OAK RIDGE National Laboratory

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

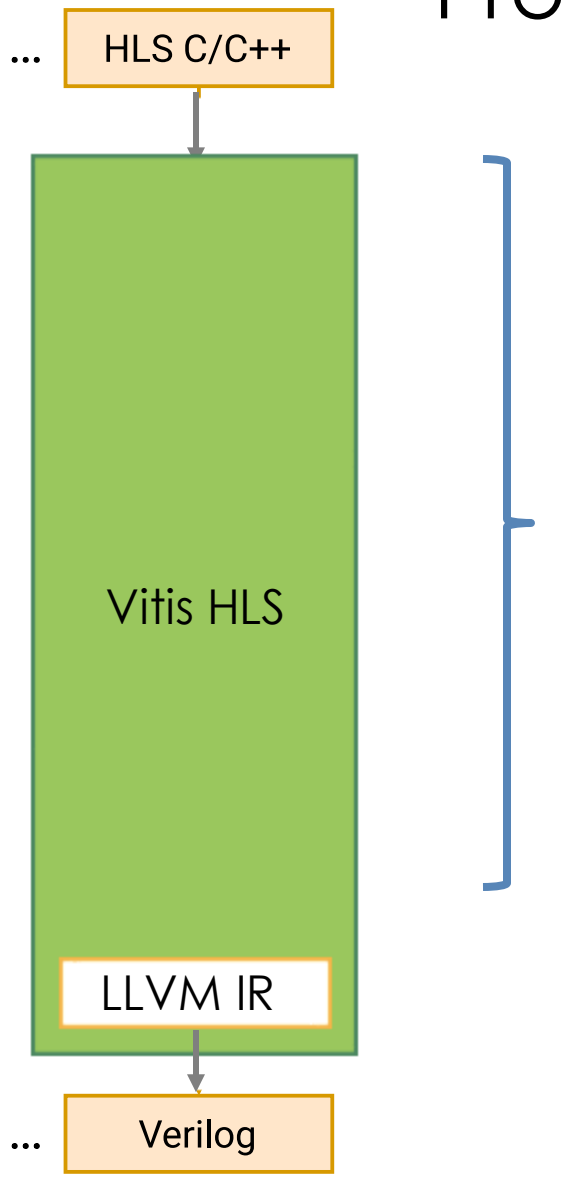
Anthony Cabrera
CSE565M Paper Presentation

