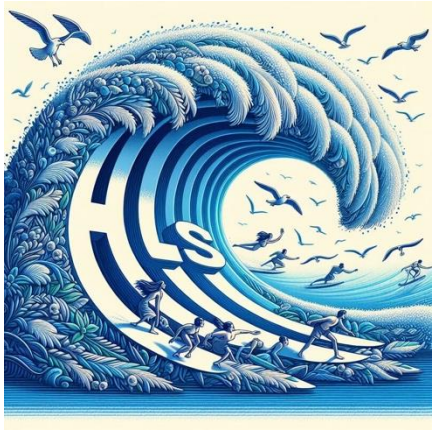
UNIVERSITY OF ILLINOIS URBANA-CHAMPAIGN

The Next Wave of HLS:
Fully Automated PyTorch-to-Accelerator Design Flow for FPGAs

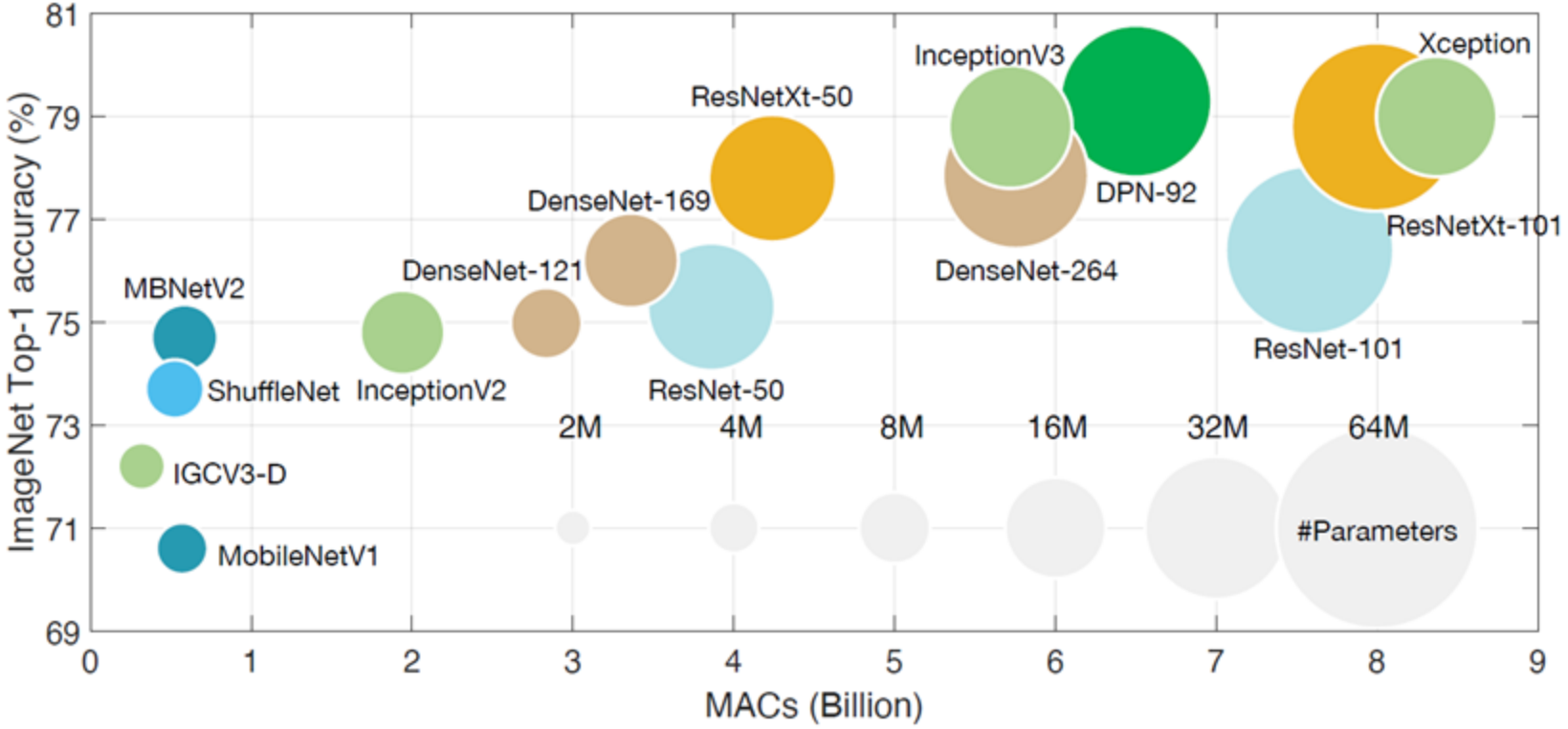


Deming Chen

Abel Bliss Professor, ECE, UIUC

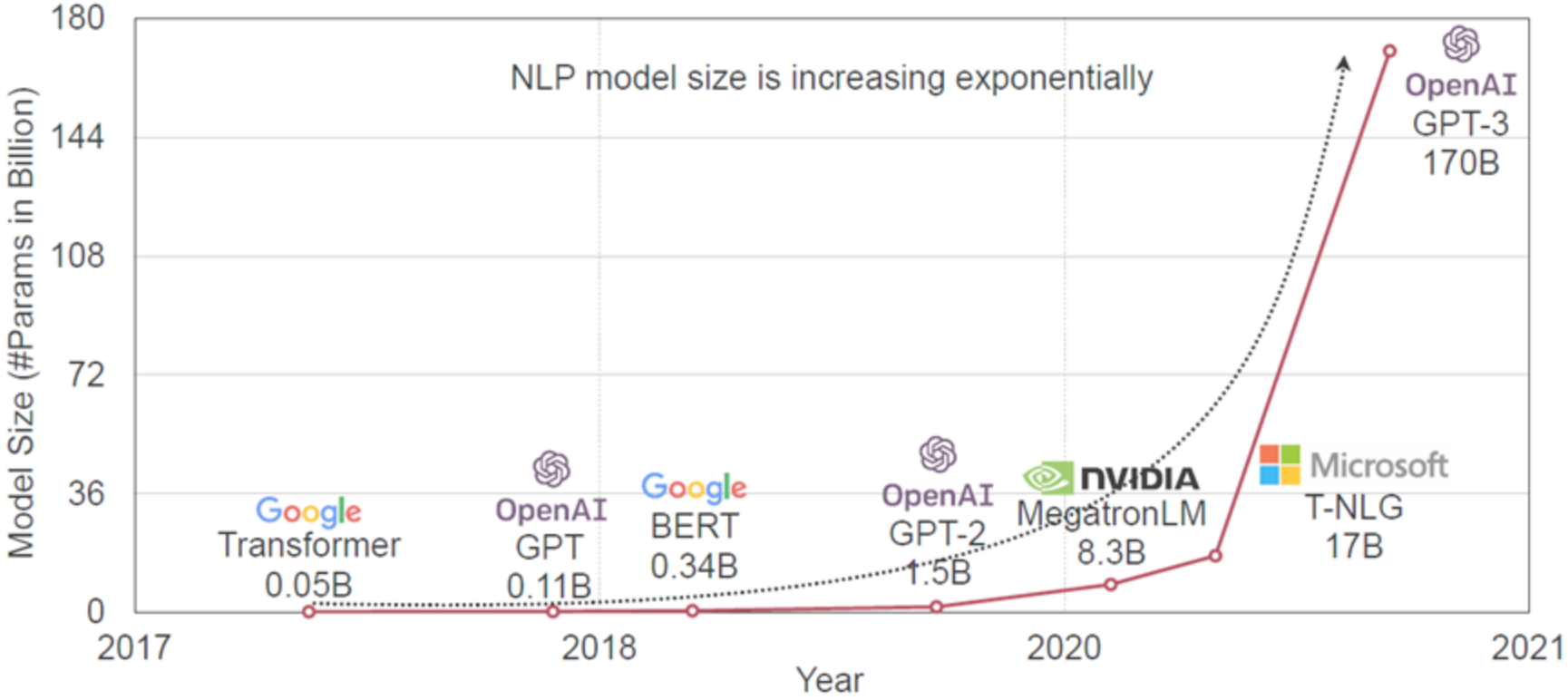
WRC, Jan 2025, Barcelona, Spain

Computational Cost of DNNs Growing



- Model Compression and Hardware Acceleration for Neural Networks: A Comprehensive Survey [Deng et al., IEEE 2020]

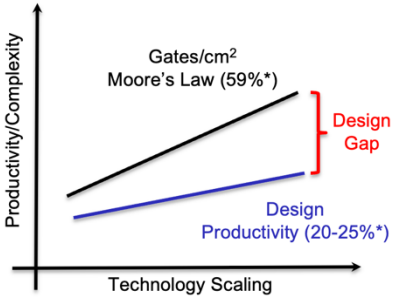
Size of Language Models Growing Exponentially



- TinyML and Efficient Deep Learning Computing [Han, 2023]

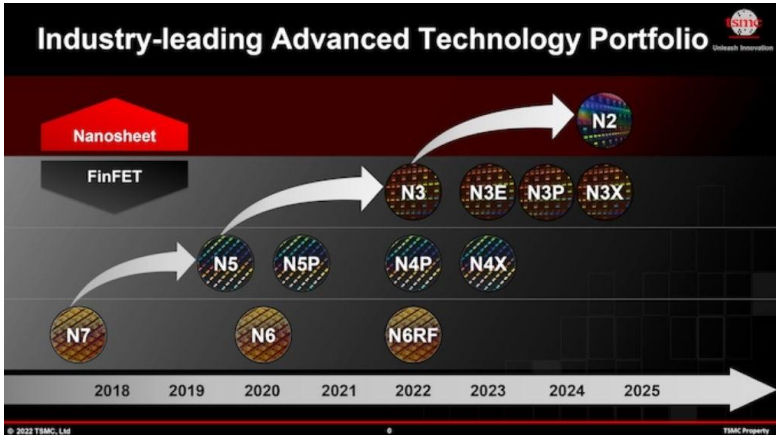
Critical need of various AI accelerators

Design Productivity Continues to Fall Behind

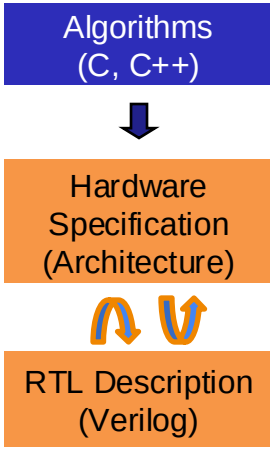


* Compound Annual Growth Rate

Source: Semico Research Corp.



https://www.phonearena.com/news/tsmc-roadmap-released_id140823



Which architecture to select?

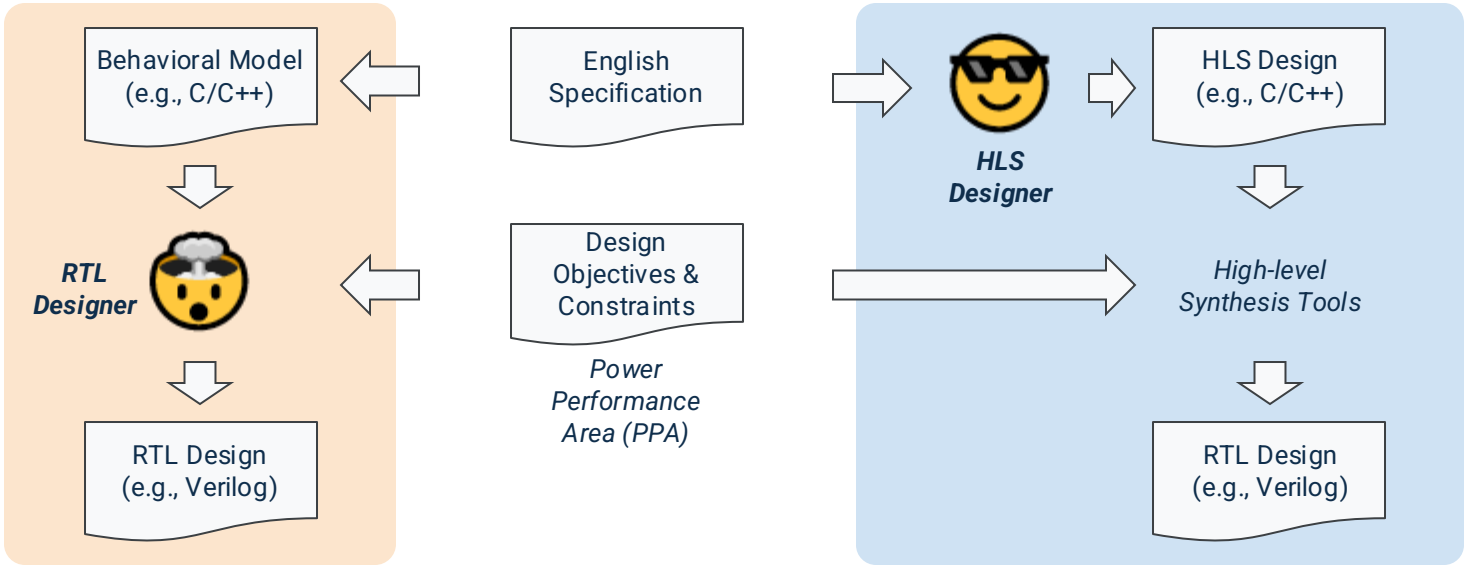
Complex and time consuming!

