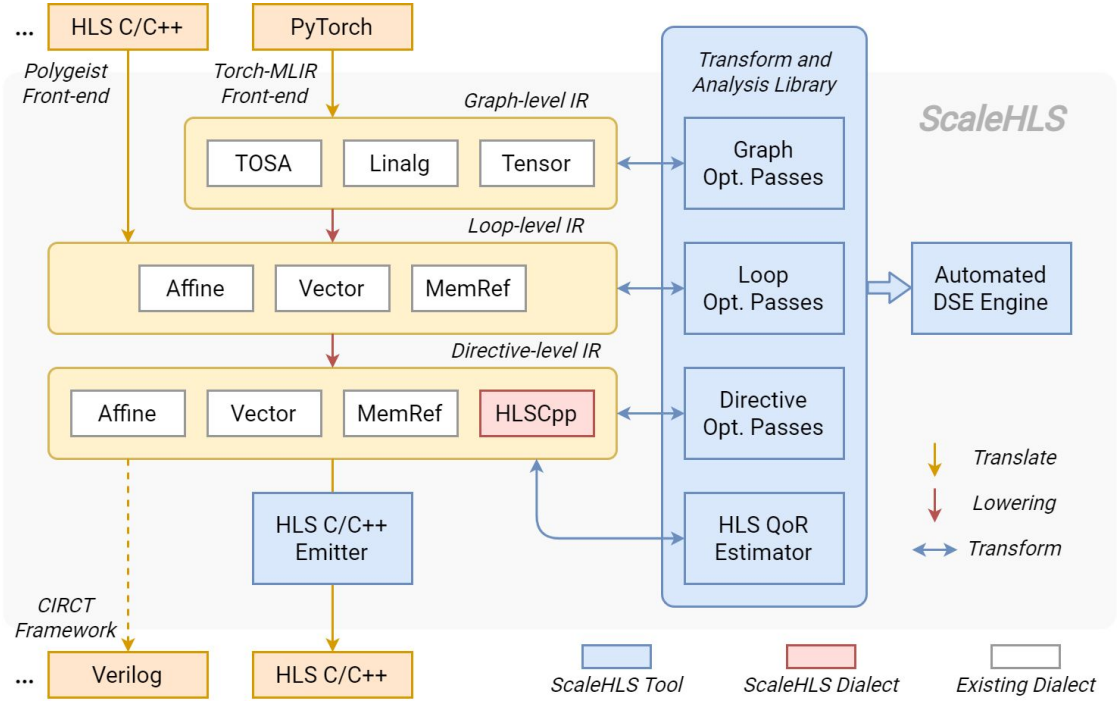
MLIR: “Meta IR” and Compiler Infrastructure (Cont’d)



Outline

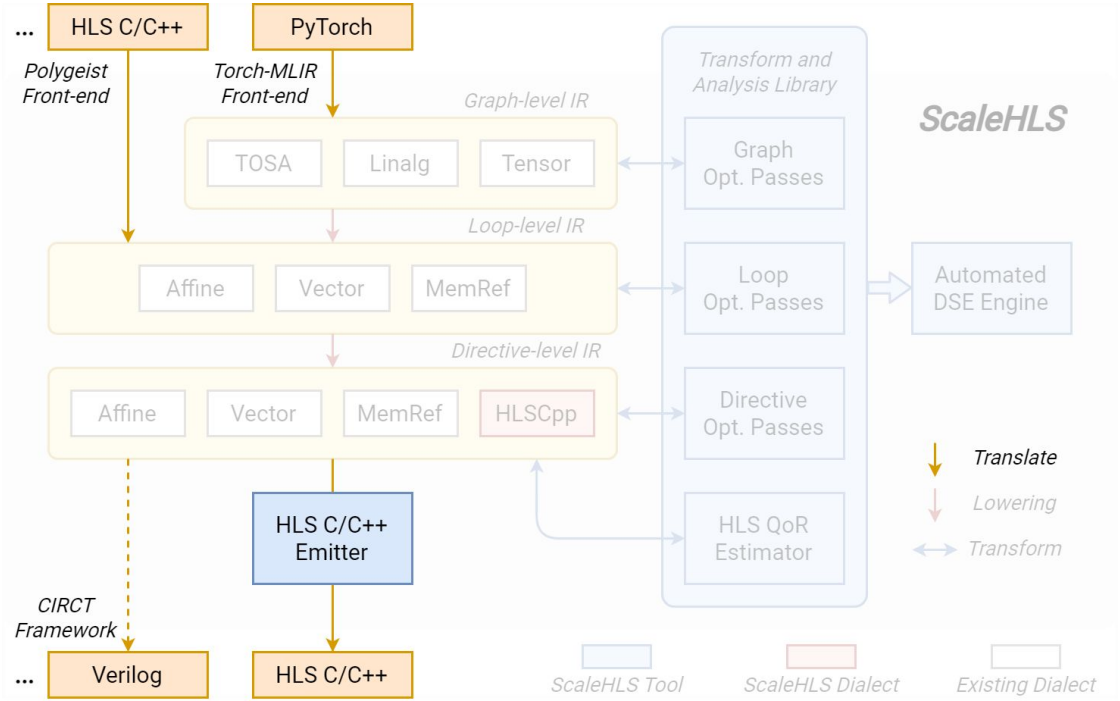
- Motivations
- Background: MLIR
- **ScaleHLS Framework**
- ScaleHLS Optimizations
- Design Space Exploration
- Evaluation Results
- HIDA (ScaleHLS 2.0)
- Conclusion

ScaleHLS Framework: Integration



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

ScaleHLS Framework: Integration (Cont'd)



Inputs

C/C++ Polygeist [1]

PyTorch Torch-MLIR [2]

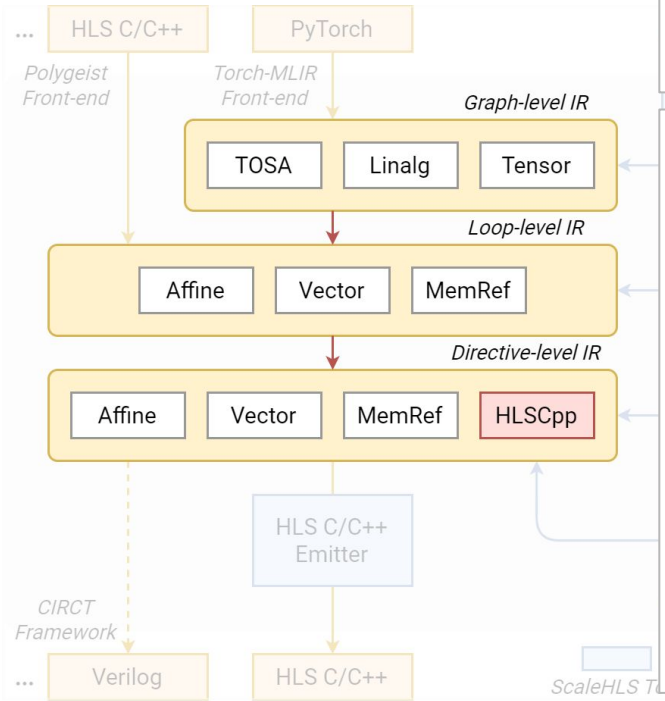
Outputs

C/C++ C/C++ Emitter

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

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

ScaleHLS Framework: Representation



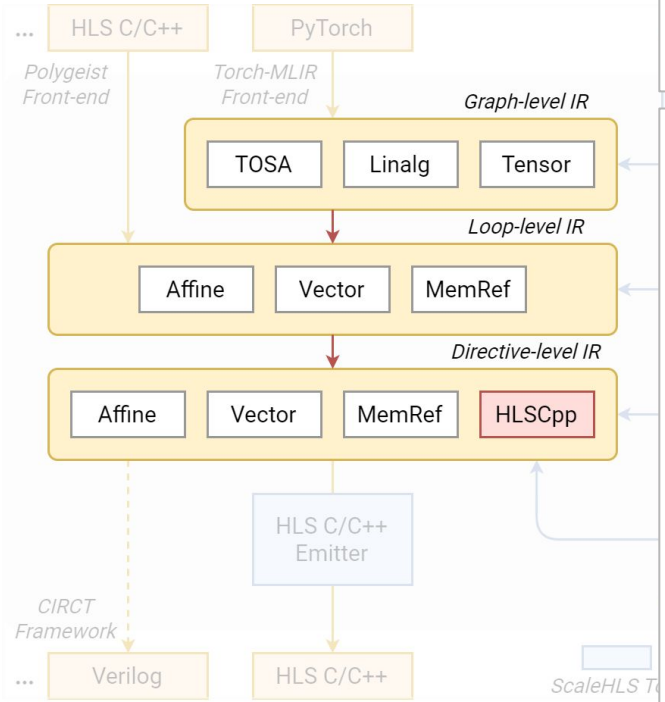

```
%O = "tosa.conv2d"(%I, %W, ...) {...} : (tensor<1x32x32x3xi8>, tensor<64x3x3x3xi8>, ...) ->
                                        tensor<1x30x30x64xi8>
```

Conv2D Graph-level MLIR

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

Conv2D Loop-level MLIR

ScaleHLS Framework: Representation (Cont'd)