October 3, 2023

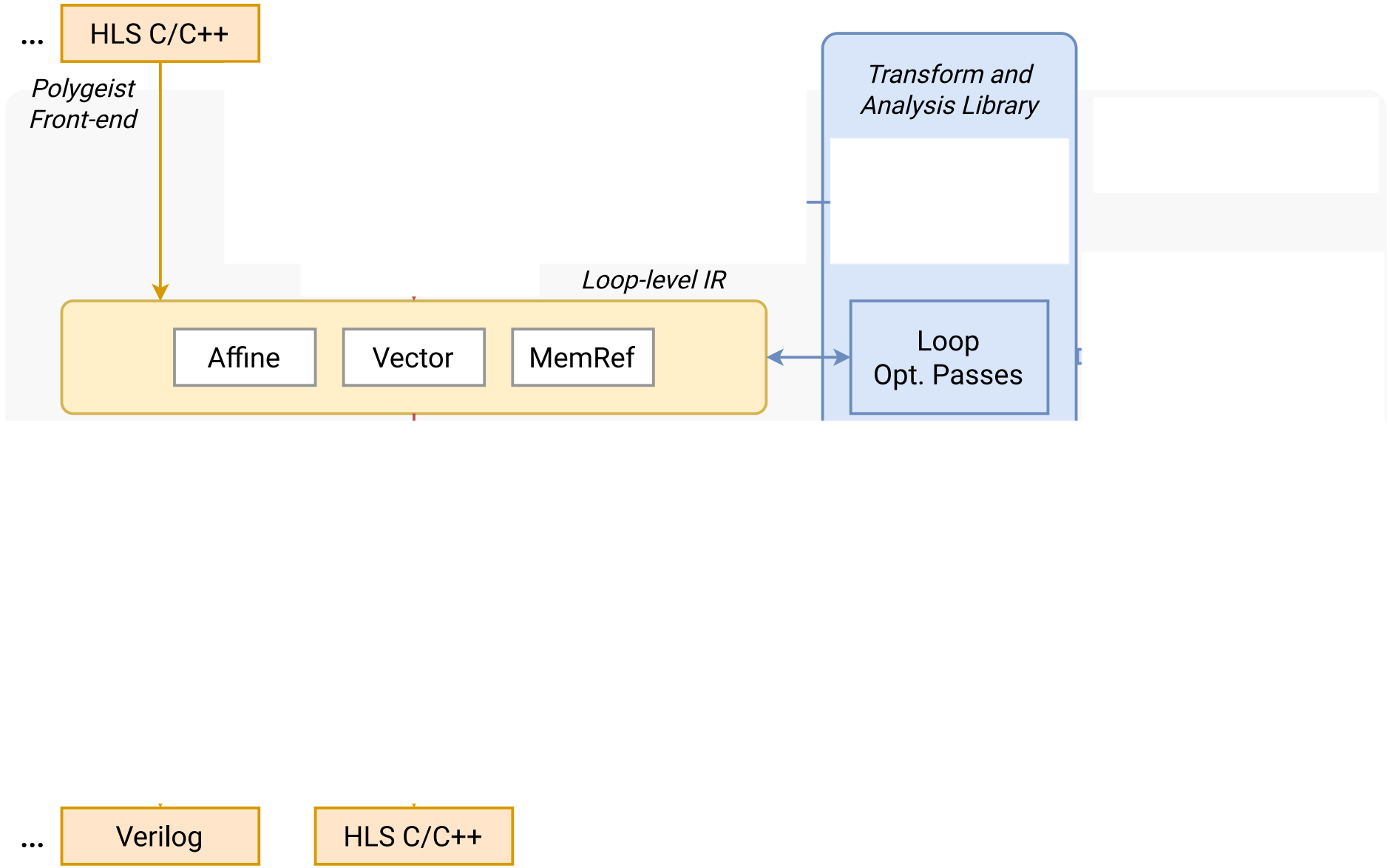
U.S. DEPARTMENT OF ENERGY

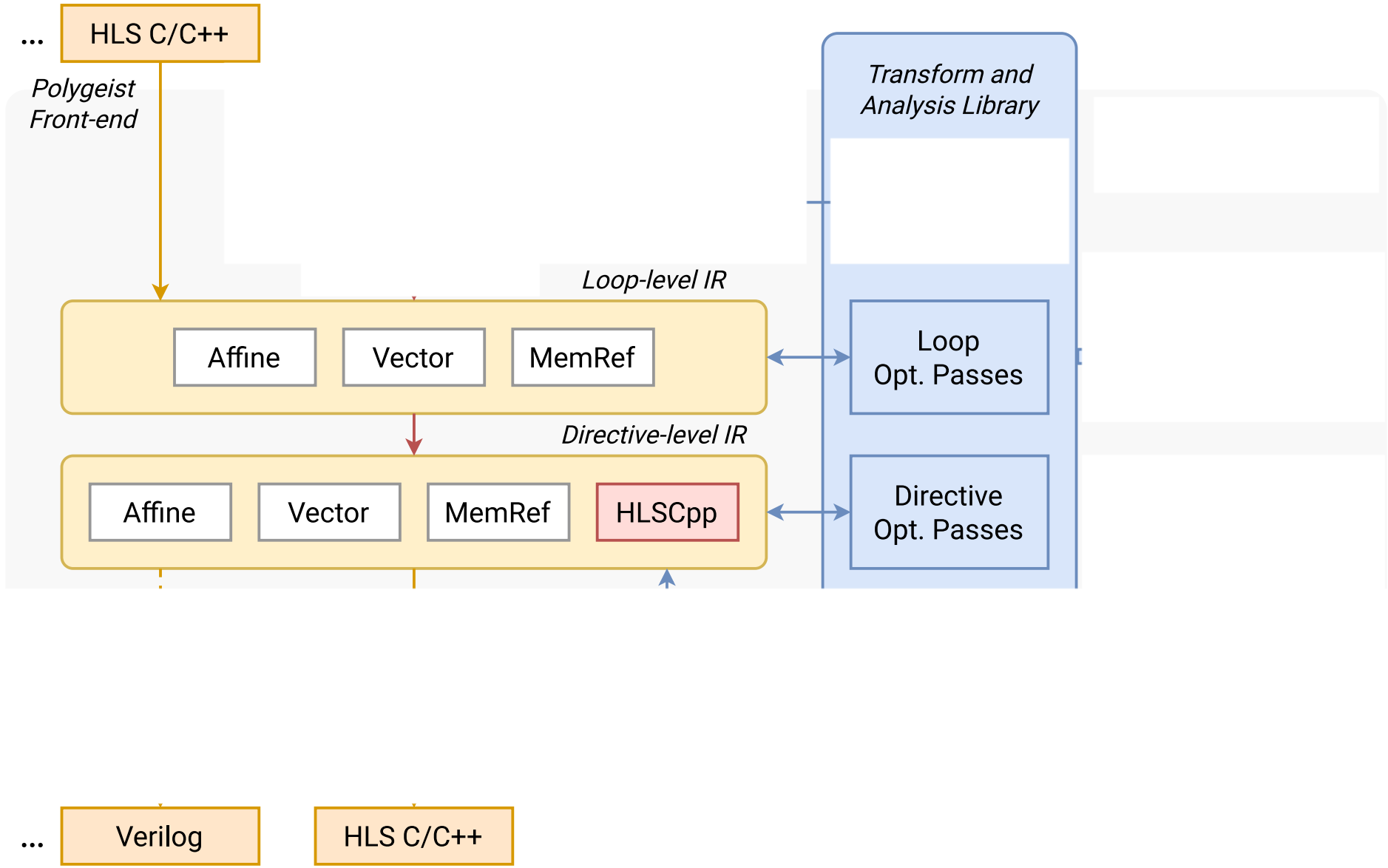
Problem Sketch/Intuition

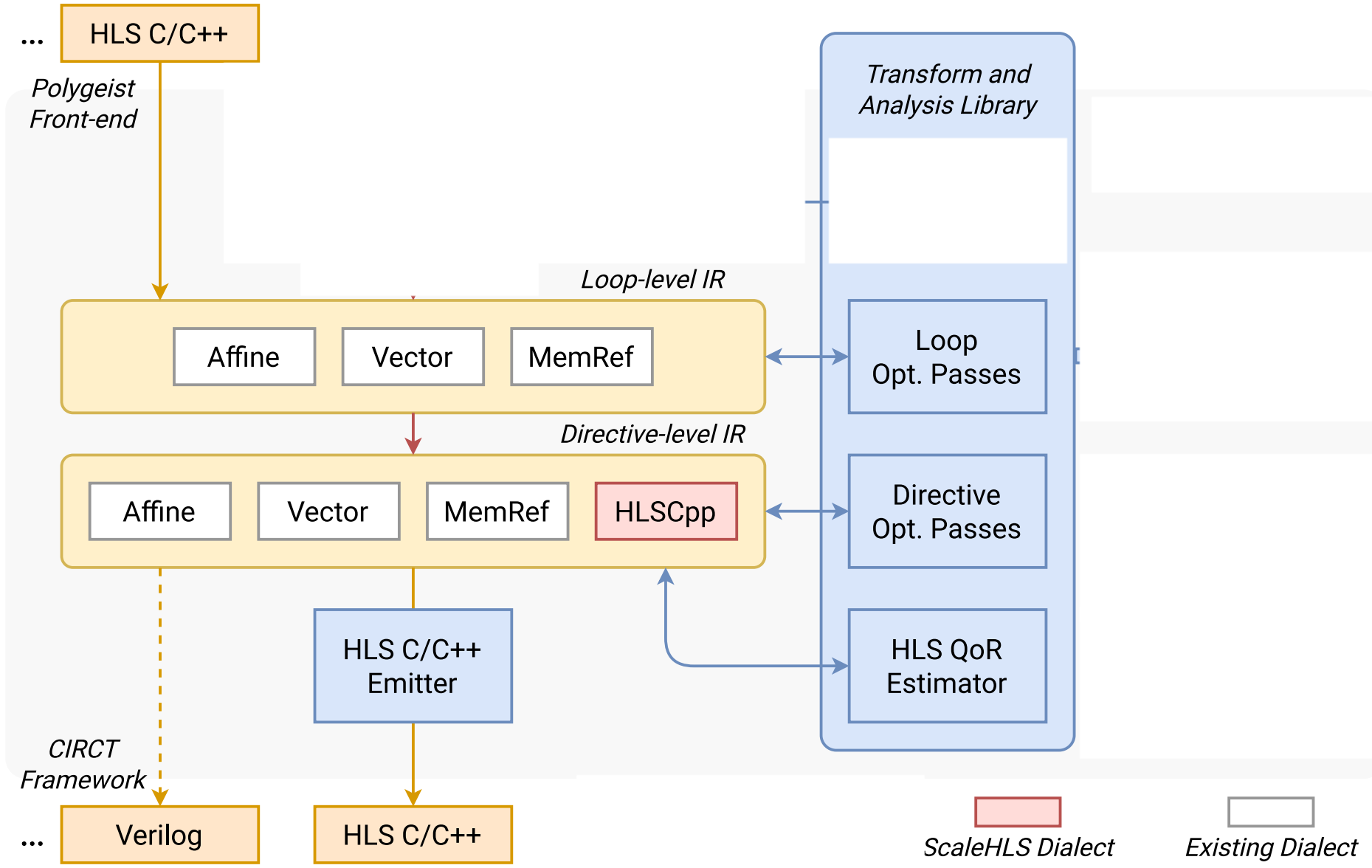
Problem Sketch/Intuition

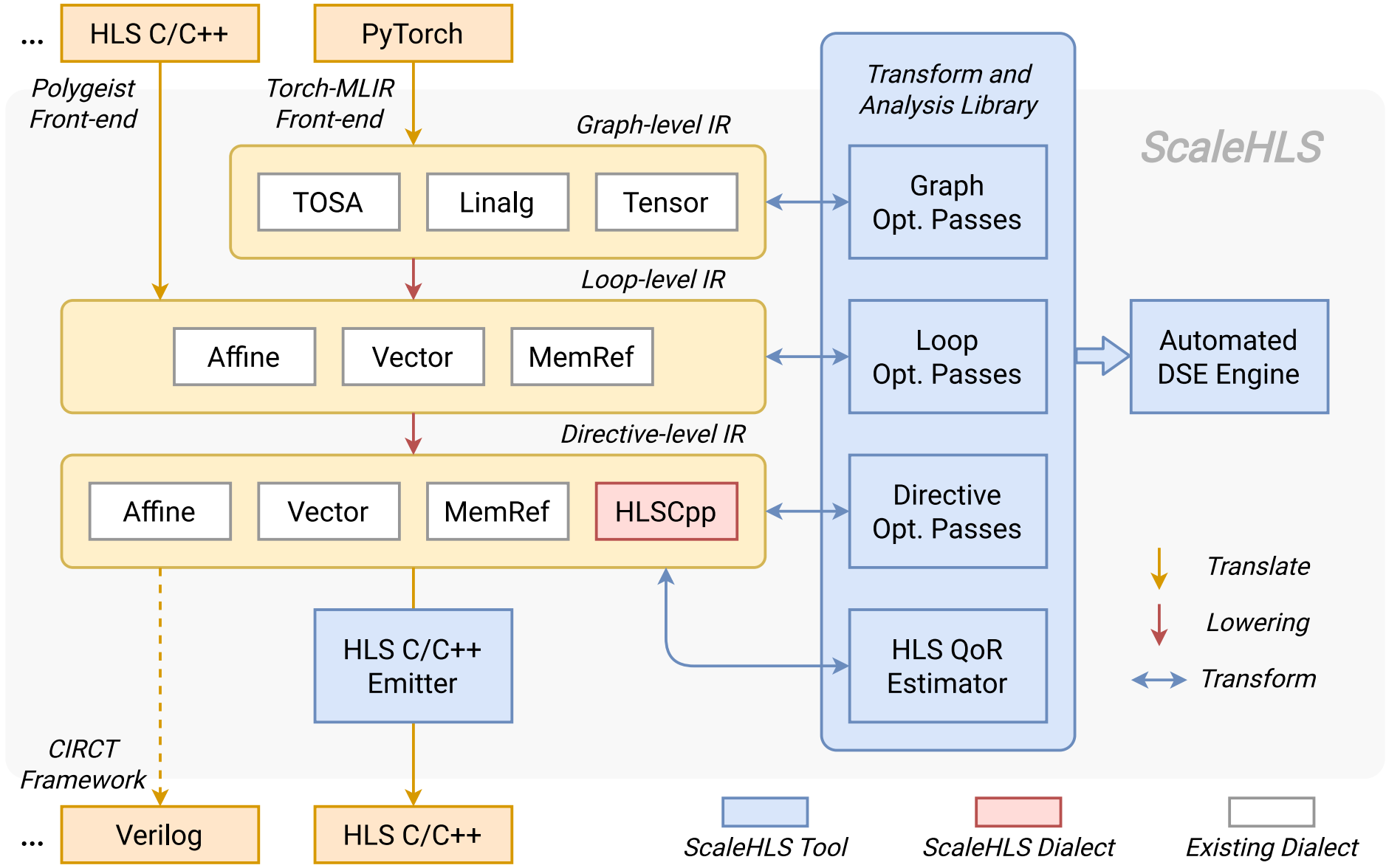


Large semantic gap!






...
HLS C/C++
Polygeist
Front-end
Loop-level IR
Affine
Vector
MemRef
Directive-level IR
Affine
Vector
MemRef
HLSCpp
HLS C/C++
Emitter
CIRCT
Framework
...
Verilog
HLS C/C++
Transform and
Analysis Library
Loop
Opt. Passes
Directive
Opt. Passes
HLS QoR
Estimator
ScaleHLS Dialect
Existing Dialect



Main Takeaway

- Current HLS tooling is built on a single, low level of abstraction, e.g., LLVM IR.
- However, the semantic gap between C/C++ and LLVM IR is very large, which makes it challenging to perform higher abstraction level optimizations.
- The goal of ScaleHLS is to fill this semantic gap with hierarchical intermediate representations that make it easier to optimize the HLS design, subsequently explore the design space.

Contributions of ScaleHLS

- Hierarchical representations of HLS designs to make reasoning about optimizations easier
- Provide optimization passes and infrastructure to operate at the level of graph, loop, and directive levels
- An automated DSE engine to find the Pareto curve between latency and space
- Provides and HLS C front-end to MLIR and an HLS C/C++ emitter

OAK RIDGE National Laboratory

Background

MLIR in a Nutshell

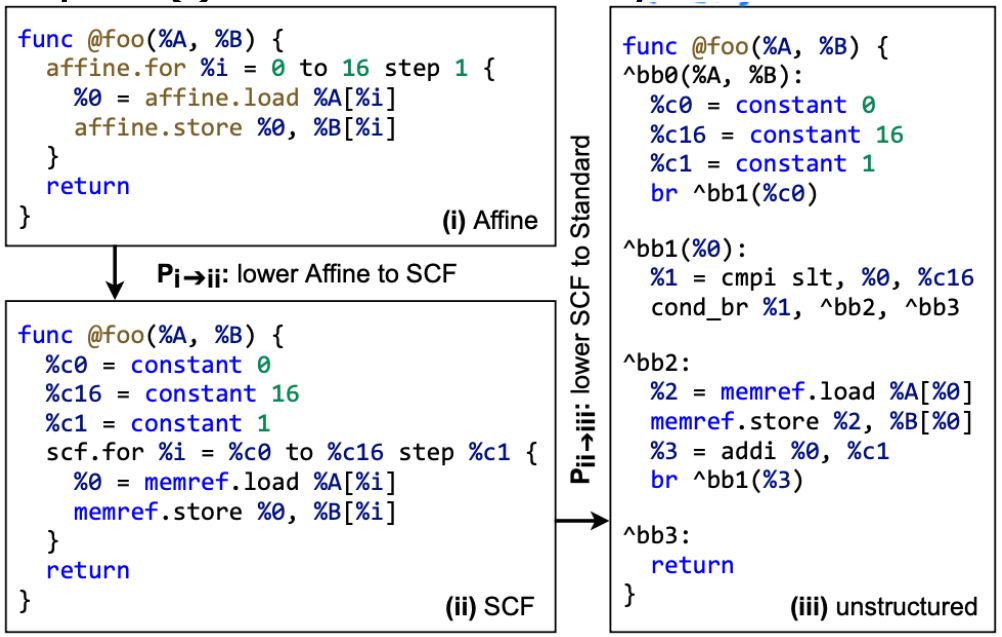
- Close the semantic gap between [input language] and LLVM IR through hierarchical intermediate representations, i.e., **progressive lowering**
- Take advantage of pre-existing SW, passes, dialects, and optimizations and plug them in to your own compiler flows

```
func @foo(%A, %B) {
^bb0(%A, %B):
  %c0 = constant 0
  %c16 = constant 16
  %c1 = constant 1
  br ^bb1(%c0)

^bb1(%0):
  %1 = cmpi slt, %0, %c16
  cond_br %1, ^bb2, ^bb3

^bb2:
  %2 = memref.load %A[%0]
  memref.store %2, %B[%0]
  %3 = addi %0, %c1
  br ^bb1(%3)

^bb3:
  return
}
```

(iii) unstructured

MLIR in a Nutshell

- Close the semantic gap between [input language] and LLVM IR through hierarchical intermediate representations, i.e., **progressive lowering**
- Take advantage of pre-existing SW, passes, dialects, and optimizations and plug them in to your own compiler flows

