HIDA: Hierarchical Dataflow Compiler for High-Level Synthesis

Hanchen Ye, Hyegang Jun, Deming Chen

Nov. 8, 2023

Outline

- ScaleHLS Recall
- Motivation
- HIDA Intermediate Representation
- HIDA Optimizations
- Evaluation Results
- Conclusion

Outline

- ScaleHLS Recall
- Motivation
- HIDA Intermediate Representation
- HIDA Optimizations
- Evaluation Results
- Conclusion

Recall: ScaleHLS 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.

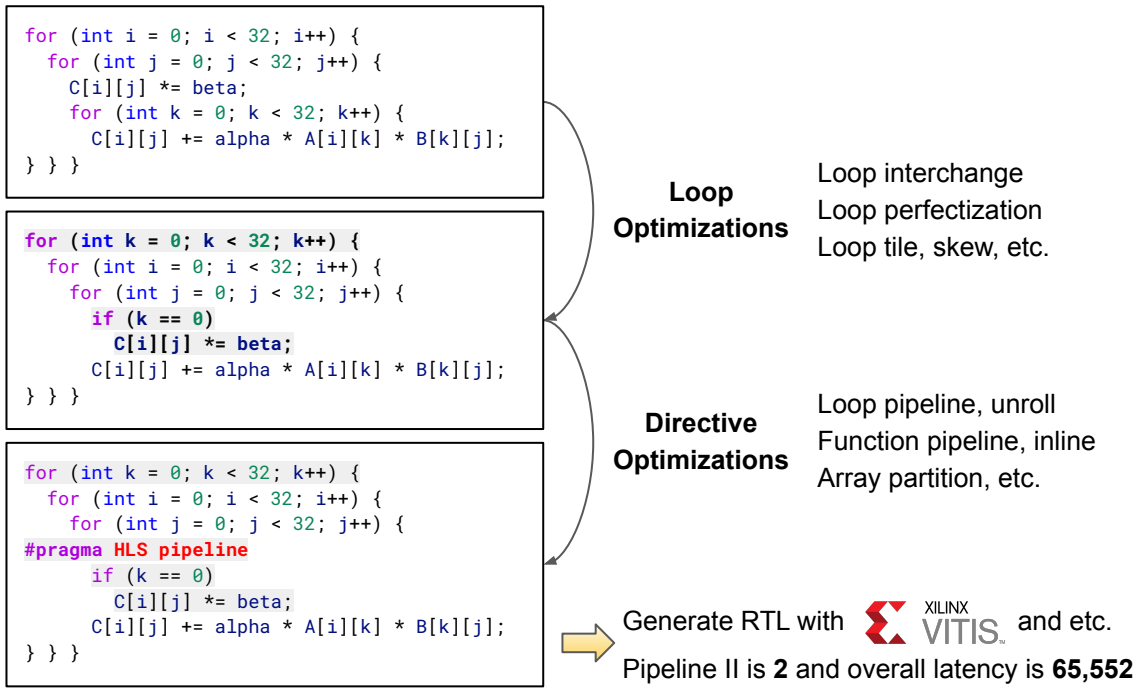
```
for (int i = 0; i < 32; i++) {
  for (int j = 0; j < 32; j++) {
    C[i][j] *= beta;
    for (int k = 0; k < 32; k++) {
#pragma HLS pipeline
      C[i][j] += alpha * A[i][k] * B[k][j];
} } }
```

Generate RTL with XILINX VITIS and etc.

Pipeline II is **5** and overall latency is **183,296**

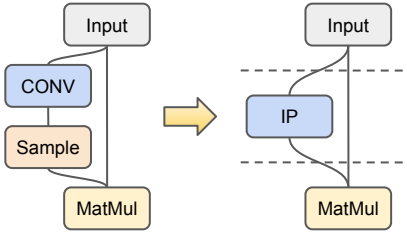
Recall: ScaleHLS Motivation (Cont.)

How do we do HLS designs?



Recall: ScaleHLS Motivation (Cont.)

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**

Recall: ScaleHLS Motivation (Cont.)

Difficulties:

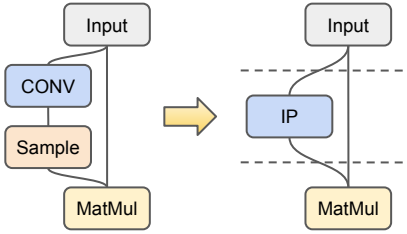
- Low-productive and error-proning
- Hard to enable automated design space exploration (DSE)
- NOT scalable!

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

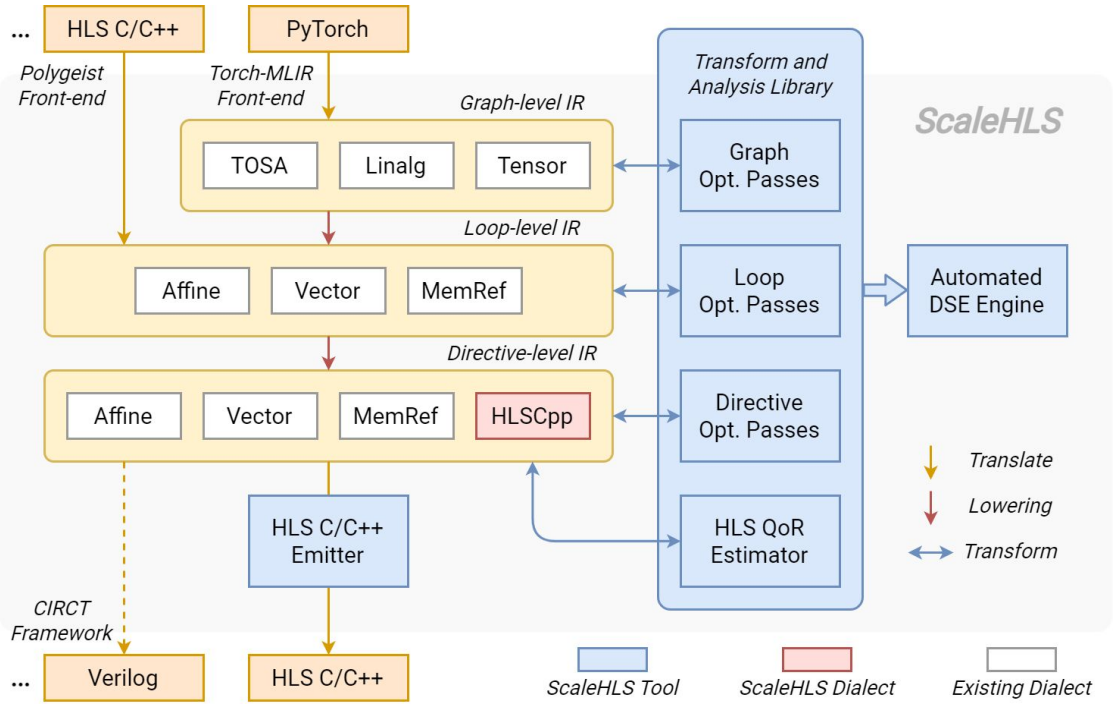
Directive Optimizations
Loop pipeline, unroll
Function pipeline, inline
Array partition, etc.

Manual Code Rewriting

Generate RTL with XILINX VITIS and etc.