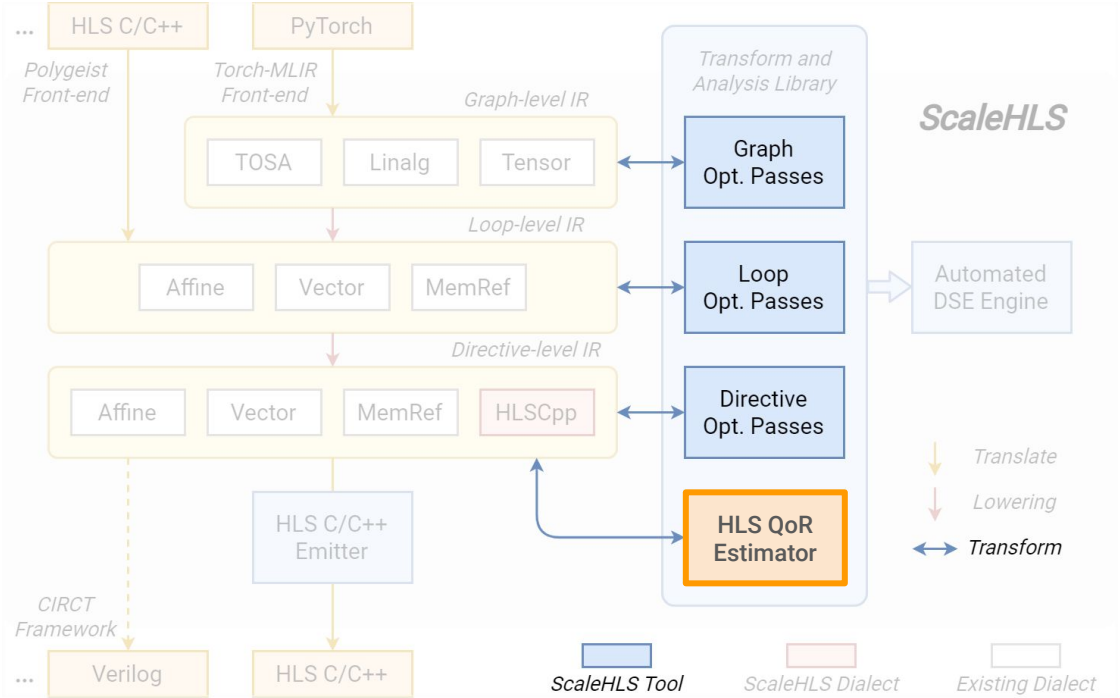

```
%O = "tosa.conv2d"(%I, %W, ...) {...} : (tensor<1x32x32x3xi8>, tensor<64x3x3x3xi8>, ...) ->
                                          tensor<1x30x30x64xi8>
```

Conv2D Graph-level MLIR

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

Conv2D Directive-level MLIR

ScaleHLS Framework: Optimization



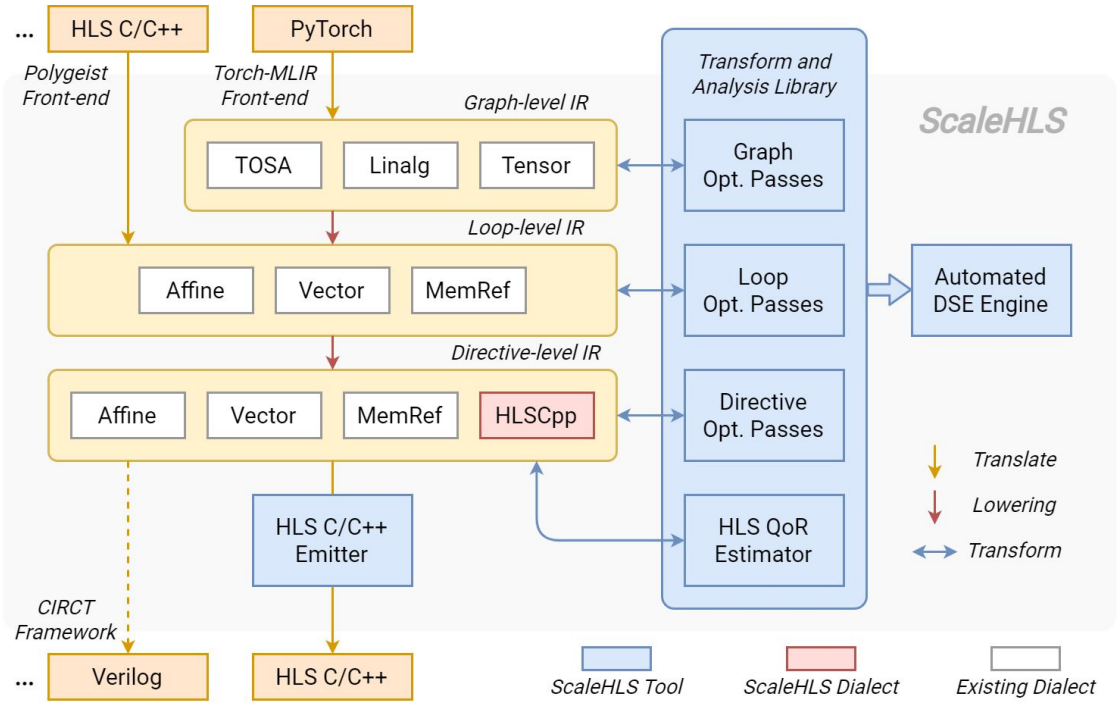
Level	ScaleHLS Passes

Graph	`-simplify-tosa-graph`
	`-legalize-dataflow`
	`-split-function`

Loop	`-affine-loop-perfectization`
	`-remove-variable-bound`
	`-affine-loop-tile`
	`-affine-loop-order-opt`
	`-affine-loop-unroll-jam`
	`-simplify-affine-if`
	`-affine-store-forward`
	`-simplify-memref-access`

Directive	`-loop-pipelining`
	`-function-pipelining`
	`-array-partition`
	`-create-hlscpp-primitive`
	`-qor-estimation`

ScaleHLS Framework



Represent It!

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

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

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

Optimize It!

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

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

Explore It!

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

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

Enable End-to-end Flow!

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

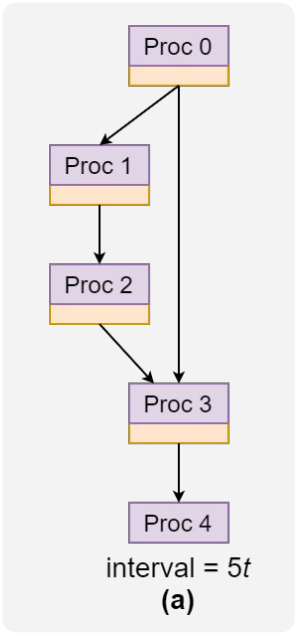
HLS C/C++ Emitter: Generate synthesizable HLS designs for downstream tools, such as Vivado HLS.

Outline

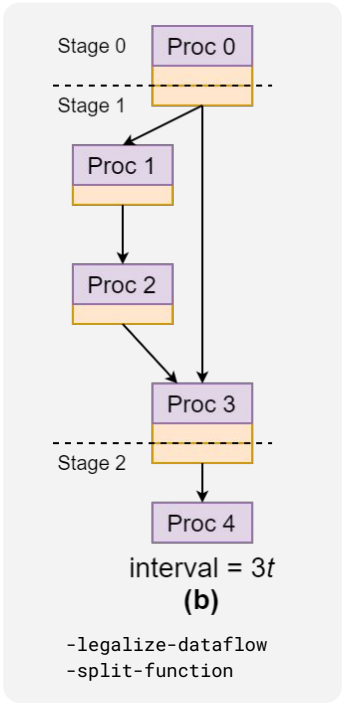
- Motivations
- Background: MLIR
- ScaleHLS Framework
- **ScaleHLS Optimizations**
- Design Space Exploration
- Evaluation Results
- HIDA (ScaleHLS 2.0)
- Conclusion

ScaleHLS Optimizations

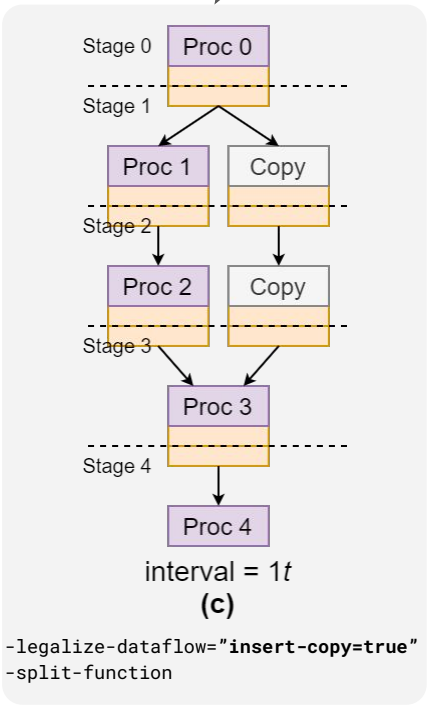
	Passes	Target	Parameters
Graph	-legalize-dataflow	function	insert-copy
	-split-function	function	min-gran

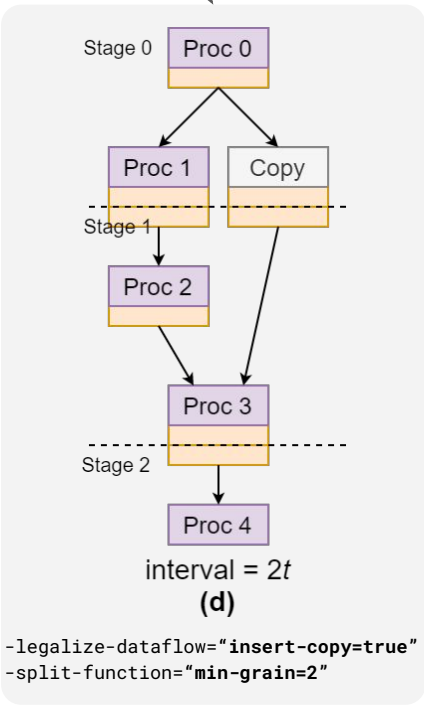


Coarse-grained Pipelining
(dataflow pragma)



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





ScaleHLS Optimizations (Cont.)