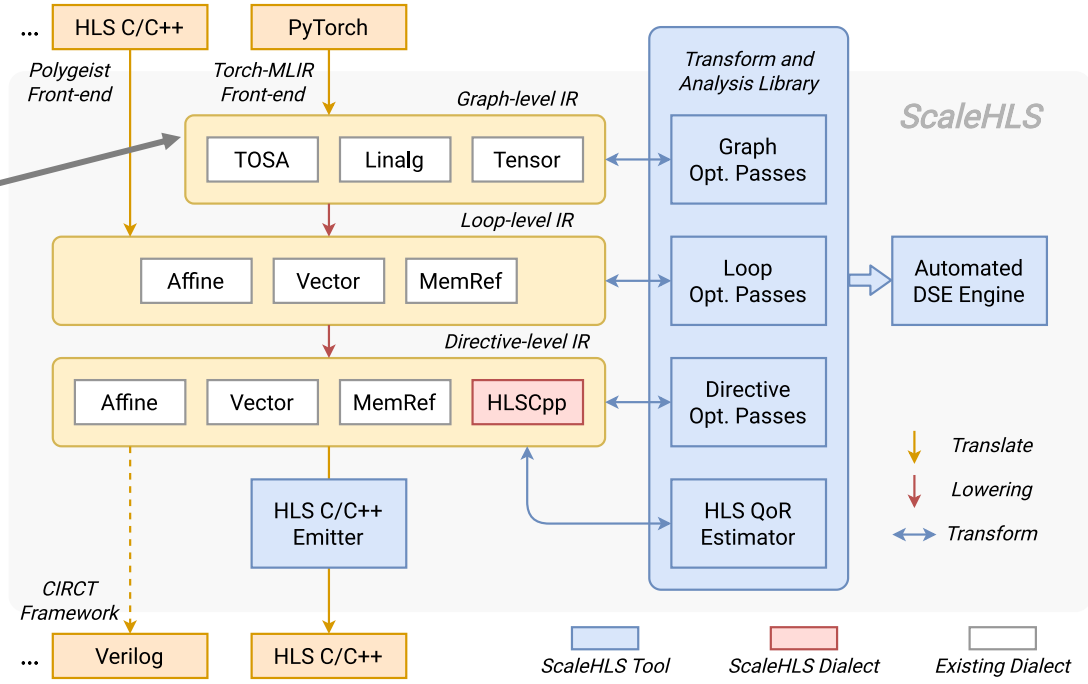


OAK RIDGE National Laboratory

ScaleHLS Design

ScaleHLS Representation:
Graph Level

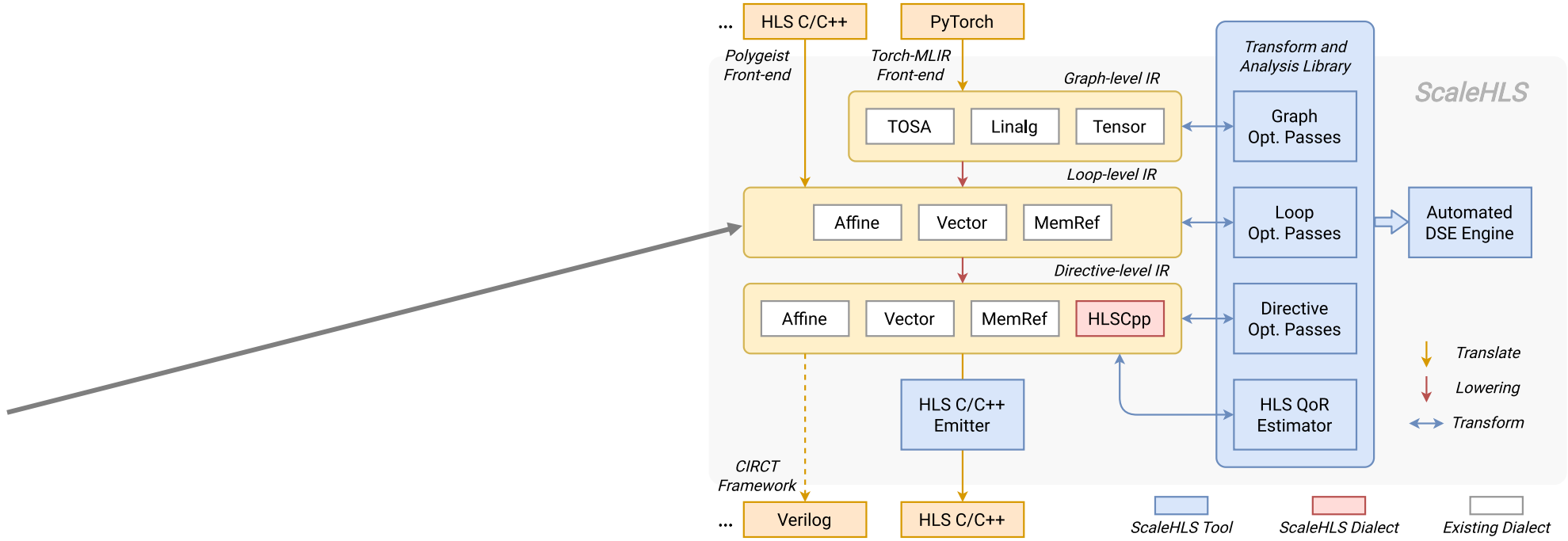
- Leverage pre-existing `onnx` dialect* for Graph-level IR and representing computation graphs
  - E.g., `%output = "onnx.Conv"(%input, %weight) {...} : (tensor<1x3x34x34xf32>, tensor<64x3x3x3xf32>) -> tensor<1x64x32x32xf32>`



* The figure has been changed since its original publication date

ScaleHLS Representation:
Loop Level

- Leverage pre-existing `affine` and `scf` dialect for Loop-level IR for loop level transformations and analysis



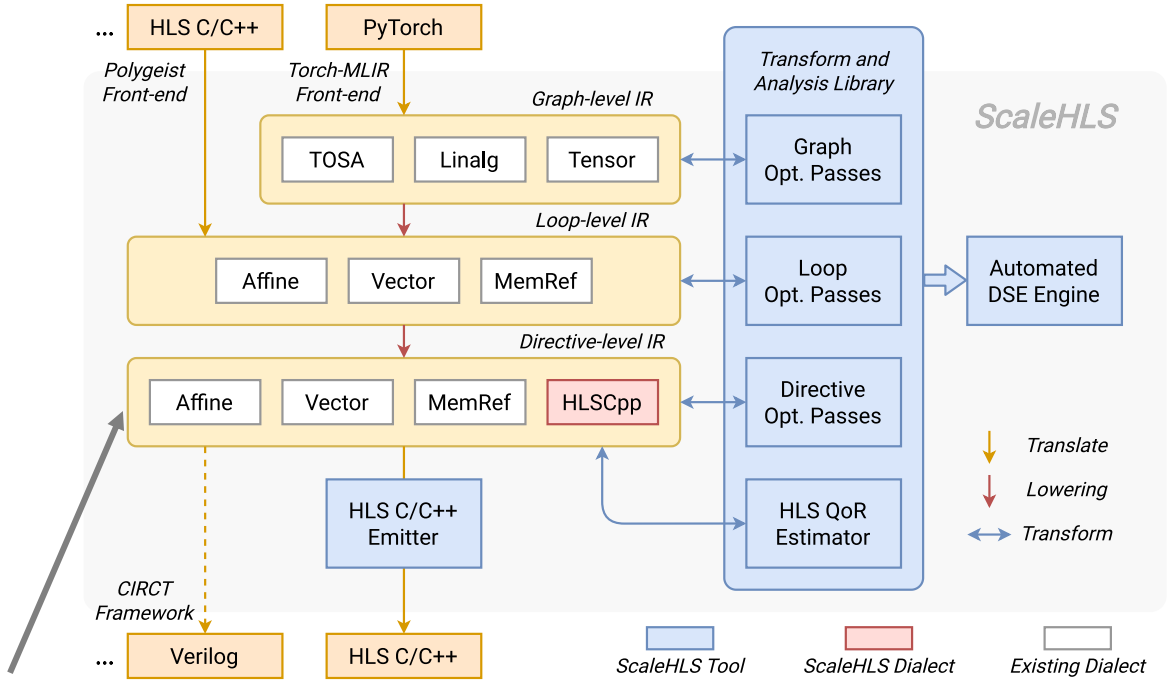
* The figure has been changed since its original publication date

ScaleHLS Representation:
HLS Level

- Develop `HLSCpp` to represent HLS-specific structures and program directives, which provides the capability of conducting directive optimizations and supports the emission of synthesizable C/C++ code

Table I
SUPPORTED HLS DIRECTIVES.

	Function	Loop	Array
Directives	dataflow pipeline inline	dataflow pipeline unroll merge	partition resource interface

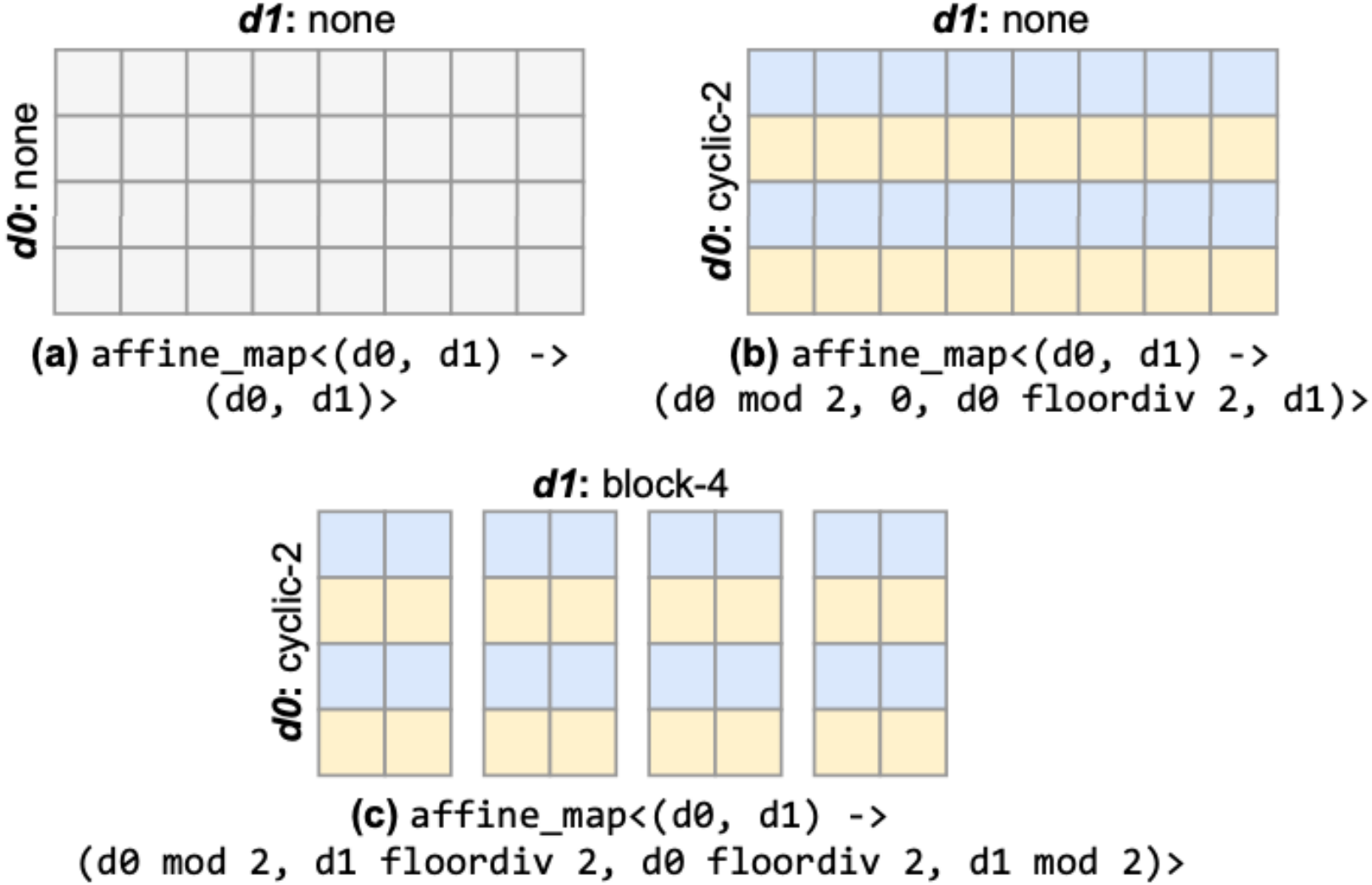


* The figure has been changed since its original publication date

Array Partitioning MLIR Representation

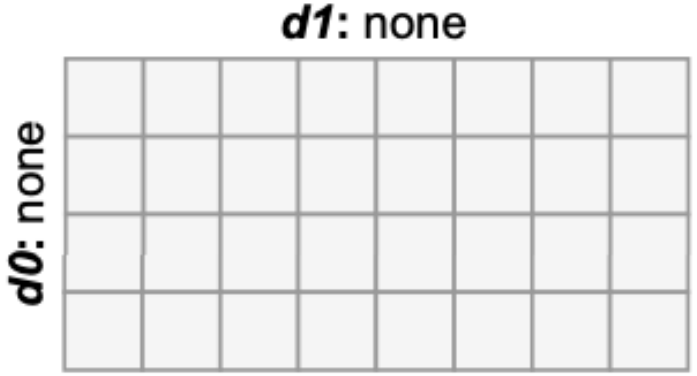
`(d0_orig, d1_orig) -> (partition idx x, partition idx y, physical idx x, physical idx y)`