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

Recall: ScaleHLS Framework



[1] Polygeist: C/C++ frontend for MLIR. https://github.com/wsmoses/Polygeist
[2] Torch-MLIR: PyTorch frontend for MLIR: https://github.com/llvm/torch-mlir
[3] CIRCT: Circuit IR Compilers and Tools https://github.com/llvm/circt

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!

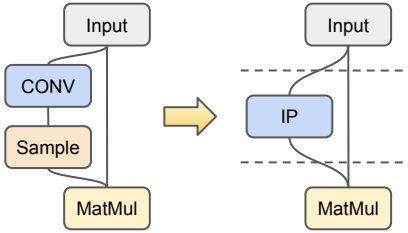
HLS C Front-end: Parse C programs into MLIR.

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

Outline

- ScaleHLS Recall
- Motivation
- HIDA Intermediate Representation
- HIDA Optimizations
- Evaluation Results
- Conclusion

Motivation: Limitations of ScaleHLS DSE



Graph Optimizations

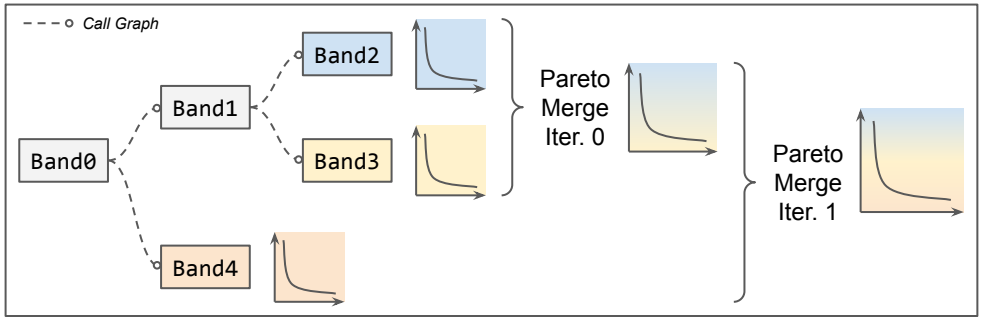
```
for (int i = 0; i < 32; i++) {
  for (int j = 0; j < 32; j++) {
    C[i][j] *= beta;
    for (int k = 0; k < 32; k++) {
      C[i][j] += alpha * A[i][k] * B[k][j];
} } }
```

Loop Optimizations

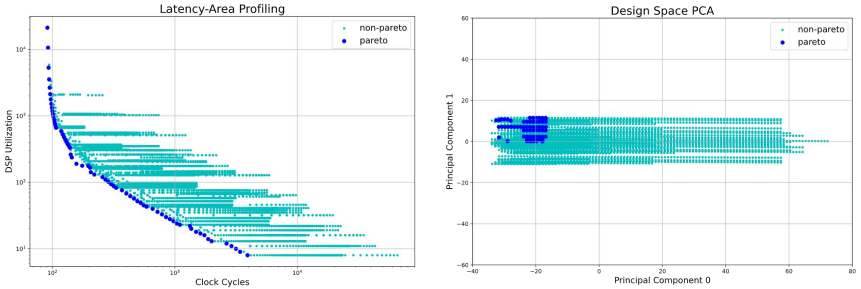
```
for (int k = 0; k < 32; k++) {
  for (int i = 0; i < 32; i++) {
    for (int j = 0; j < 32; j++) {
      if (k == 0)
        C[i][j] *= beta;
      C[i][j] += alpha * A[i][k] * B[k][j];
} } }
```

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

Motivation: 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!

Motivation: Designing dataflow architecture is hard!

Manual LeNet Accelerator Design

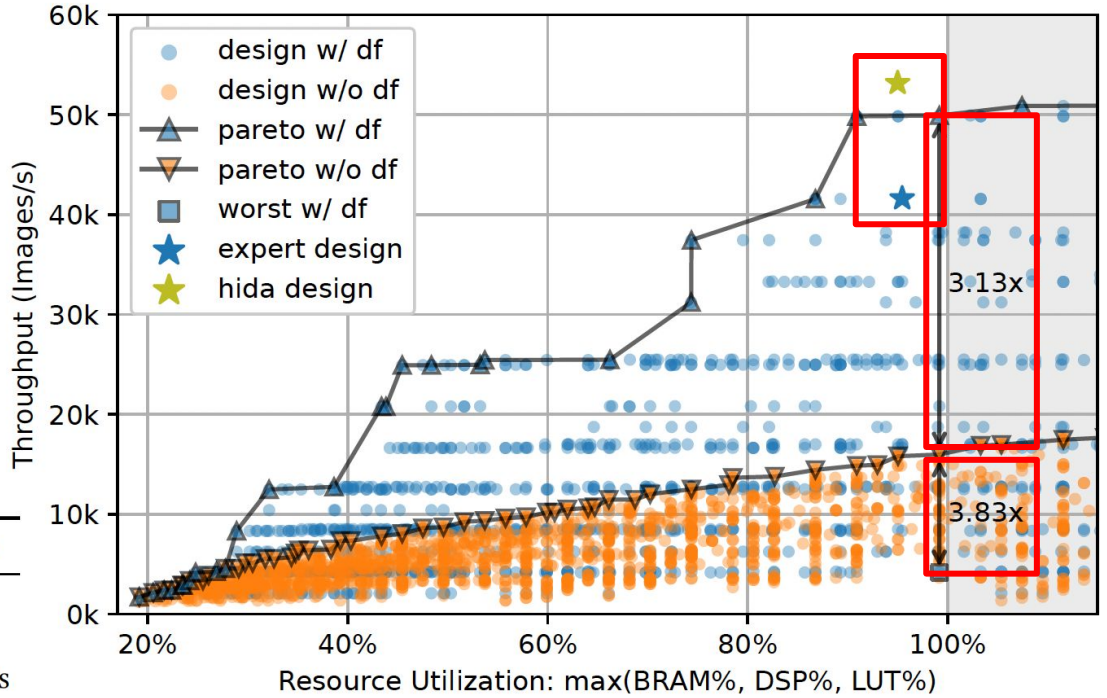
- Layer fusion
  - Convolutional layer
  - ReLU layer
  - Max pooling layer
- Parallelization
  - Batch size
  - KPF (Kernel parallel factor)
  - CPF (Channel parallel factor)
- Layer fusion and parallelization decisions are made empirically
  - The resulting design space still has 24,000 design points

Layer	Task	Factor	Range
(All Layers)	-	$BATCH$	{1, 5, 10, 15, 20}
Conv+ReLU / Pool	Task1	KPF_{task1}	{1, 2, 3, 6}
Conv+ReLU / Pool	Task2	KPF_{task2} CPF_{task2}	{1, 2, 4, 8, 16} {1, 2, 3, 6}
Conv+ReLU	Task3	KPF_{task3} CPF_{task3}	{1, 2, 3, 4, 6, 8} {1, 2, 4, 8, 16}
Linear	Task4	-	-

Motivation: Designing dataflow architecture is hard! (Cont.)