Platform	Normalized Speed-Up	Normalized Performance/ Watt	Development Time in Days
FPGA	545:1	1090:1	60
GPU	50:1	21:1	3
GPP	1:1	1:1	1

Quasi-Monte-Carlo-based Financial Option Pricing algorithm

Source: Khaled Benkrid, HEART'2010

Ongoing Need for High-Level Synthesis (HLS)



RTL Design Flow

- **Manual** optimization and scheduling
- **Long** design cycle
- **Low** portability against different PDK or PPA requirements

HLS Design Flow

- **Automated** optimization and scheduling
- **Short** design cycle
- **High** portability against different PDK or PPA requirements

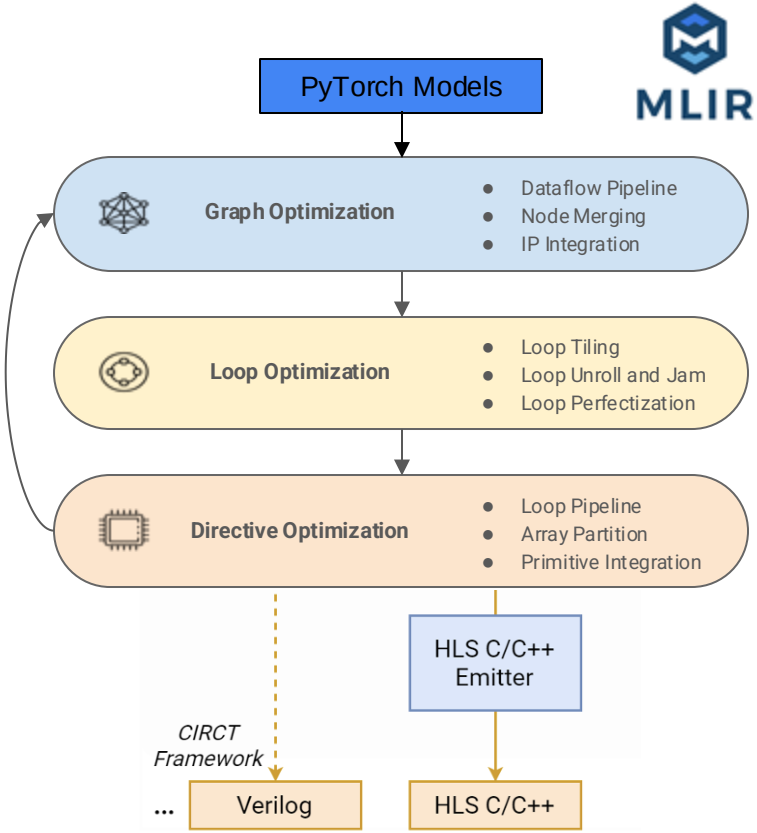
- PDK: Process design kit.

ScaleHLS: an Automated PyTorch-to-Accelerator Compiler

- New HLS solutions translating AI models to customized AI accelerators automatically.
  - By adopting PyTorch as input for AI designs (instead of traditional C/C++ for HLS), the lines of code and design simulation time can be reduced by about **10x** and **100x**, respectively.

- Such AI model-to-RTL flows pave the way for a new wave of HLS
  - Drive the high-productivity designs of AI circuits with high density, high-energy efficiency, low cost, and short design cycle.
  - Can be expanded to other non-AI domains.

- Can generate an AI chiplet from a PyTorch model automatically through this compiler



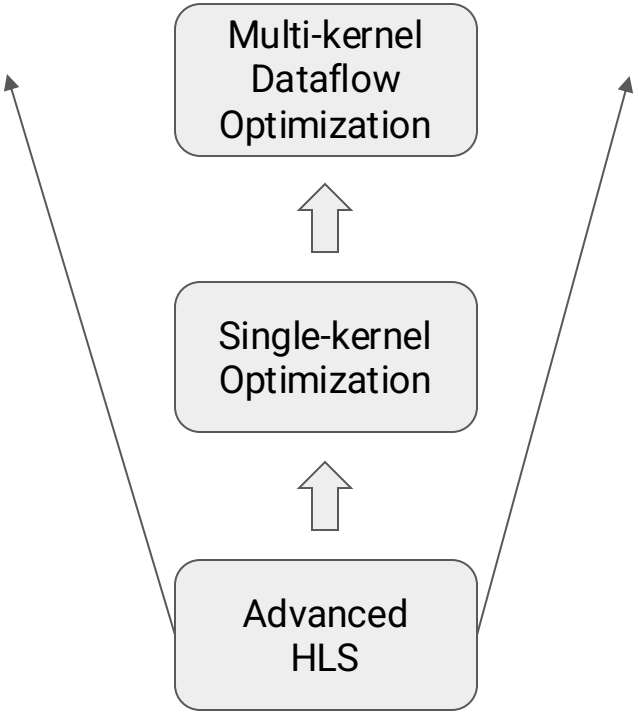
ScaleHLS Framework

https://github.com/UIUC-ChenLab/ScaleHLS-HIDA

>60k views and ~4000 downloads

[1] Ye, Hanchen, et al., "ScaleHLS: A New Scalable High-Level Synthesis Framework on Multi-Level Intermediate Representation", HPCA, 2022.
[2] Ye, Hanchen, et al., "Invited: ScaleHLS, a Scalable High-Level Synthesis Framework with Multi-level Transformations and Optimizations", DAC, 2022.
[3] Ye, Hanchen, et al., "High-level Synthesis for Domain Specific Computing", ISPD, 2023.
[4] Ye, Hanchen, et al, "ScaleFlow: High-Level Synthesis for Large Dataflow Applications", TECHCON, 2023.
[5] Jun, Greg, et al, "AutoScaleDSE: A Scalable Design Space Exploration Engine for High-Level Synthesis", TRETS, 2023.
[6] Ye, Hanchen, et al, "HIDA: A Hierarchical Dataflow Compiler for High-Level Synthesis", ASPLOS, 2024.

A New Proposed Toolchain



HIDA (ScaleHLS V2): A Dataflow Compiler for High-level Synthesis [TECHCON'23] [ASPLOS'24]

ScaleHLS: A Scalable High-level Synthesis Framework on Multi-level Intermediate Representation (MLIR) [HPCA'22] [DAC'22] [TRETS'23] [ISPD'23]

Xcelo™: A New HLS Tool with Full Automation

inspirj.T



Background and Motivation

Motivation

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**

Motivation (Cont.)

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.

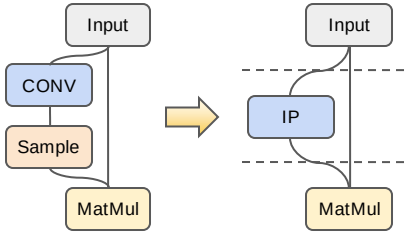
```
for (int k = 0; k < 32; k++) {
  for (int i = 0; i < 32; i++) {
    for (int j = 0; j < 32; j++) {
#pragma HLS pipeline
      if (k == 0)
        C[i][j] *= beta;
      C[i][j] += alpha * A[i][k] * B[k][j];
} } }
```

Generate RTL with XILINX VITIS and etc.

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

Motivation (Cont.)

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];
} } }
```

Graph Optimizations

Node fusion
IP integration
Task-level pipeline, 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];
} } }
```

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++) {
#pragma HLS pipeline
      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.

Generate RTL with XILINX VITIS and etc.

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

Motivation (Cont.)

Difficulties:

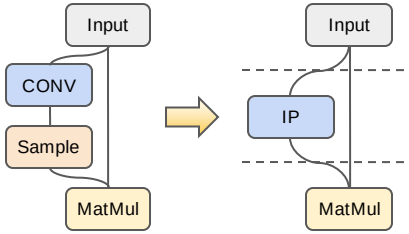
- Low-productive and error-proning
- Hard to enable automated design space exploration (DSE)
- NOT scalable!

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

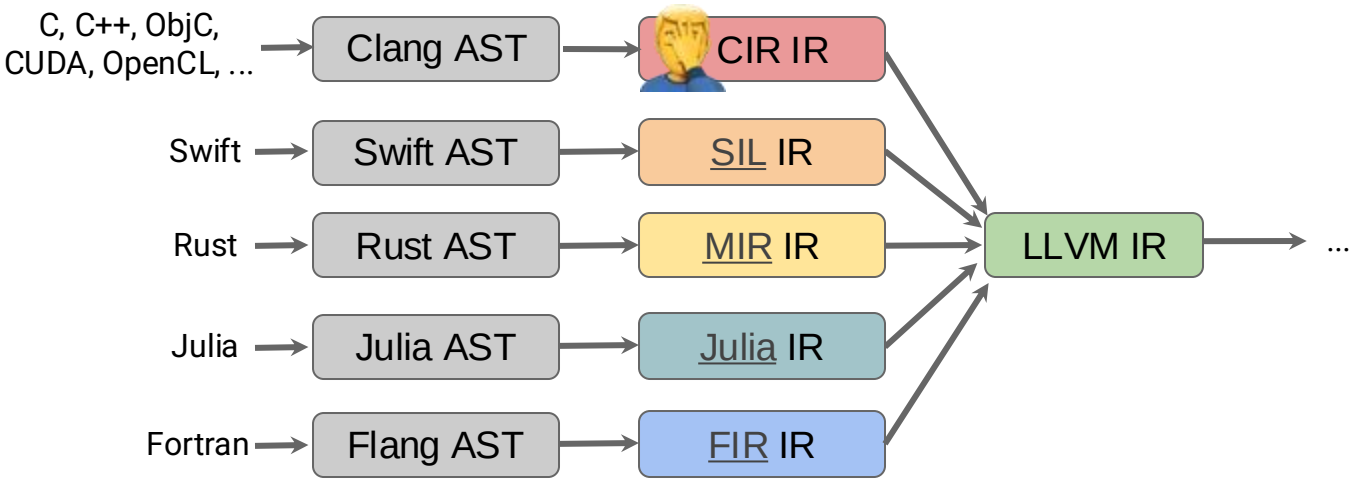
Directive Optimizations
Loop pipeline, unroll
Function pipeline, inline
Array partition, etc.

Manual Code Rewriting

Generate RTL with XILINX VITIS and etc.

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

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.

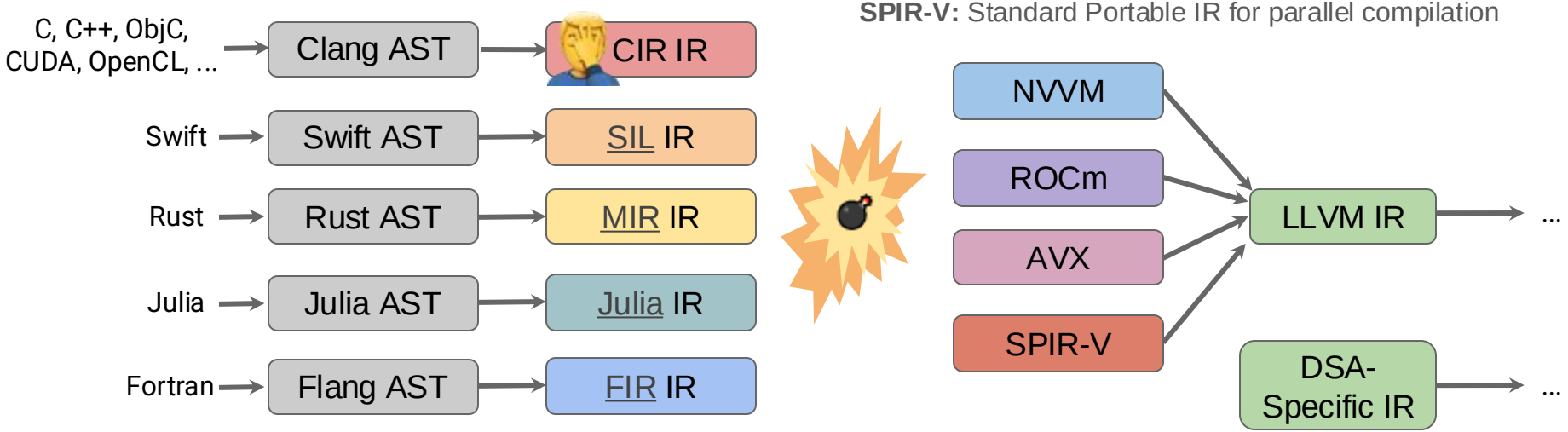
From LLVM to MLIR (Cont.)