(b) example: `(3, 6) -> ?`

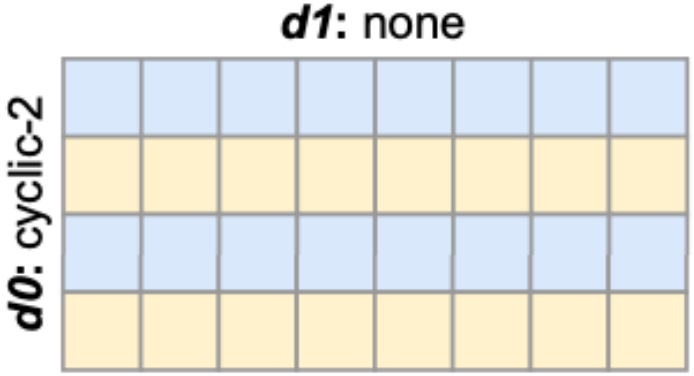


Array Partitioning MLIR Representation

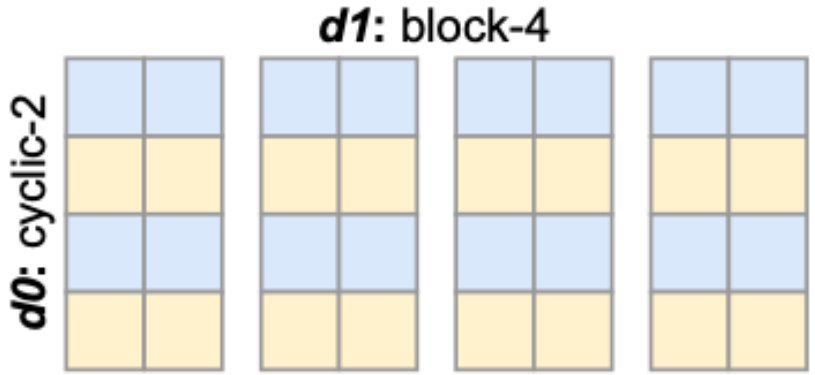
(d0$_{orig}$, d1$_{orig}$) -> (partition idx x, partition idx y, physical idx x, physical idx y)



(a) affine_map<(d0, d1) -> (d0, d1)>



(b) affine_map<(d0, d1) -> (d0 mod 2, 0, d0 floordiv 2, d1)>

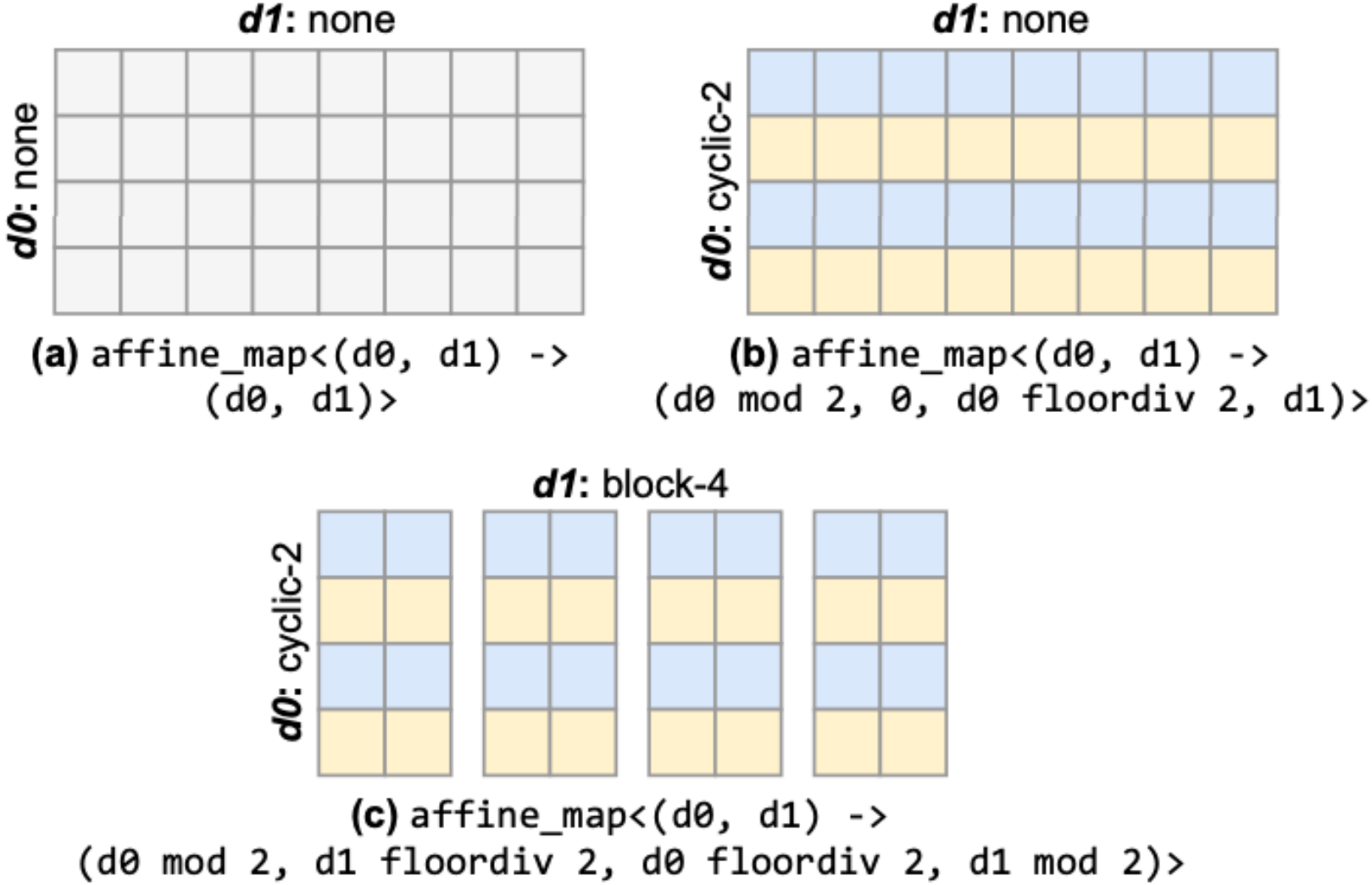


(c) affine_map<(d0, d1) -> (d0 mod 2, d1 floordiv 2, d0 floordiv 2, d1 mod 2)>

Array Partitioning MLIR Representation

`(d0_orig, d1_orig) -> (partition idx x, partition idx y, physical idx x, physical idx y)`

(b) example: `(3, 6) -> ?`

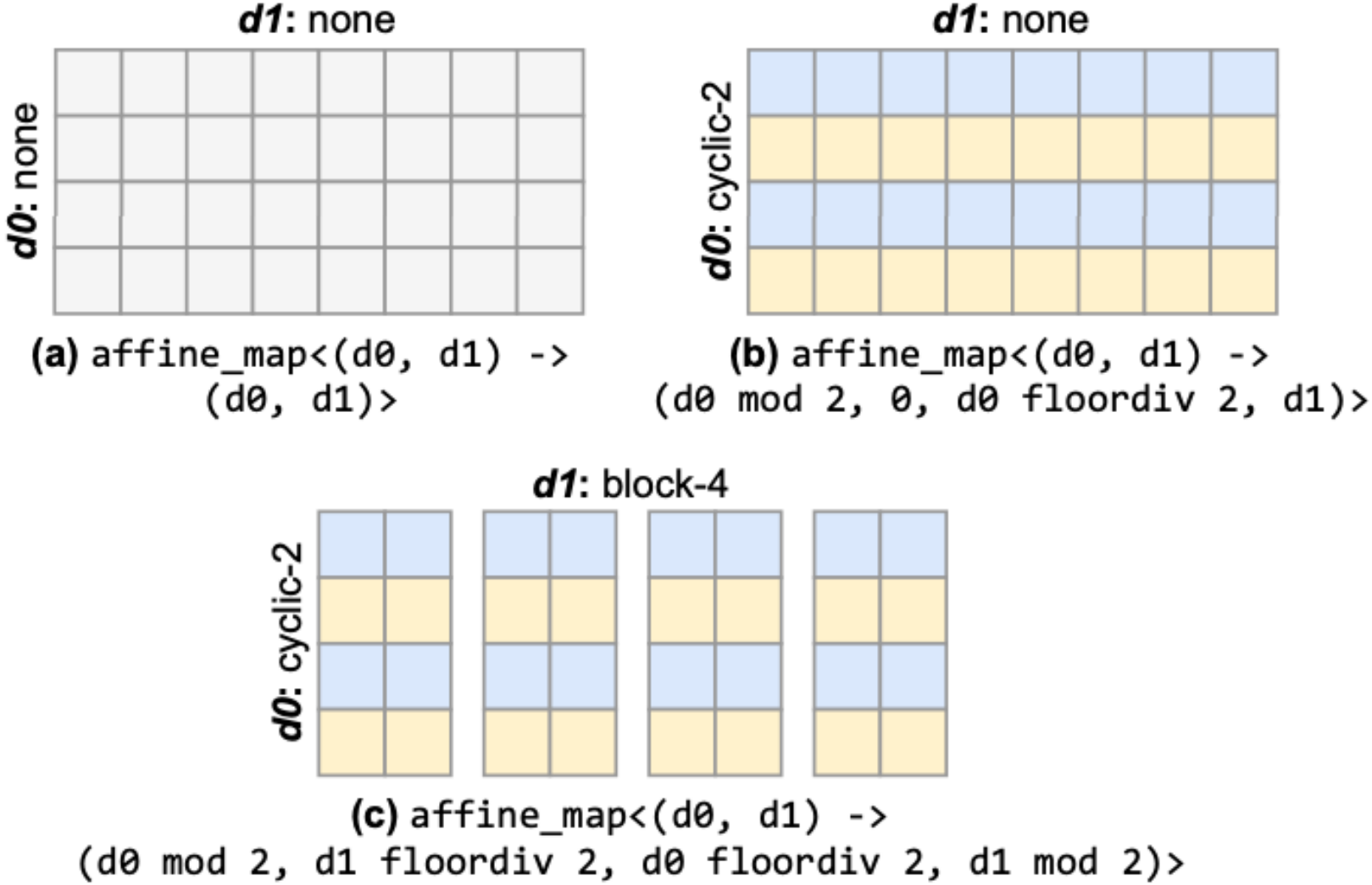


Array Partitioning MLIR Representation

`(d0orig, d1orig) -> (partition idx x, partition idx y, physical idx x, physical idx y)`

(b) example: `(3, 6)` -> ?