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

	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

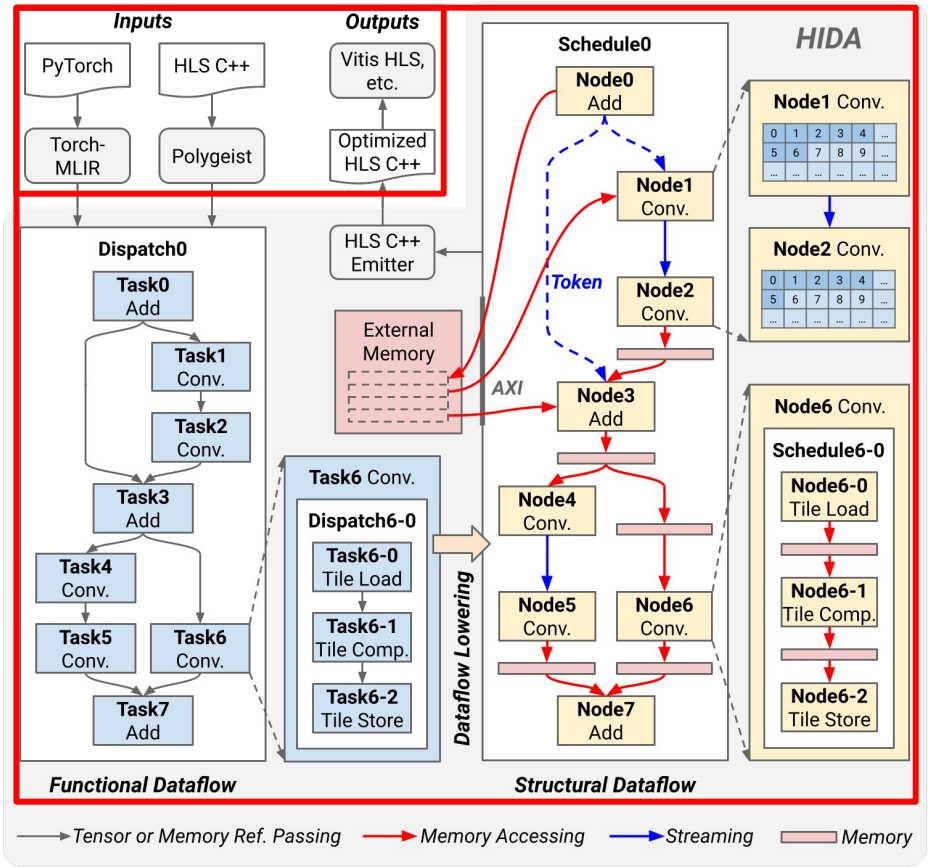


Productivity Performance Scalability

Outline

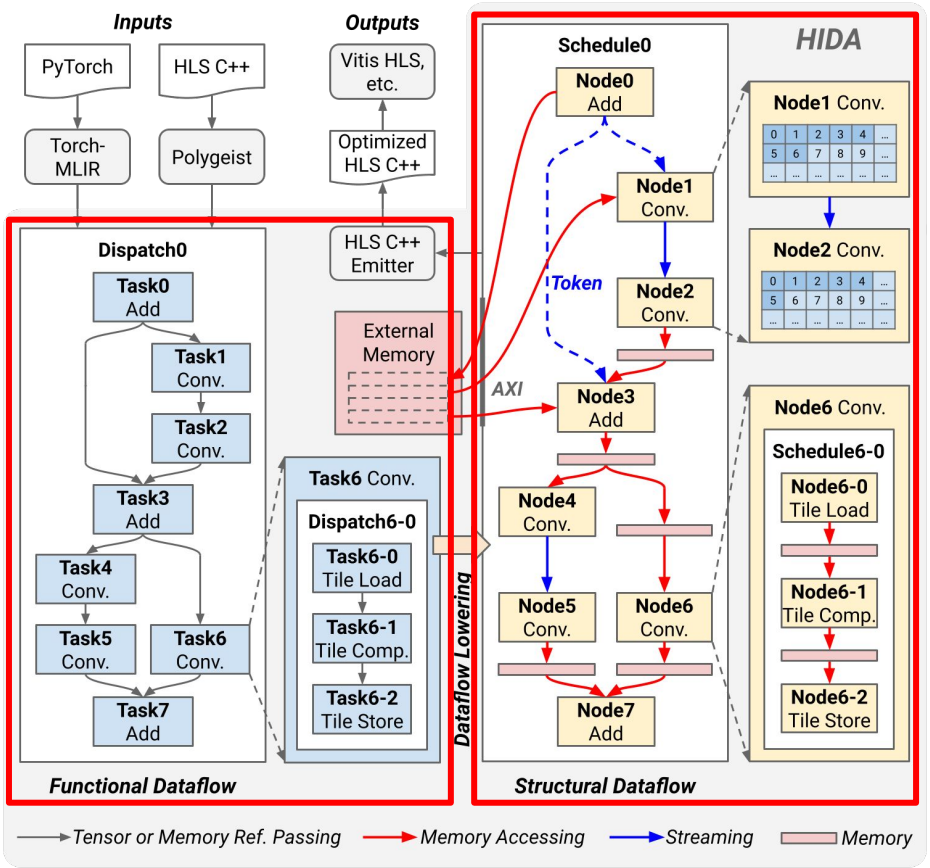
- ScaleHLS Recall
- Motivation
- HIDA Intermediate Representation
- HIDA Optimizations
- Evaluation Results
- Conclusion

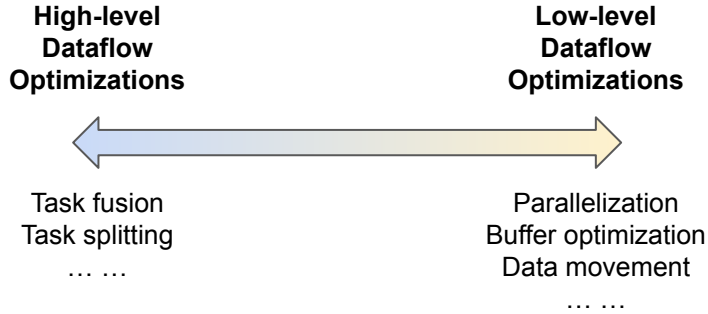
HIDA Framework



- **PyTorch** or **C/C++** as input
- Optimized **C++** dataflow design as output
- **MLIR-based** dataflow intermediate representation (IR), optimization, and code-generation

