HIDA: A Hierarchical Dataflow Compiler for High-Level Synthesis

Hanchen Ye, Hyegang Jun, Deming Chen

University of Illinois Urbana-Champaign

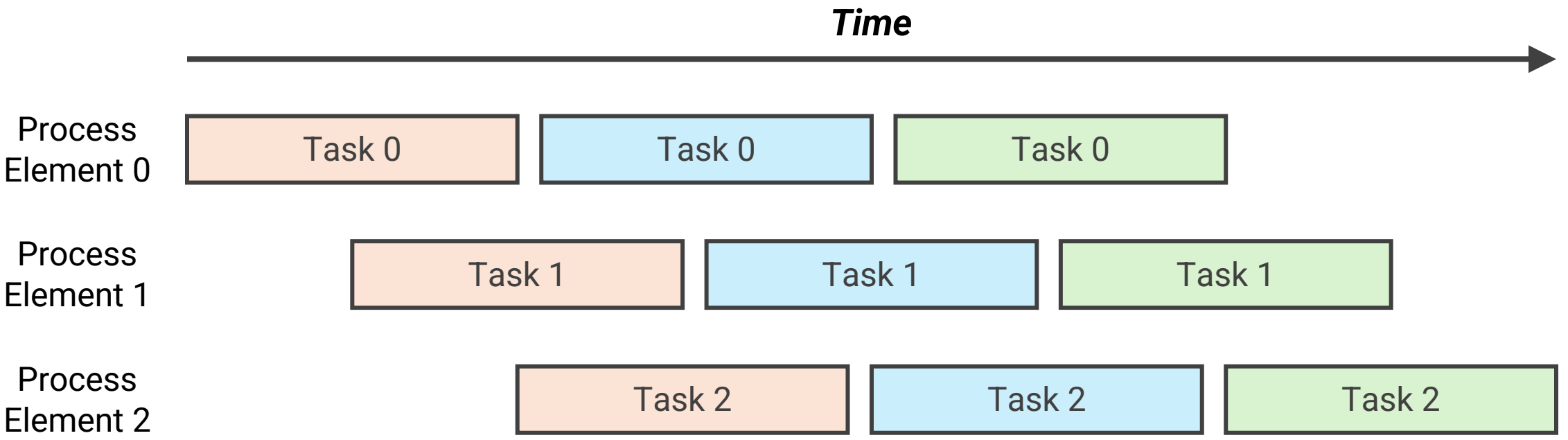
Apr. 29, 2024

Background

Why dataflow architecture?

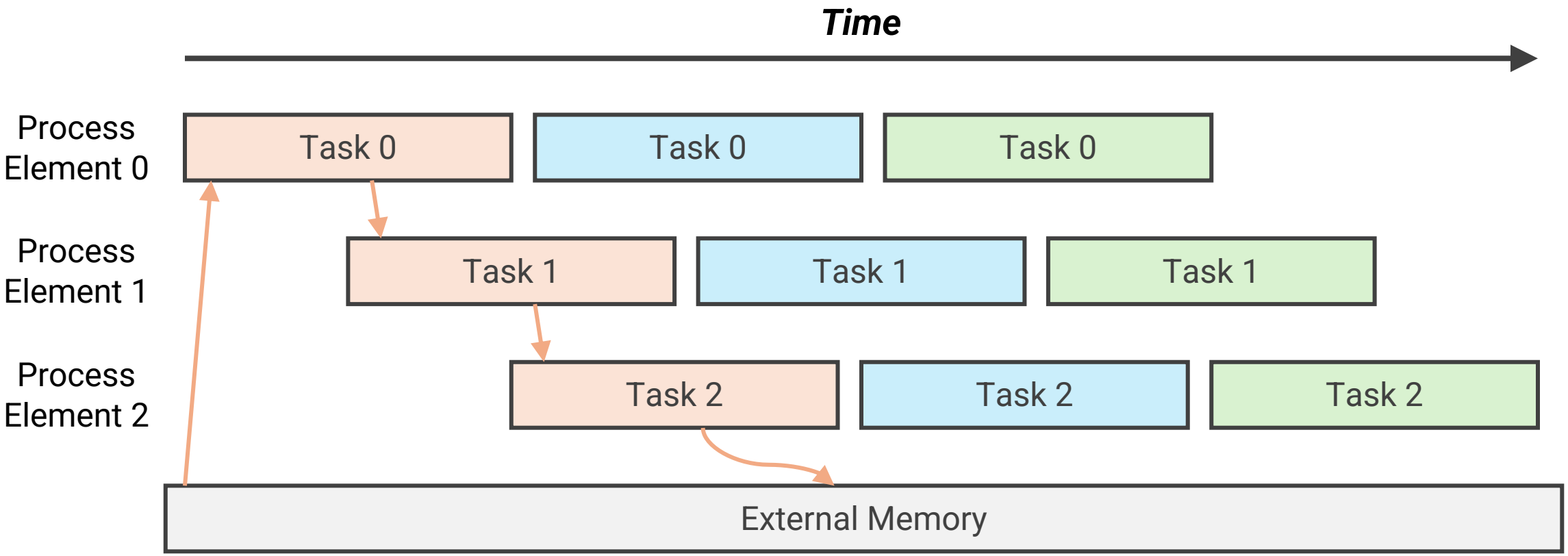
What is dataflow architecture?

- Dataflow architectures



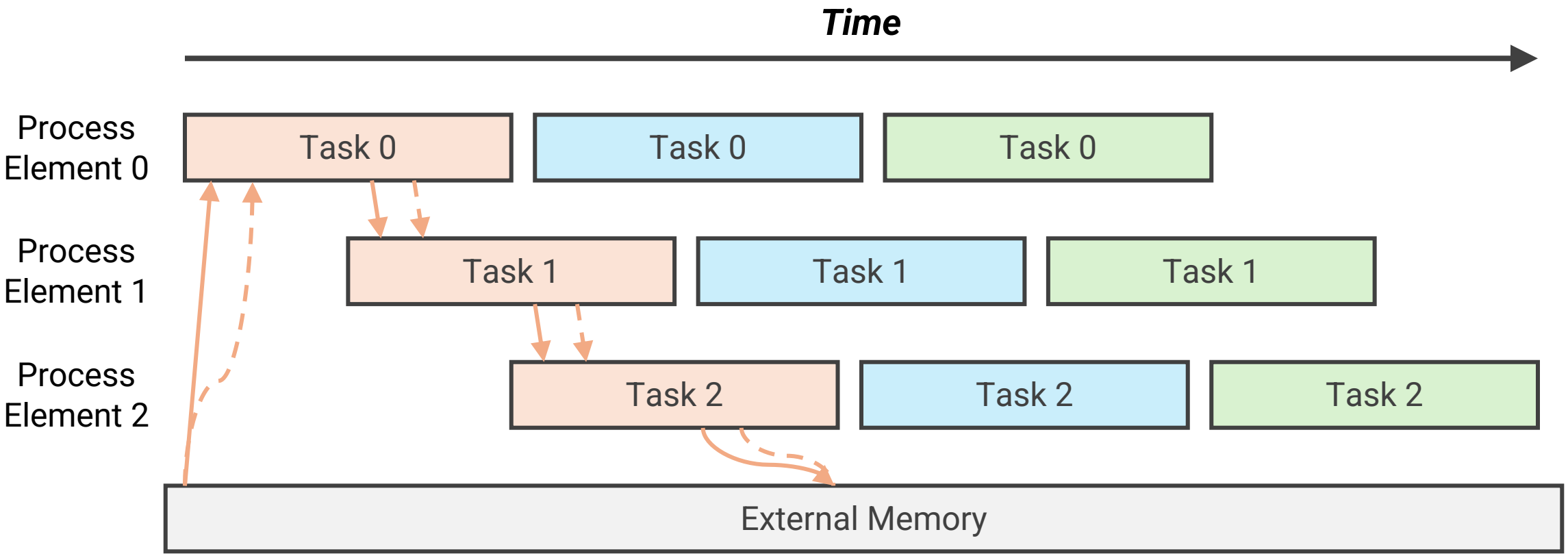
What is dataflow architecture? (Cont.)

- Dataflow architectures



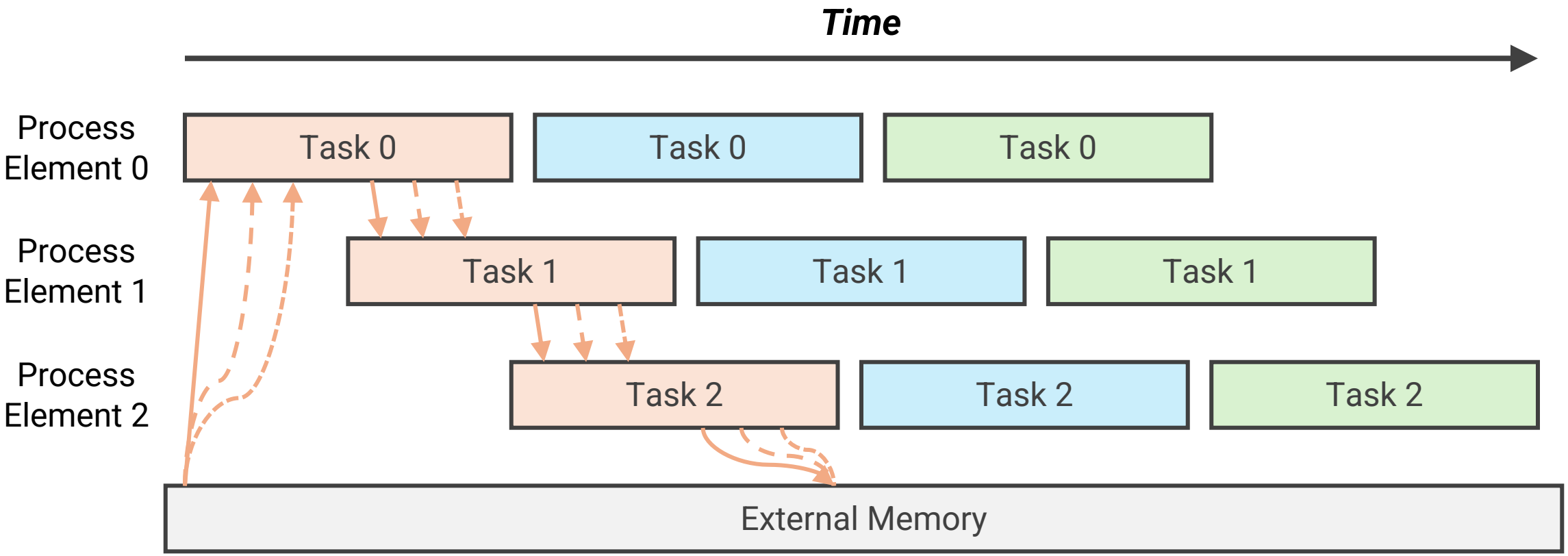
What is dataflow architecture? (Cont.)

- Dataflow architectures



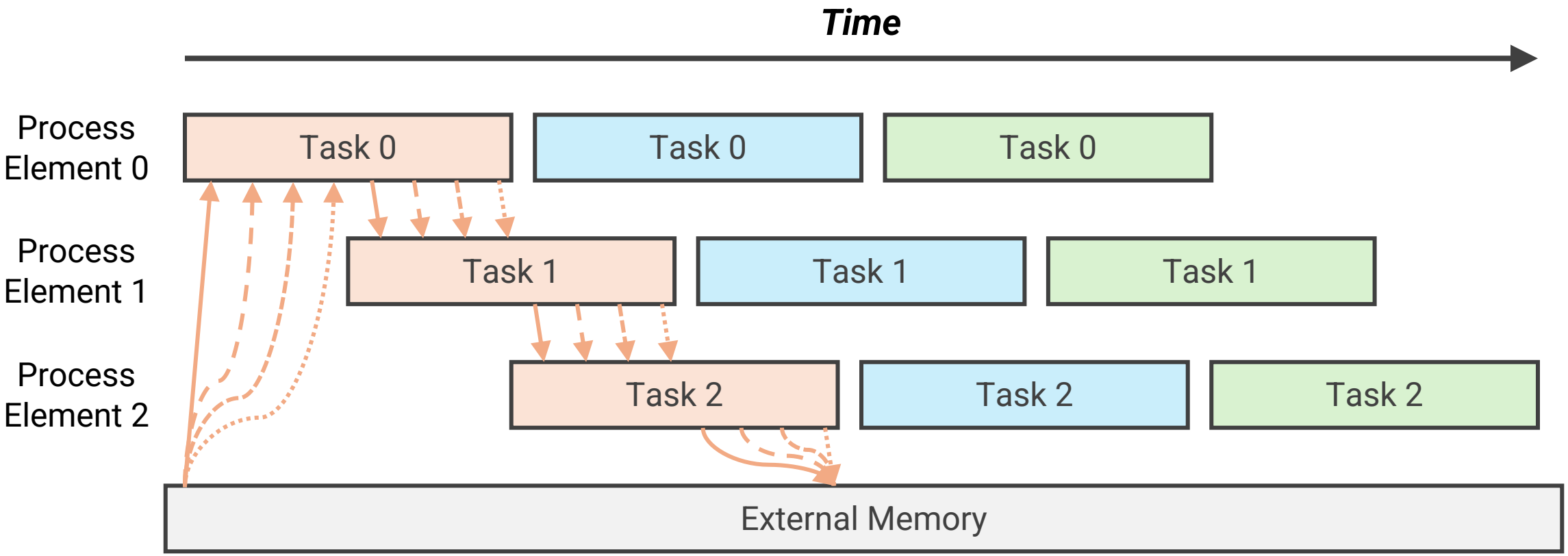
What is dataflow architecture? (Cont.)