	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.

```c
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

```c
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];
} } } }
```

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 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];
} } } }
```

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

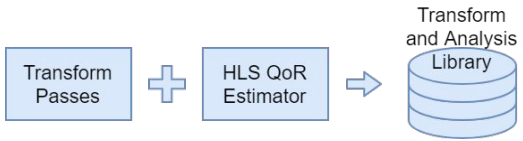
Simplify if ops;
Store ops forward;
Simplify memref ops

Optimized C emitted by the C/C++ emitter

ScaleHLS 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];
} } } }
```

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

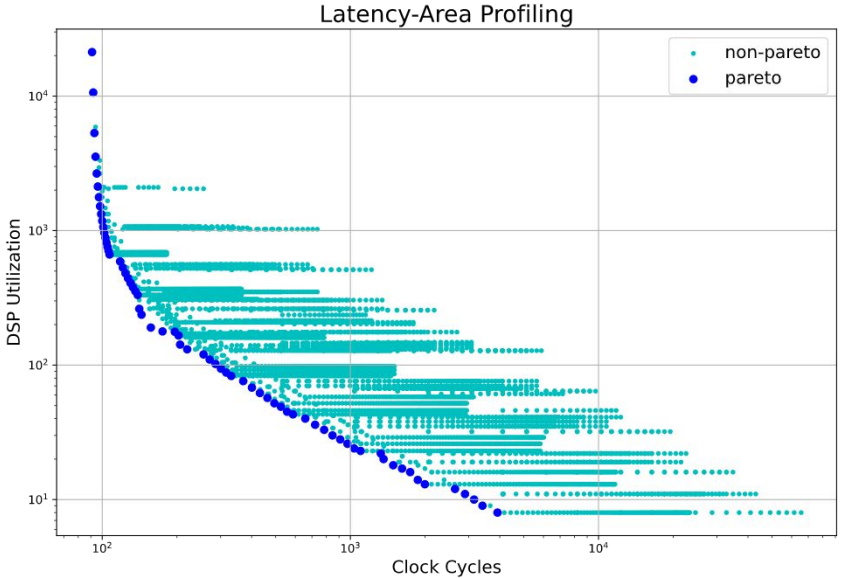
Simplify if ops;
Store ops forward;
Simplify memref ops

Optimized C emitted by the C/C++ emitter

Outline

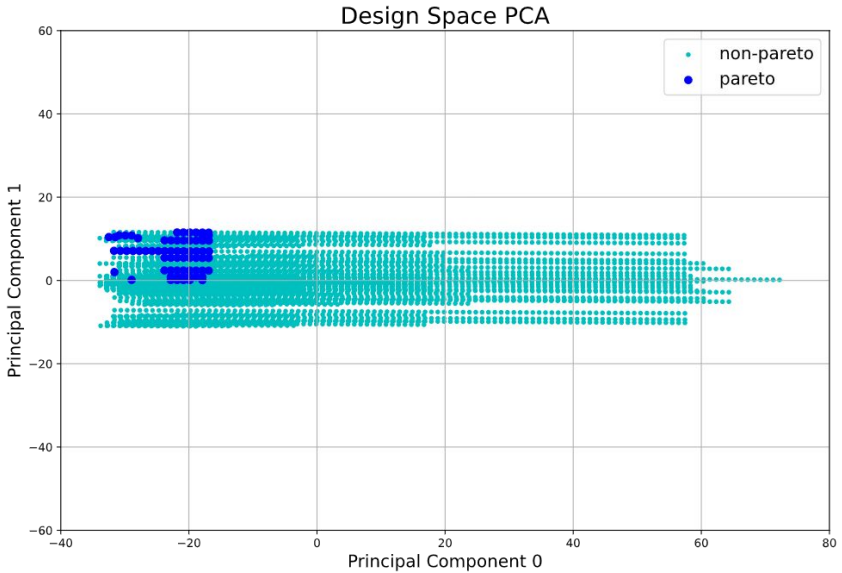
- Motivations
- Background: MLIR
- ScaleHLS Framework
- ScaleHLS Optimizations
- **Design Space Exploration**
- Evaluation Results
- HIDA (ScaleHLS 2.0)
- Conclusion

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

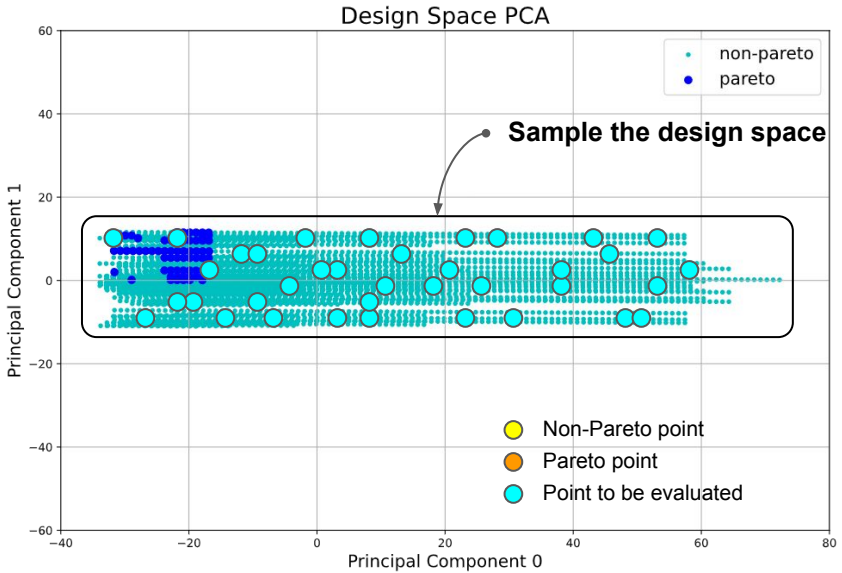


- Each parameter of a pass becomes one dimension, the original 4-dimensional design space is reduced to two dimensions through PCA
- Pareto points are located at a corner of the design space, the variance of Pareto points is much smaller than the overall variance

Design Space Exploration (Cont.)

DSE algorithm:

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

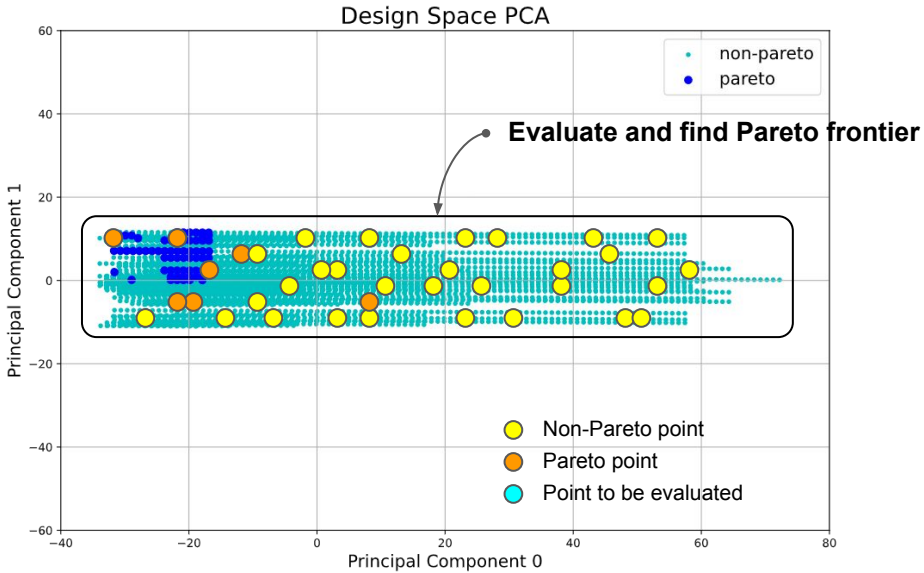


- Each parameter of a pass becomes one dimension, the original 4-dimensional design space is reduced to two dimensions through PCA
- Pareto points are located at a corner of the design space, the variance of Pareto points is much smaller than the overall variance

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

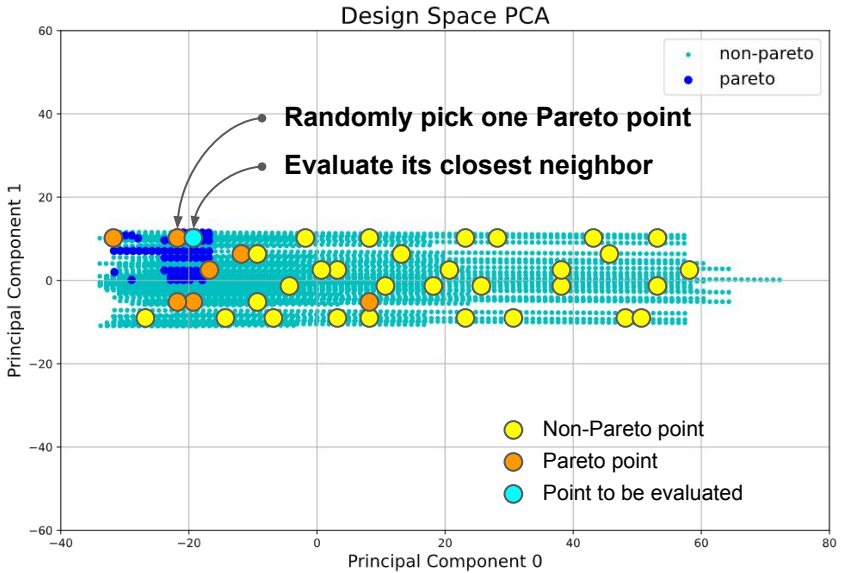


- Each parameter of a pass becomes one dimension, the original 4-dimensional design space is reduced to two dimensions through PCA
- Pareto points are located at a corner of the design space, the variance of Pareto points is much smaller than the overall variance

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

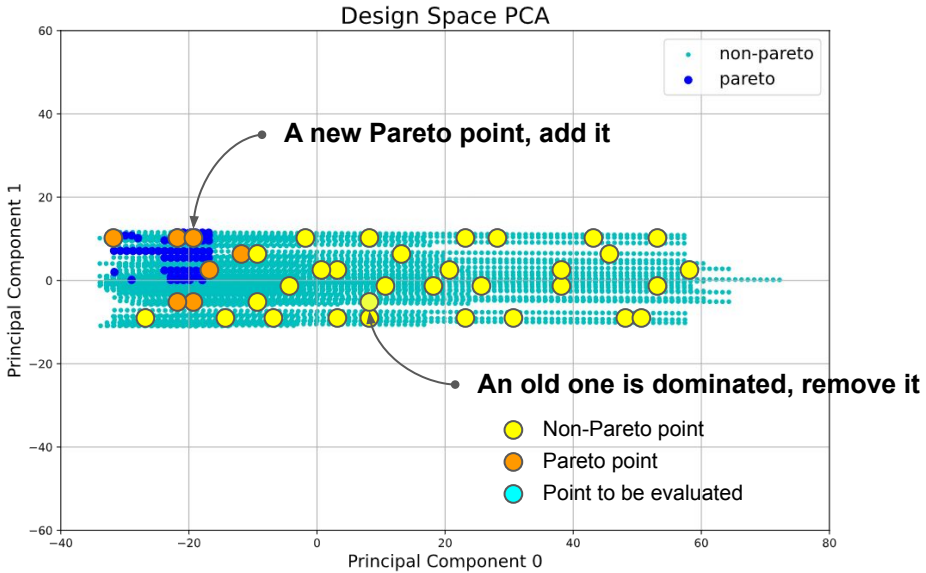


- Each parameter of a pass becomes one dimension, the original 4-dimensional design space is reduced to two dimensions through PCA