HIDA Intermediate Representation

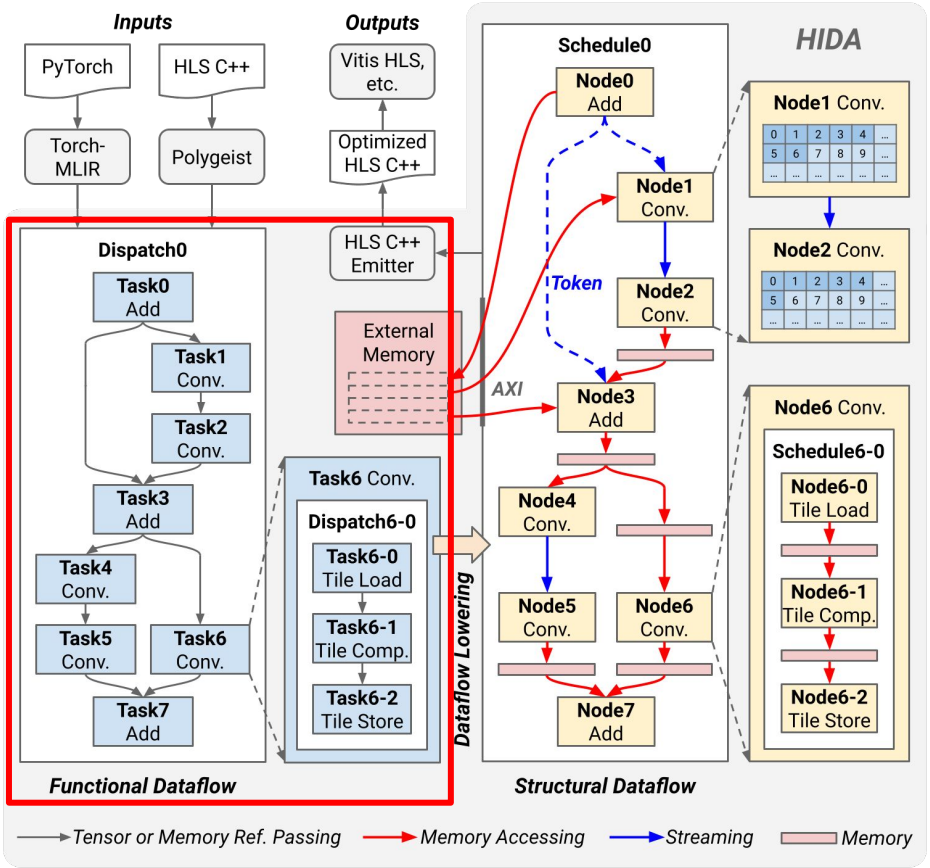




Two-level dataflow representation

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

HIDA Functional Dataflow

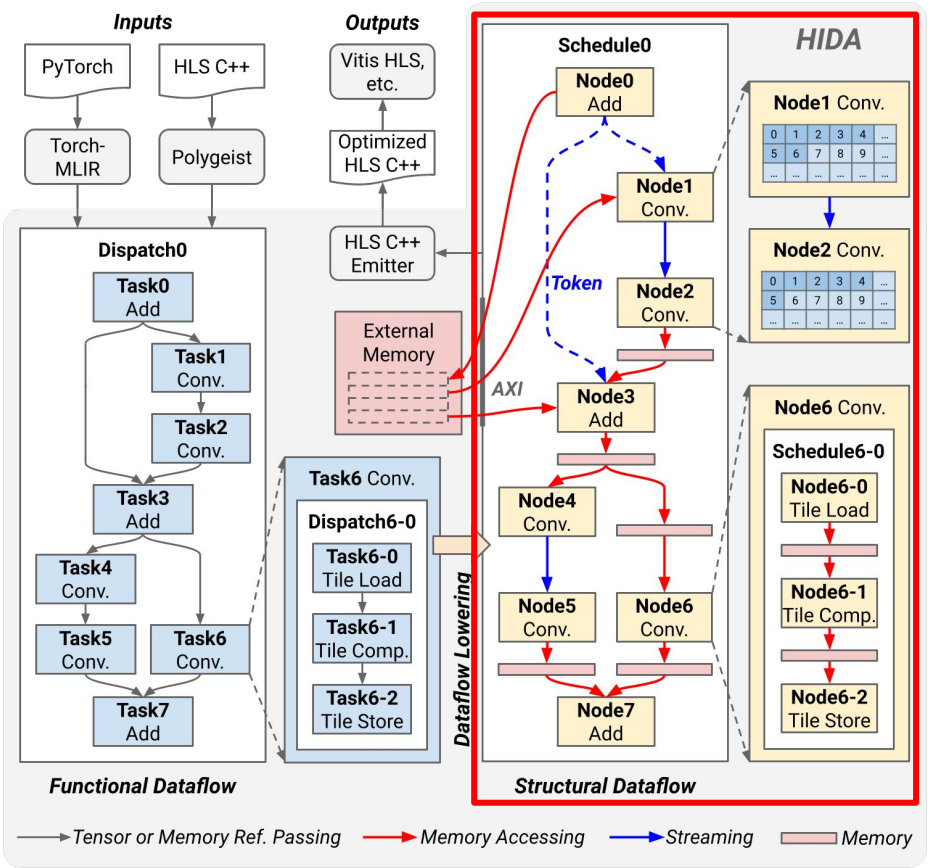



```
%tensor = hida.task() : tensor<64x64xi8> { ... }
hida.task() { ... %tensor ... }
```

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

HIDA Structural Dataflow

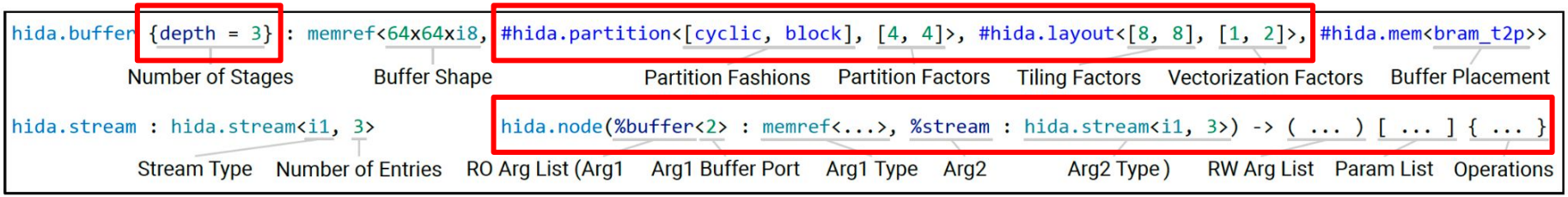



```
%buffer = hida.buffer : memref<64x64xi8, ...>
hida.node() -> (%buffer : memref<64x64xi8, ...>) { ... }
hida.node(%buffer : memref<64x64xi8, ...>) -> () { ... }
```

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

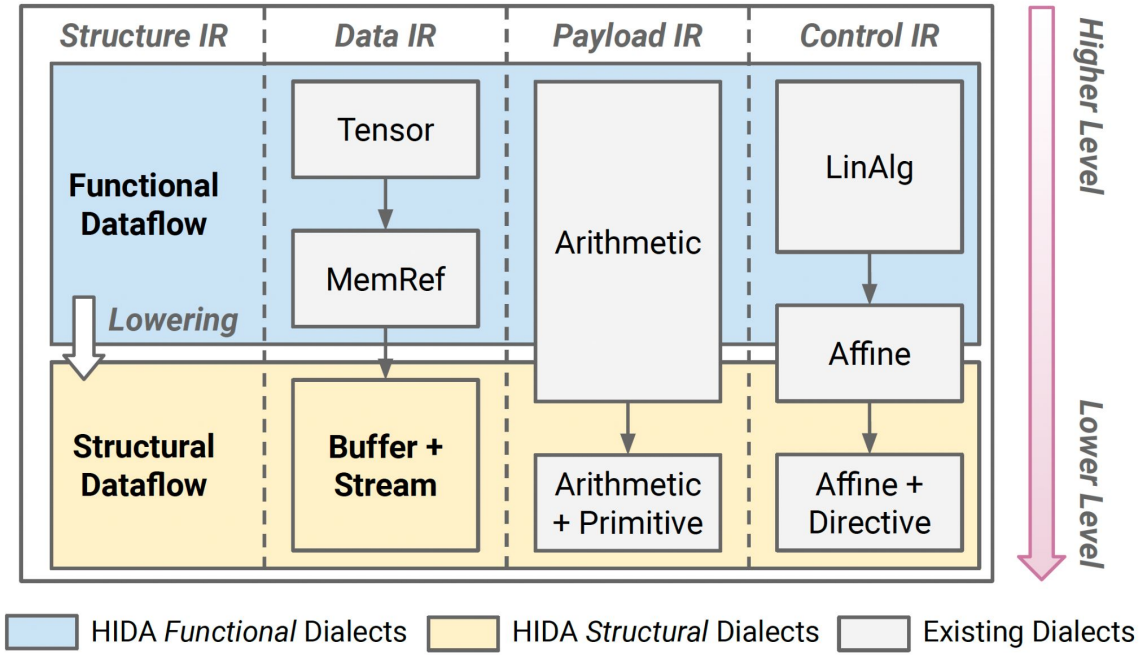
HIDA Structural Dataflow (Cont.)