- Dataflow architectures



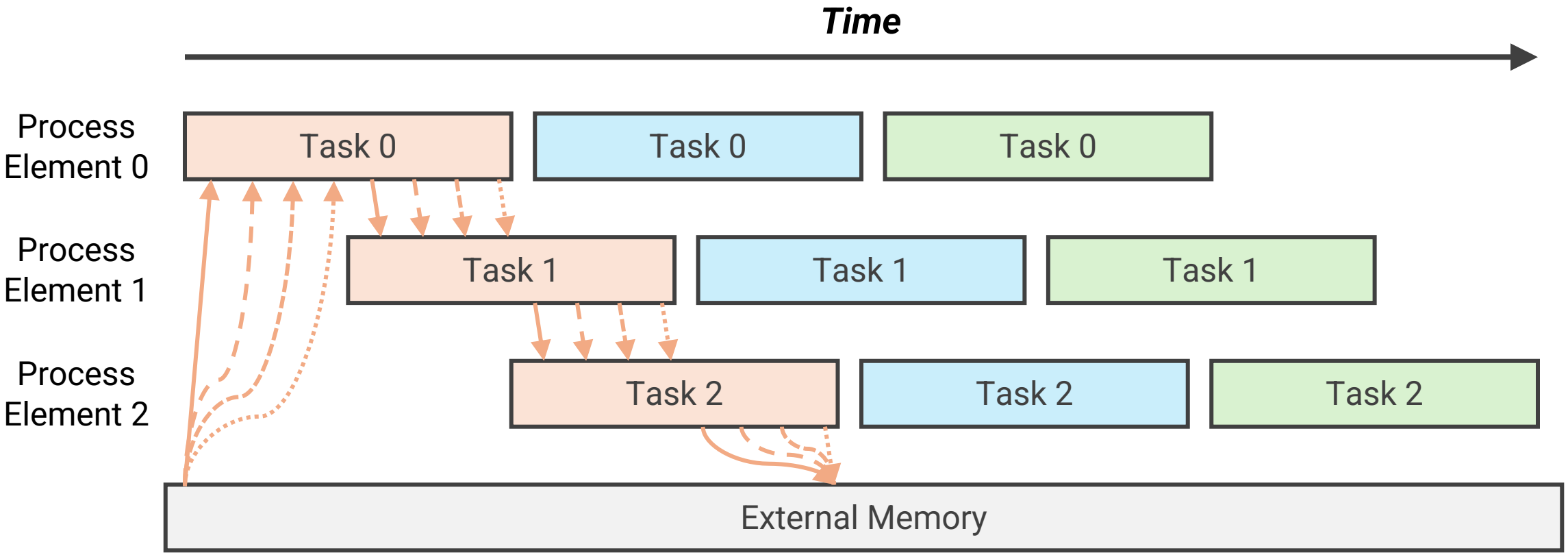
What is dataflow architecture? (Cont.)

- Dataflow architectures



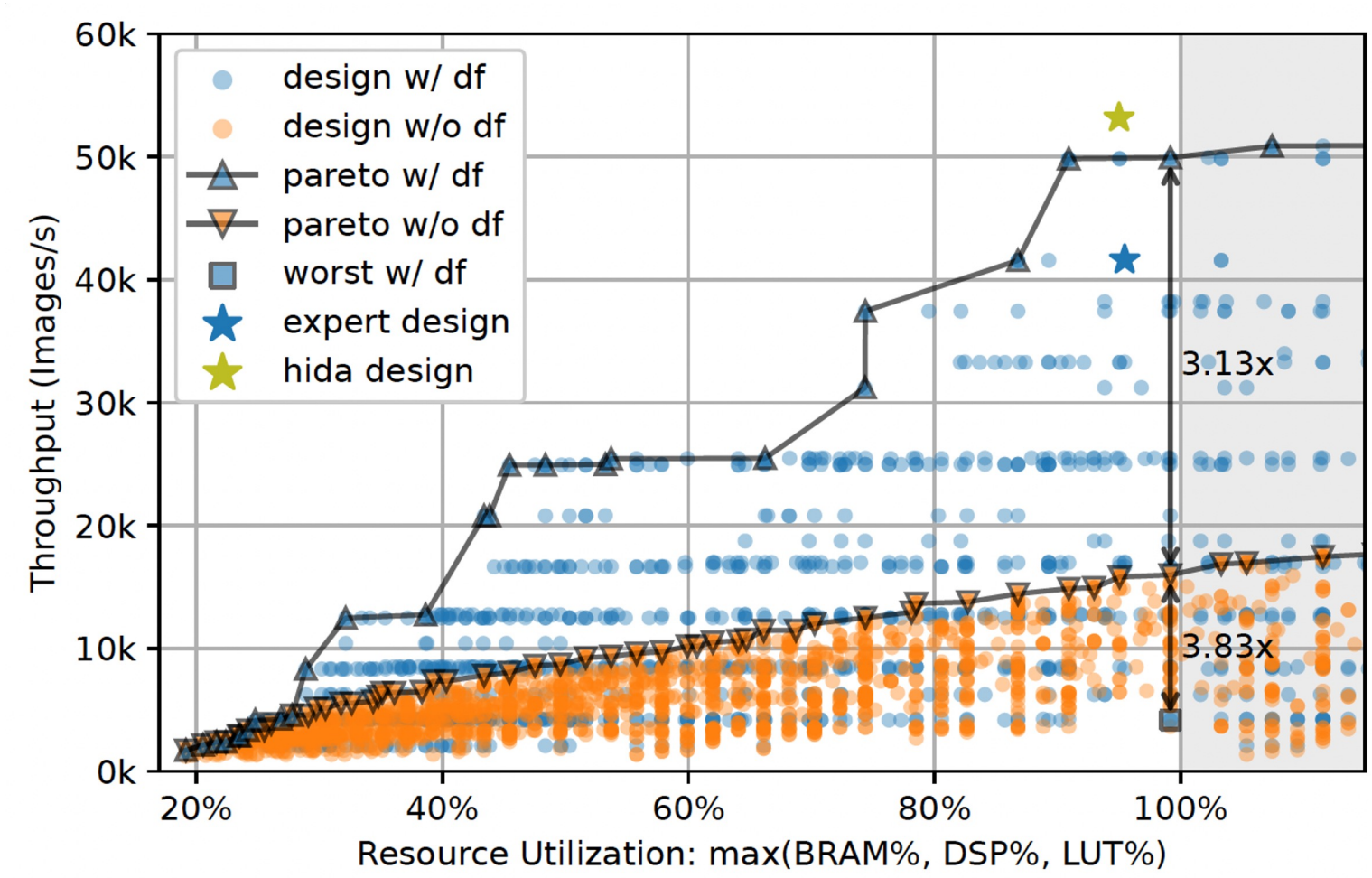
What is dataflow architecture? (Cont.)

- Dataflow architectures



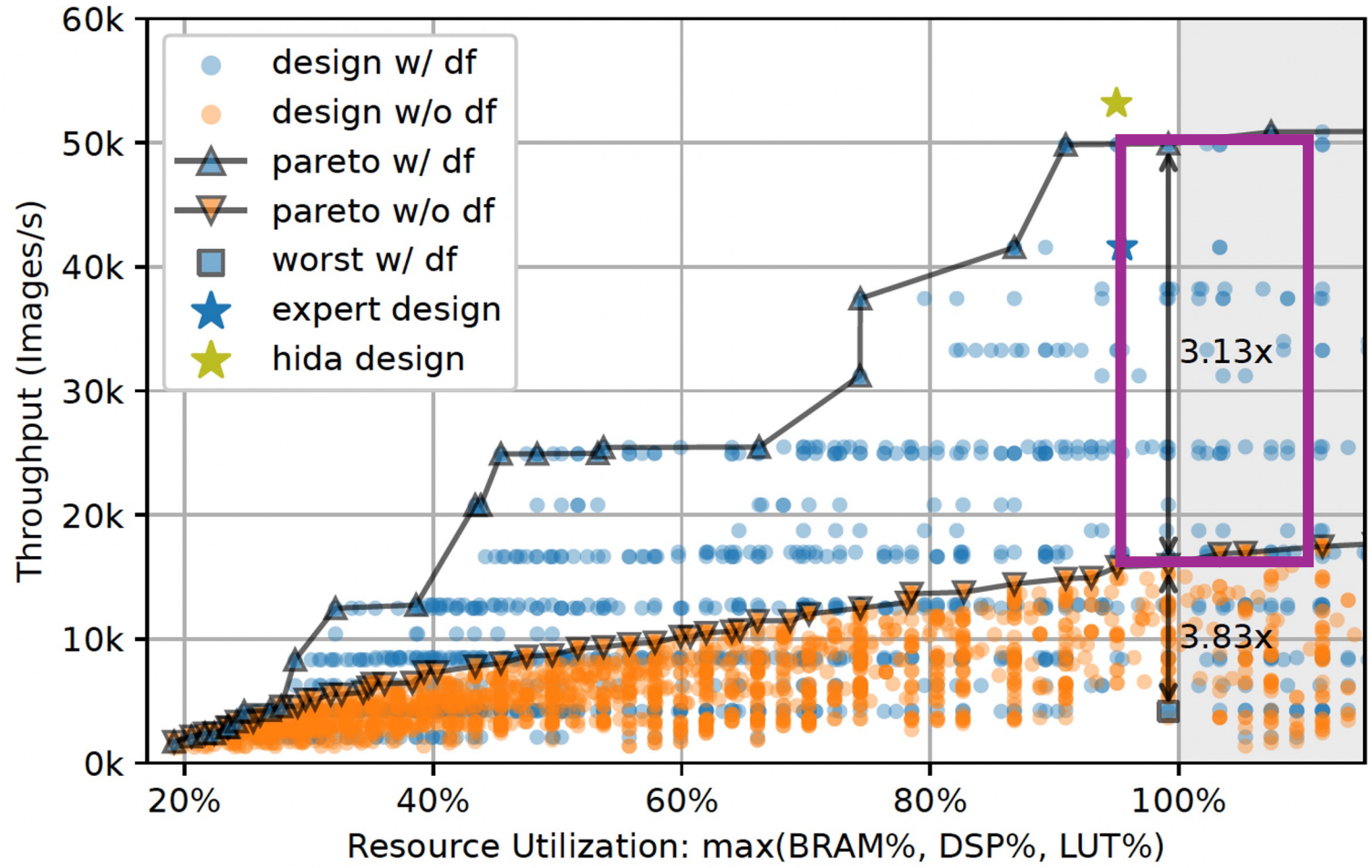
- ***Keep intermediate data on chip*** *– reduce external memory access*
- ***Overlap task execution*** *– reduce latency and on-chip memory utilization*

