Algorithm-Hardware Co-Design for Edge DNN Deployment and ML for Hardware Design

Qijing Jenny Huang

University of California, Berkeley

Berkeley
UNIVERSITY OF CALIFORNIA

MLArchSys Workshop @ ISCA 2020

Outline

- Motivation
- Codesign for Image Classification
  - Synetgy: Shift-based DiracDeltaNet [FPGA'19]
- Codesign for Object Detection
  - Deformable Convolution [EMC2'19]
- Reinforcement Learning for HLS
  - Autophase: Compiler Phase-Ordering [MLSys'20]
- Conclusion

Embedded Computer Vision

Applications

Robots

Drones

Autonomous Vehicles

Security cameras

Mobile phones

CV Kernels/Tasks

Image Classification

Object Detection

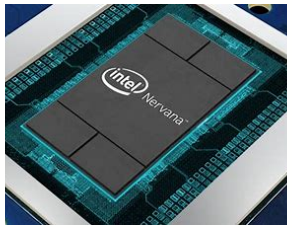
Semantic Segmentation

Embedded Platforms

CPU

GPU

FPGA

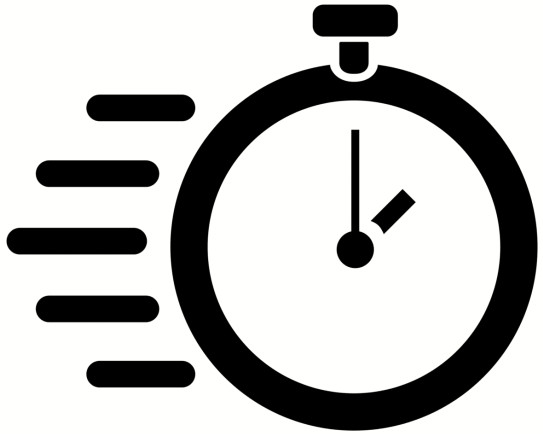


ASIC

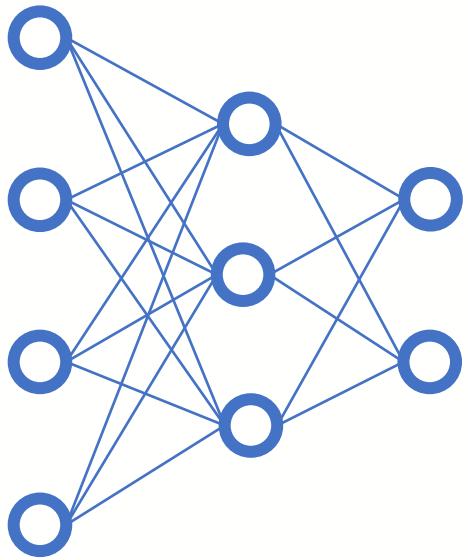
Goals for Embedded CV

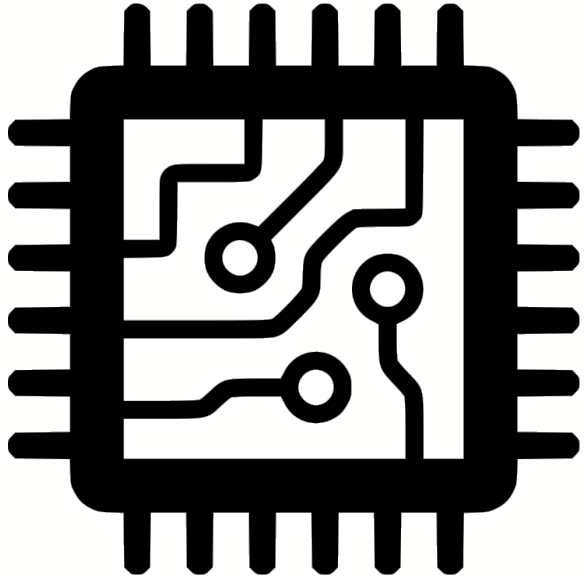
Accuracy

Efficiency



How to improve accuracy and efficiency?





Design better ConvNet

Design better hardware

Motivation

- Why codesign algorithm and hardware?
  - **Inefficient Model Designs** – many CV tasks use large inefficient models and operations solely optimized for accuracy
  - **Limited Hardware Resources** – embedded devices have limited compute resources and a strict energy and power budget
  - **Real-time Requirements** – accelerators must guarantee response within certain time constraints
- Goals: codesign **algorithms** and **accelerators** that *satisfy embedded system constraints and fall on the pareto optimal curve of the accuracy-latency tradeoff.*

Outline

- Motivation
- Codesign for Image Classification
  - Synetgy: Shift-based DiracDeltaNet [FPGA'19]
- Codesign for Object Detection
- Reinforcement Learning for HLS
  - Autophase: Compiler Phase-Ordering [MLSys'20]
- Conclusion

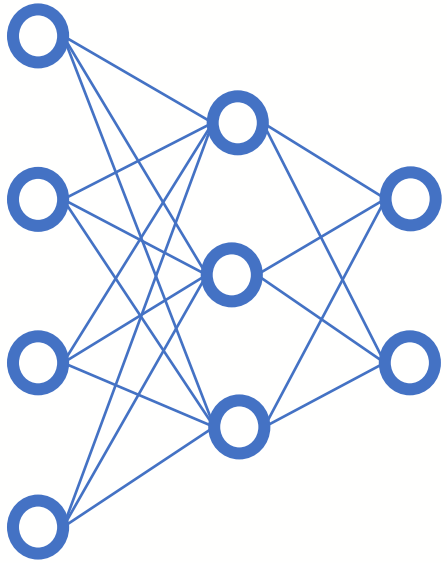
Synetgy: Algorithm-hardware Co-design for ConvNet Accelerators on Embedded FPGAs

Berkeley
UNIVERSITY OF CALIFORNIA

[1] Yang, Yifan, Huang, Qijing, et al. "Synetgy: Algorithm-hardware co-design for convnet accelerators on embedded fpgas." FPGA 2019.