- Pareto points are located at a corner of the design space, the variance of Pareto points is much smaller than the overall variance

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



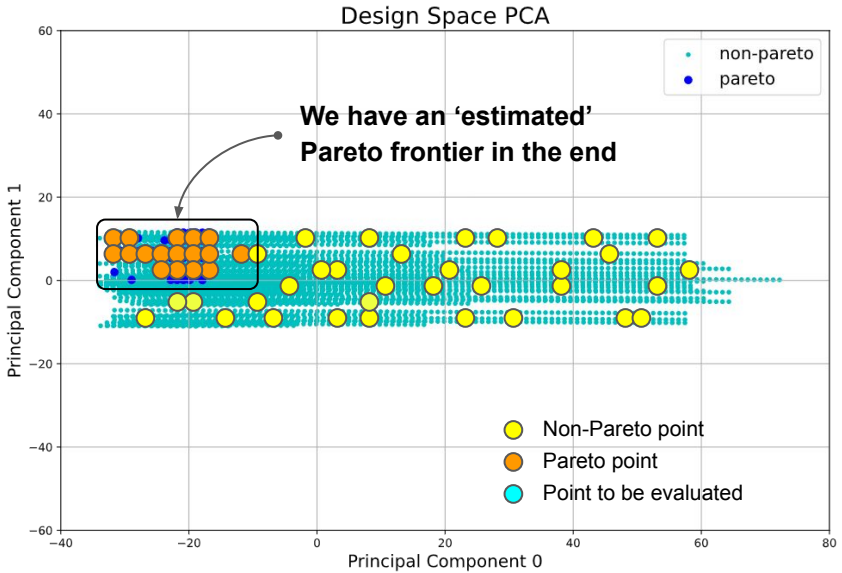
- Each parameter of a pass becomes one dimension, the original 4-dimensional design space is reduced to two dimensions through PCA
- Pareto points are located at a corner of the design space, the variance of Pareto points is much smaller than the overall variance

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

Outline

- Motivations
- Background: MLIR
- ScaleHLS Framework
- ScaleHLS Optimizations
- Design Space Exploration
- **Evaluation Results**
- HIDA (ScaleHLS 2.0)
- Conclusion

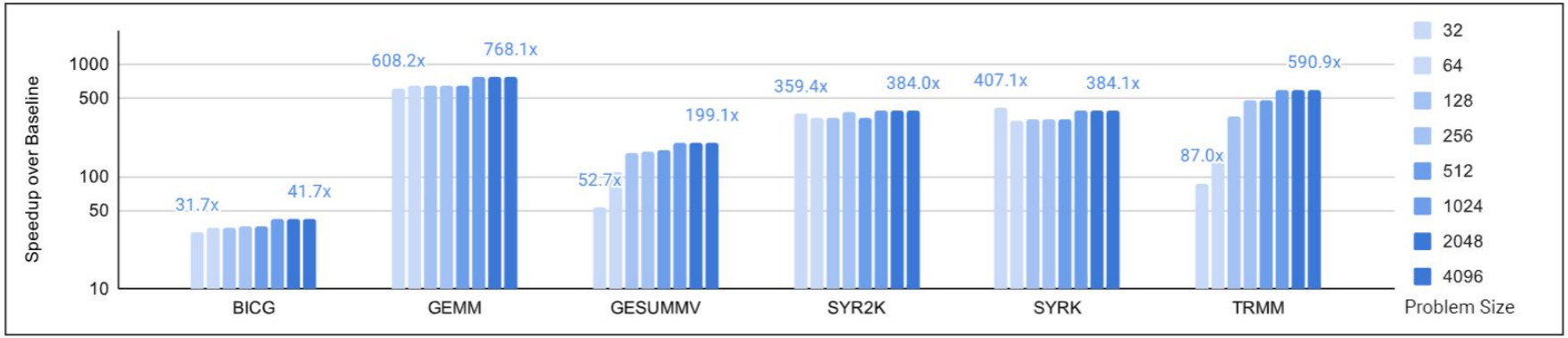
DSE Results of Computation Kernel

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.

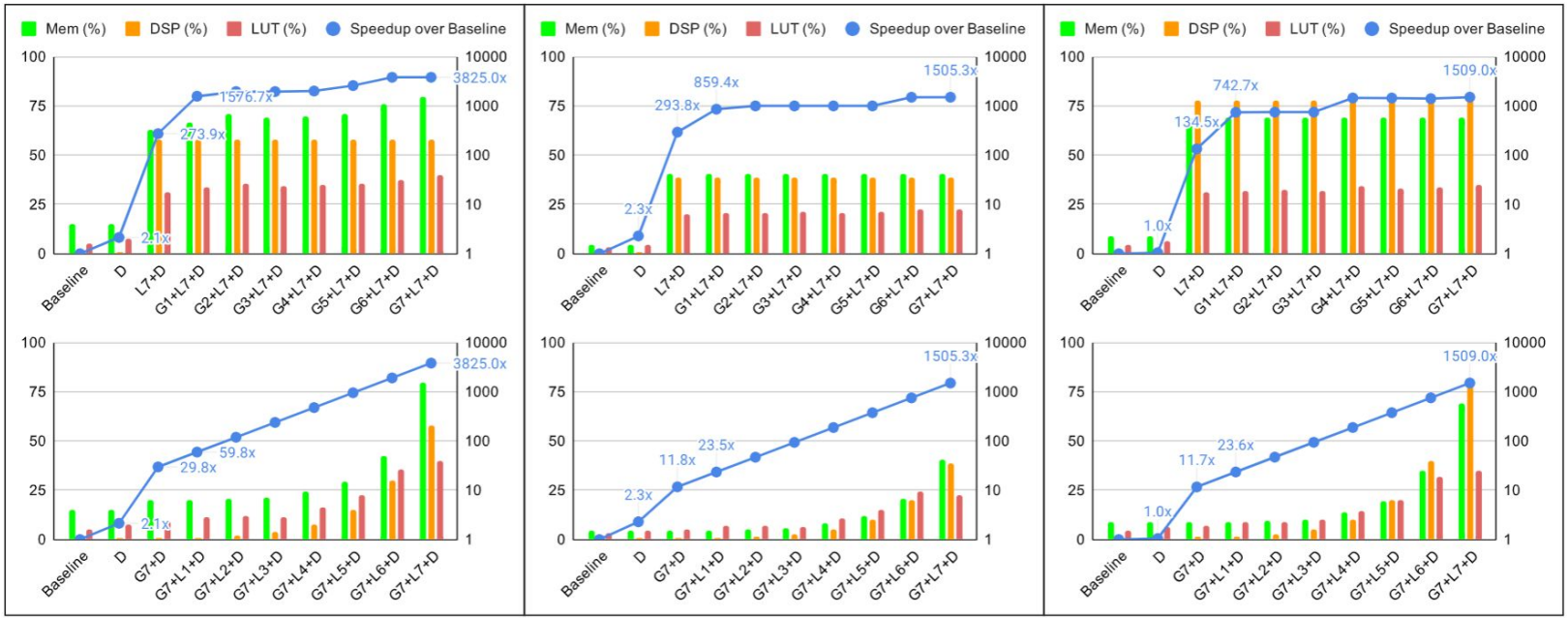
Optimization Results of DNN Models

Model	Speedup	Runtime (seconds)	Memory (SLR Util. %)	DSP (SLR Util. %)	LUT (SLR Util. %)	FF (SLR Util. %)	Our DSP 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

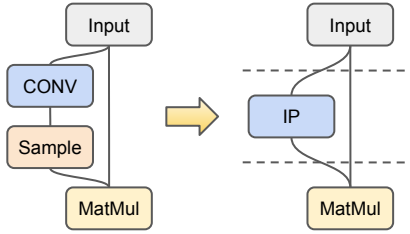
Ablation study of DNN models

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.

Outline

- Motivations
- Background: MLIR
- ScaleHLS Framework
- ScaleHLS Optimizations
- Design Space Exploration
- Evaluation Results
- **HIDA (ScaleHLS 2.0)**
- Conclusion

Limitation of ScaleHLS



Graph Optimizations

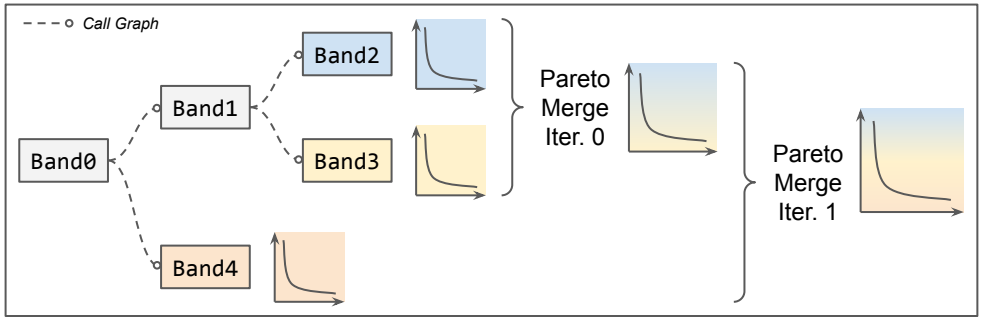
```
for (int i = 0; i < 32; i++) {
  for (int j = 0; j < 32; j++) {
    C[i][j] *= beta;
    for (int k = 0; k < 32; k++) {
      C[i][j] += alpha * A[i][k] * B[k][j];
} } }
```

Loop Optimizations

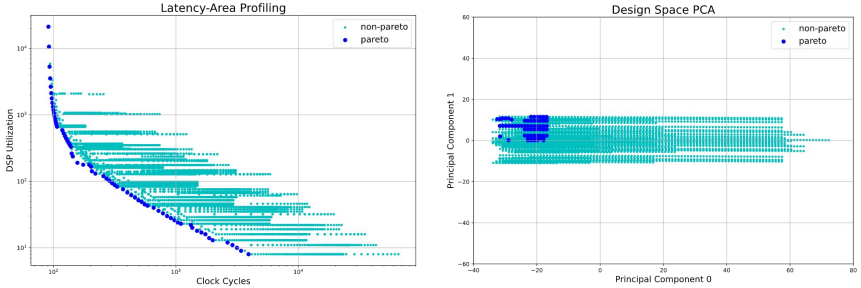
```
for (int k = 0; k < 32; k++) {
  for (int i = 0; i < 32; i++) {
    for (int j = 0; j < 32; j++) {
      if (k == 0)
        C[i][j] *= beta;
      C[i][j] += alpha * A[i][k] * B[k][j];
} } }
```

Directive Optimizations

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



Step (2) Global multi-kernel Pareto curving merging



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

Limitation of ScaleHLS (Cont'd)

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

Inter-kernel Correlation

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

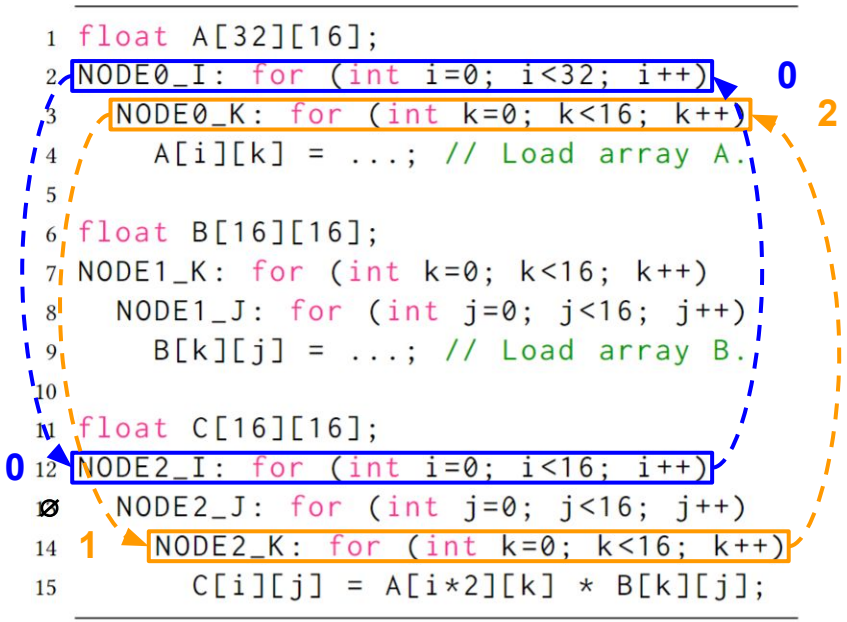
Connectedness

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

Intensity

Simply merging the local Pareto curves will not work well!