Case study: An LeNet accelerator on FPGA



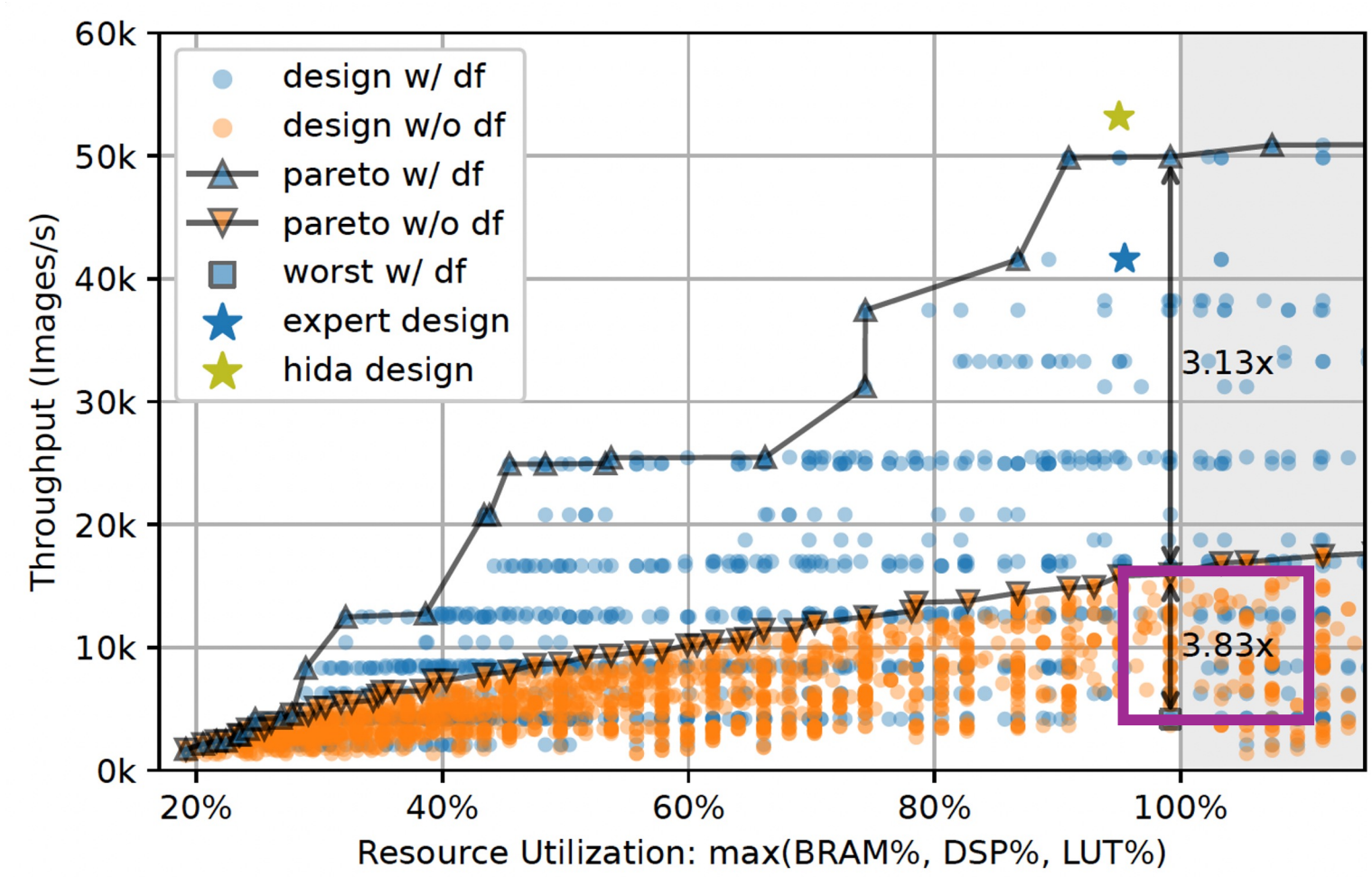
Case study: An LeNet accelerator on FPGA (Cont.)

- Dataflow designs are Pareto dominating



Case study: An LeNet accelerator on FPGA (Cont.)

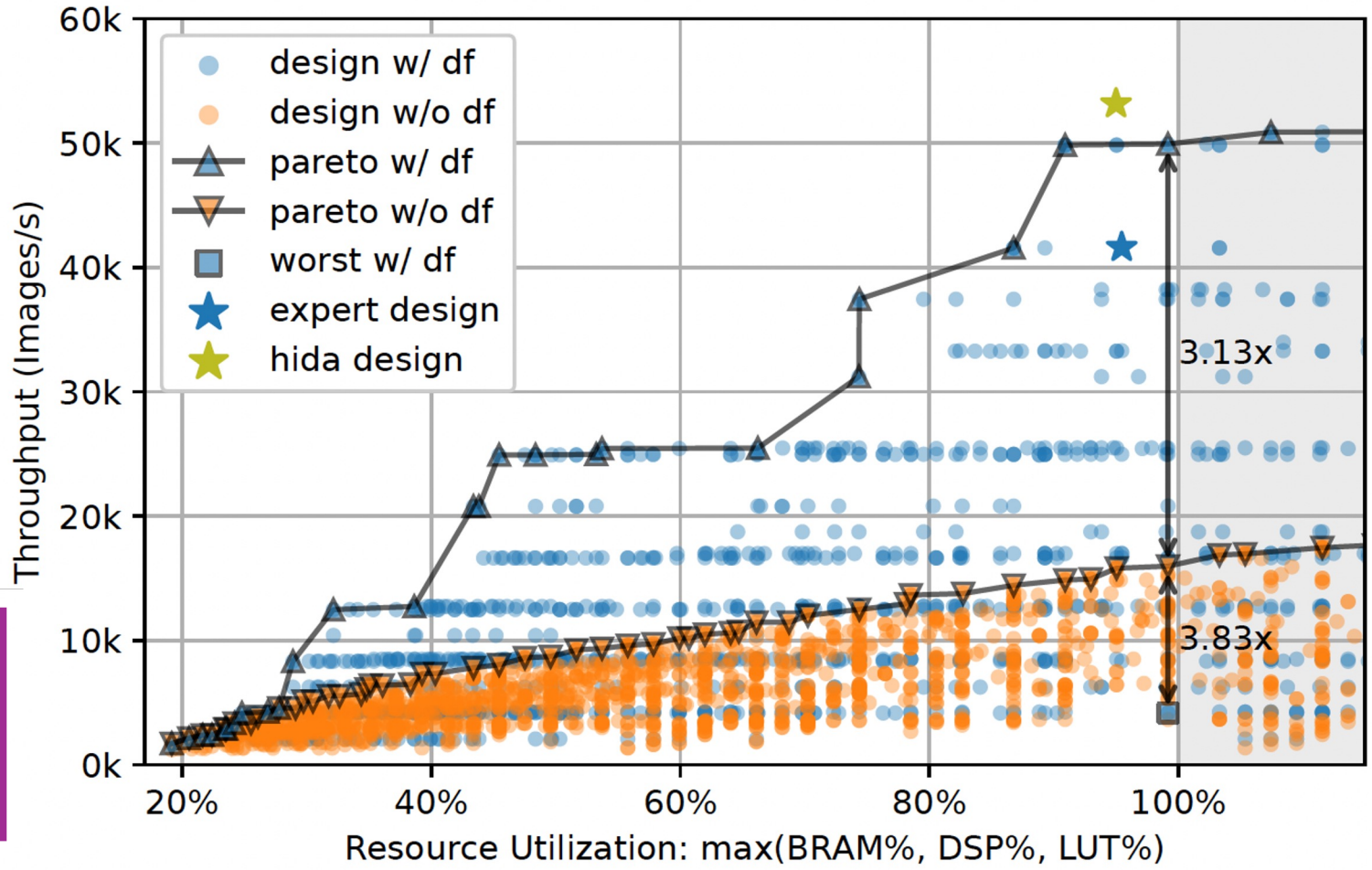
- Dataflow designs are Pareto dominating
- Dataflow cannot guarantee a good trade-off



Case study: An LeNet accelerator on FPGA (Cont.)

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

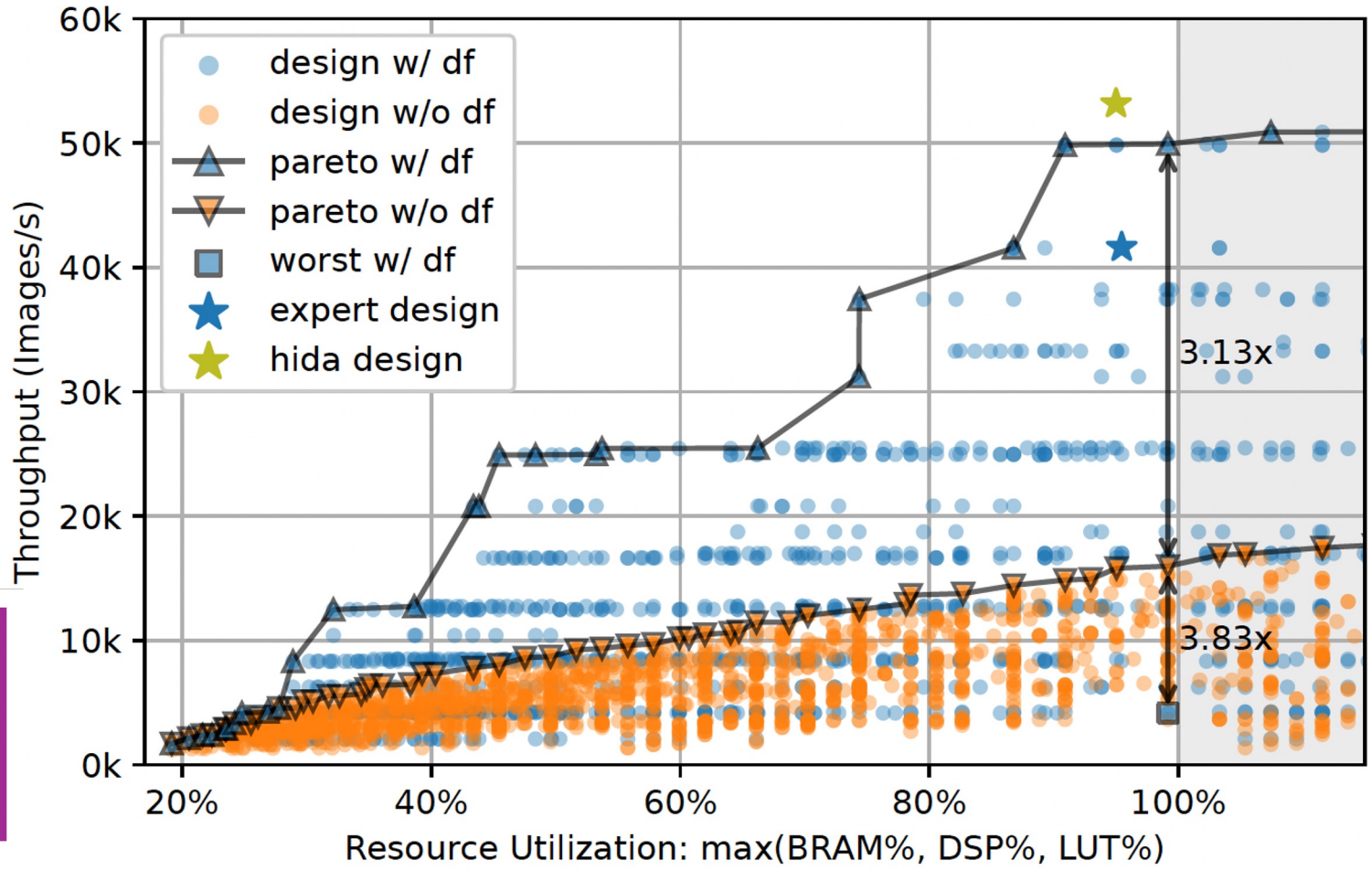
	Expert	Exhaustive	HIDA
Resource Util.	95.5%	99.2%	95.0%
Throu. (Imgs/s)	41.6k	49.9k	53.2k
Develop Cycle	40 hours	210 hours	9.9 mins



Case study: An LeNet accelerator on FPGA (Cont.)

- Dataflow designs are Pareto dominating
- Dataflow cannot guarantee a good trade-off
- Dataflow design space is large to comprehend
- Automated tool can outperform 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



Why designing dataflow architecture is hard?

Motivation

Why designing dataflow architecture is hard?