`d0 mod 2` = 3 mod 2 = 1



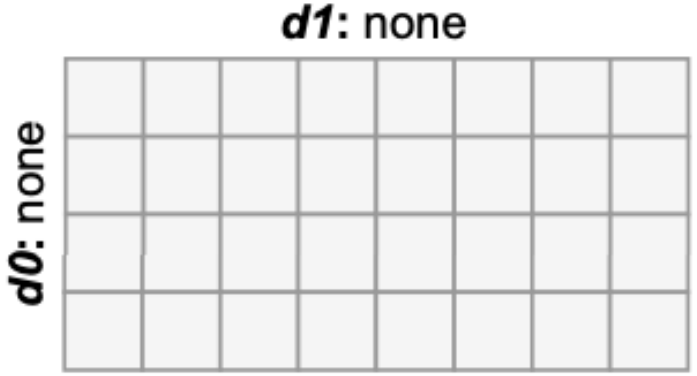
Array Partitioning MLIR Representation

`(d0_orig, d1_orig) -> (partition idx x, partition idx y, physical idx x, physical idx y)`

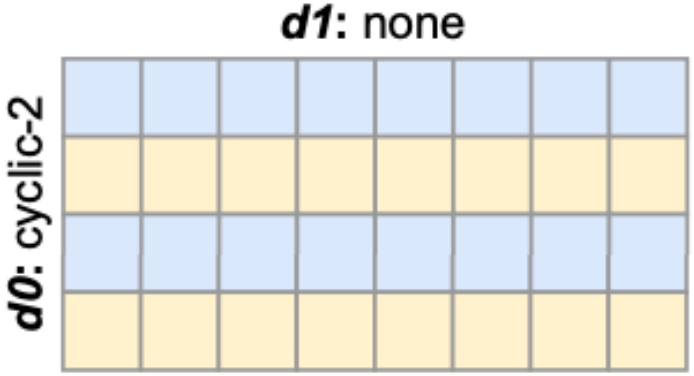
(b) example: `(3, 6)` -> ?

`d0 mod 2` = 3 mod 2 = 1

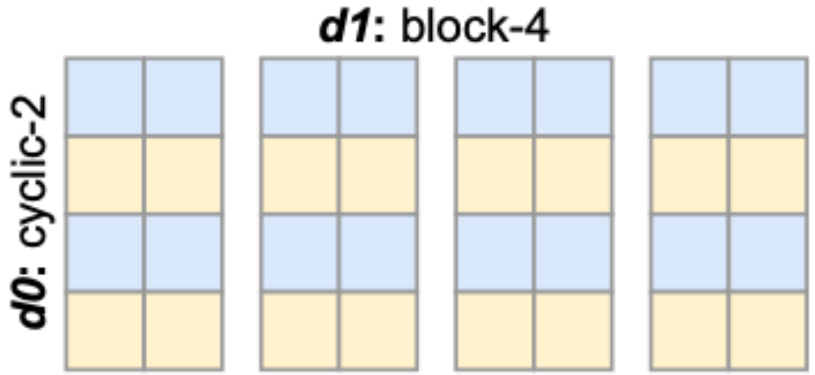
0



(a) `affine_map<(d0, d1) -> (d0, d1)>`



(b) `affine_map<(d0, d1) -> (d0 mod 2, 0, d0 floordiv 2, d1)>`



(c) `affine_map<(d0, d1) -> (d0 mod 2, d1 floordiv 2, d0 floordiv 2, d1 mod 2)>`

Array Partitioning MLIR Representation

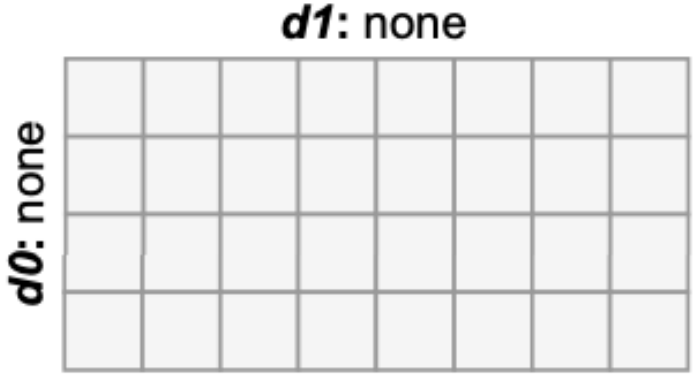
`(d0orig, d1orig) -> (partition idx x, partition idx y, physical idx x, physical idx y)`

(b) example: `(3, 6)` -> ?

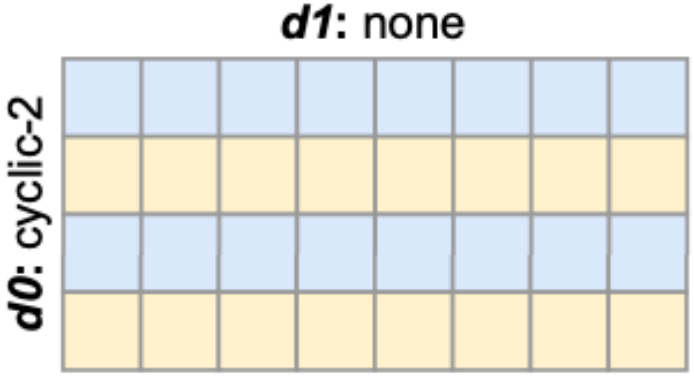
`d0 mod 2` = 3 mod 2 = 1

0

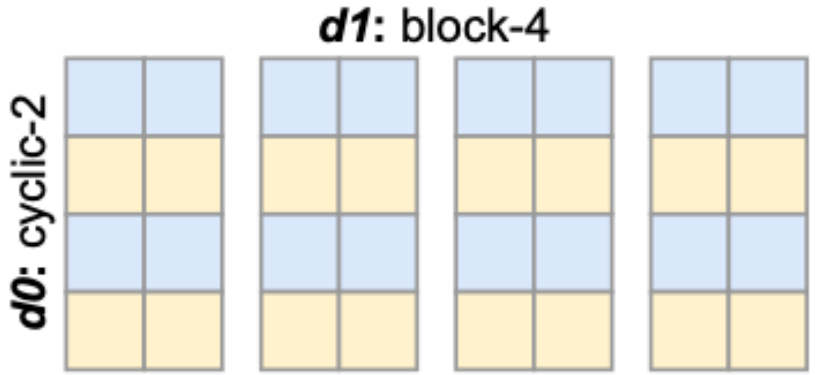
`d0 floordiv 2` = 3 `floordiv` 2 = 1



(a) `affine_map<(d0, d1) -> (d0, d1)>`



(b) `affine_map<(d0, d1) -> (d0 mod 2, 0, d0 floordiv 2, d1)>`



(c) `affine_map<(d0, d1) -> (d0 mod 2, d1 floordiv 2, d0 floordiv 2, d1 mod 2)>`

Array Partitioning MLIR Representation

`(d0orig, d1orig) -> (partition idx x, partition idx y, physical idx x, physical idx y)`

(b) example: `(3, 6)` -> ?

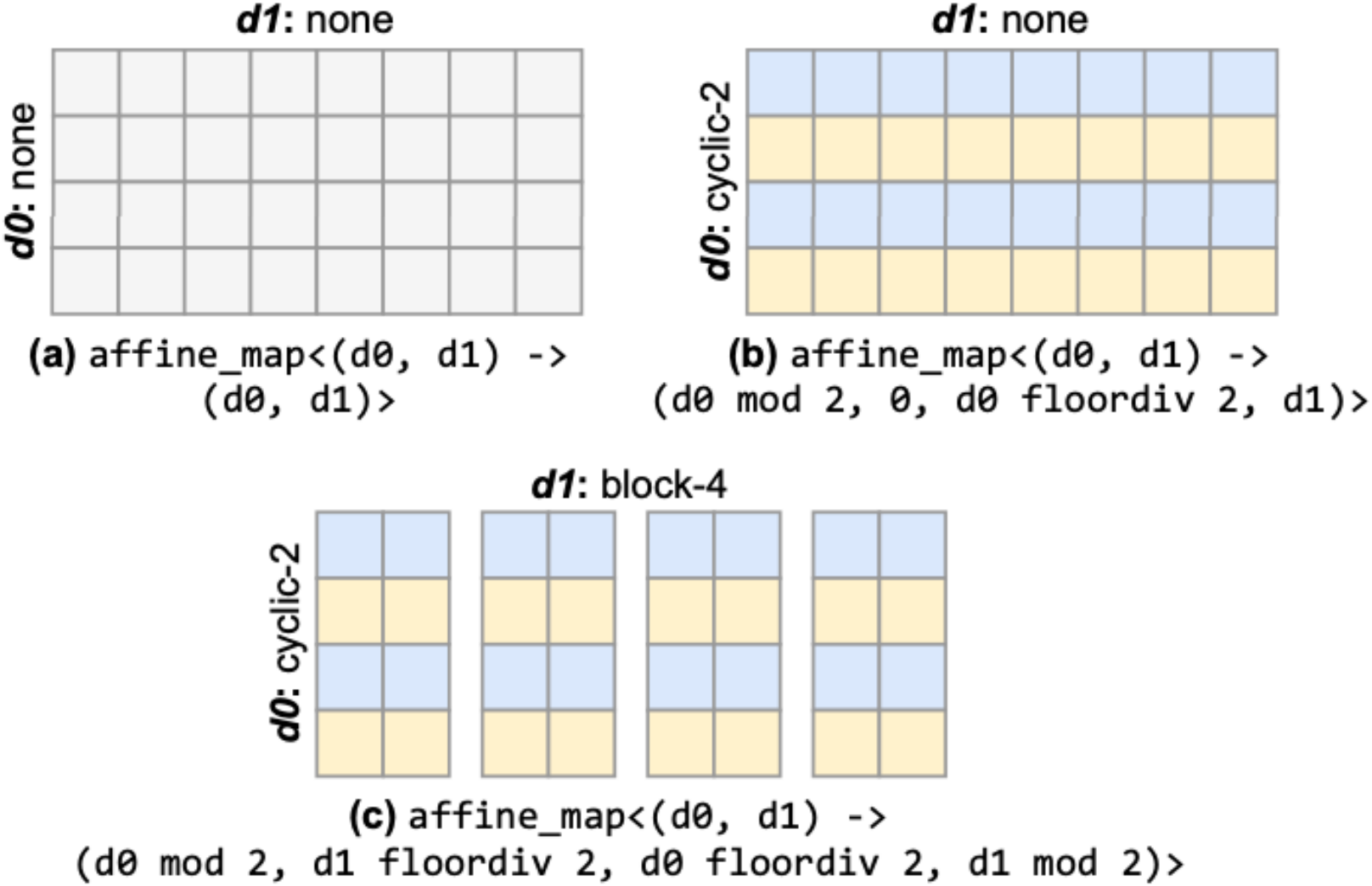
`d0 mod 2` = 3 mod 2 = 1

0

`d0 floordiv 2`

= 3 `floordiv` 2 = 1

d1 = 6



Array Partitioning MLIR Representation

`(d0orig, d1orig) -> (partition idx x, partition idx y, physical idx x, physical idx y)`