What we did in HIDA

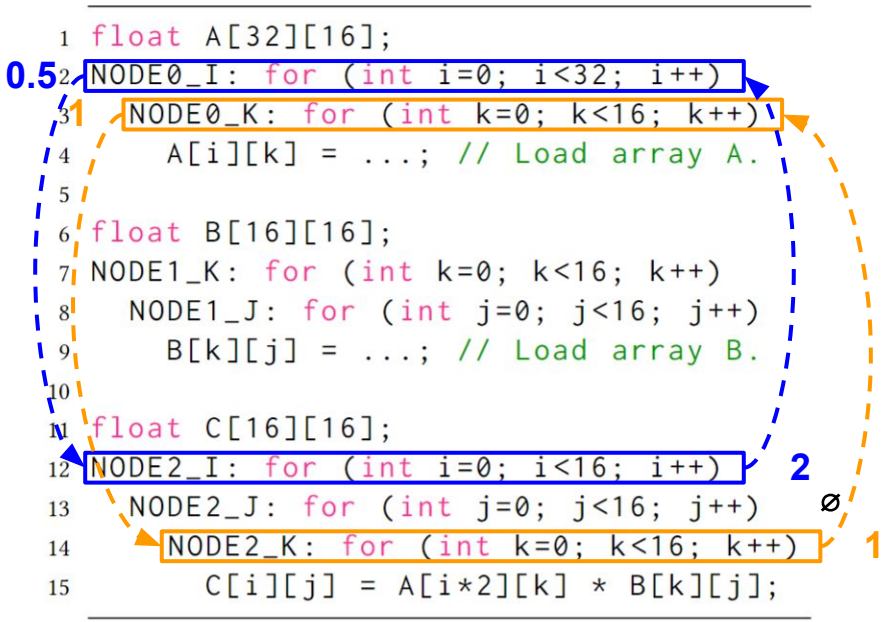


Step (1) Connectedness Analysis

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

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

What we did in HIDA (Cont'd)



Step (1) Connectedness Analysis

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

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

What we did in HIDA (Cont'd)

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

Step (2) Node Sorting

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

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

What we did in HIDA (Cont'd)

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

Step (3) Node Parallelization

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

What we did in HIDA (Cont'd)

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

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

Step (3) Node Parallelization

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

What we did in HIDA (Cont'd)

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

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

Step (3) Node Parallelization

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

What we did in HIDA (Cont'd)

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

Step (3) Node Parallelization

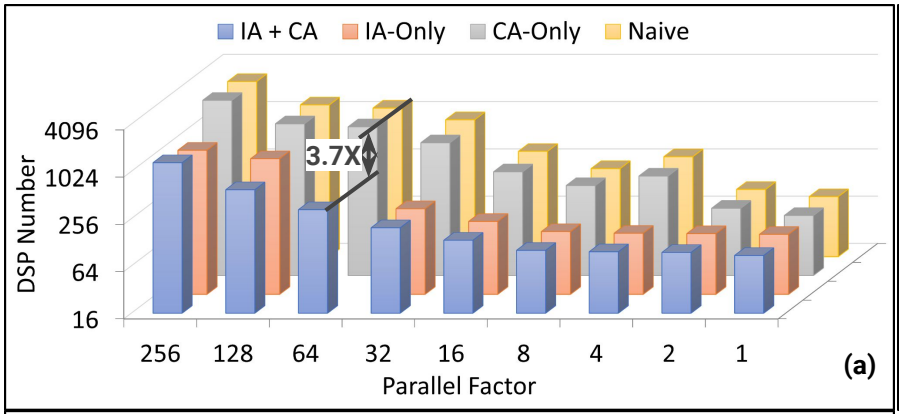
Node	Intensity	Parallel Factor		Loop Unroll Factors			
		w/o IA	w/ IA	IA+CA	IA	CA	Naive
Node0	512	32	4	[4, 1]	[2, 2]	[8, 4]	[4, 8]
Node1	256	32	2	[1, 2]	[1, 2]	[4, 8]	[4, 8]
Node2	4,096	32	32	[4, 8, 1]	[4, 8, 1]	[4, 8, 1]	[4, 8, 1]

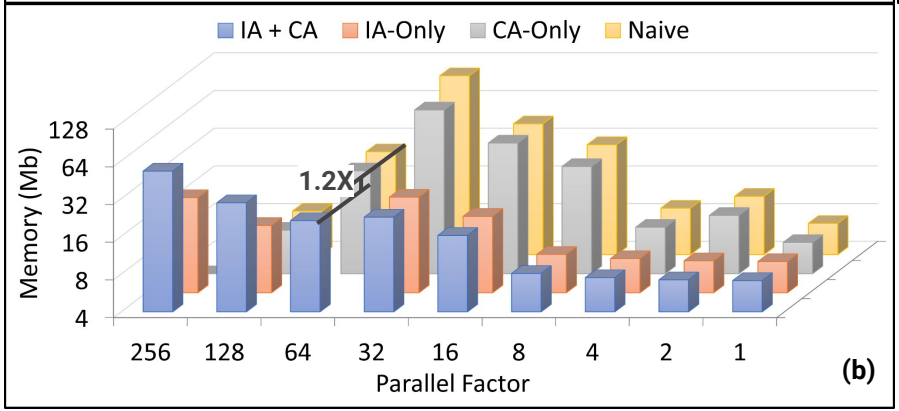
Intensity-aware (IA) Connectedness-aware (CA) HIDA DSE

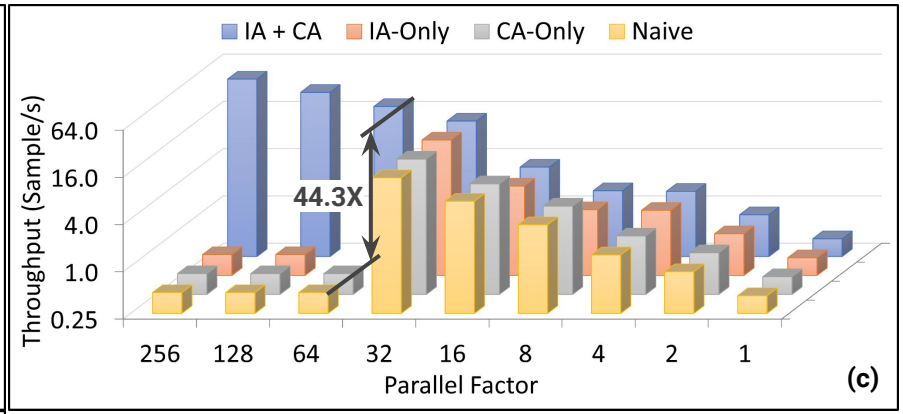
Naive ScaleHLS DSE

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

ResNet-18 Ablation Study on HIDA

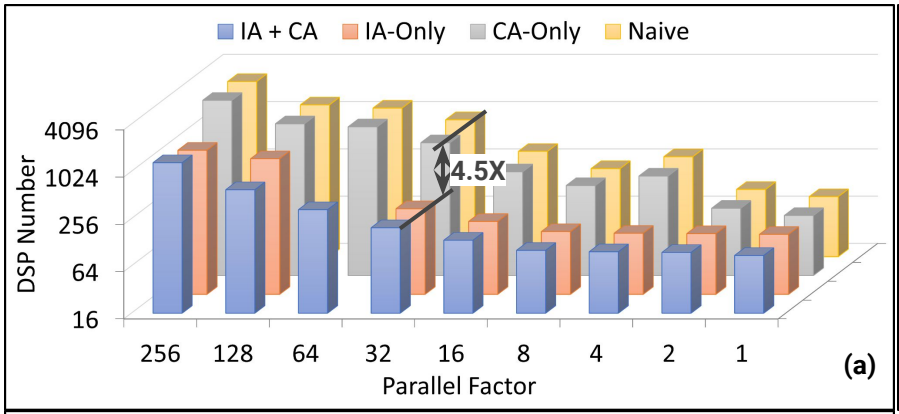


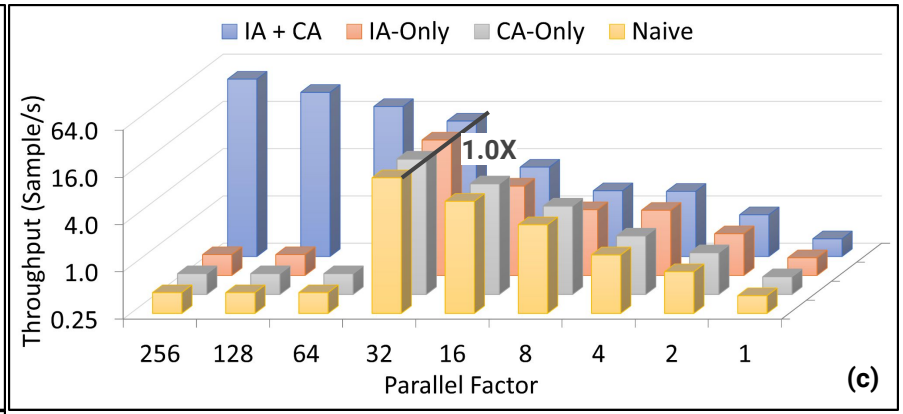


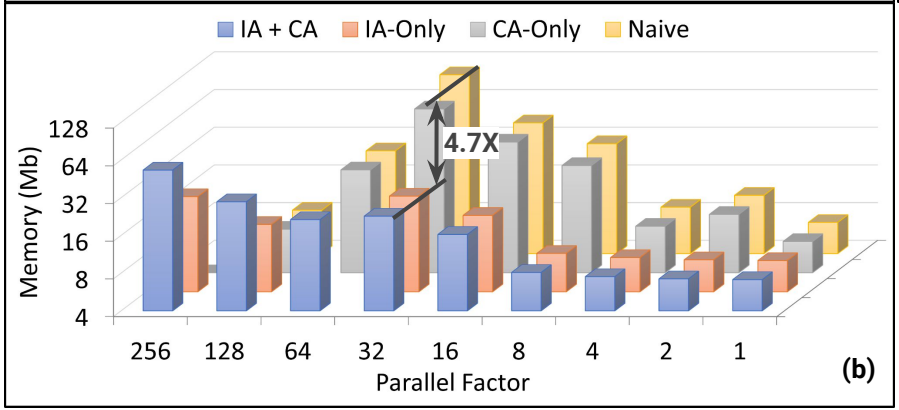


- IA+CA parallelization can determine whether the solution is scalable

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

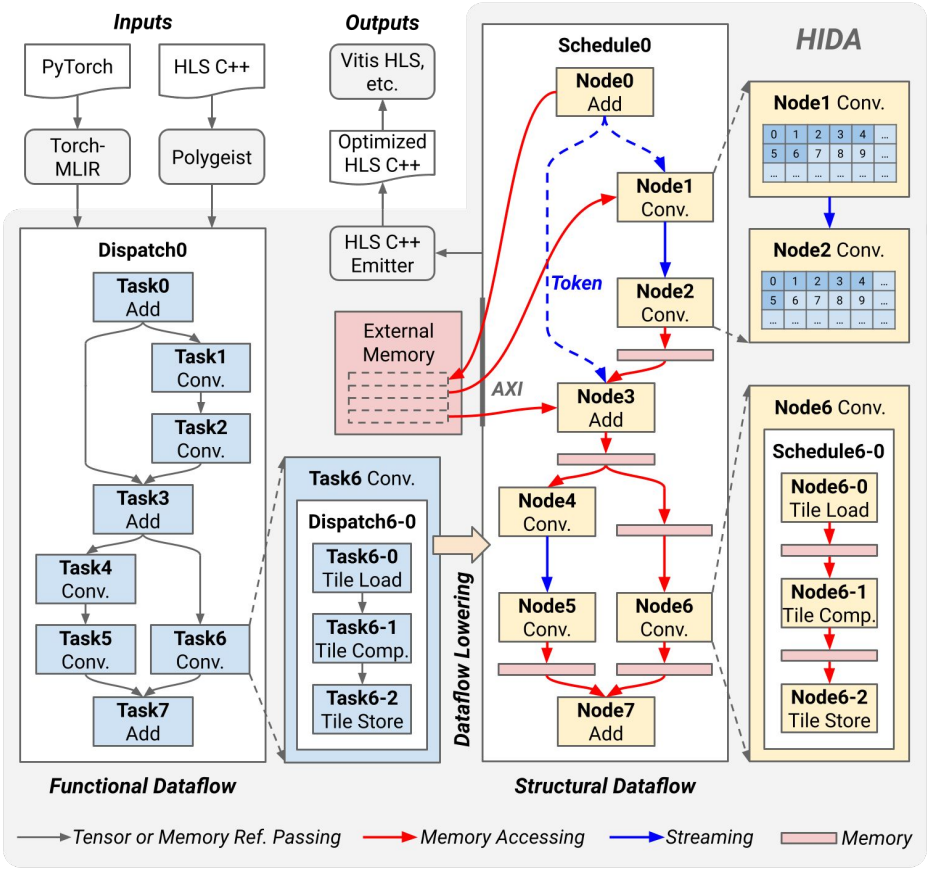






- IA+CA parallelization can determine whether the solution is scalable
- IA+CA parallelization can significantly reduce resource utilization

Intermediate Representation of HIDA



- PyTorch or C/C++ as input