* `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

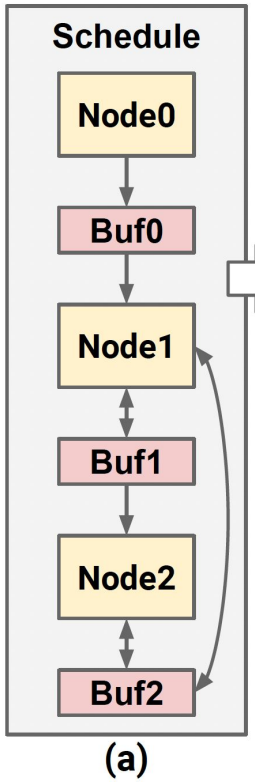
Integration with MLIR Dialects



Outline

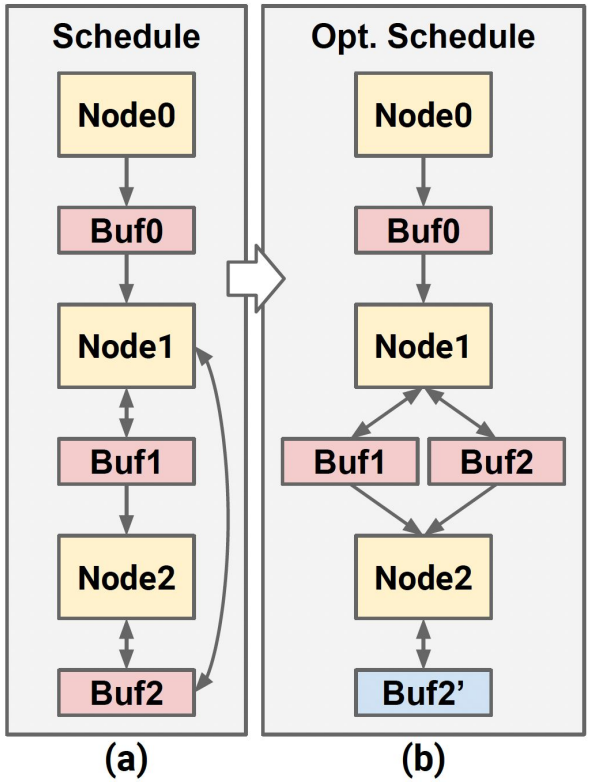
- ScaleHLS Recall
- Motivation
- HIDA Intermediate Representation
- HIDA Optimizations
- Evaluation Results
- Conclusion