NVVM: IR for Nvidia GPU

ROCm: IR for AMD GPU

AVX: IR for Intel vector extension

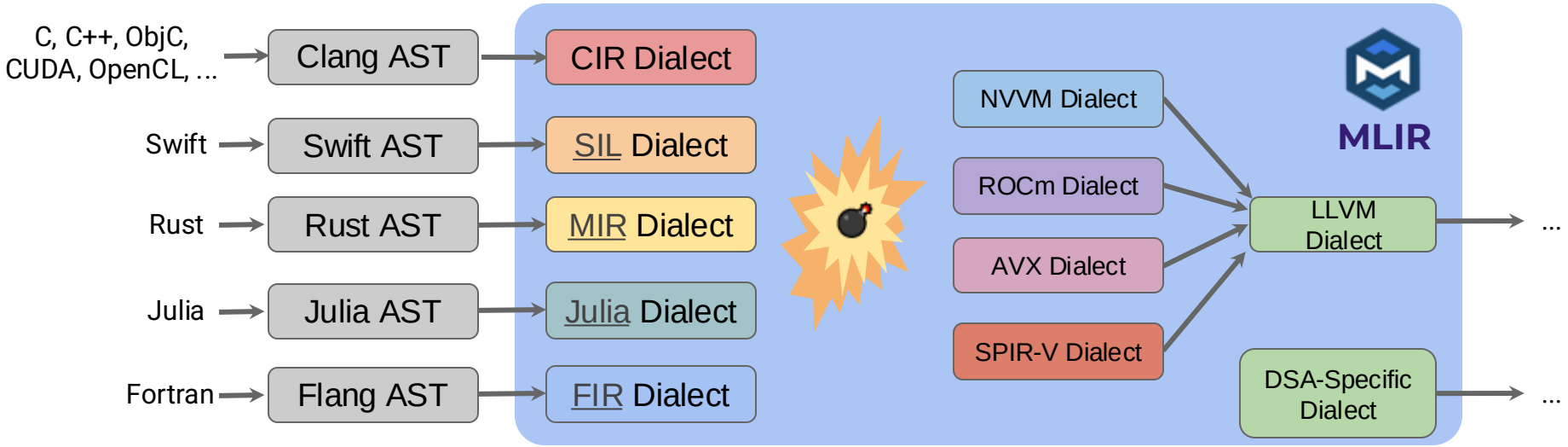
SPIR-V: Standard Portable IR for parallel compilation



- 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: “Meta IR” and Compiler Infrastructure



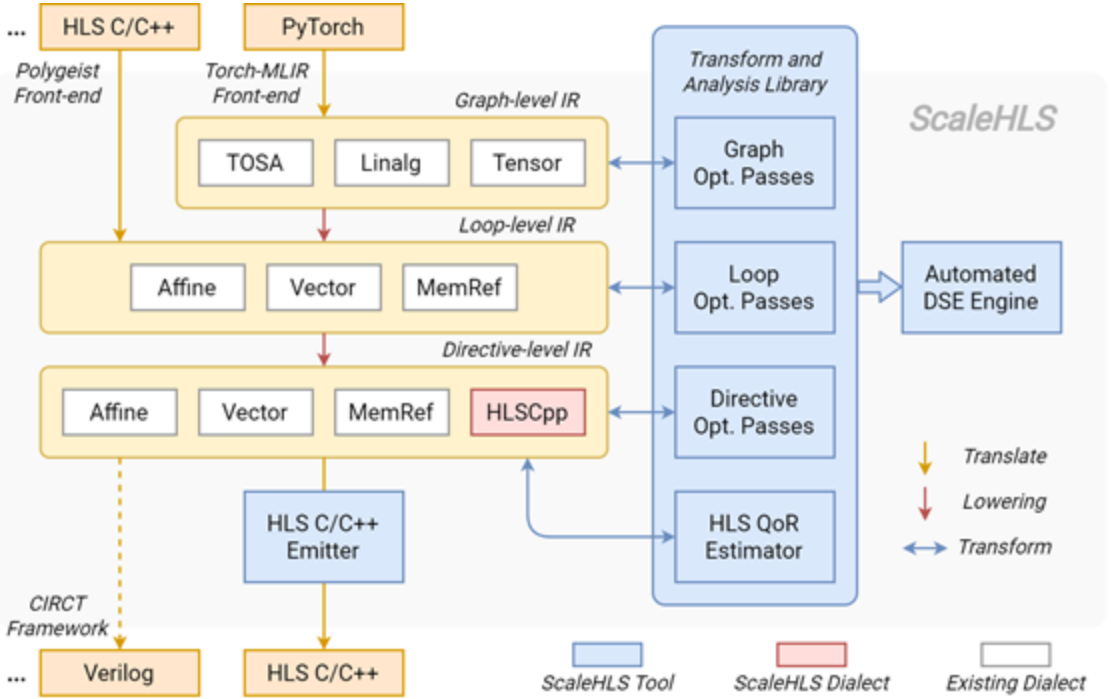
MLIR is a **“Meta IR”** and **compiler infrastructure** for:

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



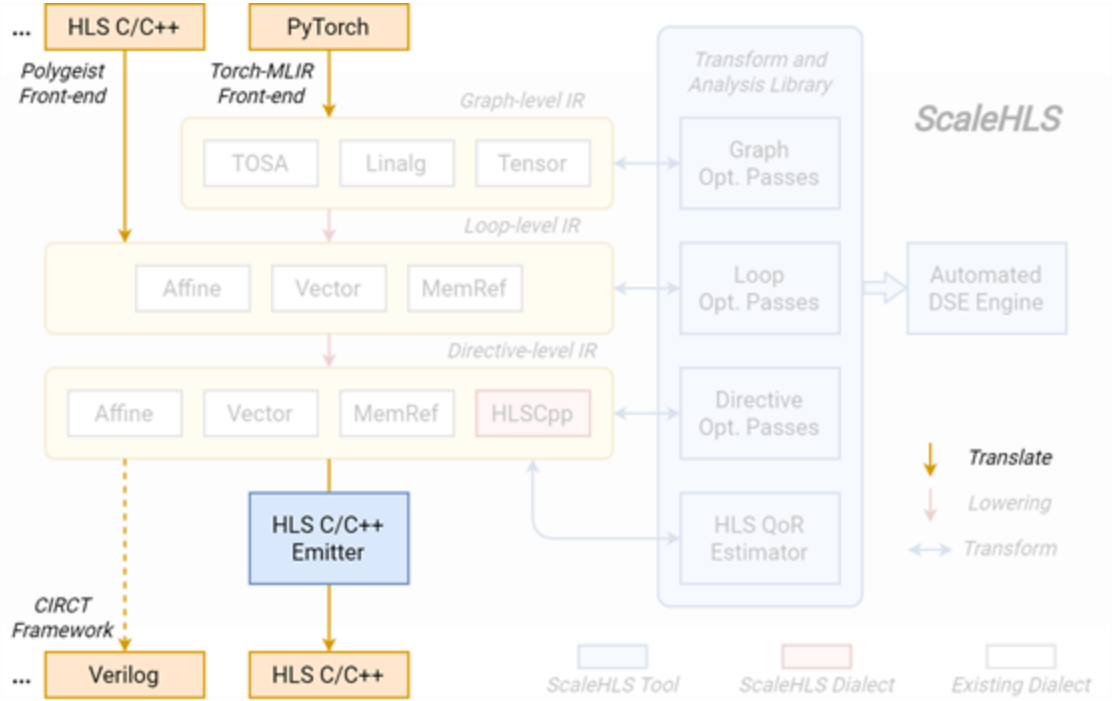
Multi-level HLS IR and Optimization

ScaleHLS Framework: Overview



[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



Inputs

C/C++ Polygeist [1]

PyTorch Torch-MLIR [2]

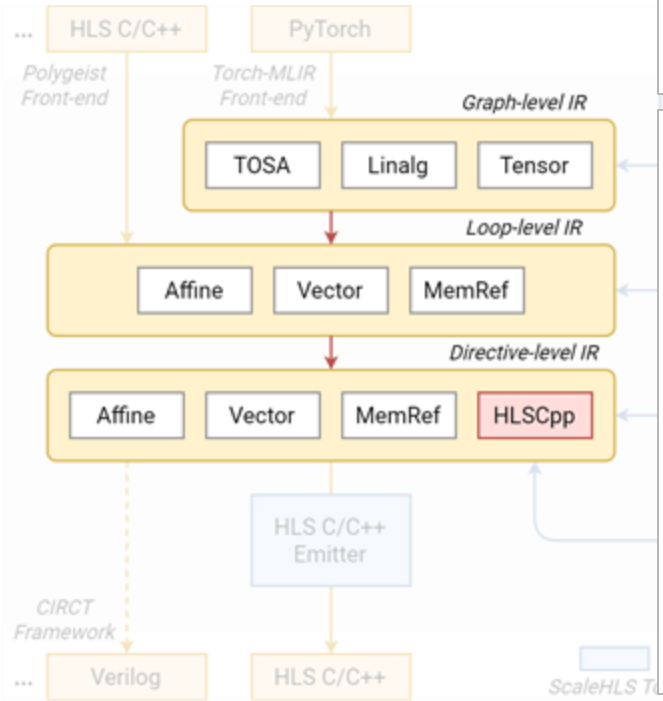
Outputs

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



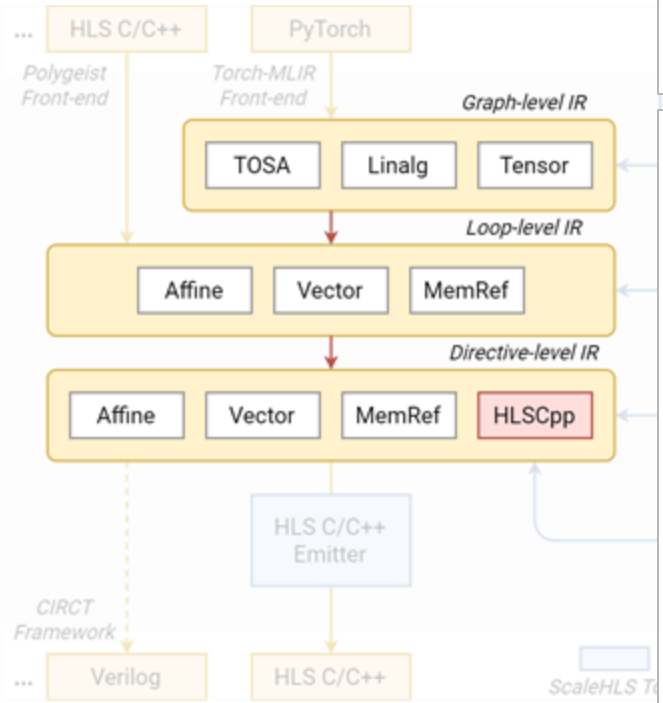

```
%O = "tosa.conv2d"(%I, %W, ...) {...} : (tensor<1x32x32x3xi8>, tensor<64x3x3x3xi8>, ...) ->
                                        tensor<1x30x30x64xi8>
```

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)



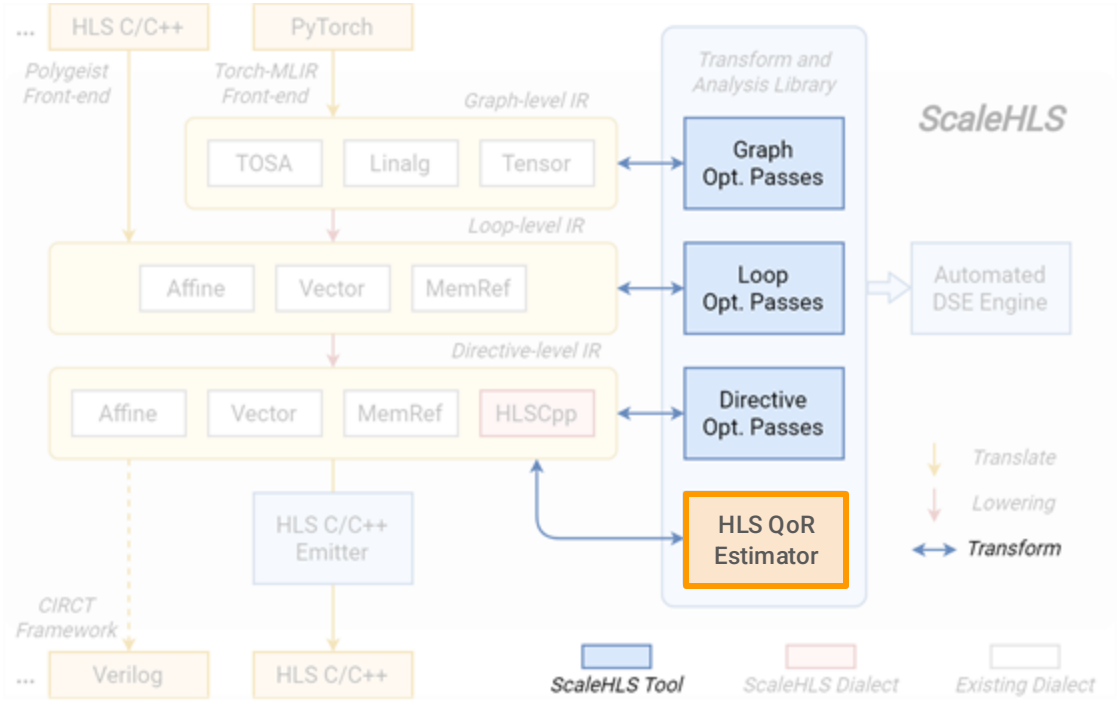

```
%O = "tosa.conv2d"(%I, %W, ...) {...} : (tensor<1x32x32x3xi8>, tensor<64x3x3x3xi8>, ...) ->
                                        tensor<1x30x30x64xi8>
```

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 Optimizations

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

Boldface ones are new passes provided by us, while others are MLIR built-in passes.