(b) example: `(3, 6) -> ?`

`d0 mod 2` = 3 mod 2 = 1

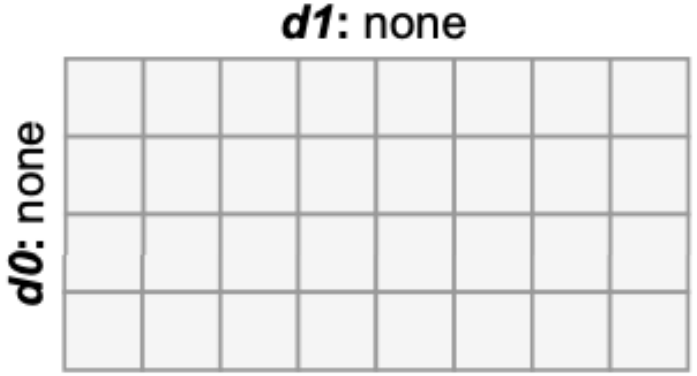
0

`d0 floordiv 2` = 3 `floordiv` 2 = 1

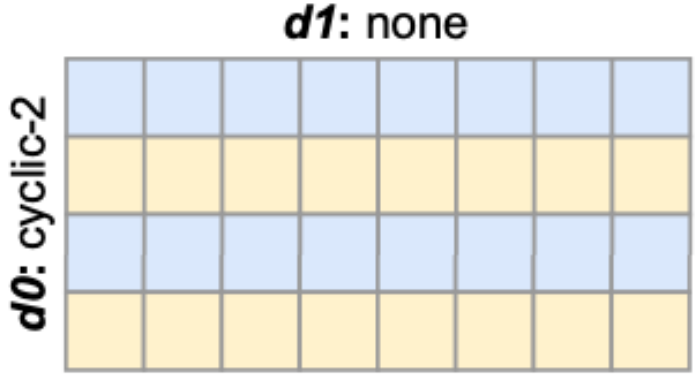
d1 = 6

Thus,

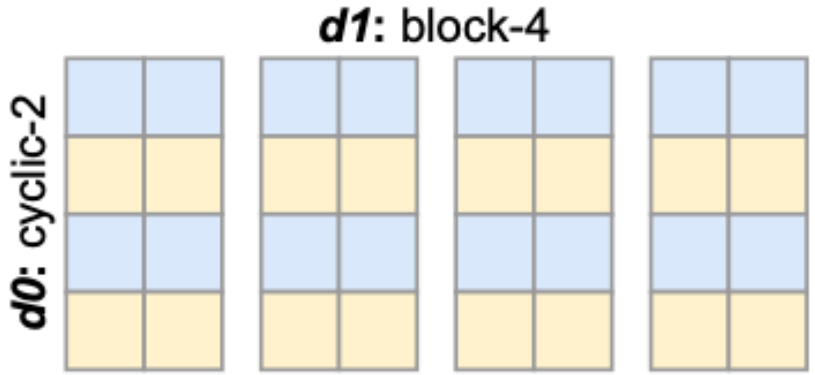
`(3, 6) -> (1, 0, 1, 6)`



(a) affine_map<(d0, d1) -> (d0, d1)>



(b) affine_map<(d0, d1) -> (d0 mod 2, 0, d0 floordiv 2, d1)>



(c) affine_map<(d0, d1) -> (d0 mod 2, d1 floordiv 2, d0 floordiv 2, d1 mod 2)>

ScaleHLS Optimization

- Apply the appropriate flavor of pass at the appropriate representation level
  - e.g., apply graph passes on graph representation
- Note the passes not in bold in the *Loop* row; those are passes that already exist and are compatible with the dialects in that level of representation

Table II
SCALEHLS PASSES.

	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 band	tile-sizes
	-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 ScaleHLS, while others are MLIR built-in passes.

Graph Transform Example

Explores tradeoff between latency and space

- Assume each Proc takes 1t time.
- The goal is to apply the `HLS dataflow` pragma to this design
- 4(a) violates the bypass path constraint of Vitis HLS
- 4(b) legalizes this dataflow by combining the offending sub-graph into it's own stage. Results in 3-stage pipeline with latency = 3t
- 4(c) aggressively legalizes by adding enough copy nodes to remove bypass path. Results in 5-stage pipeline with latency = 1t, but more HW needed to achieve this
- Latency/space tradeoff in 4(d) by introducing `split-function` pass, which groups every two stages. Latency = 2t with only one additional copy node required

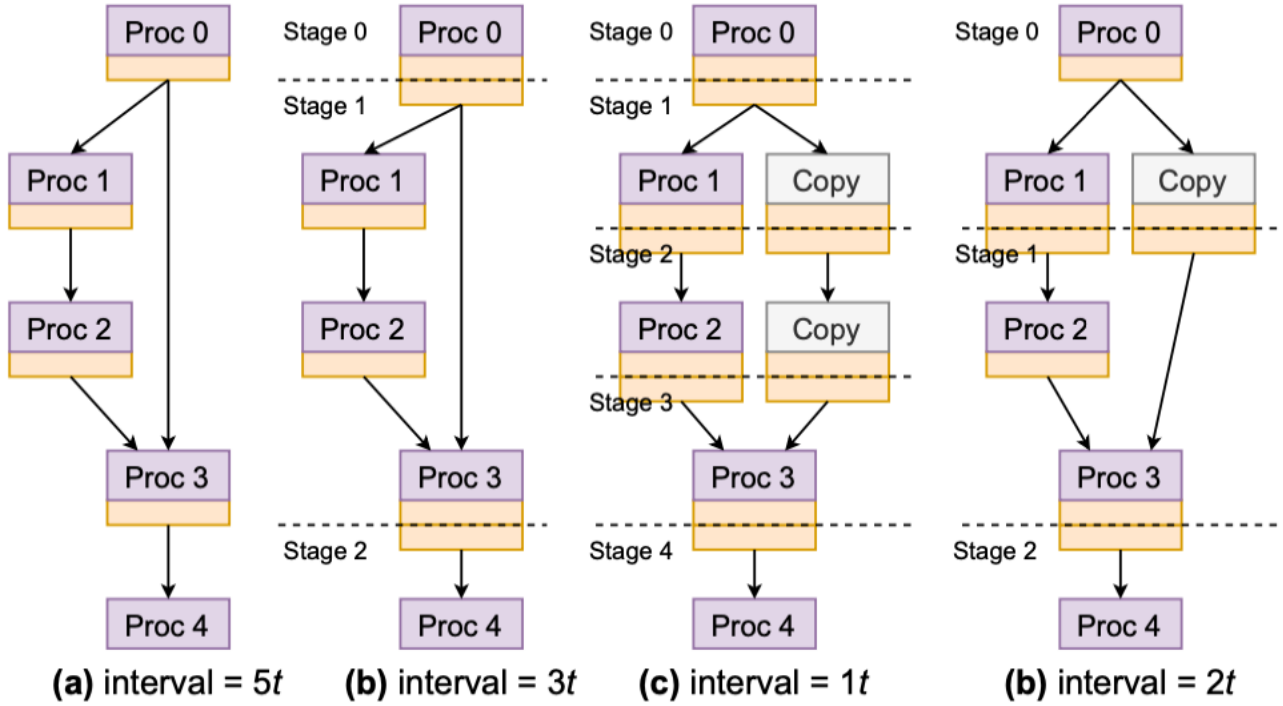


Figure 4. Graph-level dataflow optimization. (a) original dataflow; (b) legalized dataflow without copy nodes; (c) legalized dataflow with inserting copy nodes; (d) dataflow with a minimum granularity of 2.

Full Example

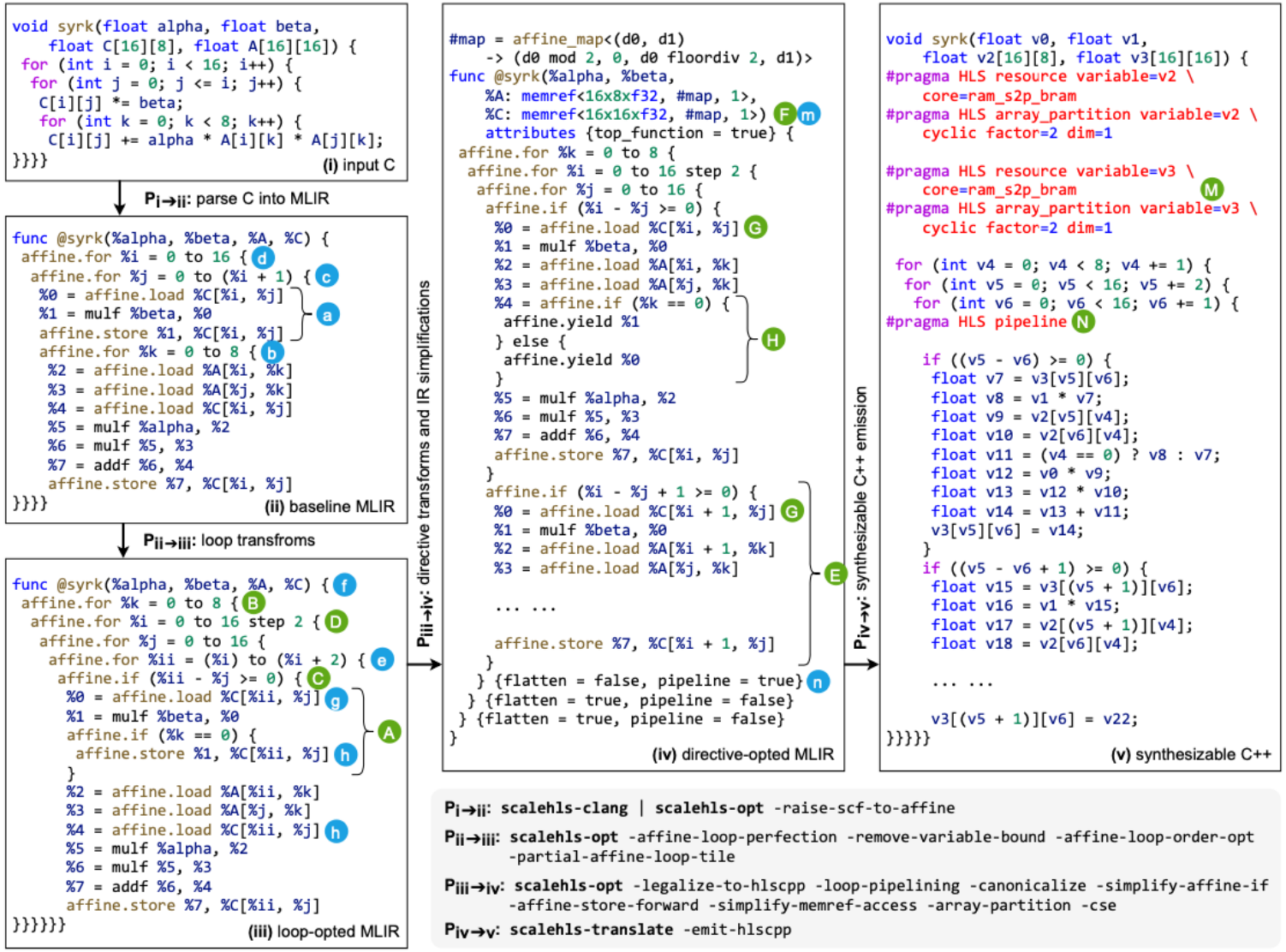


Figure 5. An SYRK computation kernel example. `scalehls-clang` compiles C program into the MLIR framework. `scalehls-opt` is the command line tool for conducting all conversion, transform, and analysis passes of ScaleHLS, while `scalehls-translate` is for the MLIR to C/C++ translation. Some operation attributes or types are omitted for simplicity.