Multiple Producer Elimination



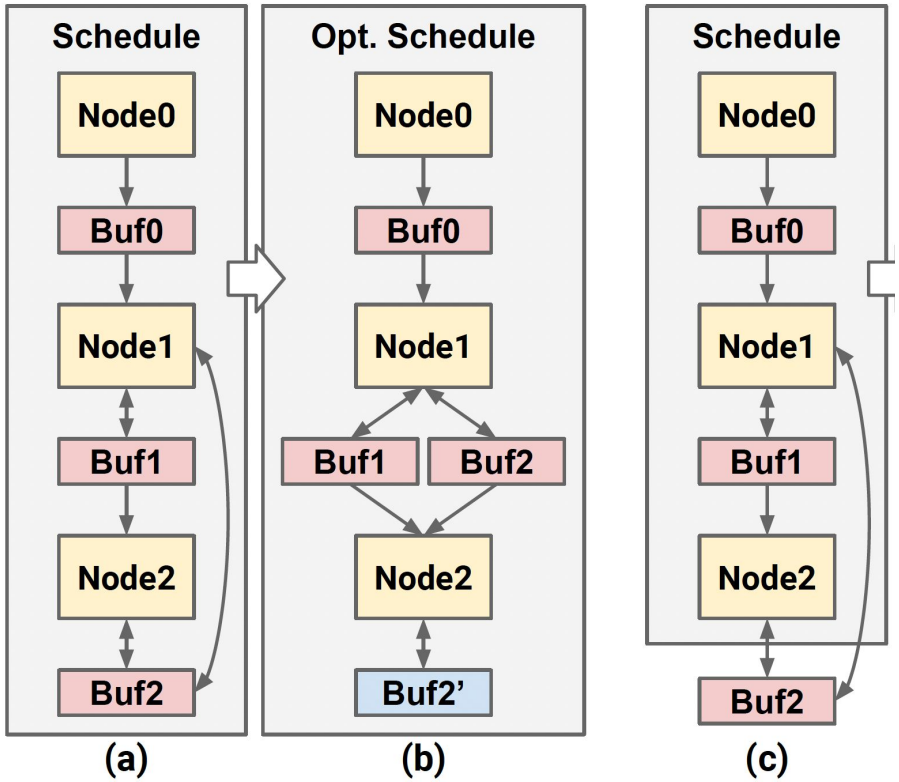
Buffer inside of the context

Multiple Producer Elimination



Buffer inside of the context

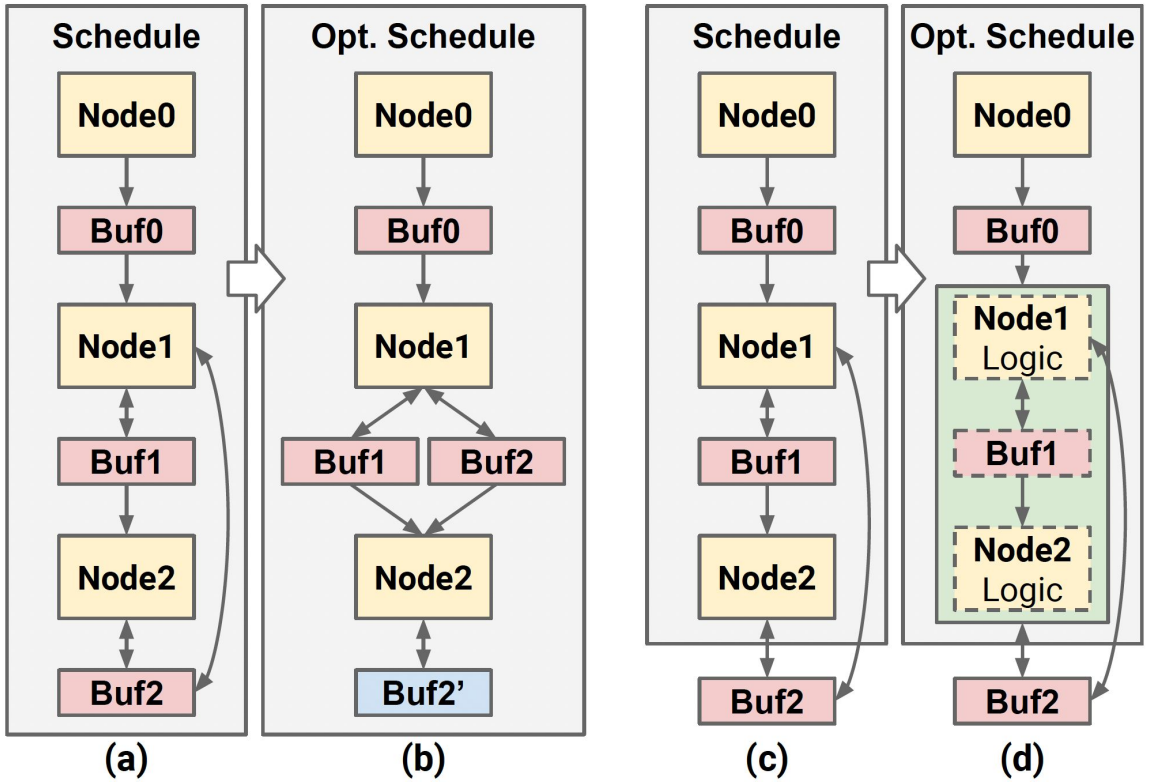
Multiple Producer Elimination



Buffer inside of the context

Buffer outside of the context

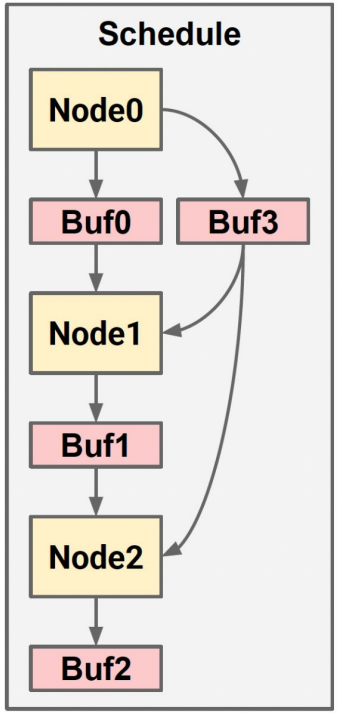
Multiple Producer Elimination



Buffer inside of the context

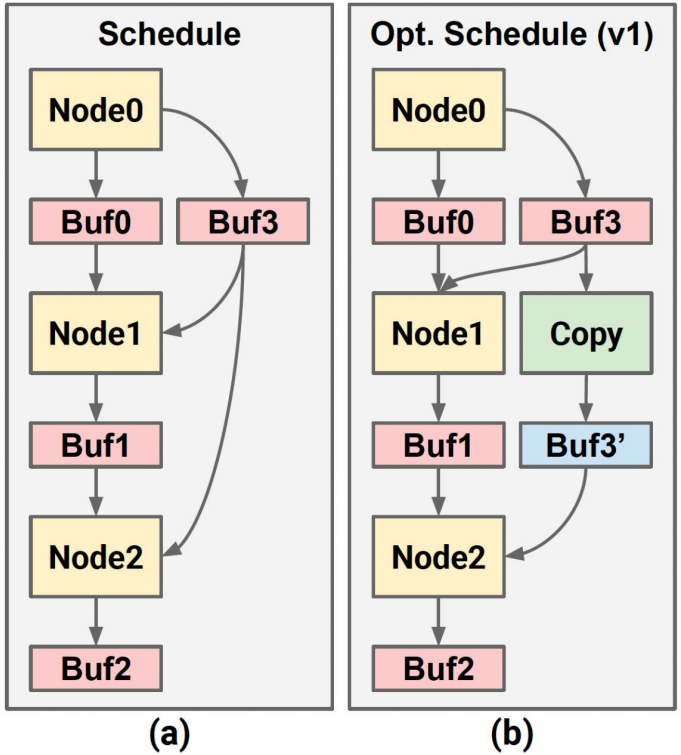
Buffer outside of the context

Data Paths Balancing



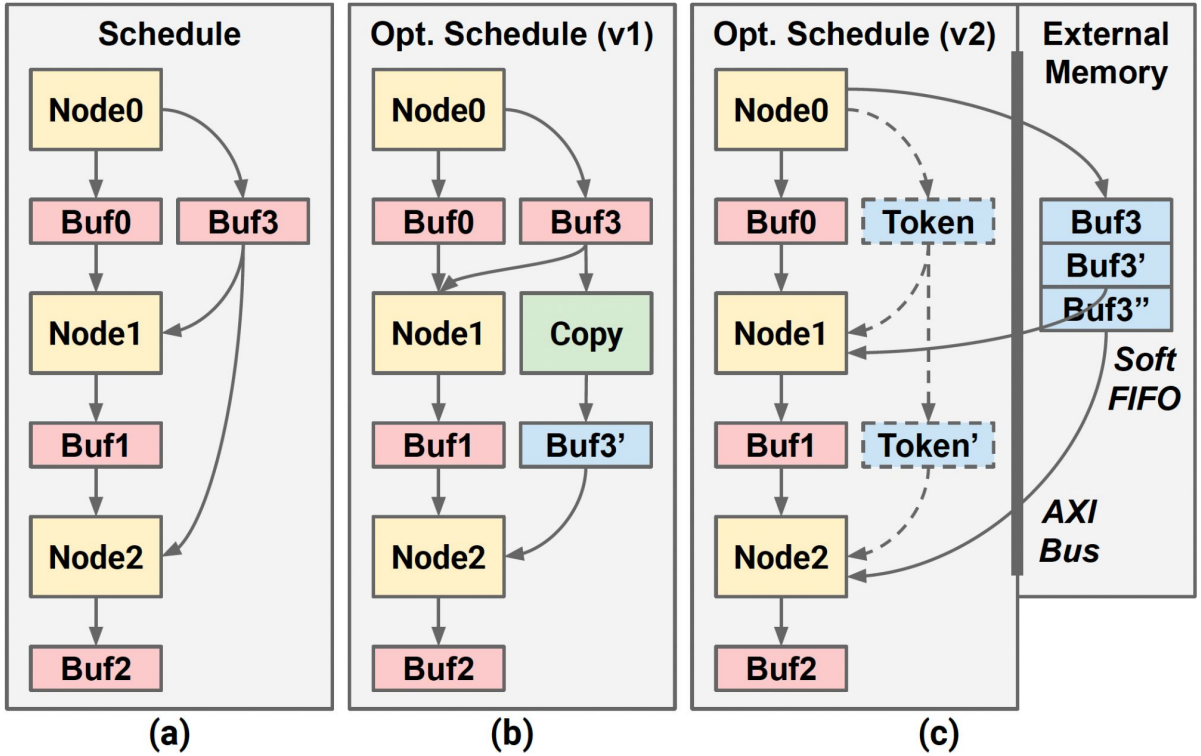
(a)