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

Baseline C

Loop and Directive Opt in MLIR



```
void syrk(int alpha, int beta, int C[32][32], int A[32][32]) {
#pragma HLS interface s_axilite port=return bundle=ctrl
#pragma HLS interface s_axilite port=alpha bundle=ctrl
#pragma HLS interface s_axilite port=beta bundle=ctrl
#pragma HLS interface bram port=C
#pragma HLS interface bram port=A

#pragma HLS resource variable=C core=ram_s2p_bram

#pragma HLS array_partition variable=A cyclic factor=2 dim=2
#pragma HLS resource variable=A core=ram_s2p_bram

  for (int k = 0; k < 32; k += 2) {
    for (int i = 0; i < 32; i += 1) {
      for (int j = 0; j < 32; j += 1) {
#pragma HLS pipeline II = 3
        if ((i - j) >= 0) {
          int v7 = C[i][j];
          int v8 = beta * v7;
          int v9 = A[i][k];
          int v10 = A[j][k];
          int v11 = (k == 0) ? v8 : v7;
          int v12 = alpha * v9;
          int v13 = v12 * v10;
          int v14 = v11 + v13;
          int v15 = A[i][(k + 1)];
          int v16 = A[j][(k + 1)];
          int v17 = alpha * v15;
          int v18 = v17 * v16;
          int v19 = v14 + v18;
          C[i][j] = v19;
} } } } }
```

Optimized C emitted by the C/C++ emitter

ScaleHLS 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 unrolling, 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

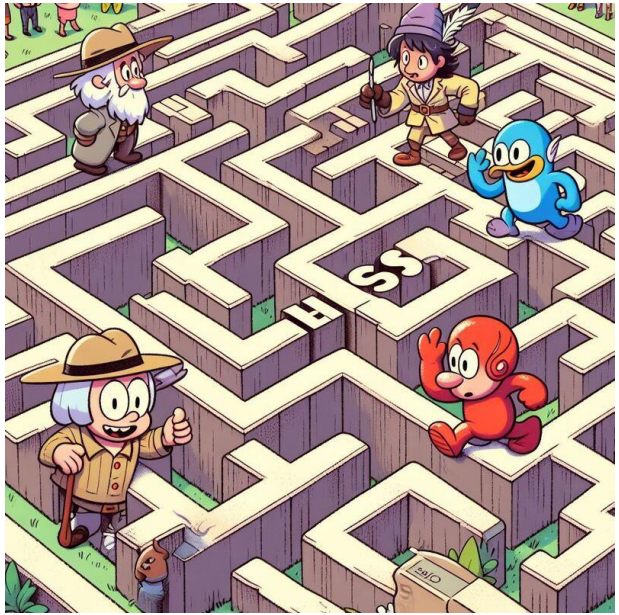
Loop order permutation; Loop unroll

Remove variable loop bound

Loop pipeline

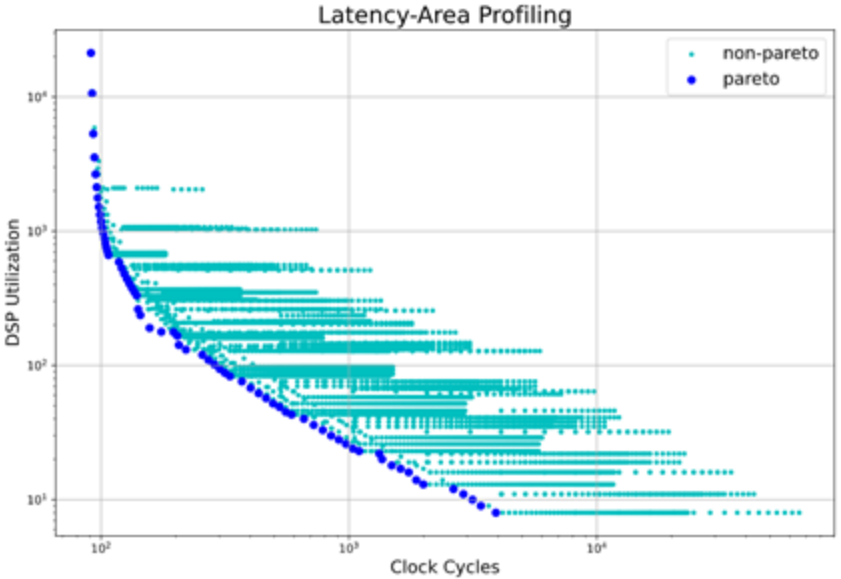
Simplify if ops;
Store ops forward;
Simplify memref ops

Optimized C emitted by the C/C++ emitter



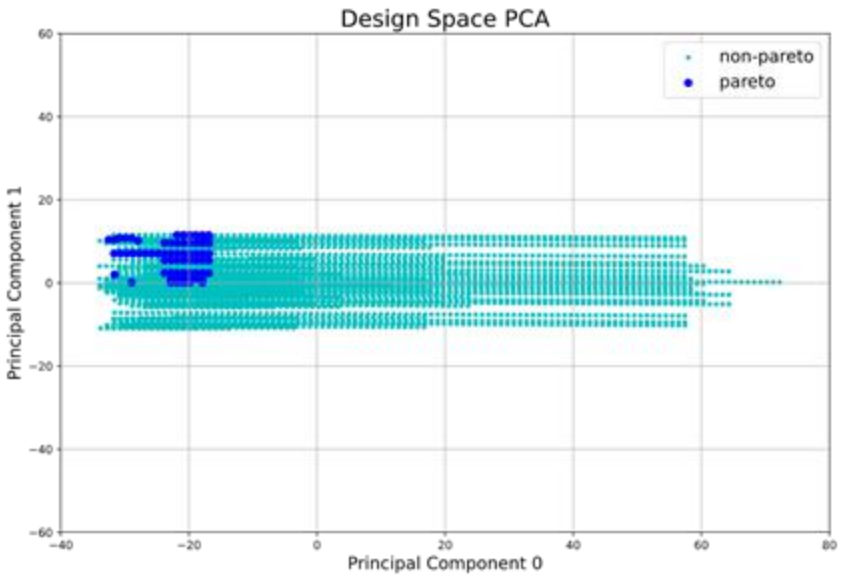
Single-kernel Design Space Exploration (DSE)

Intra-node Design Space Exploration



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

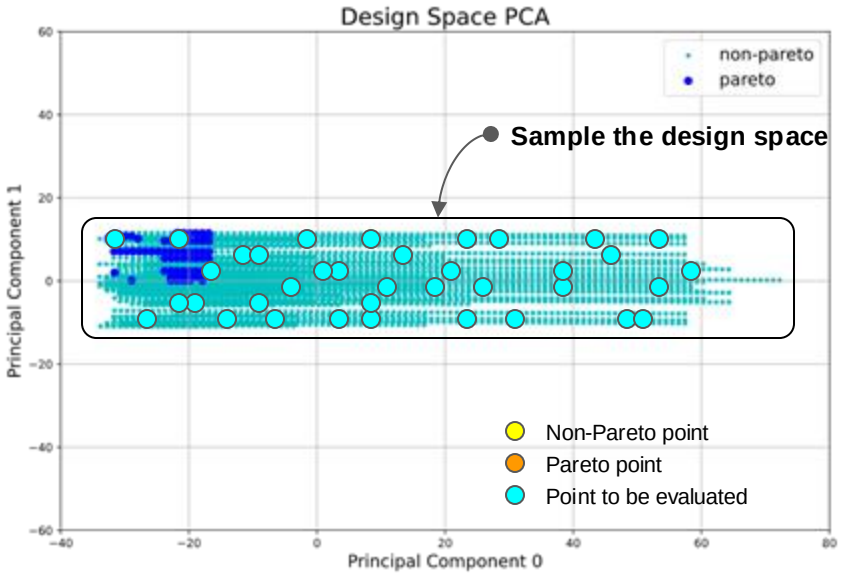


- Each parameter of a pass becomes one dimension, the original multi-dimensional design space is reduced to two dimensions through PCA
- Pareto points are located at a corner of the design space, the variance of Pareto points is much smaller than the overall variance

Intra-node Design Space Exploration (Cont.)

DSE algorithm:

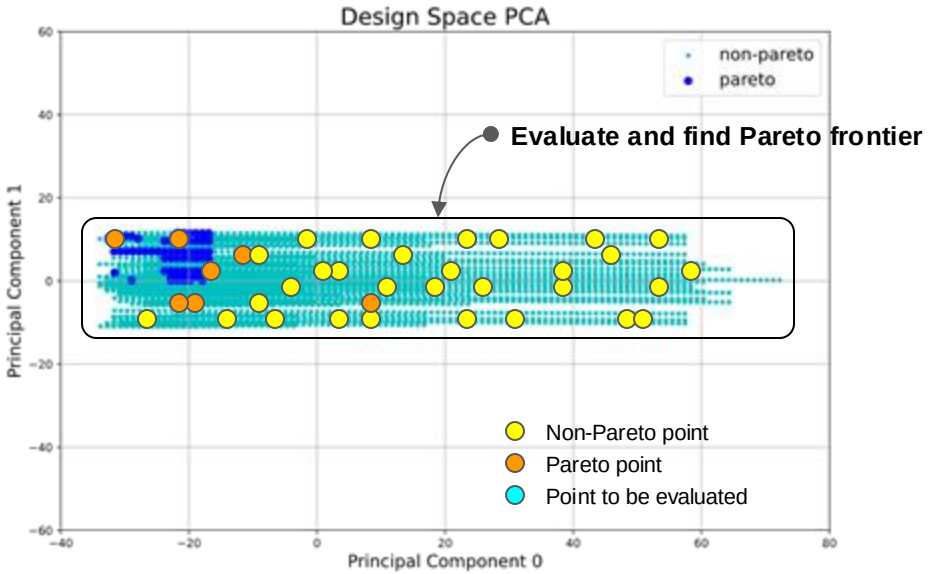
1. Sample the whole design space and evaluate each sampled design point with the QoR estimator



Intra-node Design Space Exploration (Cont.)

DSE algorithm:

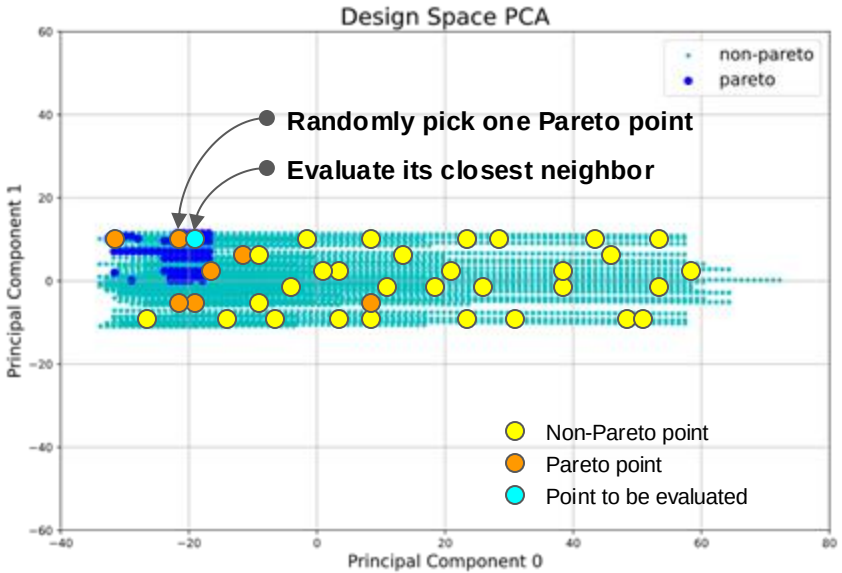
1. Sample the whole design space and evaluate each sampled design point with the QoR estimator
2. Extract the Pareto frontier from all evaluated design points



Intra-node Design Space Exploration (Cont.)

DSE algorithm:

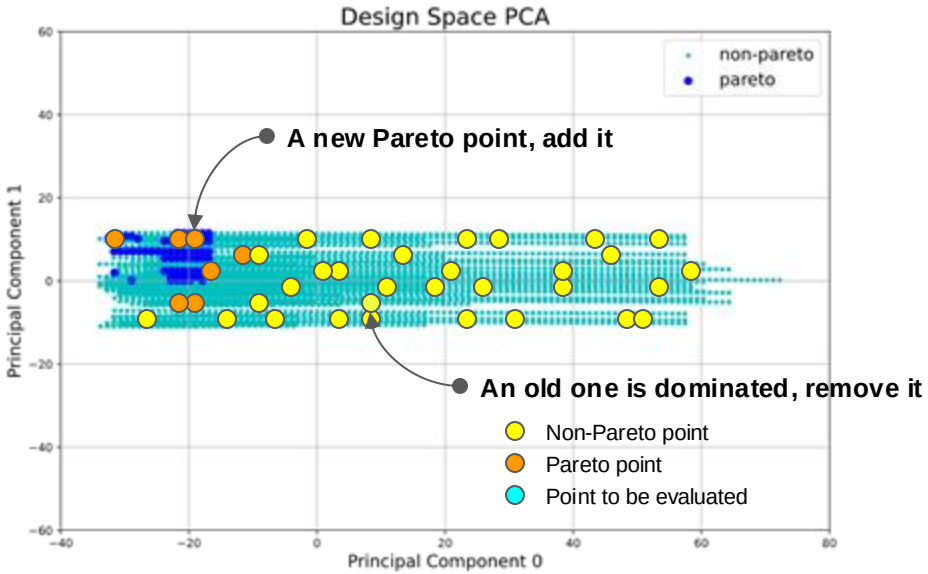
1. Sample the whole design space and evaluate each sampled design point with the QoR estimator
2. Extract the Pareto frontier from all evaluated design points
3. Evaluate the closest neighbor of a randomly selected design point in the current Pareto frontier



Intra-node Design Space Exploration (Cont.)