Two levels of dataflow optimization

Two levels of dataflow optimization

- Tensor & Linear algebra optimizations
  - Tiling, fusion, permutation, packing, etc.
- Full tensor reduction
  - Reducing full tensor to tiled partial tensors
- Task manipulation
  - Placement, scheduling, etc.
- … …

Two levels of dataflow optimization

High-level Dataflow Optimizations

Low-level Dataflow Optimizations

- Tensor & Linear algebra optimizations
  - Tiling, fusion, permutation, packing, etc.
- Full tensor reduction
  - Reducing full tensor to tiled partial tensors
- Task manipulation
  - Placement, scheduling, etc.
- … …

- Ping-pong Buffer optimizations
  - Placement, partitioning, etc.
- Stream channel optimizations
  - Placement, vectorization, sizing, etc.
- Task optimizations
  - Pipelining, vectorization, etc.
- Backend-specific optimizations
  - FPGA, AMD AI Engine, etc.
- … …

Two levels of dataflow optimization

High-level Dataflow Optimizations

Low-level Dataflow Optimizations

- Tensor & Linear algebra optimizations
  - Tiling, fusion, permutation, packing, etc.
- Full tensor reduction
  - Reducing full tensor to tiled partial tensors
- Task manipulation
  - Placement, scheduling, etc.
- … …

- Ping-pong Buffer optimizations
  - Placement, partitioning, etc.
- Stream channel optimizations
  - Placement, vectorization, sizing, etc.
- Task optimizations
  - Pipelining, vectorization, etc.
- Backend-specific optimizations
  - FPGA, AMD AI Engine, etc.
- … …

Two levels of optimization are at distinct abstraction levels

Inter-kernel correlation

```
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 (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];
```

- 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

Inter-kernel correlation (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];
```

- 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

Inter-kernel correlation (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];
```

- 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

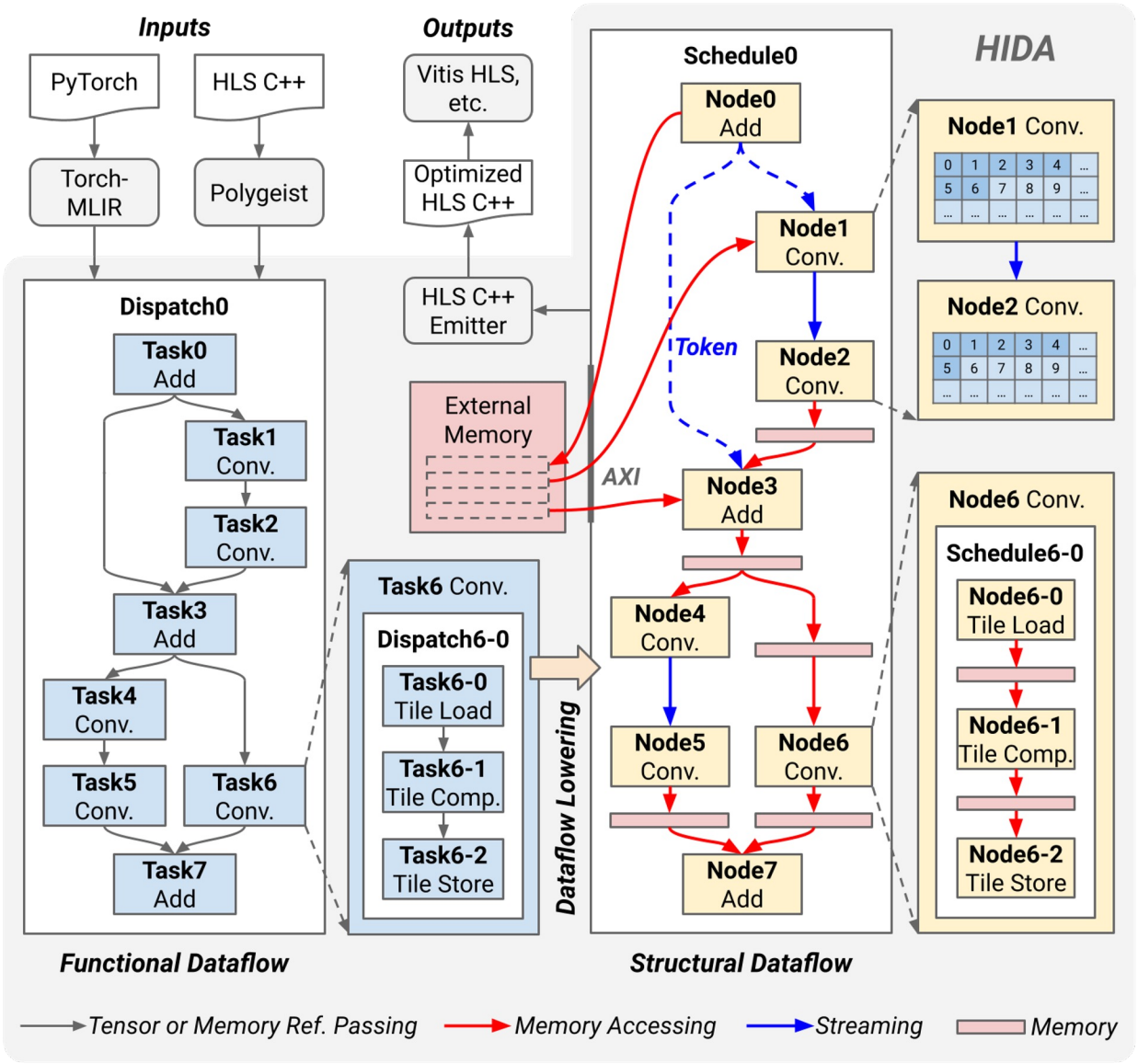
Intensity

Optimizing kernels separately can lead to poor global performance

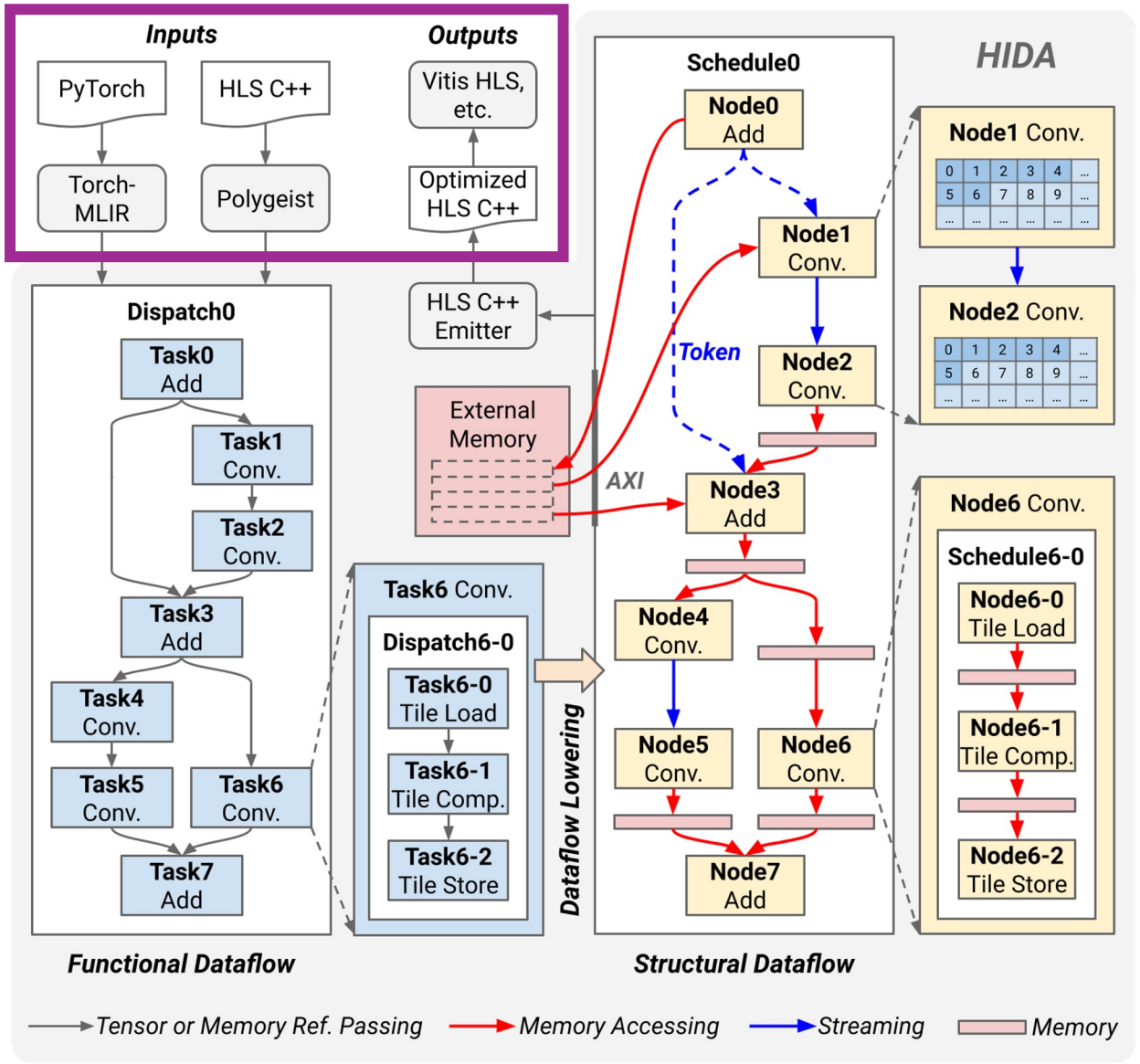
HIDA Framework

Hierarchical dataflow representation and optimization

HIDA framework overview

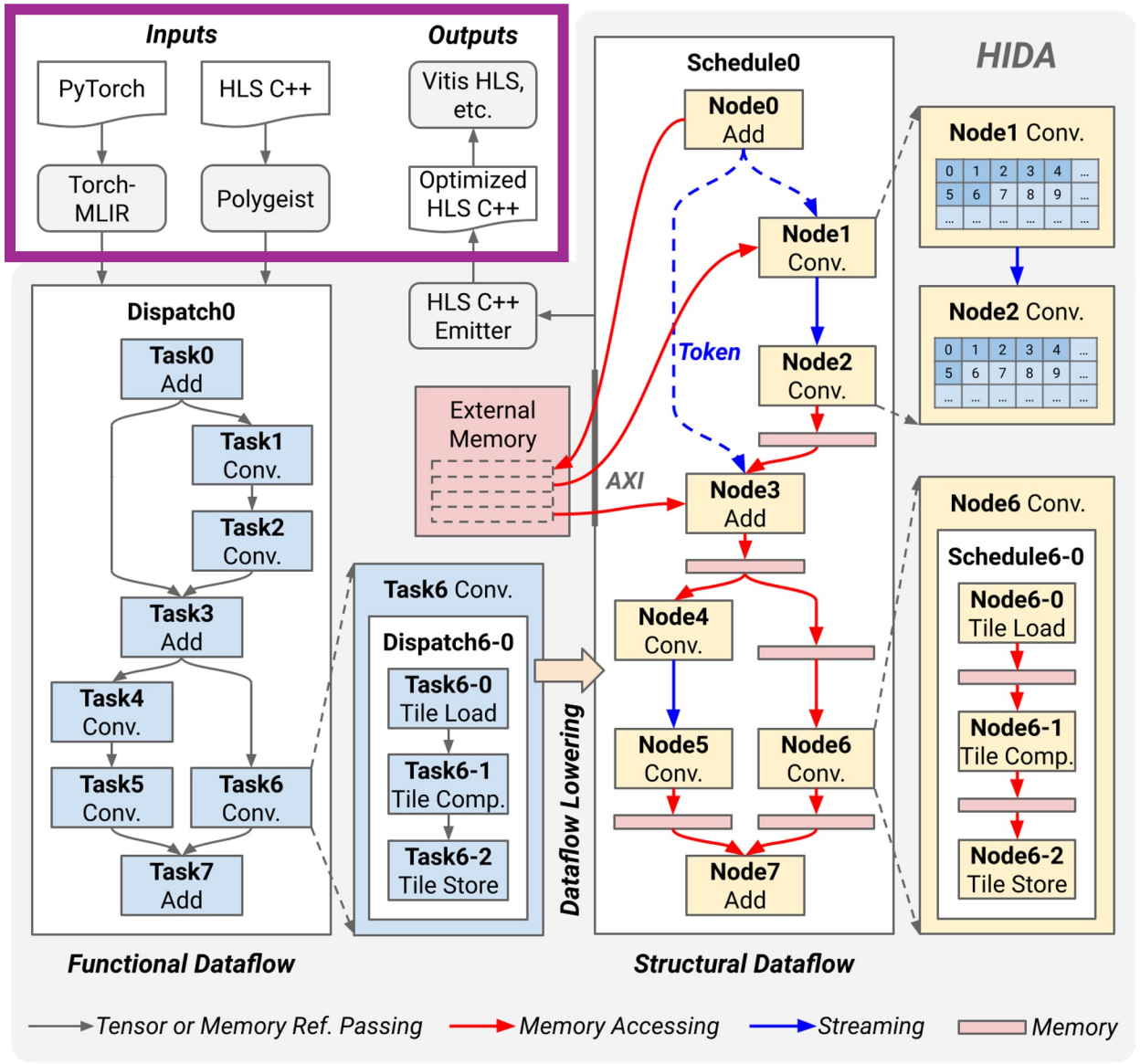


HIDA framework overview (Cont.)