Full Example

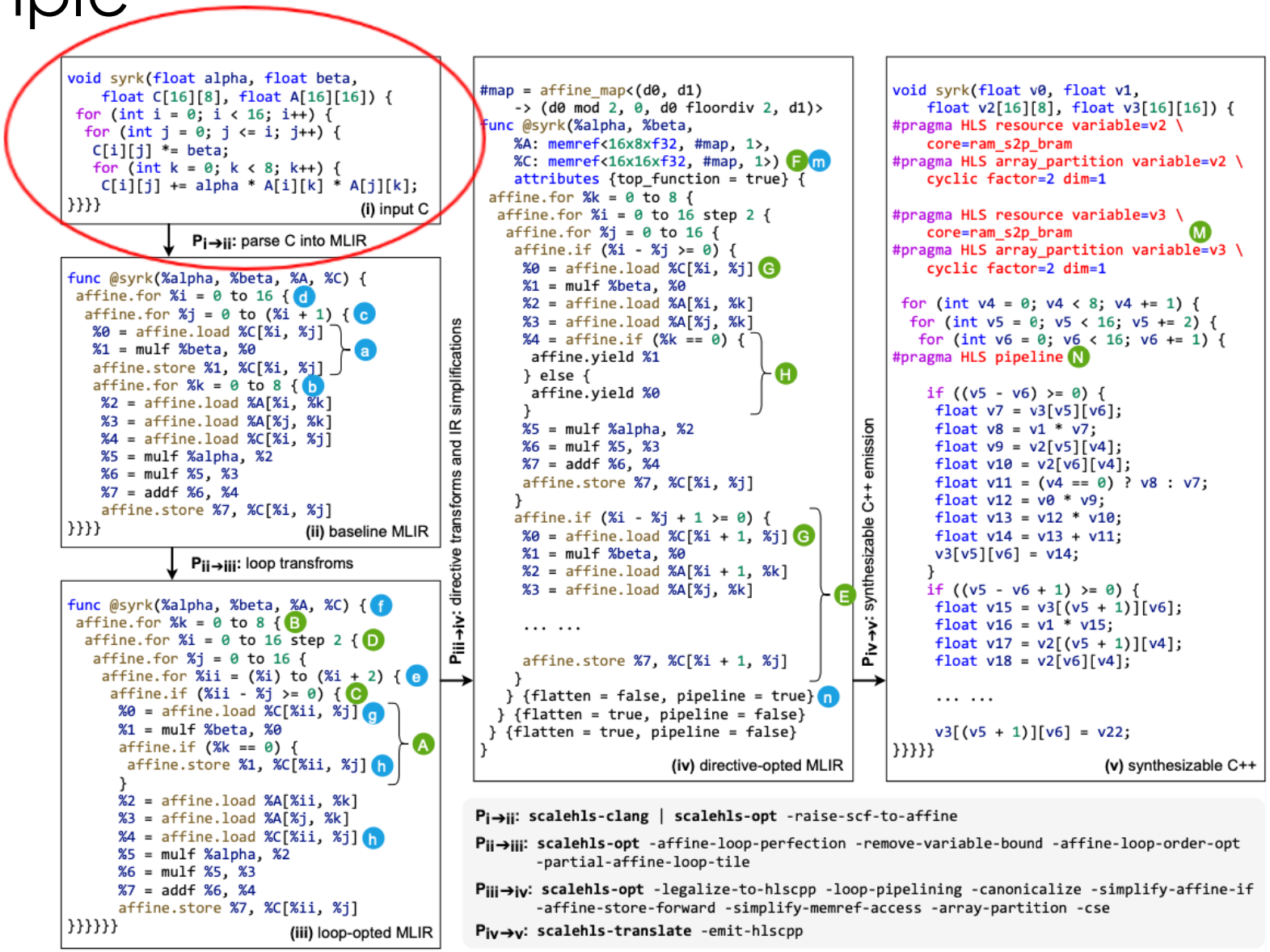


Figure 5. An SYRK computation kernel example. `scalehls-clang` compiles C program into the MLIR framework. `scalehls-opt` is the command line tool for conducting all conversion, transform, and analysis passes of ScaleHLS, while `scalehls-translate` is for the MLIR to C/C++ translation. Some operation attributes or types are omitted for simplicity.

Full Example

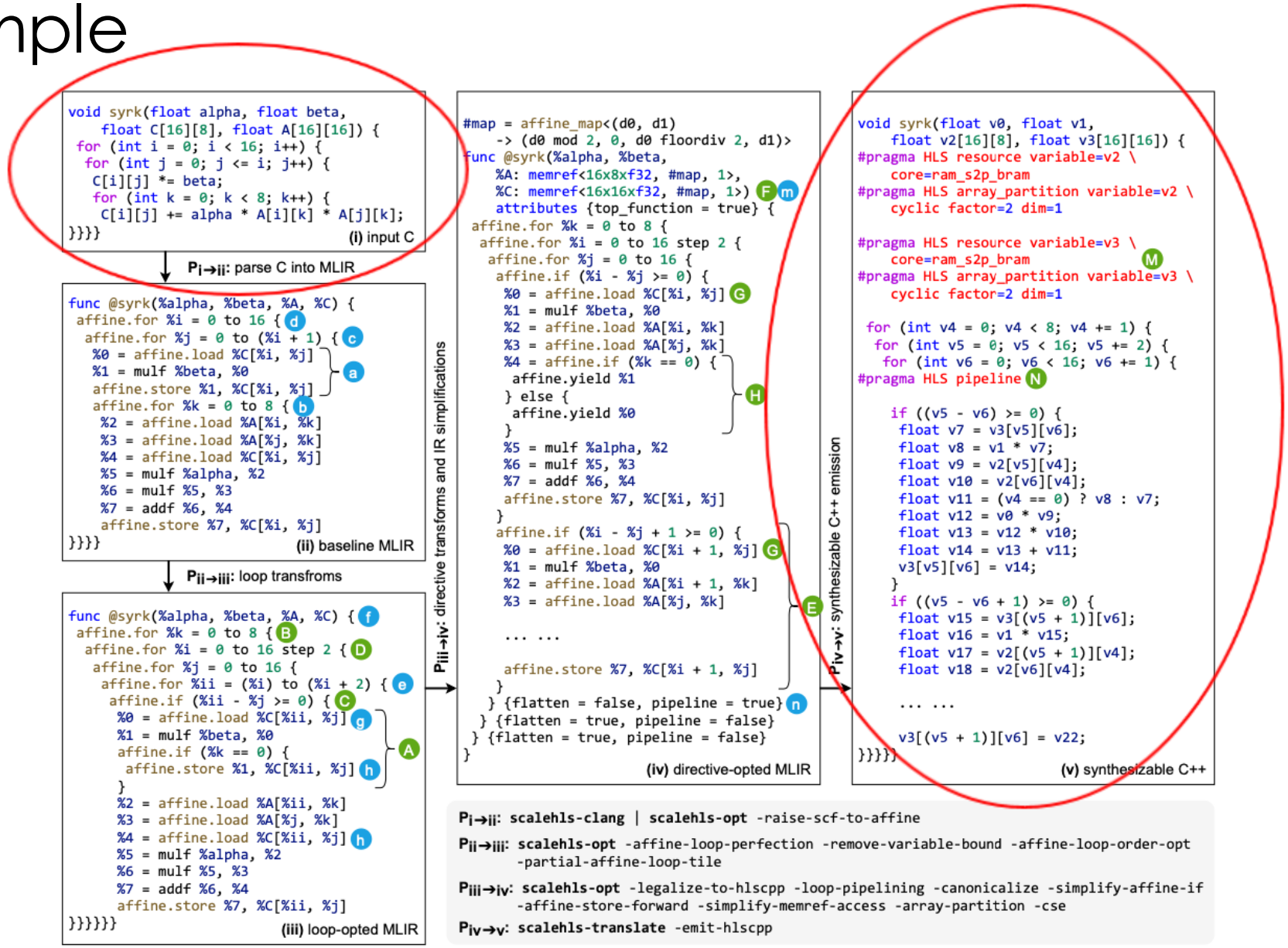


Figure 5. An SYRK computation kernel example. `scalehls-clang` compiles C program into the MLIR framework. `scalehls-opt` is the command line tool for conducting all conversion, transform, and analysis passes of ScaleHLS, while `scalehls-translate` is for the MLIR to C/C++ translation. Some operation attributes or types are omitted for simplicity.