DSE algorithm:

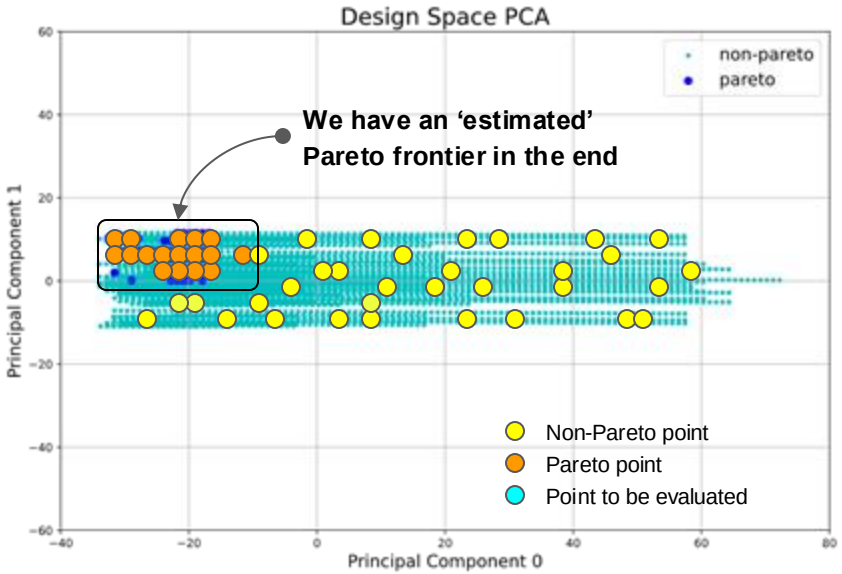
1. Sample the whole design space and evaluate each sampled design point with the QoR estimator
2. Extract the Pareto frontier from all evaluated design points
3. Evaluate the closest neighbor of a randomly selected design point in the current Pareto frontier
4. Repeat step (2) and (3) to update the discovered Pareto frontier



Intra-node 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.

Single-Kernel DSE Results

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. **LP** and **RVB** denote Loop Perfectization and Remove Variable Bound, respectively.
3. 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.

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.

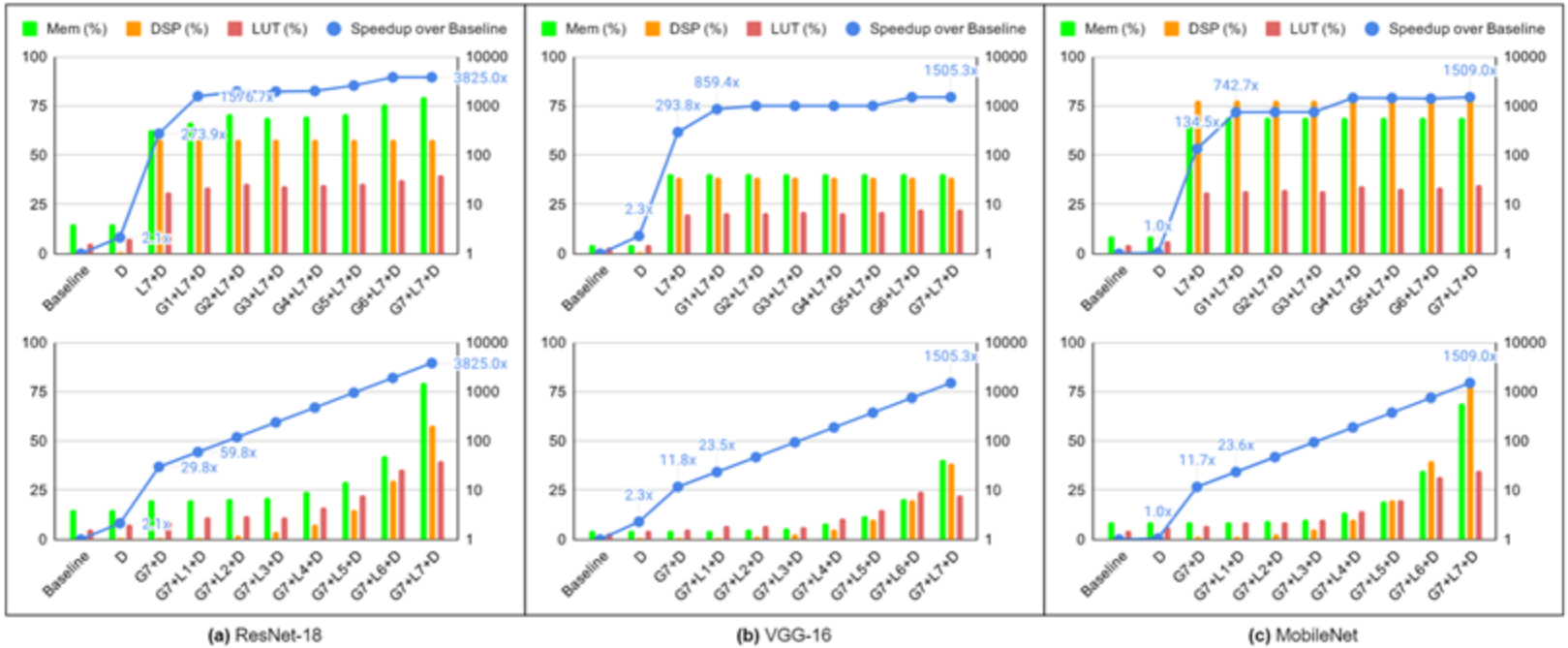
Optimization Results of DNN Models

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.)



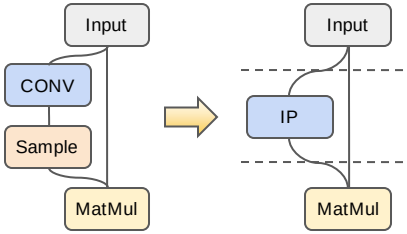
Ablation study of DNN models

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.



Dataflow IR and Optimization

Limitations of ScaleHLS DSE



Graph Optimizations

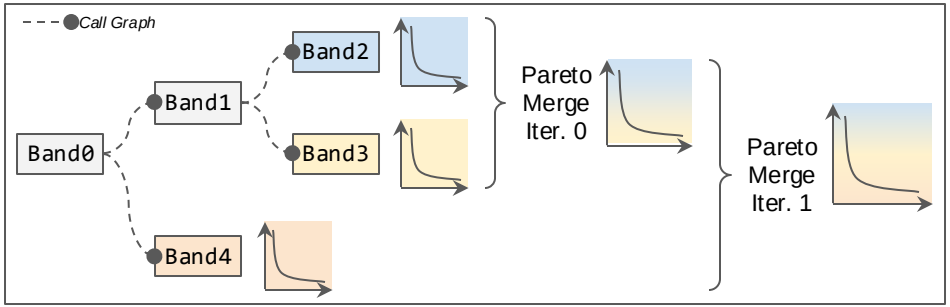
```
for (int i = 0; i < 32; i++) {
  for (int j = 0; j < 32; j++) {
    C[i][j] *= beta;
    for (int k = 0; k < 32; k++) {
      C[i][j] += alpha * A[i][k] * B[k][j];
} } }
```

Loop Optimizations

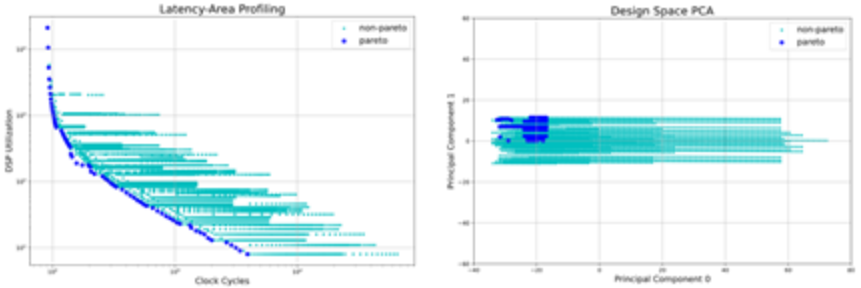
```
for (int k = 0; k < 32; k++) {
  for (int i = 0; i < 32; i++) {
    for (int j = 0; j < 32; j++) {
      if (k == 0)
        C[i][j] *= beta;
      C[i][j] += alpha * A[i][k] * B[k][j];
} } }
```

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

Any issues for this approach?