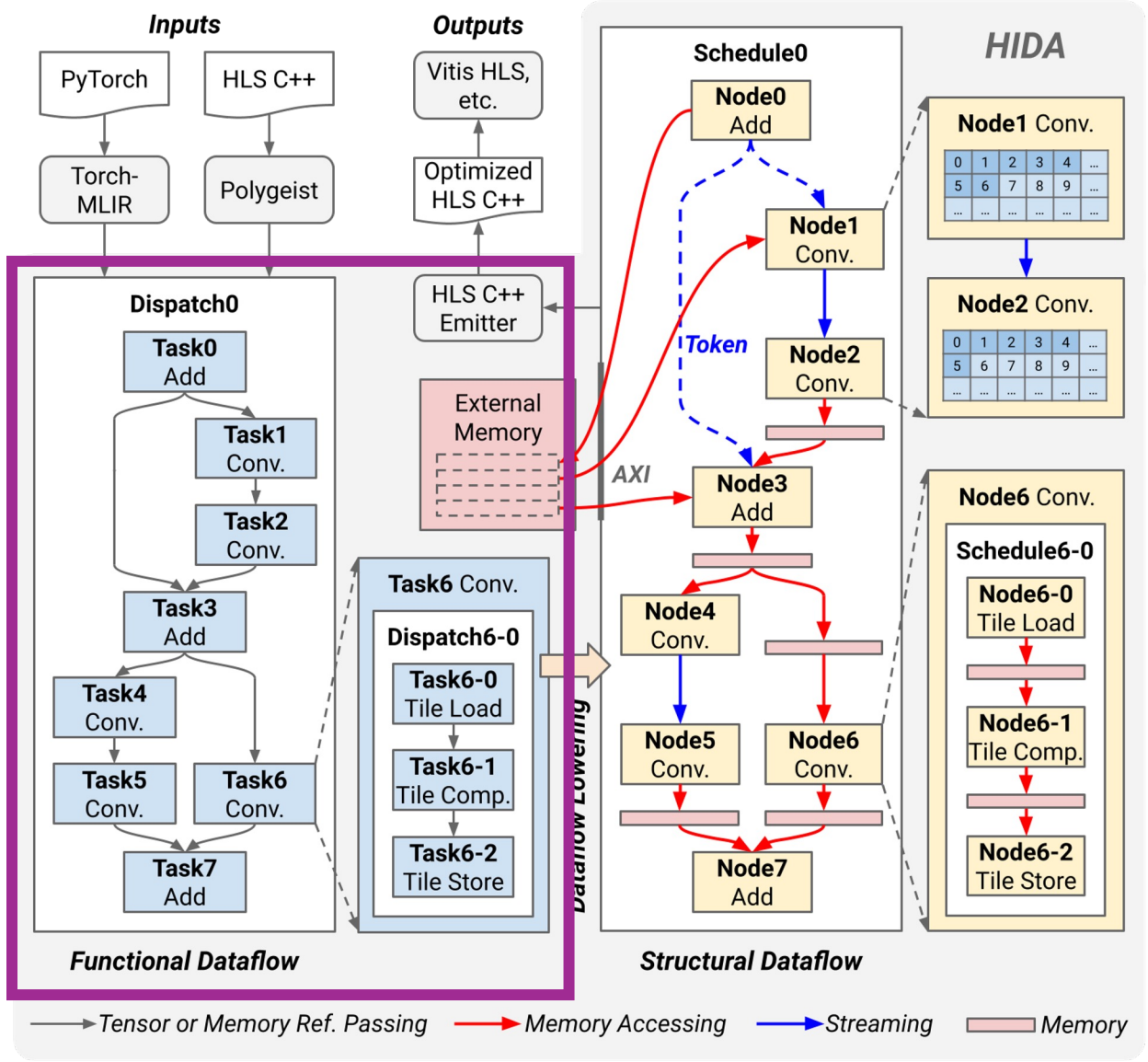
- PyTorch or C/C++ as input
- Optimized C++ dataflow design as output

HIDA framework overview (Cont.)



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

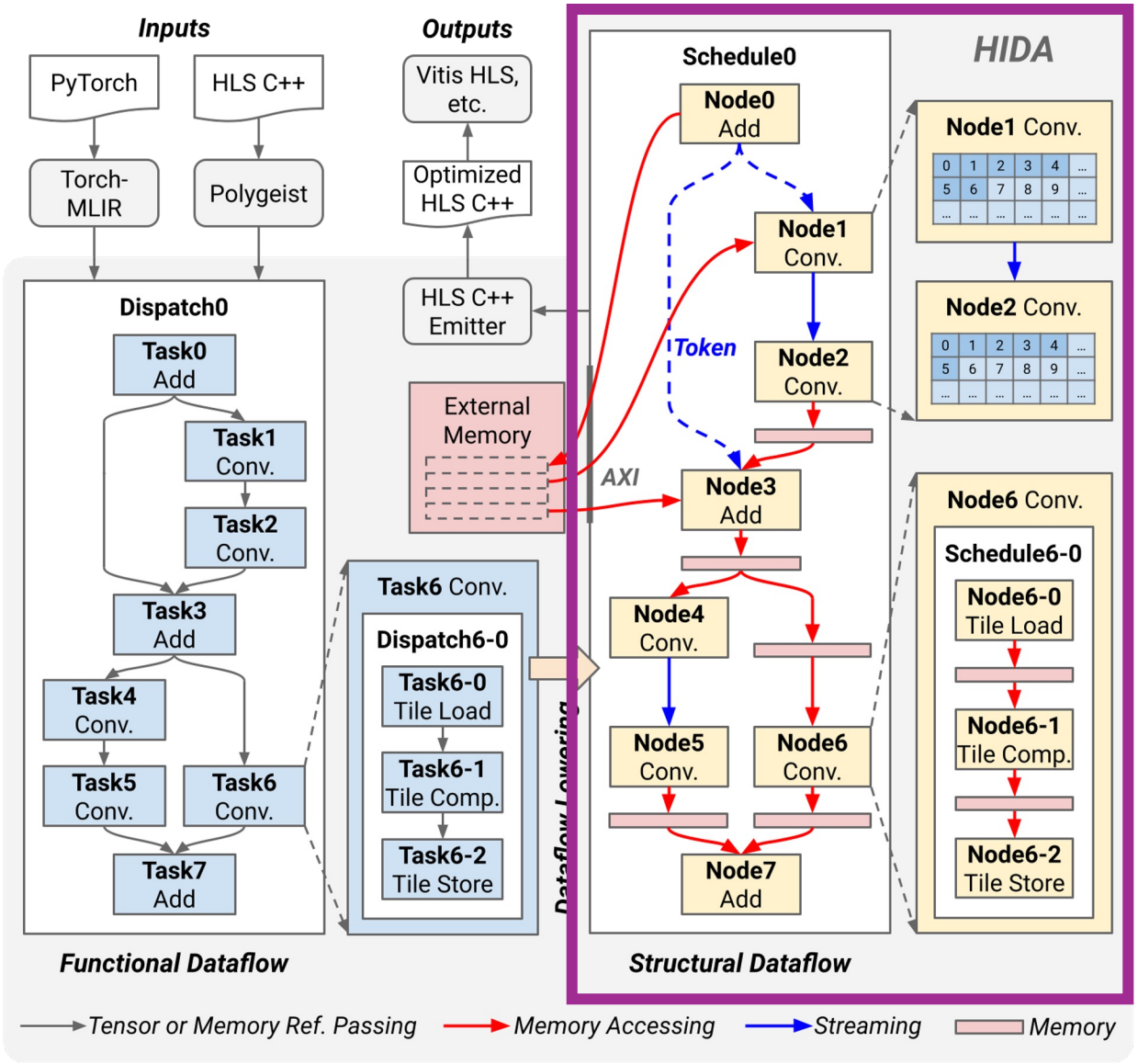
HIDA functional dataflow




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

- 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 manipulation
- Carry high-level optimizations

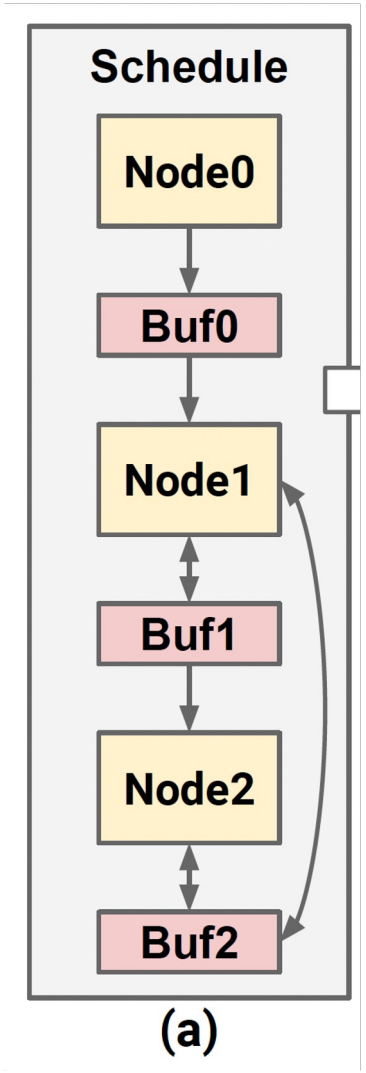
HIDA structural dataflow




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

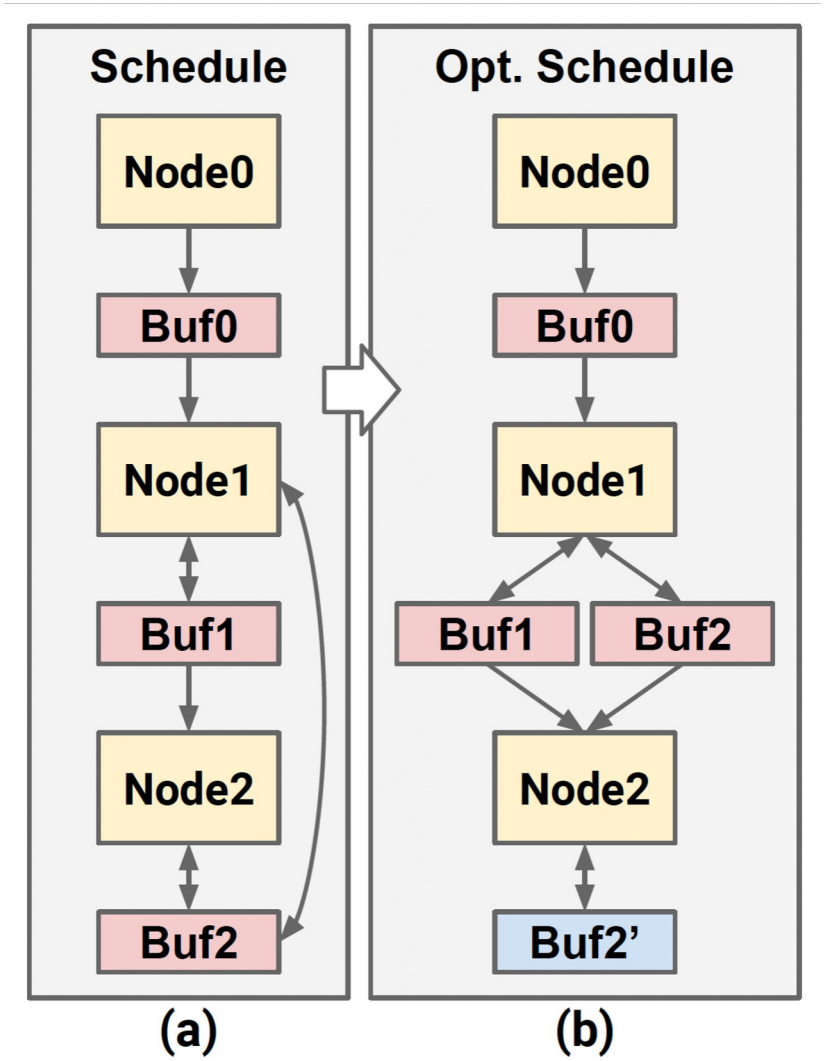
- Buffer representation
  - Support both ping-pong buffer and stream channels
- Isolated from above
  - Each node has its own context
  - Decouple inter-node and intra-node dataflow optimization
- Carry low-level optimizations