Full Example

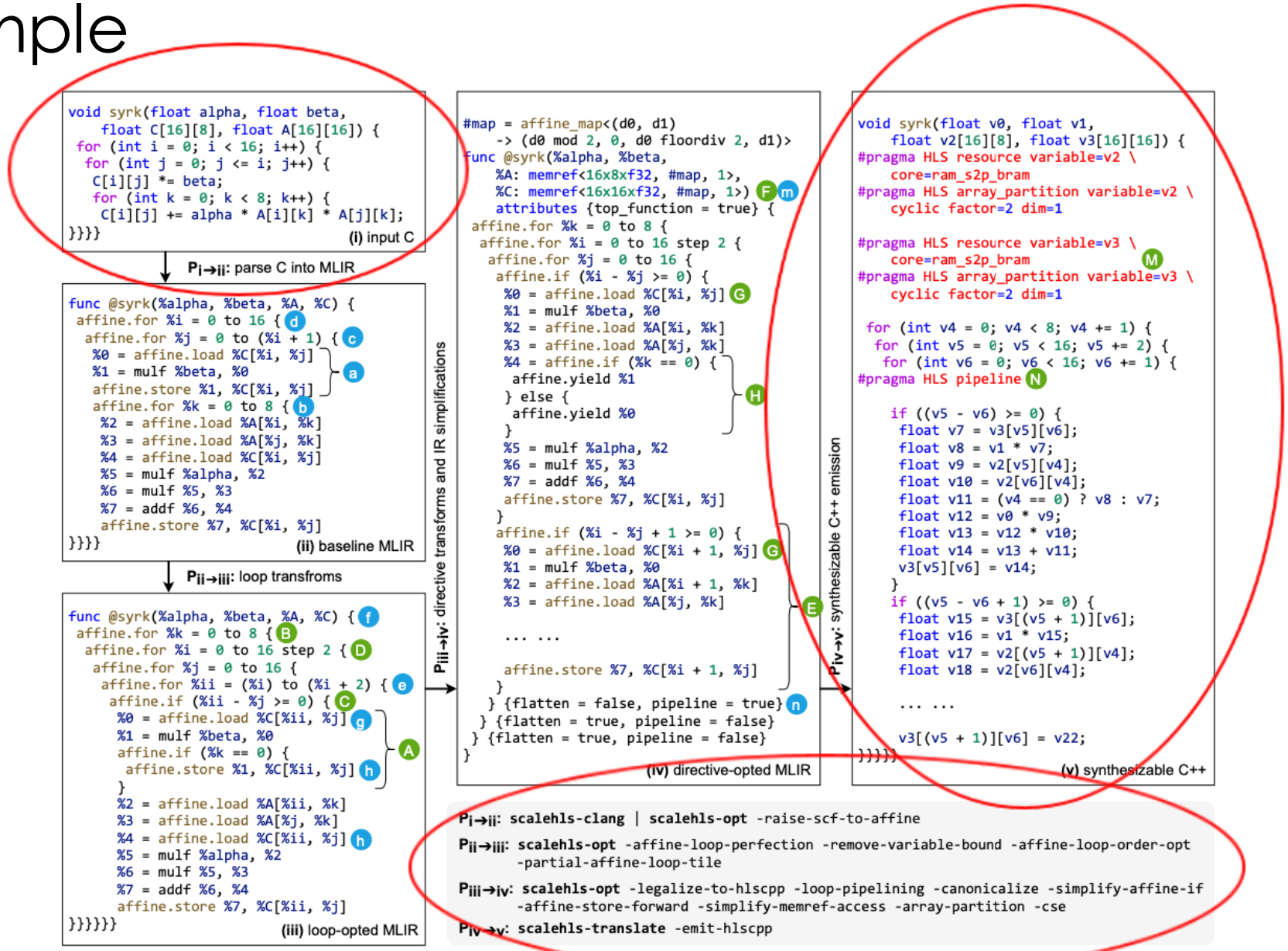


Figure 5. An SYRK computation kernel example. `scalehls-clang` compiles C program into the MLIR framework. `scalehls-opt` is the command line tool for conducting all conversion, transform, and analysis passes of ScaleHLS, while `scalehls-translate` is for the MLIR to C/C++ translation. Some operation attributes or types are omitted for simplicity.

Automatic Design Space Exploration

- Before emitting final design, perform automated DSE to find the Pareto frontier for the latency/area tradeoff
  - DSE, in this case, is just experimenting with different combinations of the available passes for ScaleHLS
- Performing PCA reveals that Pareto optimal points cluster and informs their DSE algorithm

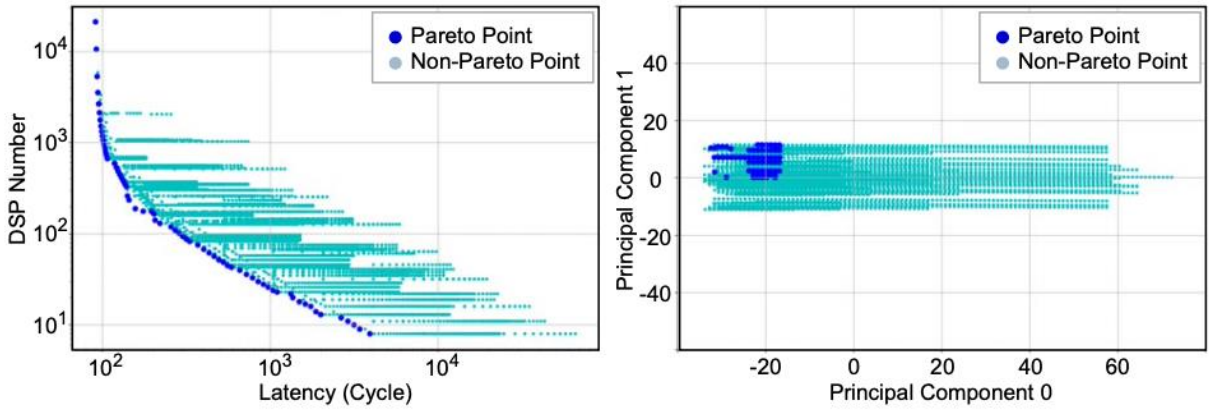


Figure 6. Design space profiling of a GEMM kernel. (a) the latency-area space; (b) PCA of the multi-dimensional design space.

OAK RIDGE National Laboratory

Results

Best Configuration for Benchmarks & Scalability Study

Table III
DSE RESULTS OF LARGE-SCALE COMPUTATION KERNELS.

Kernel	Prob. Size	Speedup	LP	RVB	Perm. Map	Tiling Sizes	Pipeline II	Array Partition Factors
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]

The data types of all kernels are 32-bits floating-points. *Speedup* is with respect to the baseline designs from PolyBench-C without the optimization of DSE. *LP* and *RVB* denote *Loop Perfectization* and *Remove Variable Bound*, respectively. In the *Loop Order Optimization*, the i-th loop in the loop nest is permuted to location $PermMap[i]$, where locations are from the outermost loop to inner.

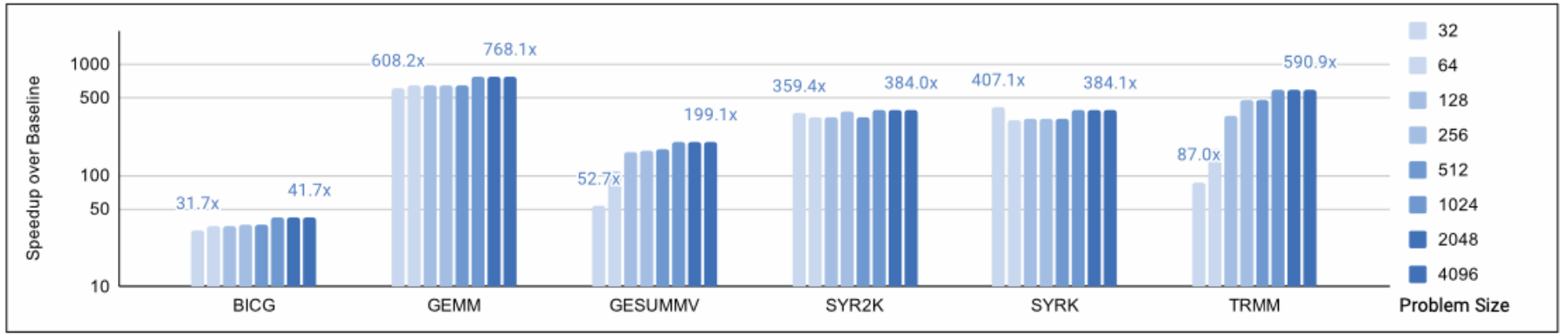


Figure 7. Scalability study of computation kernels. The problem sizes of computation kernels are scaled from 32 to 4096 and the DSE engine is launched to search for the optimized solutions under each problem size.

"Baseline" is the baseline HLS design

Result for DNN Models

Table V
OPTIMIZATION RESULTS OF REPRESENTATIVE DNN MODELS.

Model	Speedup	Runtime (seconds)	Memory (SLR Util. %)	DSP (SLR Util. %)	LUT (SLR Util. %)	Our DSP Effi. (OP/Cycle/DSP)	DSP Effi. of TVM-VTA [49]
ResNet-18	3825.0×	60.8	91.7Mb (79.5%)	1326 (58.2%)	157902 (40.1%)	1.343	0.344
VGG-16	1505.3×	37.3	46.7Mb (40.5%)	878 (38.5%)	88108 (22.4%)	0.744	0.296
MobileNet	1509.0×	38.1	79.4Mb (68.9%)	1774 (77.8%)	138060 (35.0%)	0.791	0.468

Speedup is with respect to the baseline designs compiled from PyTorch by ScaleHLS but without the multi-level optimization.

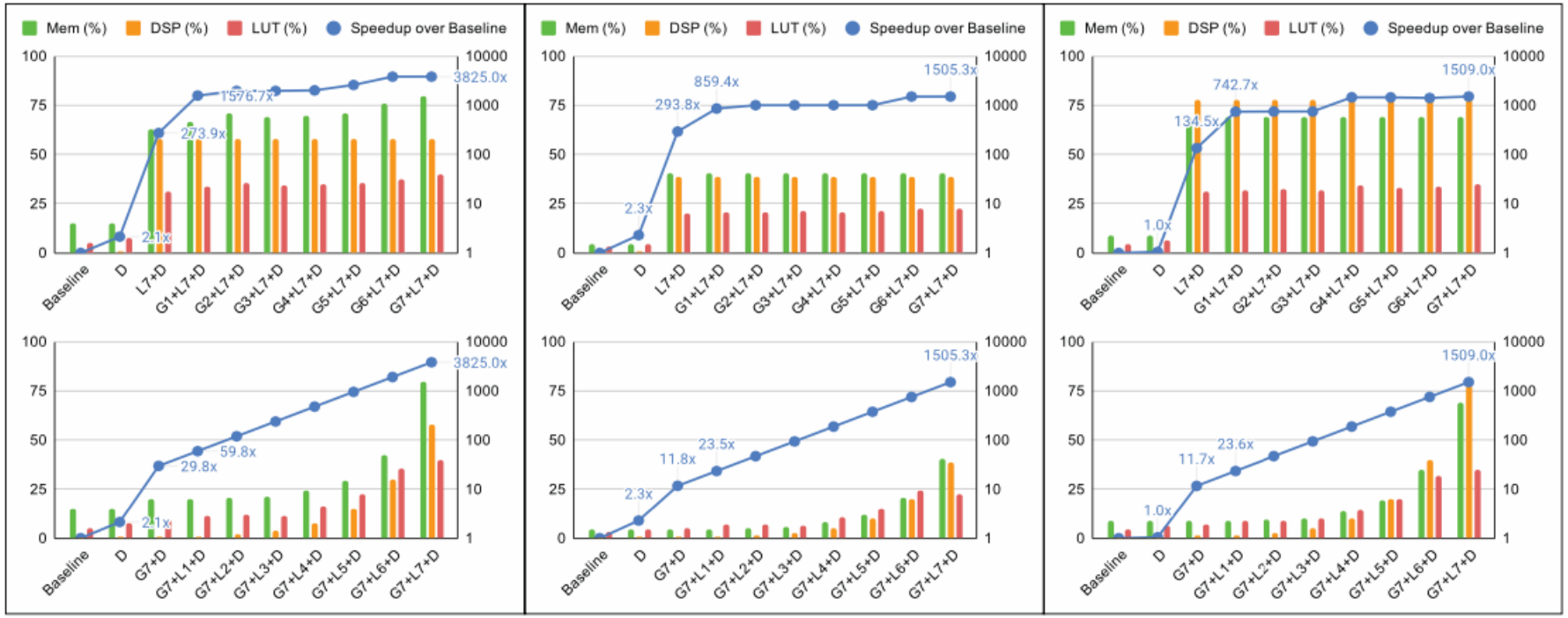


Figure 8. Ablation study of DNN models. D, $L\{n\}$, and $G\{n\}$ denote directive, loop, and graph optimizations, respectively. Larger n indicates larger loop unrolling factor and finer dataflow granularity for loop and graph optimizations, respectively.

"Baseline" is the baseline HLS design

Conclusion

- Contribution
  - An end-to-end framework that closes the semantic gap between HLS and Verilog
- Strengths
  - Novel approach to closing the semantic gap between HLS and RTL
  - Code is open-source
- Weaknesses
  - Number of benchmarked applications is small and of a similar flavor (GEMM). More applications from different domains would be beneficial
  - Approach requires using AMD software ecosystem and hardware backends