Limitations of ScaleHLS DSE (Cont.)

```
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

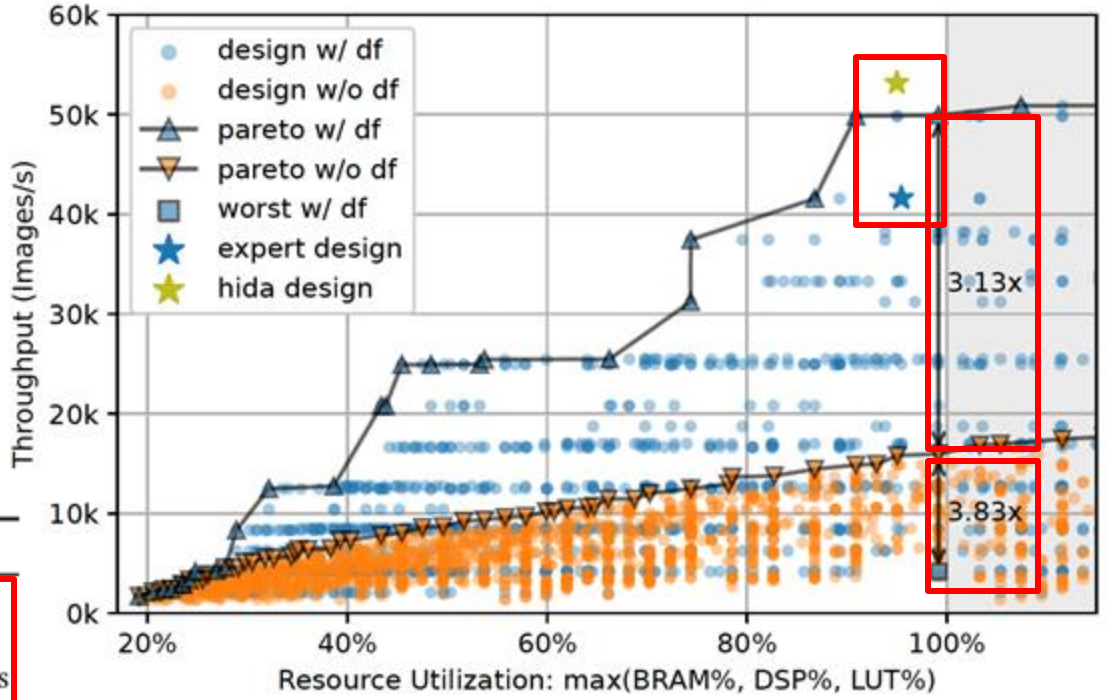
Simply merging the local Pareto curves will not work well!

Designing Dataflow Architecture Is Hard!

Manual LeNet Accelerator Design

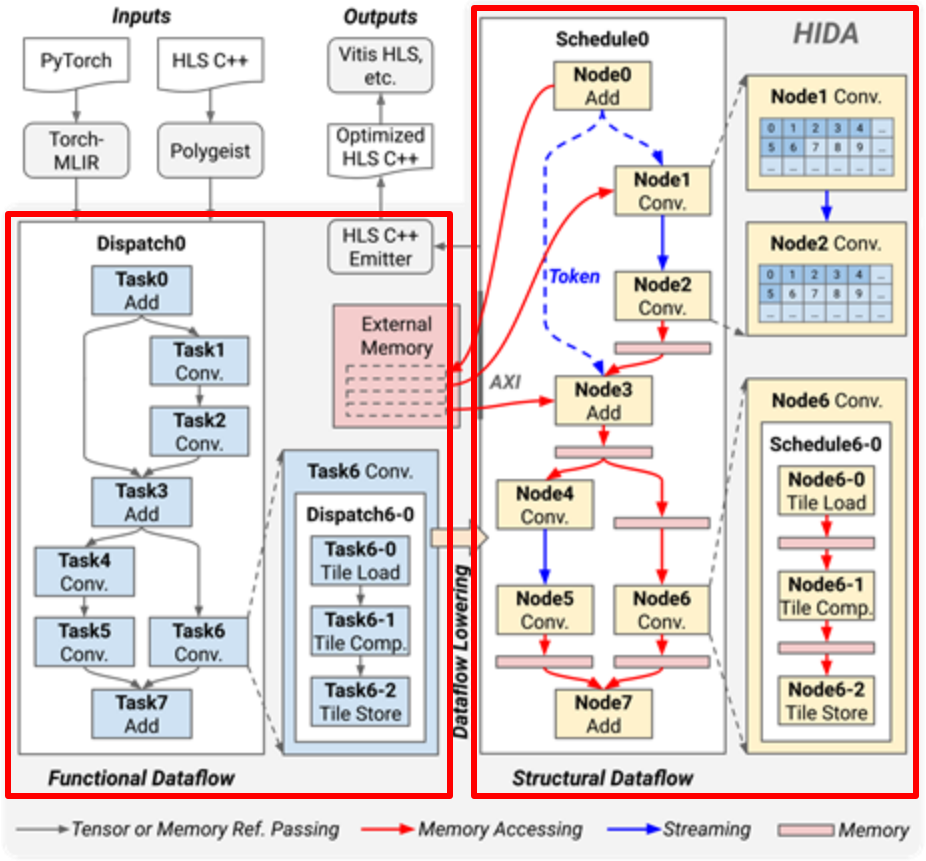
- Dataflow designs are Pareto-dominating
- Dataflow cannot guarantee a good trade-off
- Dataflow design space is difficult to comprehend
- Automated tool outperforms expert design

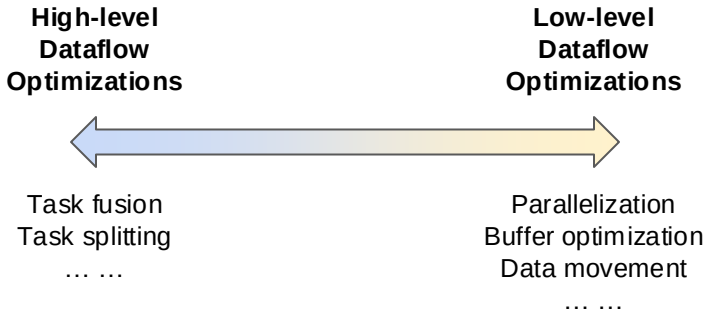
	Expert	Exhaustive	HIDA
Resource Util.	95.5%	99.2%	95.0%
Throu. (Imgs/s)	41.6k	49.9k	53.2k
Develop Cycle	40 hours	210 hours	9.9 mins



Productivity Performance Scalability

Dataflow IR

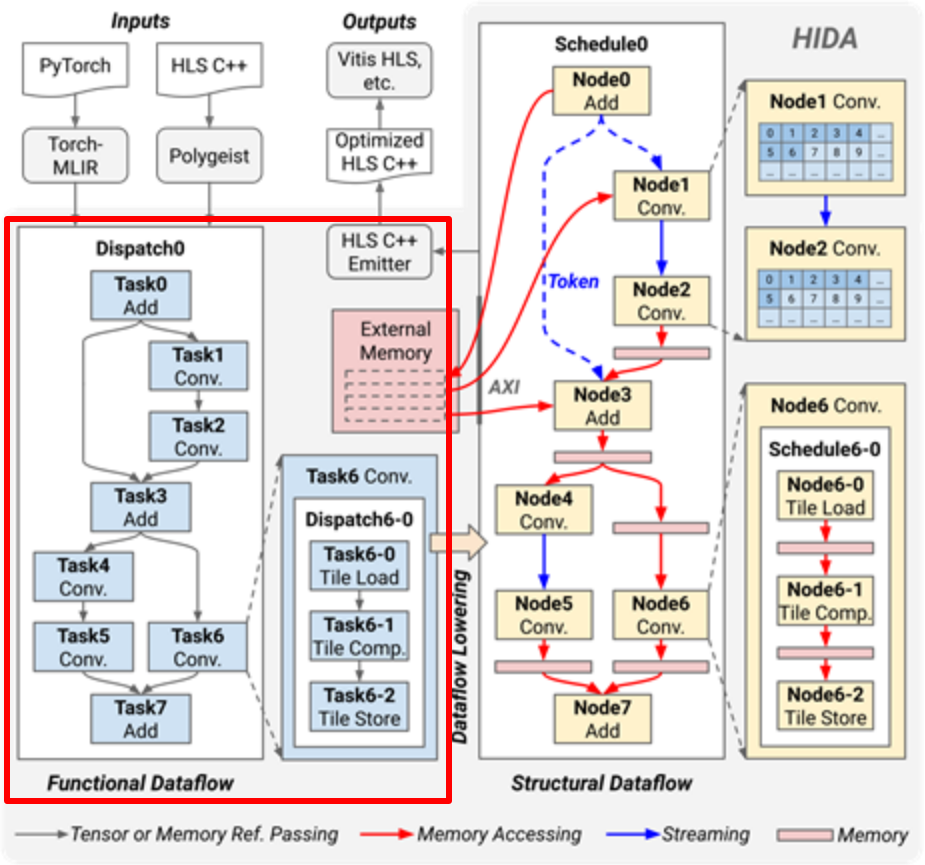




Two-level dataflow representation

- Functional dataflow
  - Capture high-level dataflow characteristics
  - Efficient dataflow manipulation
- Structural dataflow
  - Capture low-level micro-architectures
  - Efficient scheduling and parallelization

Dataflow IR: Functional Dataflow

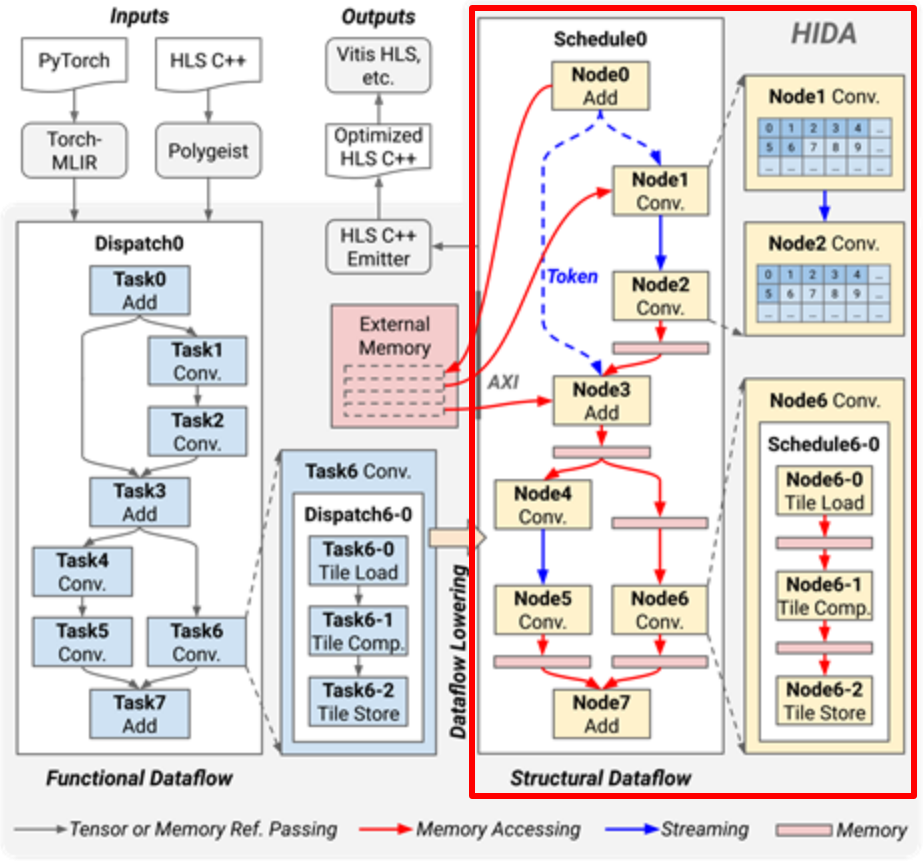



```
%tensor = hida.task() : tensor<64x64xi8> { ... }
hida.task() { ... %tensor ... }
```

Functional Dataflow

- Hierarchical structure
  - Support multiple levels of dataflow
  - Inside of Task6, the tile load, computation, and store are further dataflowed
- Transparent from above
  - All tasks share the same global context
  - Support efficient task fusion and splitting

Dataflow IR: Structural Dataflow

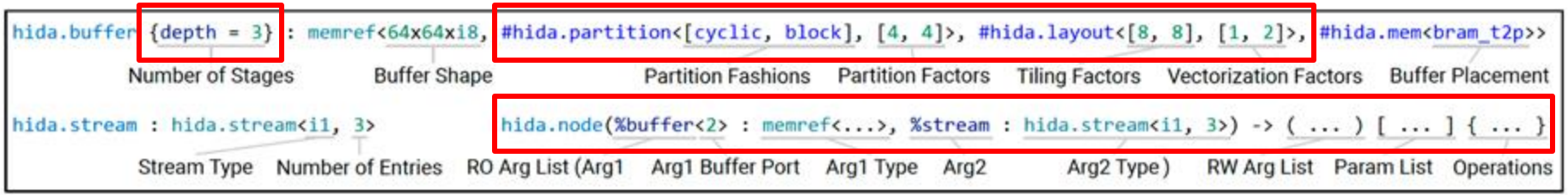



```
%buffer = hida.buffer : memref<64x64xi8, ...>
hida.node() -> (%buffer : memref<64x64xi8, ...>) { ... }
hida.node(%buffer : memref<64x64xi8, ...>) -> () { ... }
```

Structural Dataflow

- Explicit buffer representation
  - Support both memory-mapped and stream buffers
- Isolated from above
  - Each node has its own context
  - Decouple inter-node and intra-node dataflow optimization

Dataflow IR: Structural Dataflow (Cont.)



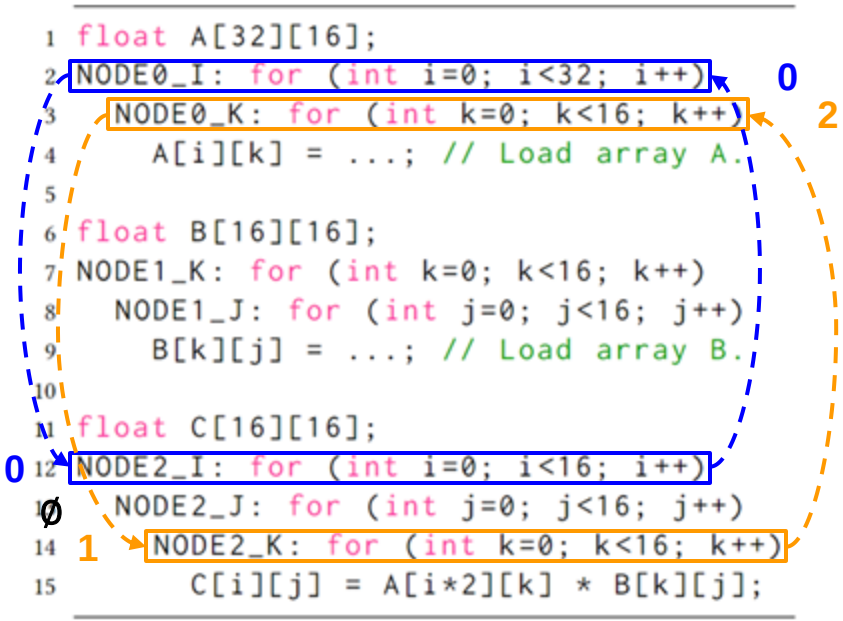
* `buffer`, `stream`, and node operation syntax in structural dataflow. *RO* and *RW* denote read-only and read-write.

- Multi-stage buffer representation
  - Support complicated schedulings, e.g., multi-line buffer
- Affine-based partition, tiling, and vectorization representation
  - Support automatic buffer optimization upon affine analyses
- Explicit buffer memory effect representation
  - Avoid unnecessary inter-node analysis



Multi-kernel Dataflow-aware DSE

Multi-kernel Dataflow-aware DSE

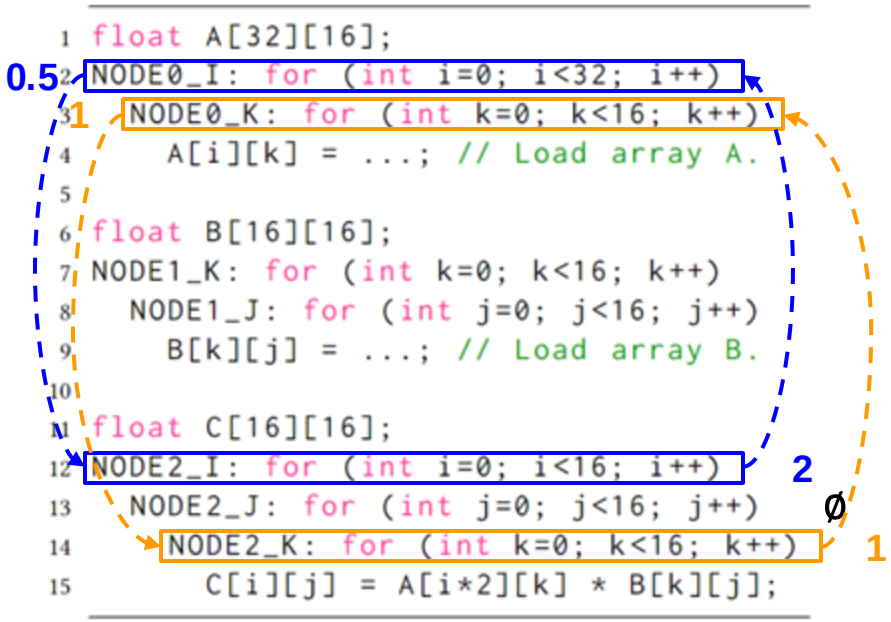


Step (1) Connectedness Analysis

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

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

Multi-kernel Dataflow-aware DSE (Cont.)



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, Ø, 1]	[0, 2]	[0.5, 1]	[2, Ø, 1]
Node1	Node2	B	[Ø, 1, 0]	[2, 1]	[1, 1]	[Ø, 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

Multi-kernel Dataflow-aware DSE (Cont.)

```
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**

Multi-kernel Dataflow-aware DSE (Cont.)

```
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
  - Local DSE without constraints
  - Solution unroll factors: [4, 8, 1]

Multi-kernel Dataflow-aware DSE (Cont.)