Data Paths Balancing



On-chip balancing

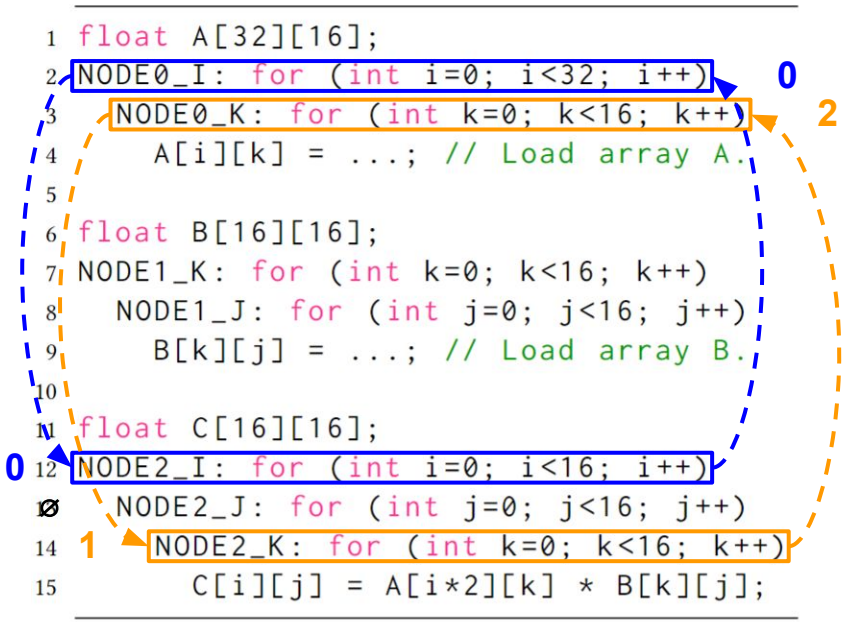
Data Paths Balancing



On-chip balancing

Off-chip balancing

HIDA Design Space Exploration

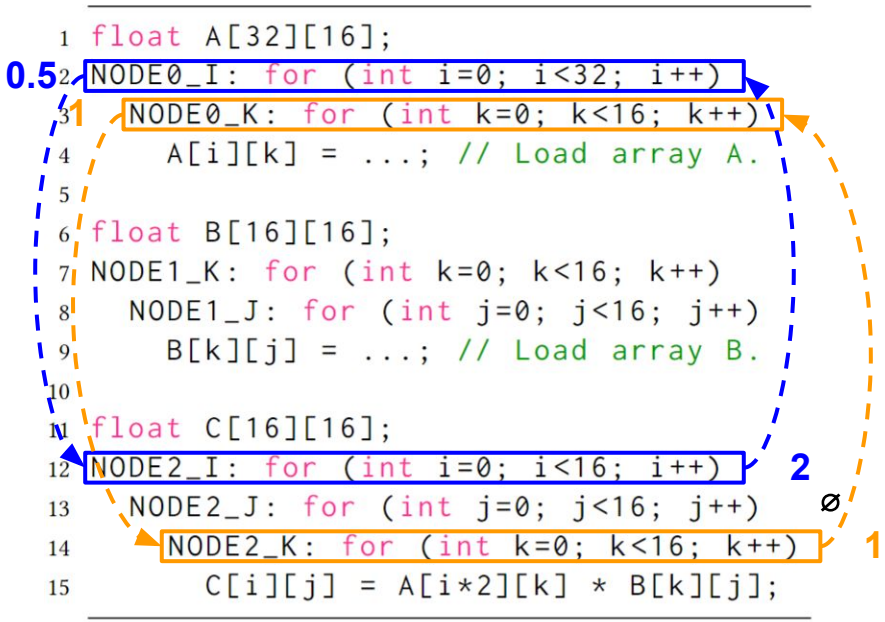


Step (1) Connectedness Analysis

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

HIDA Design Space Exploration (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, ∅, 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

HIDA Design Space Exploration (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**

HIDA Design Space Exploration (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
  - ScaleHLS DSE without constraints
  - Solution unroll factors: [4, 8, 1]

HIDA Design Space Exploration (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 (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]

HIDA Design Space Exploration (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 Node0 and 1 (32*256/4096)
  - ScaleHLS DSE with connectedness constraints
  - Solution unroll factors: [1, 2]

HIDA Design Space Exploration (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

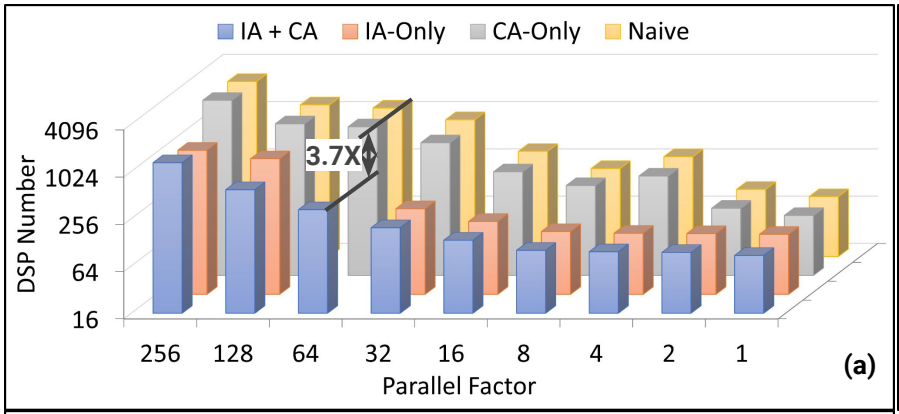
Node	Intensity	Parallel Factor		Loop Unroll Factors			
		w/o IA	w/ IA	IA+CA	IA	CA	Naive
Node0	512	32	4	[4, 1]	[2, 2]	[8, 4]	[4, 8]
Node1	256	32	2	[1, 2]	[1, 2]	[4, 8]	[4, 8]
Node2	4,096	32	32	[4, 8, 1]	[4, 8, 1]	[4, 8, 1]	[4, 8, 1]

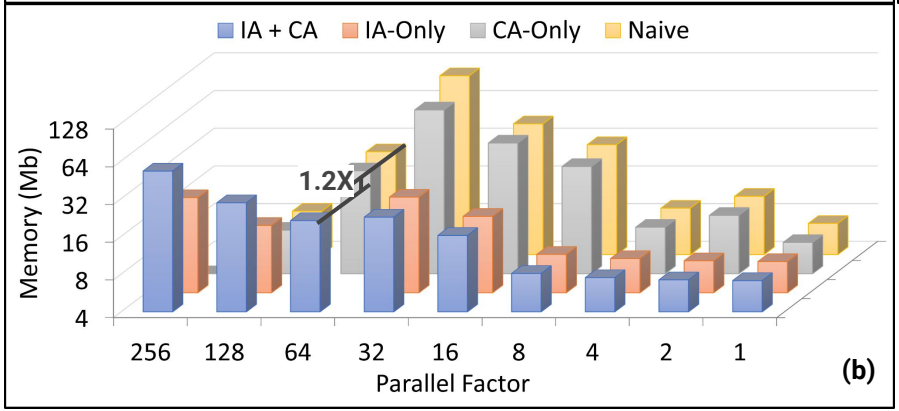
Intensity-aware (IA) Connectedness-aware (CA) HIDA DSE