Optimization #1: Multiple producer elimination



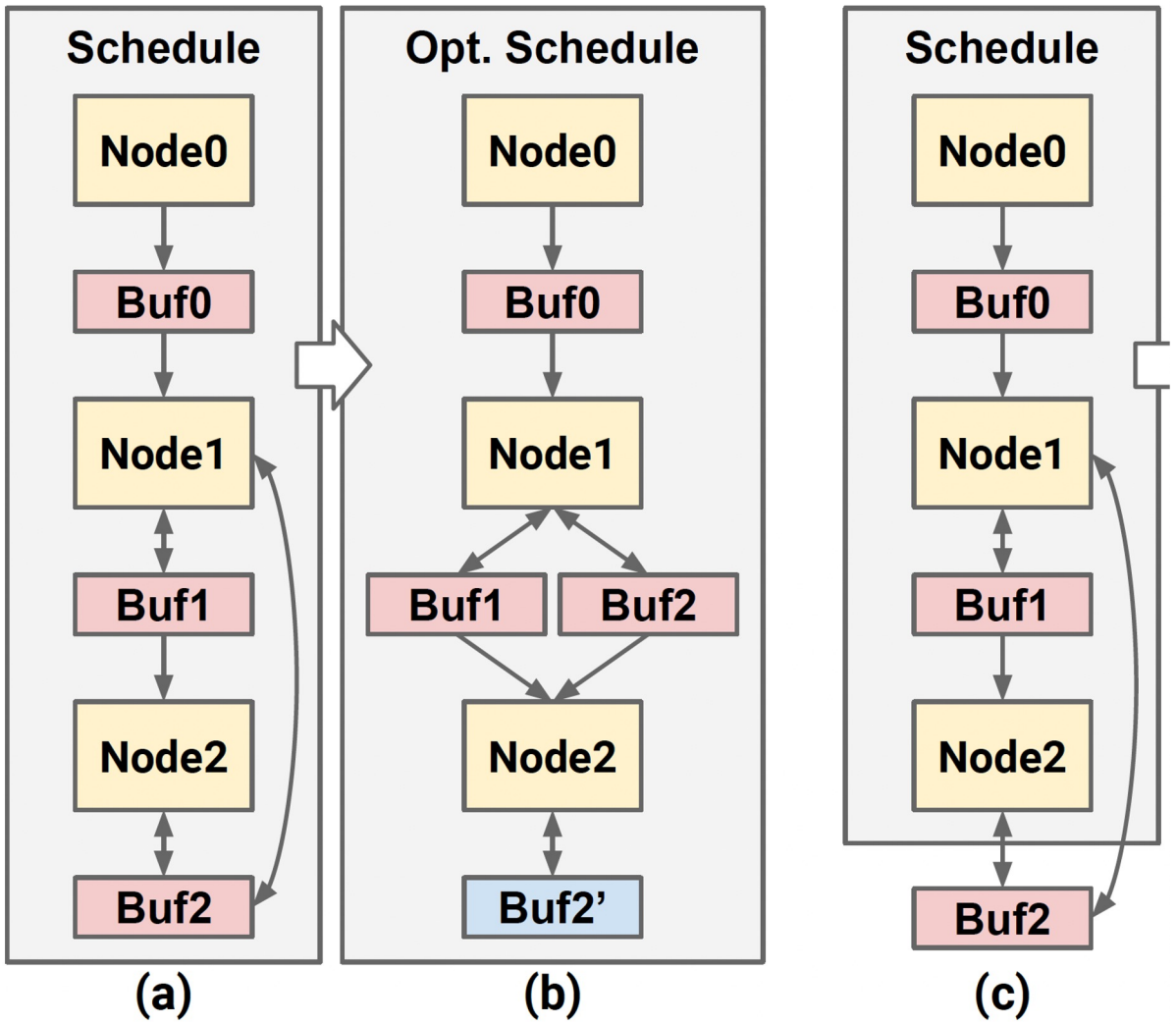
- Buffer inside of the context

Optimization #1: Multiple producer elimination (Cont.)



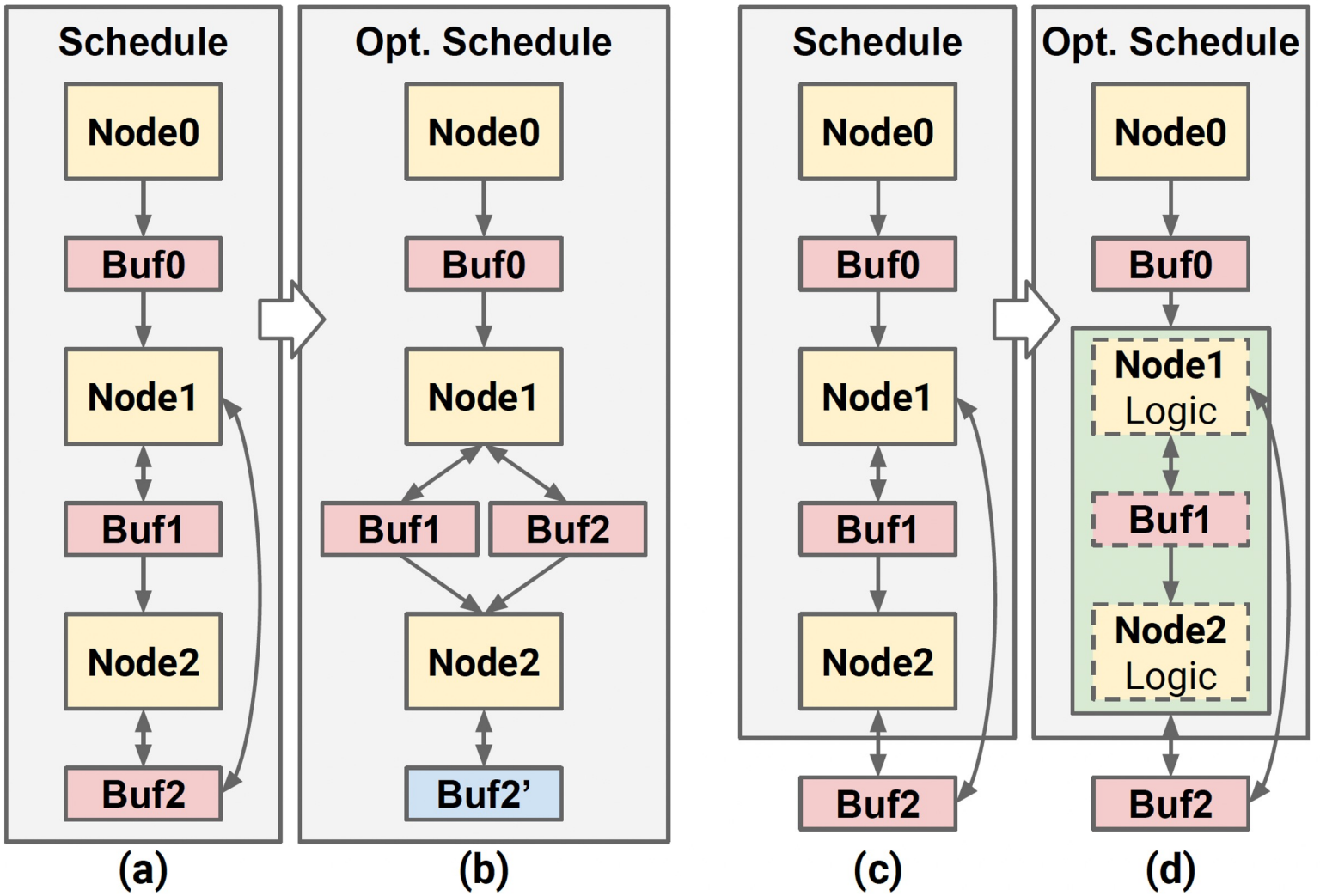
- Buffer inside of the context

Optimization #1: Multiple producer elimination (Cont.)



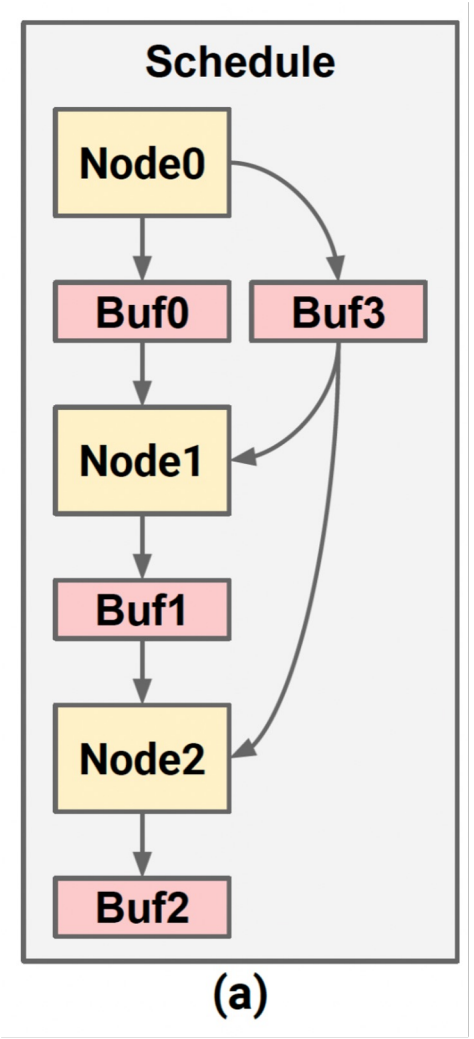
- Buffer inside of the context
- Buffer outside of the context

Optimization #1: Multiple producer elimination (Cont.)