```
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
  - Local 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]

Multi-kernel Dataflow-aware DSE (Cont.)

```
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 Node1 and 2
  - Local DSE with connectedness constraints
  - Solution unroll factors: [1, 2]

Multi-kernel Dataflow-aware DSE (Cont.)

```
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

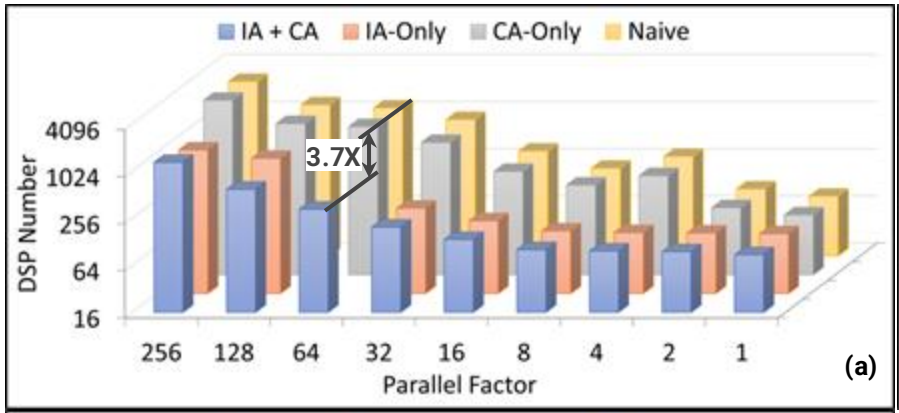
Node	Intensity	Parallel Factor		Loop Unroll Factors			
		w/o IA	w/ IA	IA+CA	IA	CA	Naive
Node0	512	32	4	[4, 1]	[2, 2]	[8, 4]	[4, 8]
Node1	256	32	2	[1, 2]	[1, 2]	[4, 8]	[4, 8]
Node2	4,096	32	32	[4, 8, 1]	[4, 8, 1]	[4, 8, 1]	[4, 8, 1]

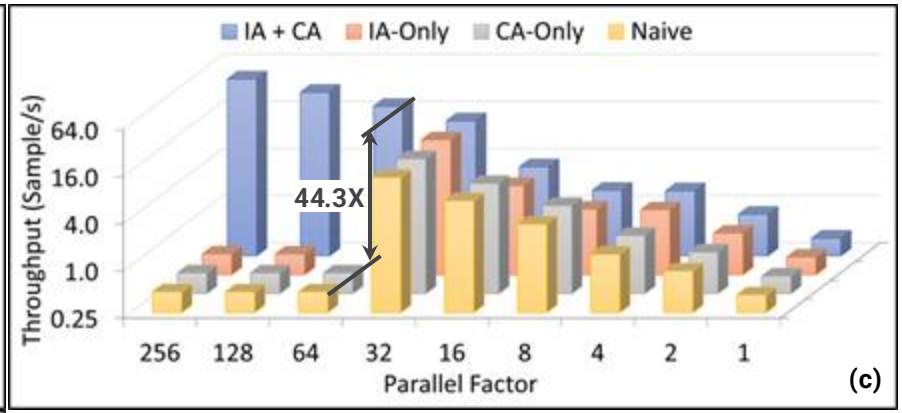
Intensity-aware (IA)
Connectedness-aware (CA)
HIDA DSE

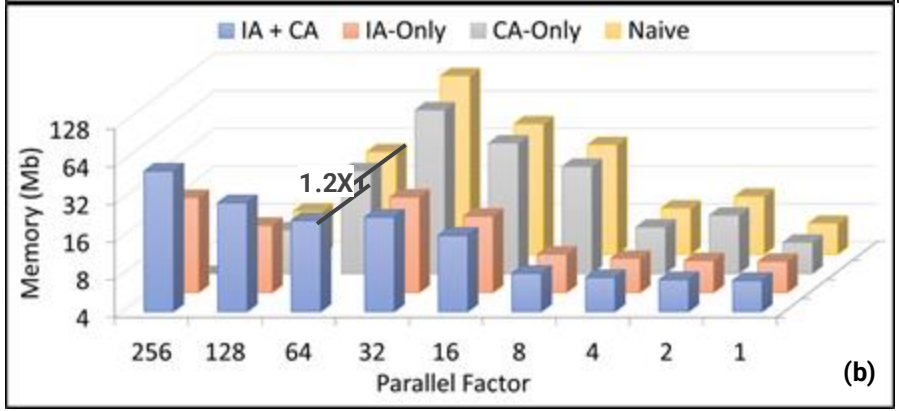
Naive DSE

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

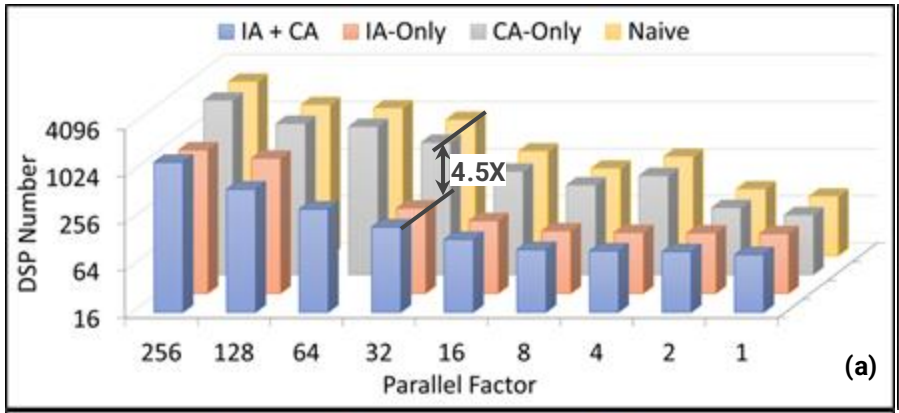


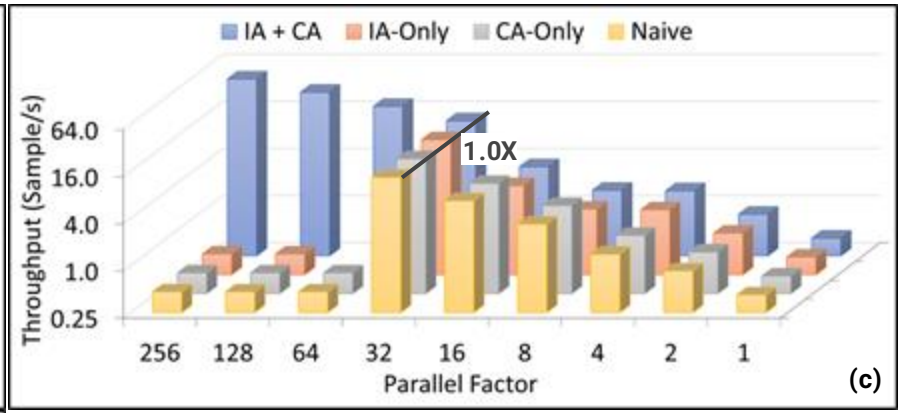


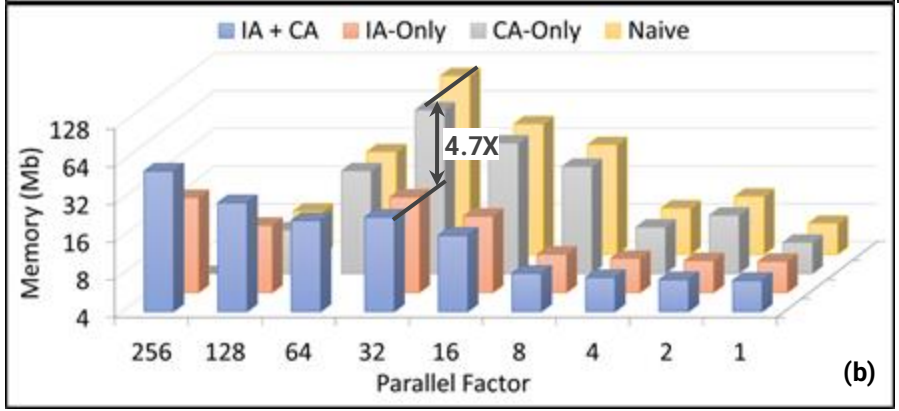


- IA+CA parallelization can determine whether the solution is scalable

ResNet-18 Ablation Study on HIDA (Cont.)







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

Dataflow-aware Results on DNN Models

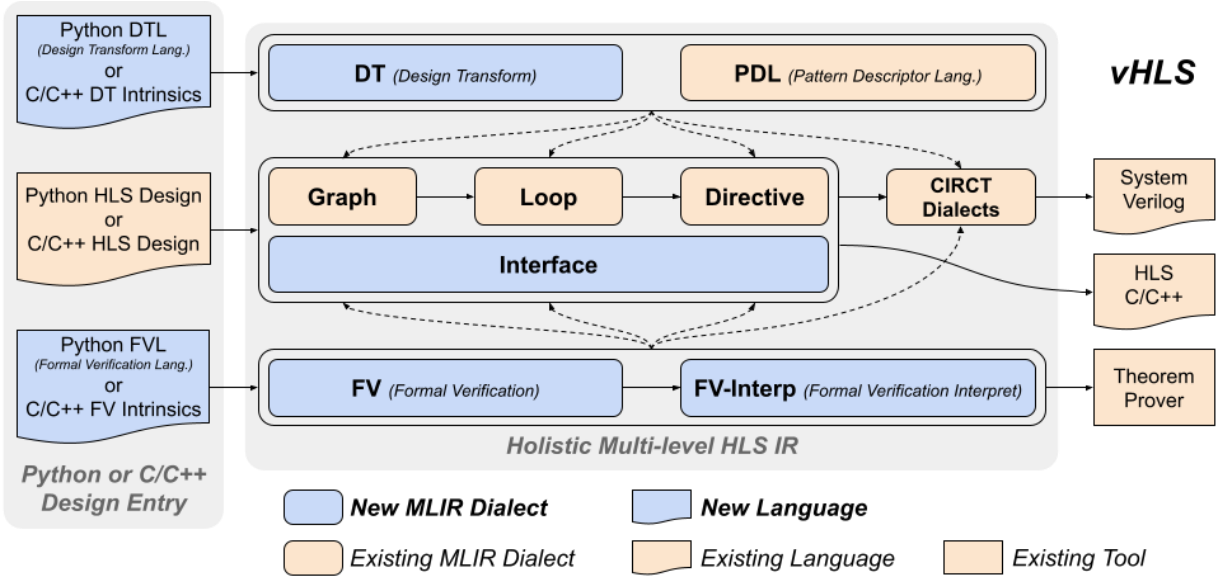
Model	HIDA Compile Time (s)	LUT Number	DSP Number	Throughput (Samples/s)*			DSP Efficiency*		
				HIDA	DNNBuilder [75]	ScaleHLS [68]	HIDA	DNNBuilder [75]	ScaleHLS [68]
ResNet-18	83.1	142.1k	667	45.4	-	3.3 (13.88×)	73.8%	-	5.2% (14.24×)
MobileNet	110.8	132.9k	518	137.4	-	15.4 (8.90×)	75.5%	-	9.6% (7.88×)
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.



Future Opportunities

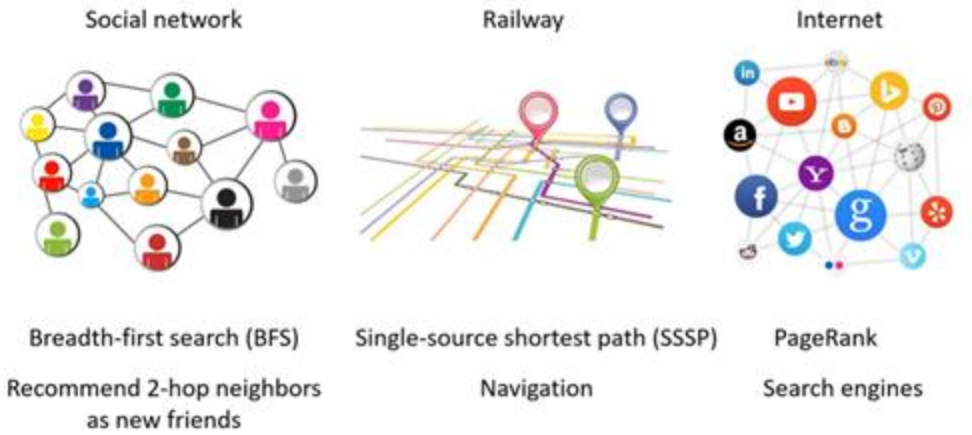
Opportunities (1): HLS Verification



Prove the correctness!