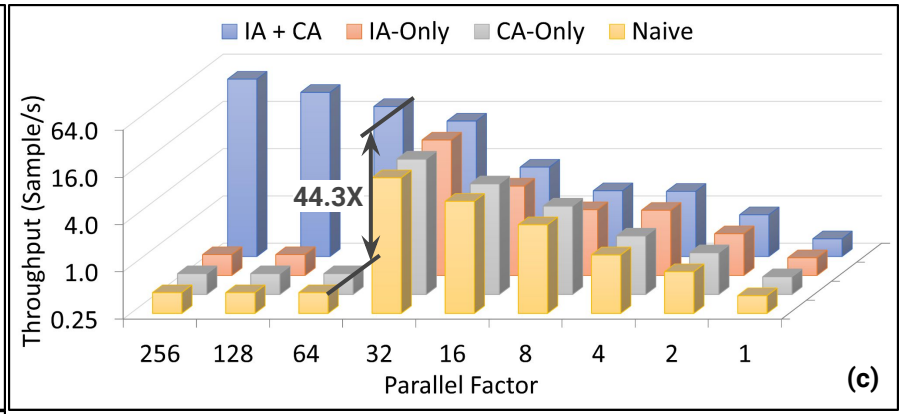
Naive ScaleHLS DSE

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

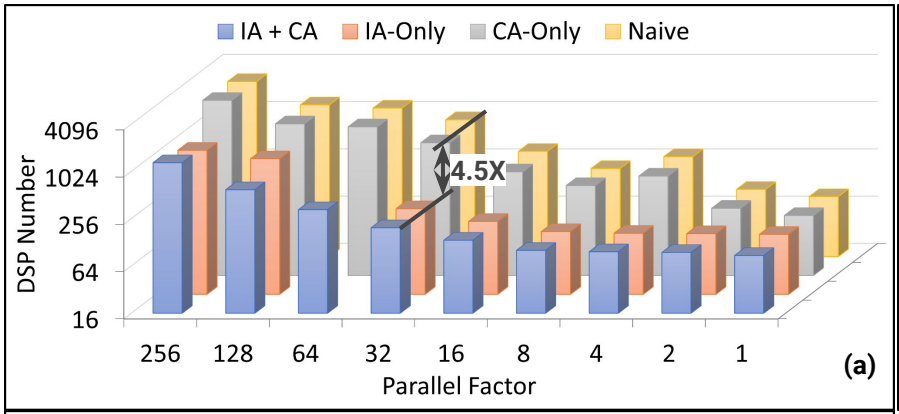


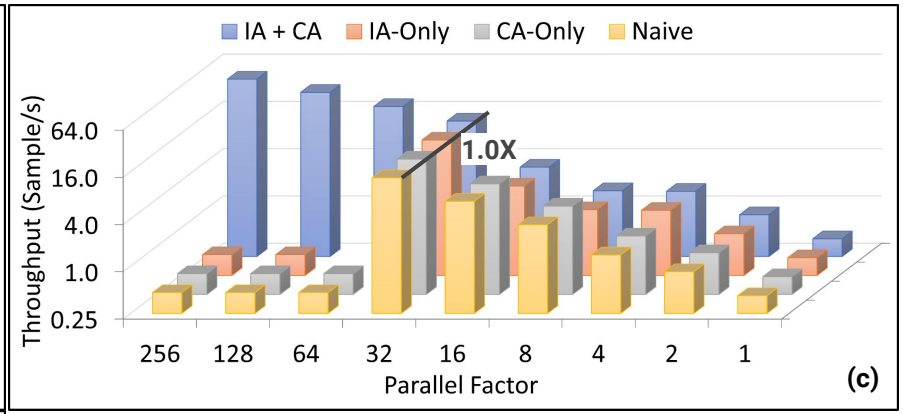


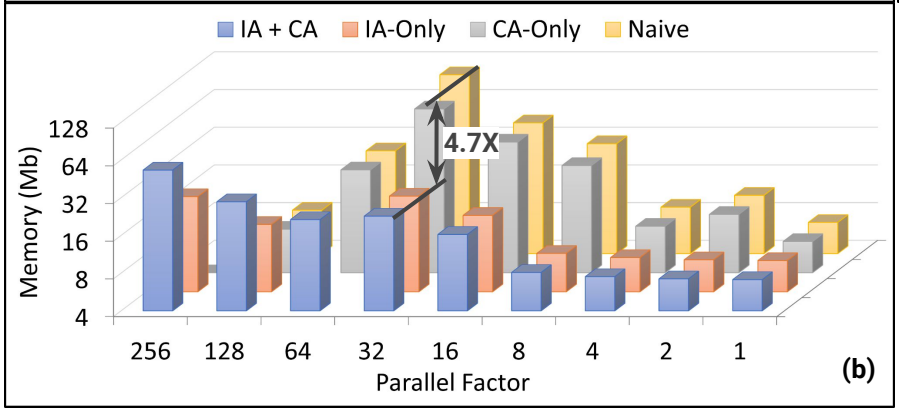


- IA+CA parallelization can determine whether the solution is scalable

ResNet-18 Ablation Study on HIDA (Cont.)







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

Outline

- ScaleHLS Recall
- Motivation
- HIDA Intermediate Representation
- HIDA Optimizations
- Evaluation Results
- Conclusion

HIDA Results on DNN Models

Kernel	HIDA Compile Time (s)	LUT Number	FF Number	DSP Number	Throughput (Samples/s)*			
					HIDA	ScaleHLS [70]	SOFF [37]	Vitis [34]
2mm	0.65	38.8k	27.4k	269	239.22	122.39 (1.95×)	30.67 (7.80×)	1.23 (194.88×)
3mm	0.79	38.7k	27.8k	243	175.43	92.33 (1.90×)	-	1.04 (167.99×)
atax	2.06	44.6k	34.6k	260	1,021.39	932.26 (1.10×)	2,173.17 (0.47×)	103.18 (9.90×)
bicg	0.72	16.0k	15.1k	61	2,869.69	2,869.61 (1.00×)	2,295.75 (1.25×)	104.19 (27.54×)
correlation	0.91	14.5k	12.3k	66	67.33	59.77 (1.13×)	3.96 (16.99×)	1.32 (50.97×)
gesummv	0.60	34.2k	22.8k	232	31,685.68	31,685.68 (1.00×)	3,466.70 (9.14×)	266.65 (118.83×)
jacobi-2d	1.98	91.4k	56.6k	352	257.27	128.63 (2.00×)	-	2.71 (94.95×)
mvt	0.42	23.8k	16.5k	162	9,979.04	4,989.02 (2.00×)	870.01 (11.47×)	62.13 (160.62×)
seidel-2d	3.59	5.5k	2.5k	4	0.14	0.14 (1.00×)	-	0.11 (1.28×)
symm	1.05	14.9k	9.5k	74	2.62	2.62 (1.00×)	-	2.02 (1.29×)
syr2k	0.69	14.3k	12.8k	78	27.68	27.67 (1.00×)	-	1.44 (19.23×)
Geo. Mean	**0.99**					**1.29×**	**4.49×**	**31.08×**

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

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.

Conclusion

HIDA GitHub Repository

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

- We propose a two-level dataflow intermediate representation, Functional and Structural dataflow, to capture the dataflow characteristics
- We propose a dataflow optimizer for efficient dataflow optimization and parallelization
- We demonstrate 8.54x and 1.29x higher throughputs over the SOTA HLS optimization framework and RTL-based neural networks accelerator