- Optimized C++ dataflow design as output
- Two-level dataflow representation
  - **Functional** dataflow
  - **Structural** dataflow
- Decoupled functional and structural dataflow optimization

HIDA Results on DNN Models

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

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

On-board evaluations are in progress

Outline

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

ScaleHLS is Open-Sourced!

ScaleHLS GitHub Repository

https://github.com/hanchenye/scalehls

For HLS Researchers

1. Rapidly implement new HLS optimization algorithms on top of the multi-level IR
2. Investigate new DSE algorithms using the transform and analysis library
3. Rapidly build an end-to-end HLS optimization flow and demonstrate your awesome works!

For HLS Users

1. Optimize HLS designs using the multi-level optimization passes
2. Avoid premature design choices by using the QoR estimator to estimate the latency and utilization
3. Find optimized HLS designs with the automated DSE engine

Conclusion

1. 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.
2. ScaleHLS enables an end-to-end compilation pipeline supporting both C/C++ and PyTorch as input.
3. An automated and extensible DSE engine is developed to search for optimal solutions in the multi-dimensional design spaces.
4. 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.

Readings (Attached in Materials)

1. Ye, Hanchen, Cong Hao, Jianyi Cheng, Hyunmin Jeong, Jack Huang, Stephen Neuendorffer, and Deming Chen. "**ScaleHLS: A New Scalable High-Level Synthesis Framework on Multi-Level Intermediate Representation.**" In 2022 IEEE International Symposium on High-Performance Computer Architecture (HPCA), 2022.
2. Ye, Hanchen, Jun, Hyegang, and Chen, Deming. "**HIDA: A Hierarchical Dataflow Compiler for High-Level Synthesis.**" In 2024 ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS), 2024.
3. Lattner, Chris, Mehdi Amini, Uday Bondhugula, Albert Cohen, Andy Davis, Jacques Pienaar, River Riddle, Tatiana Shpeisman, Nicolas Vasilache, and Oleksandr Zinenko. "**MLIR: Scaling compiler infrastructure for domain specific computation.**" In 2021 IEEE/ACM International Symposium on Code Generation and Optimization (CGO), 2021.

Thanks! Q&A

Hanchen Ye, Oct. 5

UNIVERSITY OF
ILLINOIS
URBANA-CHAMPAIGN

Appendix

PyTorch Compilation Walkthrough in MLIR

ML Model Compilation: PyTorch to Torch Dialect