- Correct-by-construction hardware design paradigm demands verifiable HLS flow bridging the gap between architecture model and RTL design.
- Key components:
  - Intermediate representation (IR) with formal semantics
  - Equivalence checking or verifiable HLS transformations
  - Formal interface verification

Opportunities (2): Sparse Computation with HLS



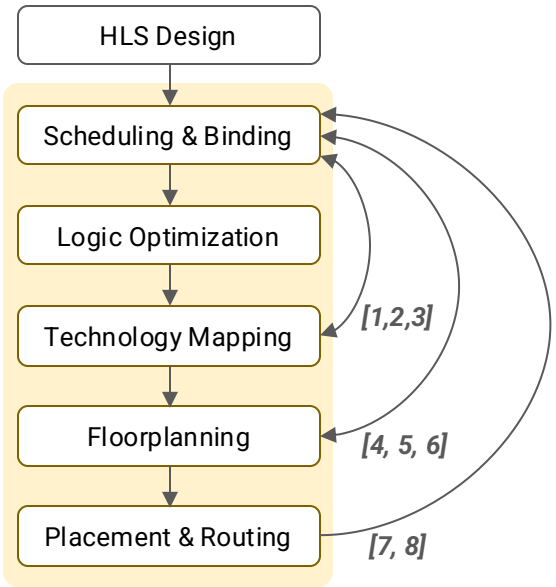
Sparse computation is ubiquitous

- Sparsity is an important optimization for many algorithms, e.g., graph processing, transformer. However, the acceleration and automation of sparse computation have long been challenging.
- HLS is a promising technique for sparsity due to its flexibility and efficiency:
  - **GraphLily** [1] and **ThunderGP** [2]: HLS framework for graph processing
  - **Y. Du et al.** [3]: HLS-based SPMV on HBM
  - **S. Huang et al.** [4]**:** Sparse DNN on FPGA
- Advanced sparse compilers:
  - **SparseTIR** [5], **TACO** [6], **MLIR-SparseTensor** [7] and more…

Skip the zeros!

[1] Hu, Yuwei, et al. "GraphLily: Accelerating graph linear algebra on HBM-equipped FPGAs." ICCAD. 2021.
[2] Chen, Xinyu, et al. "ThunderGP: HLS-based graph processing framework on FPGAs." FPGA. 2021.
[3] Du, Yixiao, et al. "High-performance sparse linear algebra on HBM-equipped FPGAs using HLS: A case study on SPMV." FPGA. 2022.
[4] Huang, Sitao, et al. "Accelerating sparse deep neural networks on FPGAs." HPEC. 2019.
[5] Ye, Zihao, et al. "SparseTIR: Composable abstractions for sparse compilation in deep learning." ASPLOS. 2023.
[6] Kjolstad, Fredrik, et al. "Taco: A tool to generate tensor algebra kernels." ASE. 2017.
[7] https://mlir.llvm.org/docs/Dialects/SparseTensorOps/

Opportunities (3): Backend-aware HLS

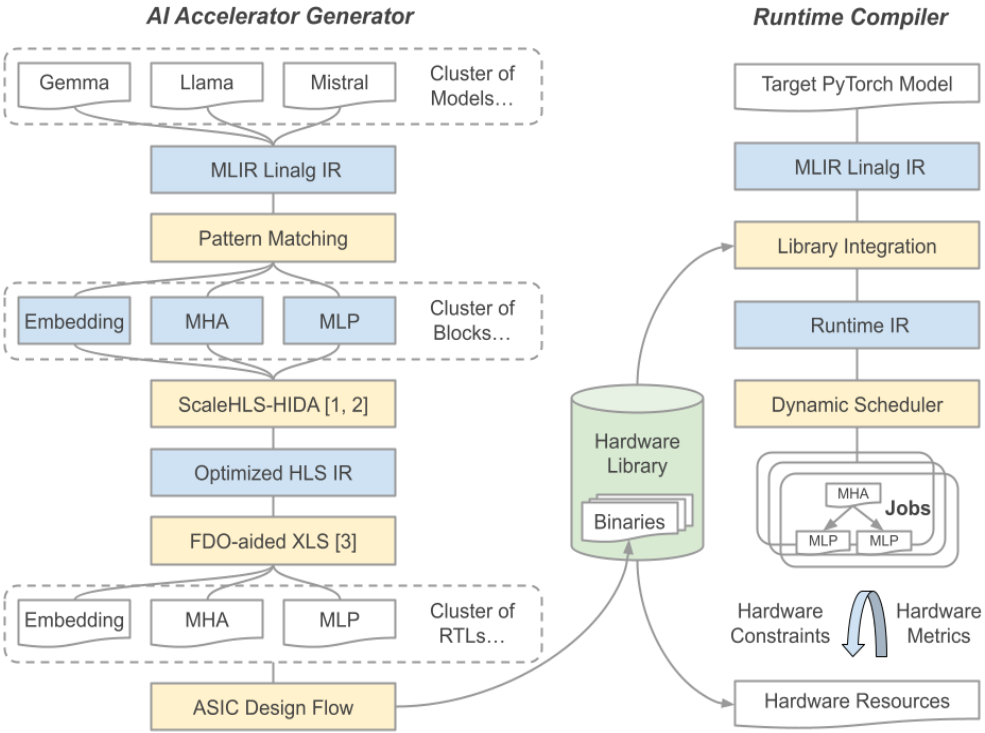


- Backend results feedback or estimation (e.g., logic synthesis, placement, and routing) can substantially improve the quality of HLS scheduling & binding (e.g., frequency and area).
- Technology mapping-aware:
  - **M. Tan at al.** [1]**:** Formulate LUT-mapping into HLS scheduling
  - **C. Rizzi et al.** [2]**:** Feedback mapping results to improve the buffer insertion of dynamic-scheduled circuits
  - **MapBuf** [3]**:** Simultaneous buffer insertion and mapping
- Floorplanning-aware:
  - **AutoBridge** [4]**:** Floorplanning-aware coarse-grained pipelining
  - **FADO** [5]**:** Floorplanning-aware HLS directive co-optimization
  - **G. Lucas** [6]**:** Layout-driven HLS scheduling
- Placement & routing-aware:
  - **LOPASS** [7]**:** Interconnect-aware FPGA synthesis
  - **H. Zhang et al.** [8]**:** Feedback post-placement and routing results to improve the node and wire delay estimation in HLS scheduling

[1] Tan, Mingxing, et al. "Mapping-aware constrained scheduling for LUT-based FPGAs." FPGA. 2015.
[2] Rizzi, Carmine, Andrea Guerrieri, and Lana Josipovic. "An Iterative Method for Mapping-Aware Frequency Regulation in Dataflow Circuits." DAC. 2023.
[3] Hanyu Wang, Carmine Rizzi, and Lana Josipović. "MapBuf: Simultaneous Technology Mapping and Buffer Insertion for HLS Performance Optimization." IWLS. 2023.
[4] Guo, Licheng, et al. "AutoBridge: Coupling coarse-grained floorplanning and pipelining for high-frequency HLS design on multi-die FPGAs." FPGA. 2021.
[5] Du, Linfeng, et al. "FADO: Floorplan-Aware Directive Optimization for High-Level Synthesis Designs on Multi-Die FPGAs." FPGA. 2023.
[6] Lucas, Gregory, and Deming Chen. "Variation-aware layout-driven scheduling for performance yield optimization." ICCAD. 2010.
[7] Chen, Deming, et al. "LOPASS: A low-power architectural synthesis system for FPGAs with interconnect estimation and optimization." TVLSI. 2009.
[8] Zheng, Hongbin, et al. "Fast and effective placement and routing directed high-level synthesis for FPGAs." FPGA. 2014.

Extract the extra bit of performance!

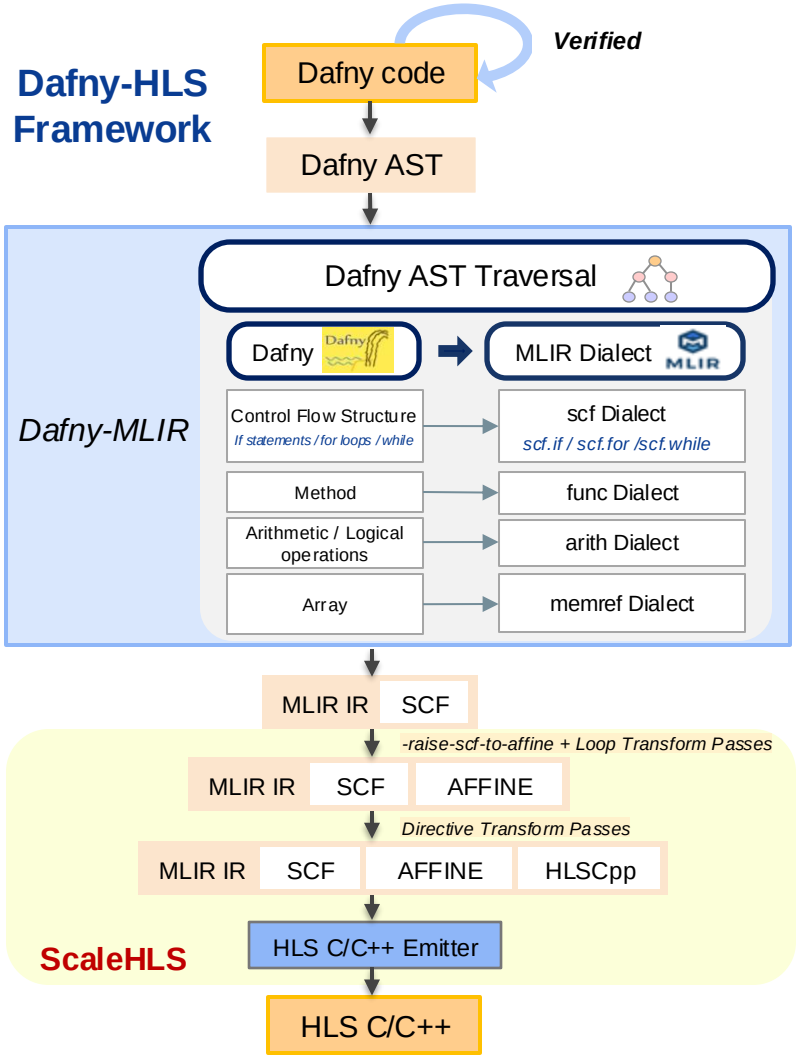
Opportunities (4): AI Accelerator for a Class of Models



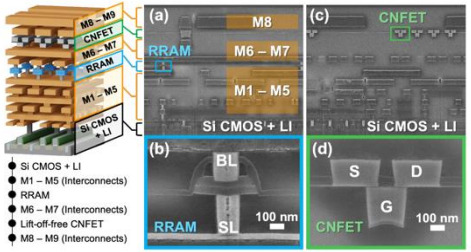
- A new proposed flow that can eventually generate an RTL AI accelerator for a class of ML models automatically
  - Can be ASIC, FPGA, chiplet
- The AI accelerator consists of hardware building blocks for the specific class
  - affiliated with a hardware library (designed or generated)
- A corresponding runtime compiler that maps/schedules a model to the accelerator

Opportunities (5): Dafny-HLS: HLS for Verifiable AI Chiplet

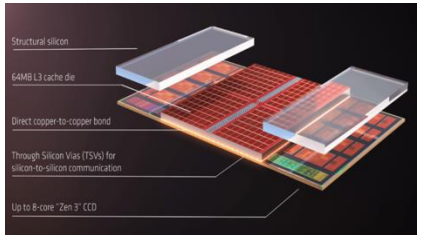
- First end-to-end approach to generate synthesizable C code from Dafny
  - Dafny code is converted to MLIR IR through AST traversal.
  - Integrated with ScaleHLS framework to optimize MLIR IR and generate synthesizable C code.
- Dafny-HLS minimizes the need for developers to manually detect errors
  - Utilizes Dafny's **formal verification engine** to ensure correctness of synthesizable C codes.
- Target AI domains and beyond
  - Implementing neural networks with Dafny-HLS can lead to high performance and reliability of AI models and AI chiplets.



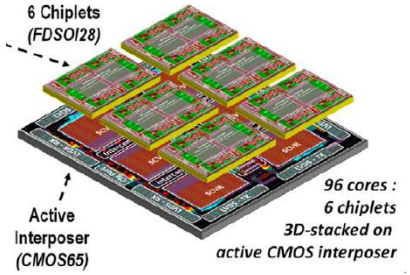
Opportunities (6): Packaging + Chiplet Co-design for 2.5D/3D-ICs



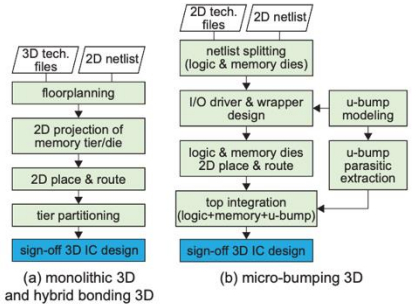
Monolithic 3D. [VLSI, 2023]. Stanford



3D V-Cache technology stacks the cache on a processor. AMD



96-Core, 6 Chiplets, 3D-Stacked. [ISSCC, 2020]. CEA-LETI



Design flows for 3D ICs. [DAC, 2021]. Georgia Tech.

- 2.5D and 3D-ICs demand ever more system-level and high-level synthesis to improve the programmability and manage the intra- and inter-chip latency, bandwidth, and energy.
- Packaging + Chiplet co-design
  - Add a chiplet abstract layer into MLIR
  - Data flow across chiplets
  - Modeling various packaging options as an essential co-design dimension
  - Thermal modeling
  - Communication modeling
  - …



The Journey Has Just Begun!

Acknowledgement

Collaborators:

Hanchen Ye, Hyegang (Greg) Jun, Cong (Callie) Hao, Jianyi Cheng, Hyunmin (Paul) Jeong, Jack Huang, Stephen Neuendorffer

Sponsors:

Booz | Allen | Hamilton®

AMD XILINX

C³SR center for cognitive computing systems research

The Grainger College of Engineering
IBM-Illinois Discovery Accelerator Institute

NSF

SRC® Semiconductor Research Corporation