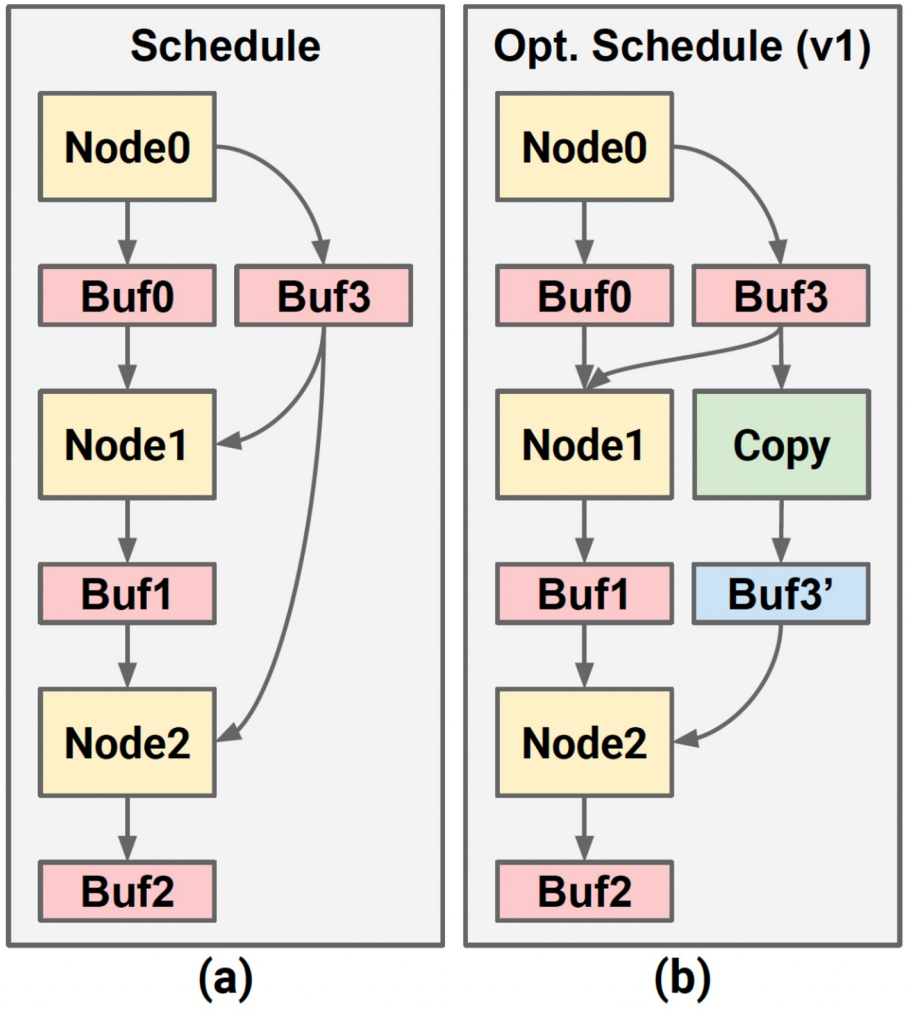


- Buffer inside of the context
- Buffer outside of the context

Optimization #2: Data paths balancing

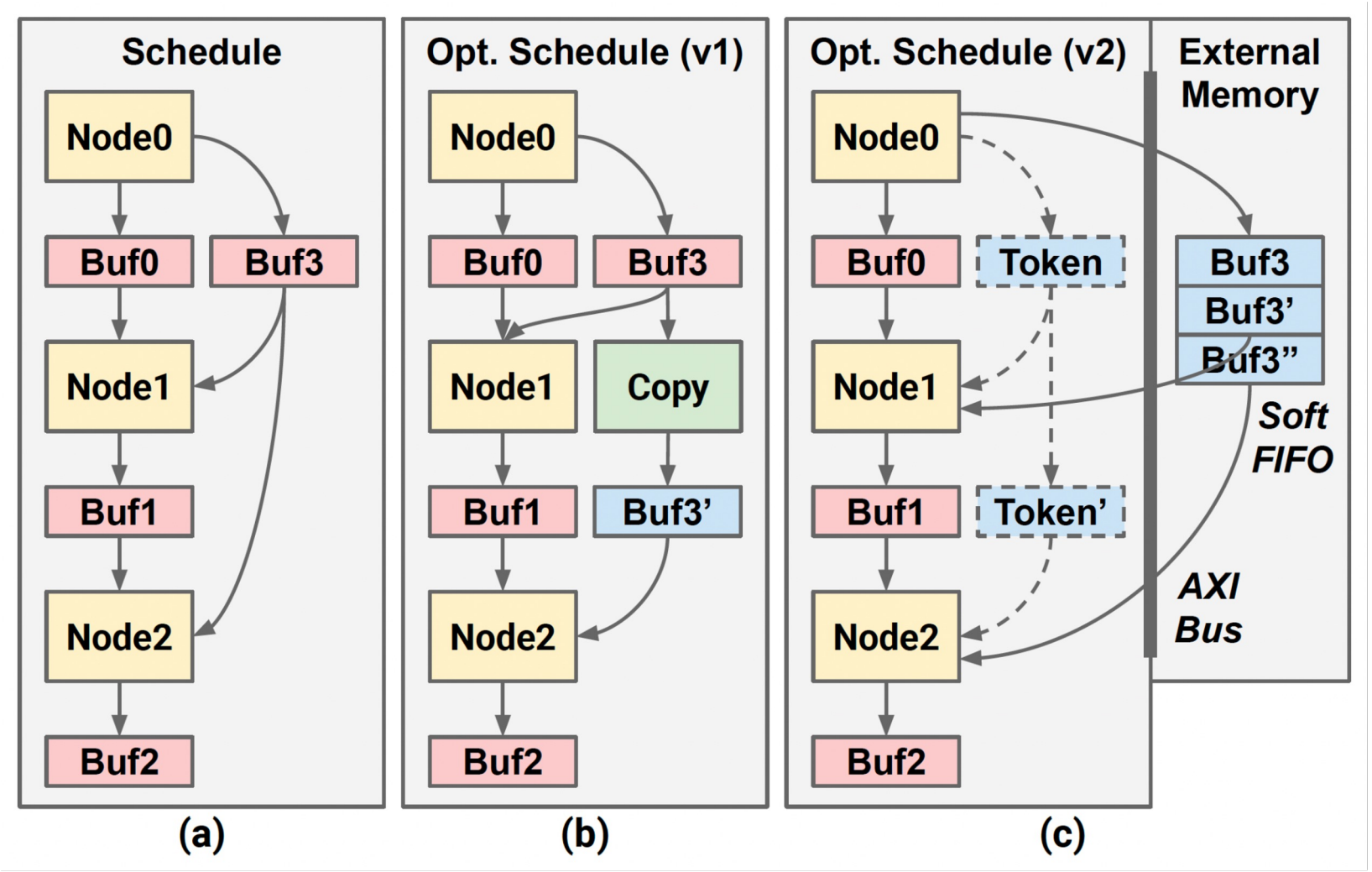


Optimization #2: Data paths balancing



- On-chip balancing

Optimization #2: Data paths balancing

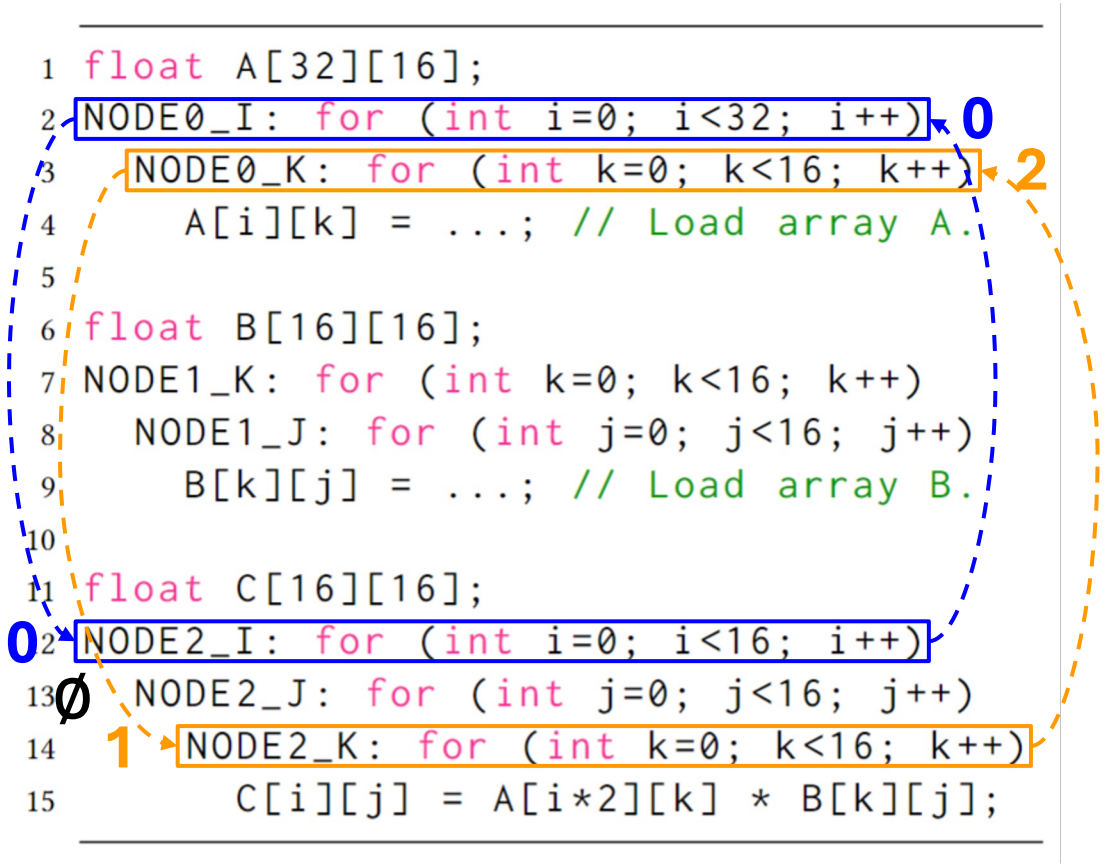


- On-chip balancing
- Off-chip balancing

HIDA Design Space Exploration

Dataflow-aware exploration

HIDA design space exploration

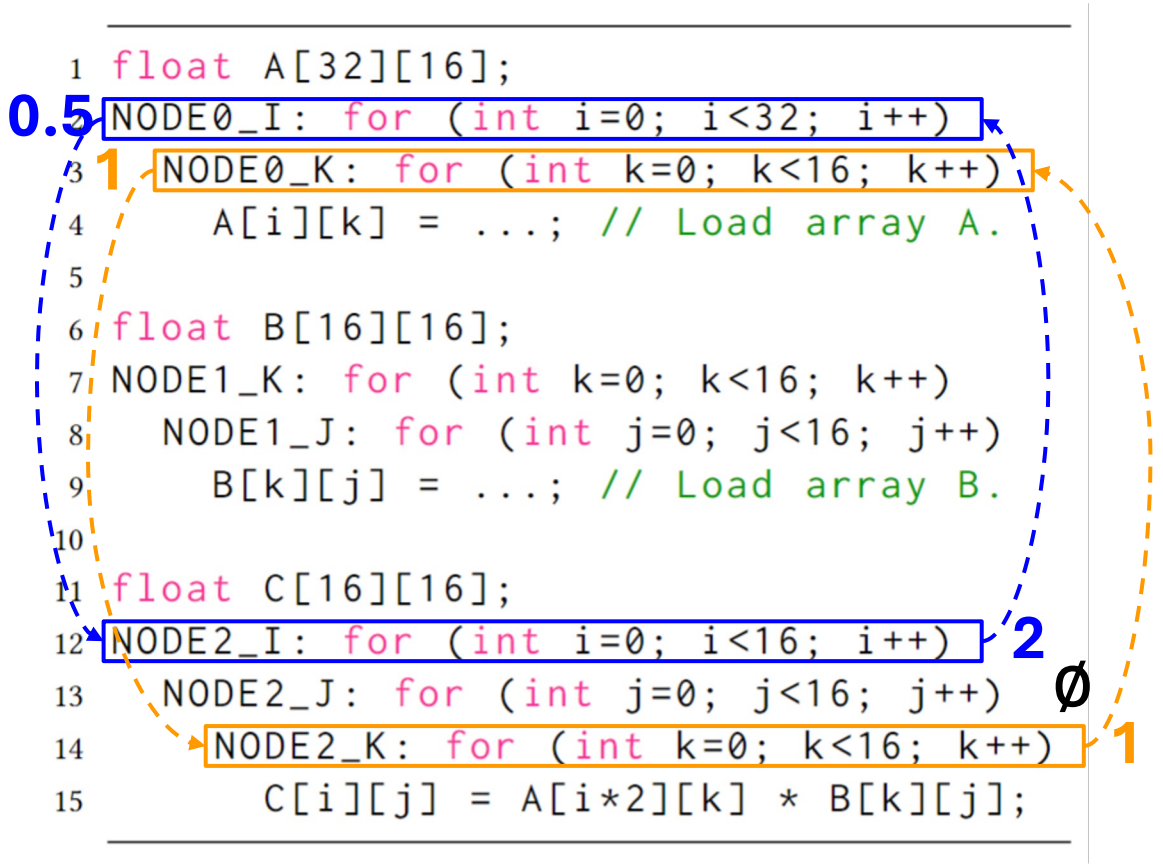


Step (1) Connectedness Analysis

Source	Target	Buffer	Permutation Map		Scaling Map	
			S-to-T	T-to-S	S-to-T	T-to-S
Node0	Node2	A	[0, 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

HIDA design space exploration (Cont.)

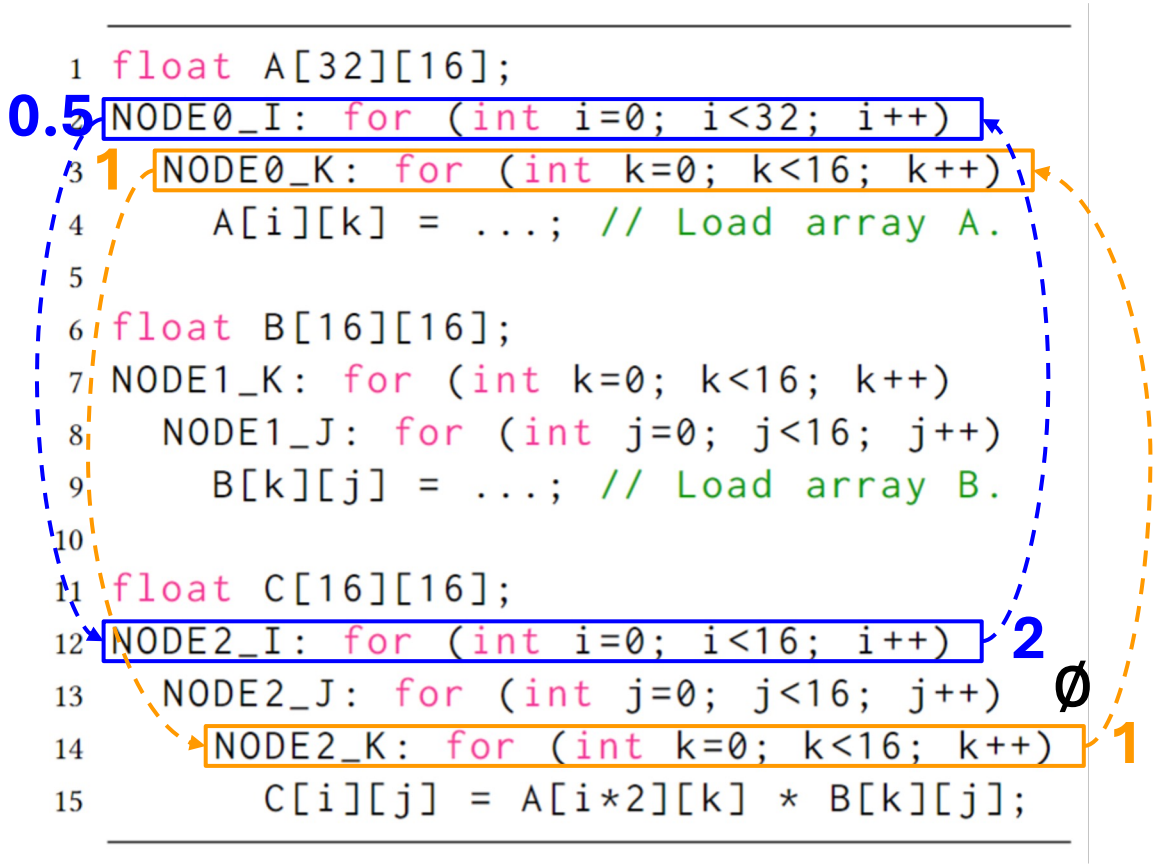


Step (1) Connectedness Analysis

Source	Target	Buffer	Permutation Map		Scaling Map	
			S-to-T	T-to-S	S-to-T	T-to-S
Node0	Node2	A	[0, 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

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, 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 / Linear algebra based
  - Demand normalized and perfect loop with affine memory access indices

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
  - Local 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)
  - 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]

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)
  - Local 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