```python
class Linear(nn.Module):
    def __init__(self):
        super(Linear, self).__init__()
        self.linear = nn.Linear(16, 10)

    def forward(self, x):
        return self.linear(x)


linear = Linear()
mlir_module = torch_mlir.compile(linear, torch.ones(
    1, 16), output_type=torch_mlir.OutputType.TORCH)
```

PyTorch Model

Torch-MLIR Front-end

```
func.func @forward(%arg0: !torch.vtensor<[1,16],f32>) -> !torch.vtensor<[1,10],f32> {
  %0 = torch.vtensor.literal(dense<"0xA270..."> : tensor<10xf32>) : !torch.vtensor<[10],f32>
  %1 = torch.vtensor.literal(dense<"0x5CE5..."> : tensor<10x16xf32>) : !torch.vtensor<[10,16],f32>
  %2 = torch.aten.linear %arg0, %1, %0 : !torch.vtensor<[1,16],f32>, !torch.vtensor<[10,16],f32>,
!torch.vtensor<[10],f32> -> !torch.vtensor<[1,10],f32>
  return %2 : !torch.vtensor<[1,10],f32>
}
```

Torch Dialect

Torch Dialect

- Front-end dialect designed for interfacing PyTorch and MLIR.
- This dialect maintains a fairly isomorphic representation with TorchScript.
- Operates on tensor objects with static ranks inferred where possible and propagated throughout the program.

Source: Torch-MLIR https://github.com/llvm/torch-mlir

ML Model Compilation: Torch to TOSA

Torch to TOSA Lowering

```
func.func @forward(%arg0: tensor<1x16xf32>) -> tensor<1x10xf32> {
  %0 = "tosa.const"() {value = dense<"0xC44B..."> : tensor<1x16x10xf32>} : () -> tensor<1x16x10xf32>
  %1 = "tosa.const"() {value = dense<"0xA270..."> : tensor<1x10xf32>} : () -> tensor<1x10xf32>
  %2 = "tosa.reshape"(%arg0) {new_shape = [1, 1, 16]} : (tensor<1x16xf32>) -> tensor<1x1x16xf32>
  %3 = "tosa.matmul"(%2, %0) : (tensor<1x1x16xf32>, tensor<1x16x10xf32>) -> tensor<1x1x10xf32>
  %4 = "tosa.reshape"(%3) {new_shape = [1, 10]} : (tensor<1x1x10xf32>) -> tensor<1x10xf32>
  %5 = "tosa.add"(%4, %1) : (tensor<1x10xf32>, tensor<1x10xf32>) -> tensor<1x10xf32>
  return %5 : tensor<1x10xf32>
}
```

Torch Dialect

TOSA (Tensor Operators Set Architecture) Dialect

- A front-end and back-end agnostic dialect representing a minimal and stable set of tensor-level operations commonly employed by Machine Learning frameworks.
- Detailed functional and numerical description enabling precise code construction for a diverse range of targets – SIMD CPUs, GPUs and custom domain-specific accelerators.

Source: Torch-MLIR https://github.com/llvm/torch-mlir

ML Model Compilation: TOSA to Linalg on Tensors

TOSA to Linalg Lowering

```
#map0 = affine_map<(d0, d1, d2) -> (d0, d2)>
#map1 = affine_map<(d0, d1, d2) -> (d2, d1)>
#map2 = affine_map<(d0, d1, d2) -> (d0, d1)>
func.func @forward(%arg0: tensor<1x16xf32>) -> tensor<1x10xf32> {
  %cst = arith.constant dense<"0xA270..."> : tensor<1x10xf32>
  %cst_0 = arith.constant dense<"0xC44B..."> : tensor<16x10xf32>
  %0 = linalg.generic {indexing_maps = [#map0, #map1, #map2], iterator_types = ["parallel", "parallel", "reduction"]}
ins(%arg0, %cst_0 : tensor<1x16xf32>, tensor<16x10xf32>) outs(%cst : tensor<1x10xf32>) {
  ^bb0(%arg1: f32, %arg2: f32, %arg3: f32):
    %1 = arith.mulf %arg1, %arg2 : f32
    %2 = arith.addf %arg3, %1 : f32
    linalg.yield %2 : f32
  } -> tensor<1x10xf32>
  return %0 : tensor<1x10xf32>
}
```

Tensor + Arith + Linalg Dialects

Linalg (Linear Algebra) Dialect

- High-level and structured representation of linear algebra operators
- Designed for driving transformations including buffer allocation, parametric tiling, vectorization, etc.

ML Model Compilation: Bufferization

Func, Arith, and Linalg Bufferization