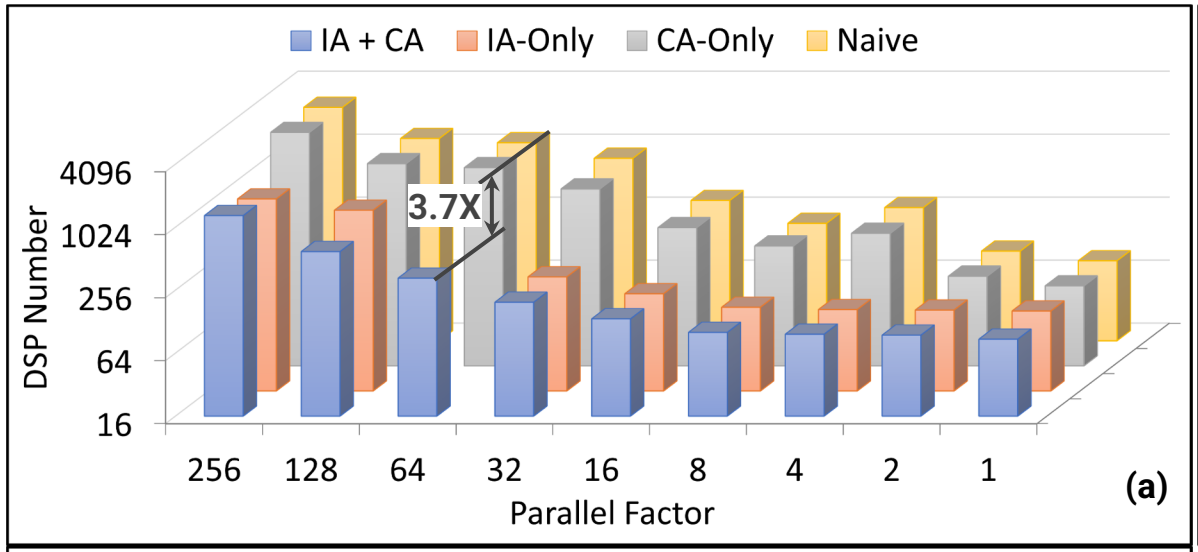
Naive
Local
DSE

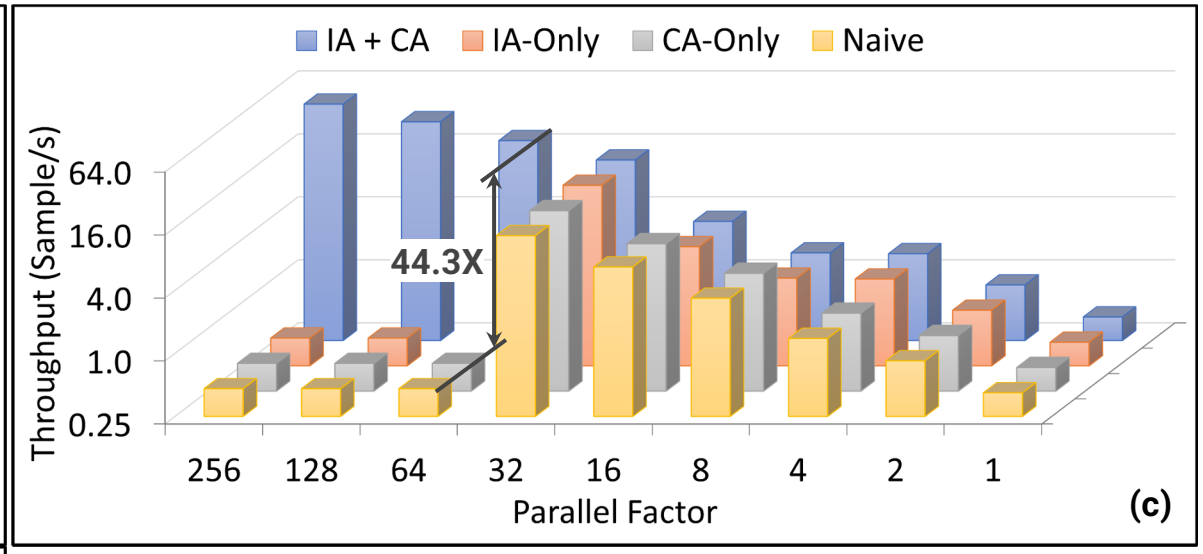
Array	Array Partition Factors				Bank Number				
	IA+CA	IA	CA	Naive	IA+CA	IA	CA	Naive	
A	[8, 1]	[8, 2]	[8, 4]	[8, 8]	8	16	32	64	***8x***
B	[1, 8]	[2, 8]	[4, 8]	[8, 8]	8	16	32	64	***8x***
C	[4, 8]	[4, 8]	[4, 8]	[4, 8]	32	32	32	32	***1x***

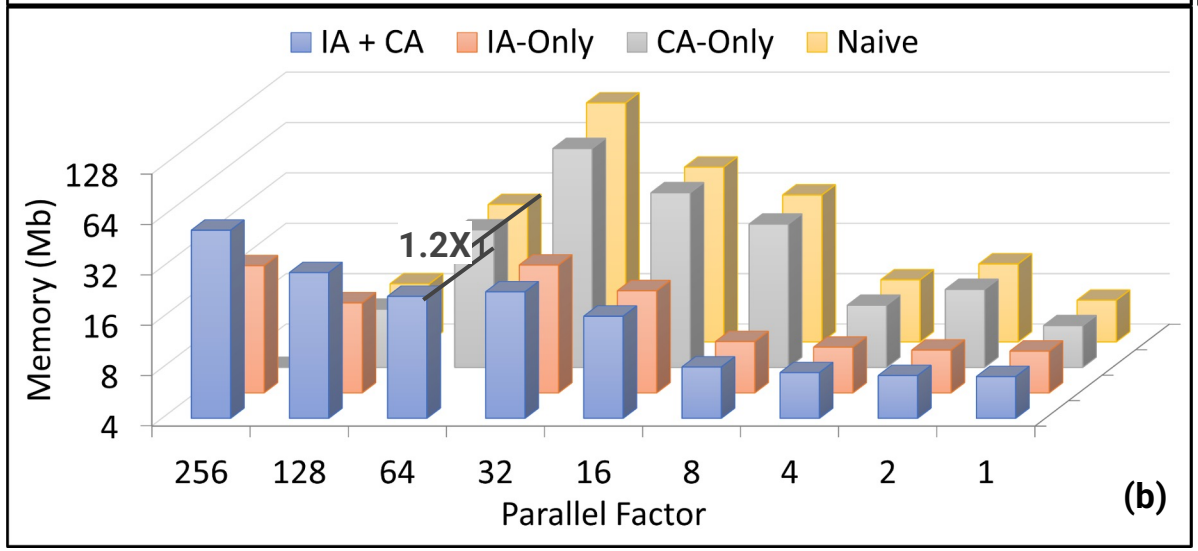
Experimental Results

… on C/C++ and PyTorch benchmarks

ResNet-18 ablation study on HIDA

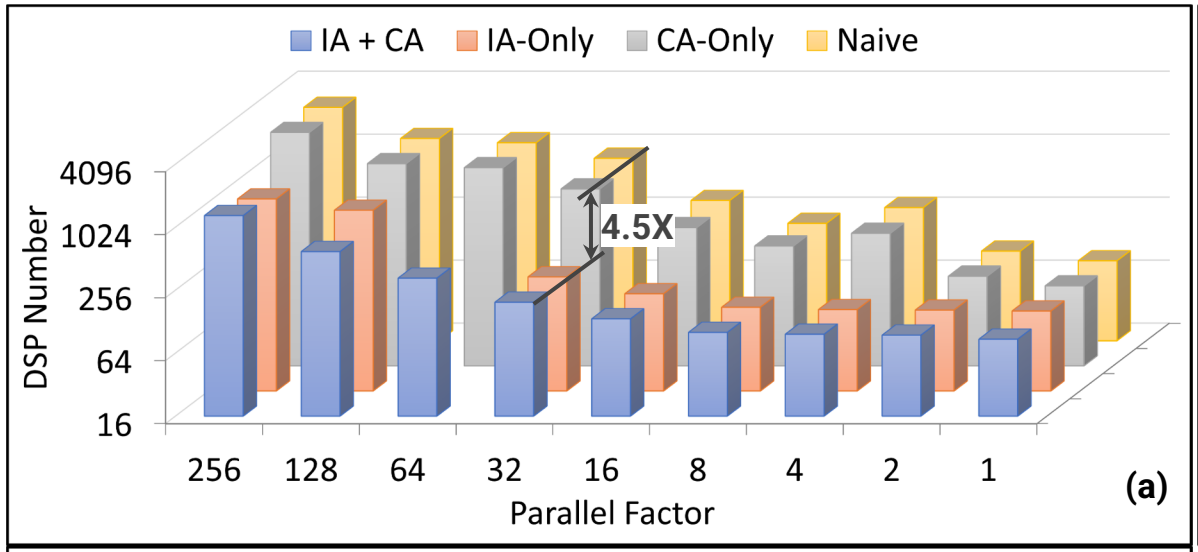


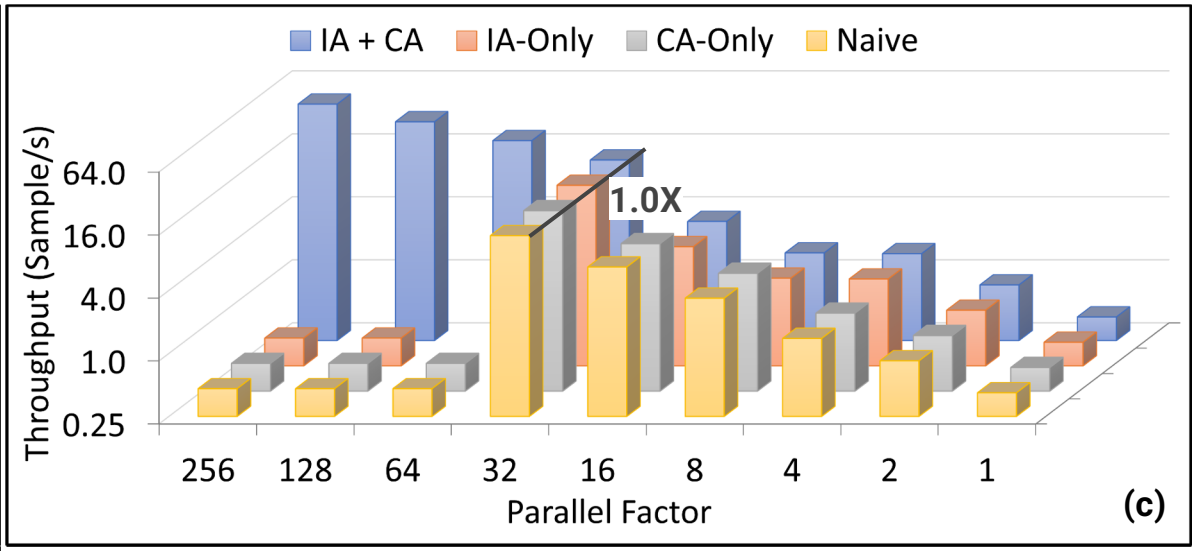


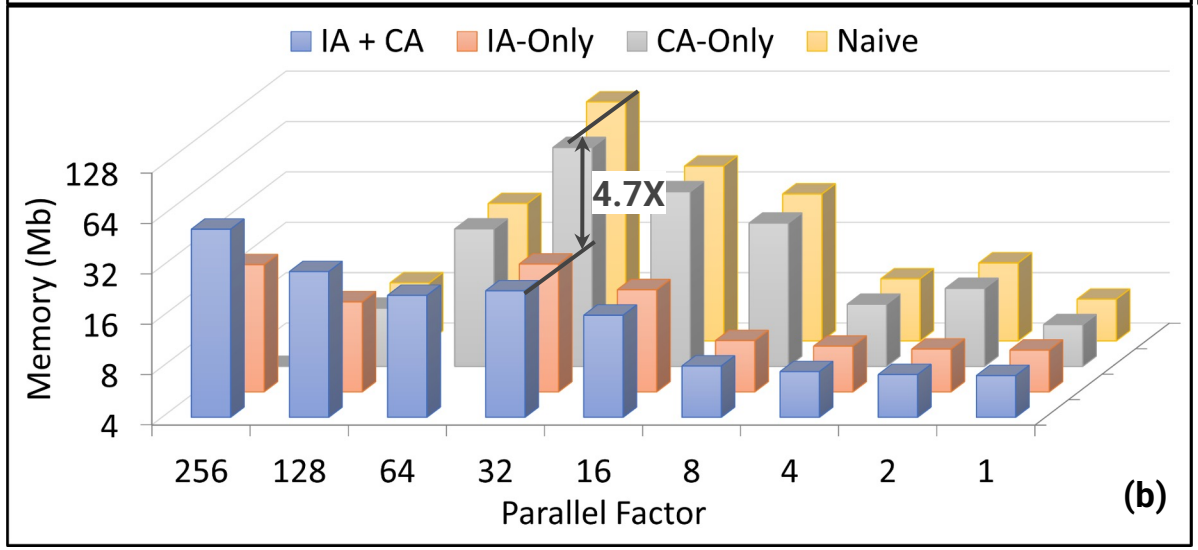


- IA+CA parallelization can determine whether the solution is scalable

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

HIDA results on C++ kernels

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

- We propose a hierarchical dataflow compilation framework, HIDA, with two levels of dataflow representation and optimization
- We propose a connectedness-aware and intensity-aware design space exploration method to systematically parallelize dataflow designs
- Experiments show performance improvements for both C++ kernels and PyTorch models

Open-sourced on GitHub:
https://github.com/UIUC-ChenLab/ScaleHLS-HIDA

Thanks for listening!

Hanchen Ye, Hyegang Jun, Deming Chen
University of Illinois Urbana-Champaign

Apr. 29, 2024