```
#map0 = affine_map<(d0, d1, d2) -> (d0, d2)>
#map1 = affine_map<(d0, d1, d2) -> (d2, d1)>
#map2 = affine_map<(d0, d1, d2) -> (d0, d1)>
memref.global "private" constant @__constant_16x10xf32 : memref<16x10xf32> = dense<"0xC44B...">
memref.global "private" constant @__constant_1x10xf32 : memref<1x10xf32> = dense<"0xA270...">
func.func @forward(%arg0: memref<1x16xf32>, %arg1: memref<1x10xf32>) {
  %0 = memref.get_global @__constant_1x10xf32 : memref<1x10xf32>
  %2 = memref.get_global @__constant_16x10xf32 : memref<16x10xf32>
  memref.copy %0, %arg1 : memref<1x10xf32> to memref<1x10xf32>
  linalg.generic {indexing_maps = [#map0, #map1, #map2], iterator_types = ["parallel", "parallel", "reduction"]}
ins(%arg0, %2 : memref<1x16xf32>, memref<16x10xf32>) outs(%arg1 : memref<1x10xf32>) {
  ^bb0(%arg2: f32, %arg3: f32, %arg4: f32):
    %3 = arith.mulf %arg2, %arg3 : f32
    %4 = arith.addf %arg4, %3 : f32
    linalg.yield %4 : f32
  }
  return
}
```

Memref + Arith + Linalg Dialects

ML Model Compilation: Linalg to Affine


```
memref.global "private" constant @__constant_16x10xf32 : memref<16x10xf32> = dense<"0xC44B...">
memref.global "private" constant @__constant_1x10xf32 : memref<1x10xf32> = dense<"0xA270...">
func.func @forward(%arg0: memref<1x16xf32>, %arg1: memref<1x10xf32>) {
  %0 = memref.get_global @__constant_1x10xf32 : memref<1x10xf32>
  %1 = memref.get_global @__constant_16x10xf32 : memref<16x10xf32>
  memref.copy %0, %arg1 : memref<1x10xf32> to memref<1x10xf32>
  affine.for %arg2 = 0 to 10 {
    affine.for %arg3 = 0 to 16 {
      %2 = affine.load %arg0[0, %arg3] : memref<1x16xf32>
      %3 = affine.load %1[%arg3, %arg2] : memref<16x10xf32>
      %4 = affine.load %arg1[0, %arg2] : memref<1x10xf32>
      %5 = arith.mulf %2, %3 : f32
      %6 = arith.addf %4, %5 : f32
      affine.store %6, %arg1[0, %arg2] : memref<1x10xf32>
    }
  }
  return
}
```

Memref + Arith + Affine Dialects

Affine Dialect

Designed for using techniques from polyhedral compilation to make dependence analysis and loop transformations efficient and reliable.

ML Model Compilation: Affine-level Vectorization

Affine Super Vectorization

```
#map = affine_map<(d0, d1) -> (0)>
memref.global "private" constant @__constant_16x10xf32 : memref<16x10xf32> = dense<"0xC44B...">
memref.global "private" constant @__constant_1x10xf32 : memref<1x10xf32> = dense<"0xA270...">
func.func @forward(%arg0: memref<1x16xf32>, %arg1: memref<1x10xf32>) {
  %c0 = arith.constant 0 : index
  %cst = arith.constant 0.000000e+00 : f32
  %0 = memref.get_global @__constant_1x10xf32 : memref<1x10xf32>
  %1 = memref.get_global @__constant_16x10xf32 : memref<16x10xf32>
  memref.copy %0, %arg1 : memref<1x10xf32> to memref<1x10xf32>
  affine.for %arg2 = 0 to 10 step 2 {
    affine.for %arg3 = 0 to 16 {
      %2 = vector.transfer_read %arg0[%c0, %arg3], %cst {permutation_map = #map} : memref<1x16xf32>, vector<2xf32>
      %3 = vector.transfer_read %1[%arg3, %arg2], %cst : memref<16x10xf32>, vector<2xf32>
      %4 = vector.transfer_read %arg1[%c0, %arg2], %cst : memref<1x10xf32>, vector<2xf32>
      %5 = arith.mulf %2, %3 : vector<2xf32>
      %6 = arith.addf %4, %5 : vector<2xf32>
      vector.transfer_write %6, %arg1[%c0, %arg2] : vector<2xf32>, memref<1x10xf32>
    }
  }
  return
}
```

Memref + Vector + Arith + Affine Dialects

ML Model Compilation: Affine to SCF

Affine to SCF Lowering

```
#map = affine_map<(d0, d1) -> (0)>
memref.global "private" constant @__constant_16x10xf32 : memref<16x10xf32> = dense<"0xC44B...">
memref.global "private" constant @__constant_1x10xf32 : memref<1x10xf32> = dense<"0xA270...">
func.func @forward(%arg0: memref<1x16xf32>, %arg1: memref<1x10xf32>) {
  %c1 = arith.constant 1 : index    %c16 = arith.constant 16 : index    %c2 = arith.constant 2 : index
  %c10 = arith.constant 10 : index    %c0 = arith.constant 0 : index    %cst = arith.constant 0.000000e+00 : f32
  %0 = memref.get_global @__constant_1x10xf32 : memref<1x10xf32>
  %1 = memref.get_global @__constant_16x10xf32 : memref<16x10xf32>
  memref.copy %0, %arg1 : memref<1x10xf32> to memref<1x10xf32>
  scf.for %arg2 = %c0 to %c10 step %c2 {
    scf.for %arg3 = %c0 to %c16 step %c1 {
      %2 = vector.transfer_read %arg0[%c0, %arg3], %cst {permutation_map = #map} : memref<1x16xf32>, vector<2xf32>
      %3 = vector.transfer_read %1[%arg3, %arg2], %cst : memref<16x10xf32>, vector<2xf32>
      %4 = vector.transfer_read %arg1[%c0, %arg2], %cst : memref<1x10xf32>, vector<2xf32>
      %5 = arith.mulf %2, %3 : vector<2xf32>
      %6 = arith.addf %4, %5 : vector<2xf32>
      vector.transfer_write %6, %arg1[%c0, %arg2] : vector<2xf32>, memref<1x10xf32>
    }
  }
  return
}
```

Memref + Vector + Arith + SCF Dialects

SCF Dialect

Represents structured control flow.

ML Model Compilation: SCF to CF (Control Flow)


```
  ... ...
^bb1(%2: index):  // 2 preds: ^bb0, ^bb4
  %3 = arith.cmpi slt, %2, %c10 : index
  cf.cond_br %3, ^bb2(%c0 : index), ^bb5
^bb2(%4: index):  // 2 preds: ^bb1, ^bb3
  %5 = arith.cmpi slt, %4, %c16 : index
  cf.cond_br %5, ^bb3, ^bb4
^bb3:  // pred: ^bb2
  %6 = vector.transfer_read %arg0[%c0, %4], %cst {permutation_map = #map} : memref<1x16xf32>, vector<2xf32>
  %7 = vector.transfer_read %1[%4, %2], %cst : memref<16x10xf32>, vector<2xf32>
  %8 = vector.transfer_read %arg1[%c0, %2], %cst : memref<1x10xf32>, vector<2xf32>
  %9 = arith.mulf %6, %7 : vector<2xf32>
  %10 = arith.addf %8, %9 : vector<2xf32>
  vector.transfer_write %10, %arg1[%c0, %2] : vector<2xf32>, memref<1x10xf32>
  %11 = arith.addi %4, %c1 : index
  cf.br ^bb2(%11 : index)
^bb4:  // pred: ^bb2
  %12 = arith.addi %2, %c2 : index
  cf.br ^bb1(%12 : index)
^bb5:  // pred: ^bb1
  return
}
```

Memref + Vector + Arith + CF Dialects

ML Model Compilation: Lower to LLVM

Memref, Vector, Arith, and CF to LLVM Lowering

```
  ... ...
^bb1(%20: i64):  // 2 preds: ^bb0, ^bb4
  %21 = llvm.icmp "slt" %20, %5 : i64
  llvm.cond_br %21, ^bb2(%4 : i64), ^bb5
^bb2(%22: i64):  // 2 preds: ^bb1, ^bb3
  %23 = llvm.icmp "slt" %22, %7 : i64
  llvm.cond_br %23, ^bb3, ^bb4
^bb3:  // pred: ^bb2
  ... ...
  %46 = llvm.intr.masked.load %45, %36, %0 {alignment = 4 : i32} : (!llvm.ptr<vector<2xf32>>, vector<2xi1>, vector<2xf32>)
-> vector<2xf32>
  %47 = llvm.fmul %30, %41  : vector<2xf32>
  %48 = llvm.fadd %46, %47  : vector<2xf32>
  llvm.intr.masked.store %48, %45, %36 {alignment = 4 : i32} : vector<2xf32>, vector<2xi1> into !llvm.ptr<vector<2xf32>>
  %49 = llvm.add %22, %8  : i64
  llvm.br ^bb2(%49 : i64)
^bb4:  // pred: ^bb2
  %50 = llvm.add %20, %6  : i64
  llvm.br ^bb1(%50 : i64)
^bb5:  // pred: ^bb1
  llvm.return
}
```

LLVM Dialect