ConvNet Design Strategies

- Strategy 1: Use efficient models
- Strategy 2: Simplify operators
- Strategy 3: Quantize

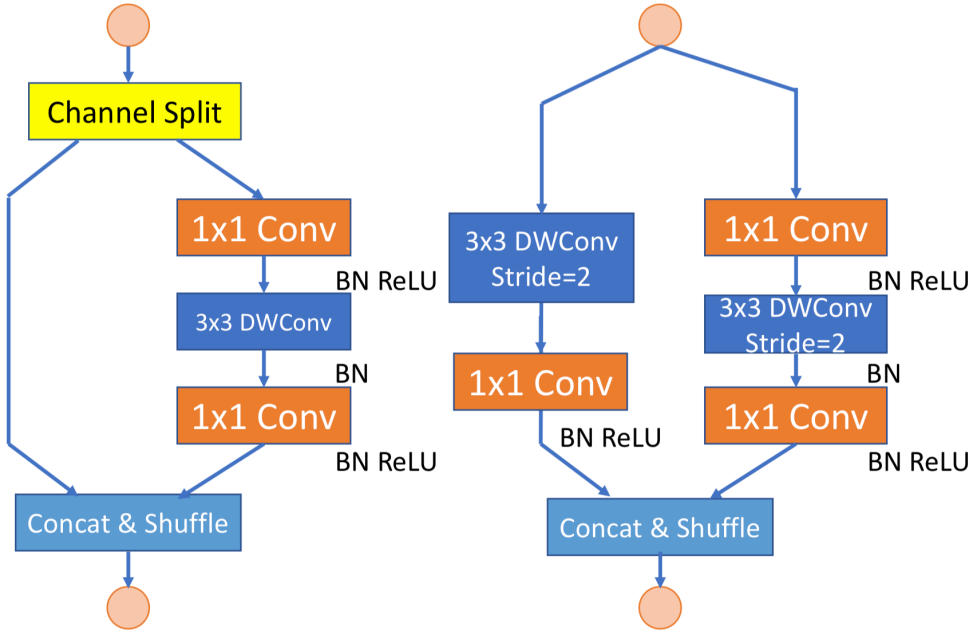


Design better ConvNet

CNN Strategy 1: Use efficient models

- **ShuffleNetV2-1.0**x [1] as our starting point
- Compared to VGG16:
  - 65x fewer OPs
  - 48x fewer parameters
  - Near equal accuracy on ImageNet

	MACs	#Params	Top-1 Acc
ShuffleNetV2-1.0x	146M	2.3M	69.4%
VGG16	15.3G	138M	71.5%

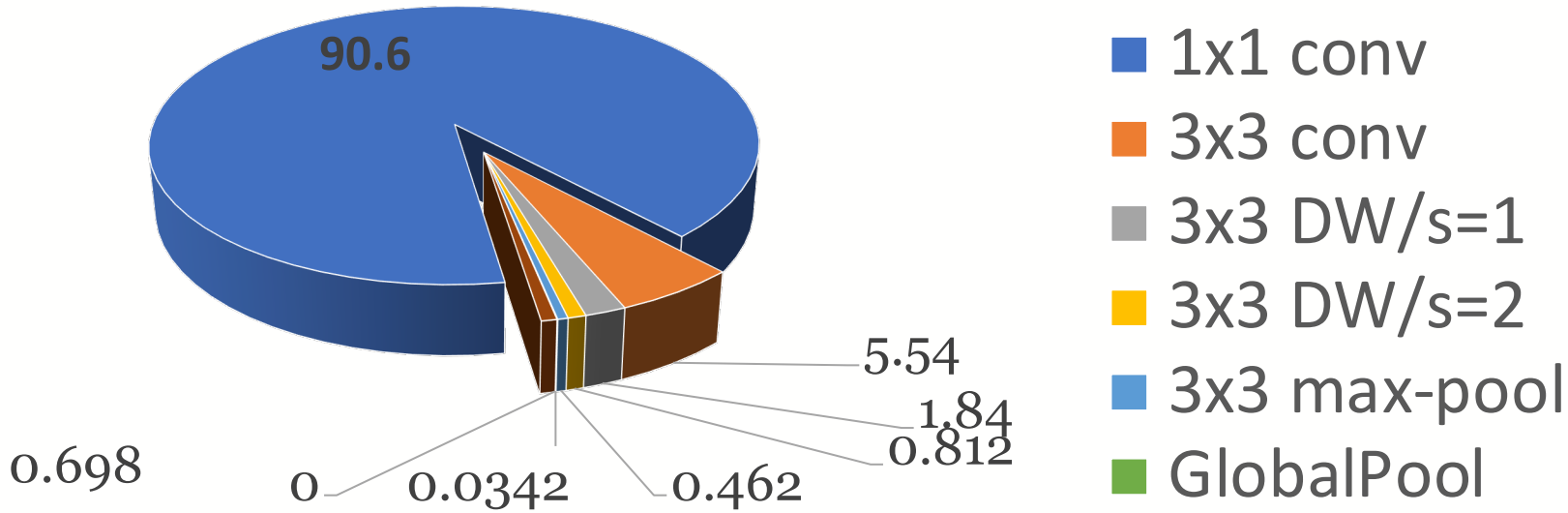


ShuffleNetV2 building blocks

[1] Ma, N., Zhang, X., Zheng, H. and Sun, J. "ShuffleNet V2: Practical Guidelines for Efficient CNN Architecture Design. " ECCV 2018

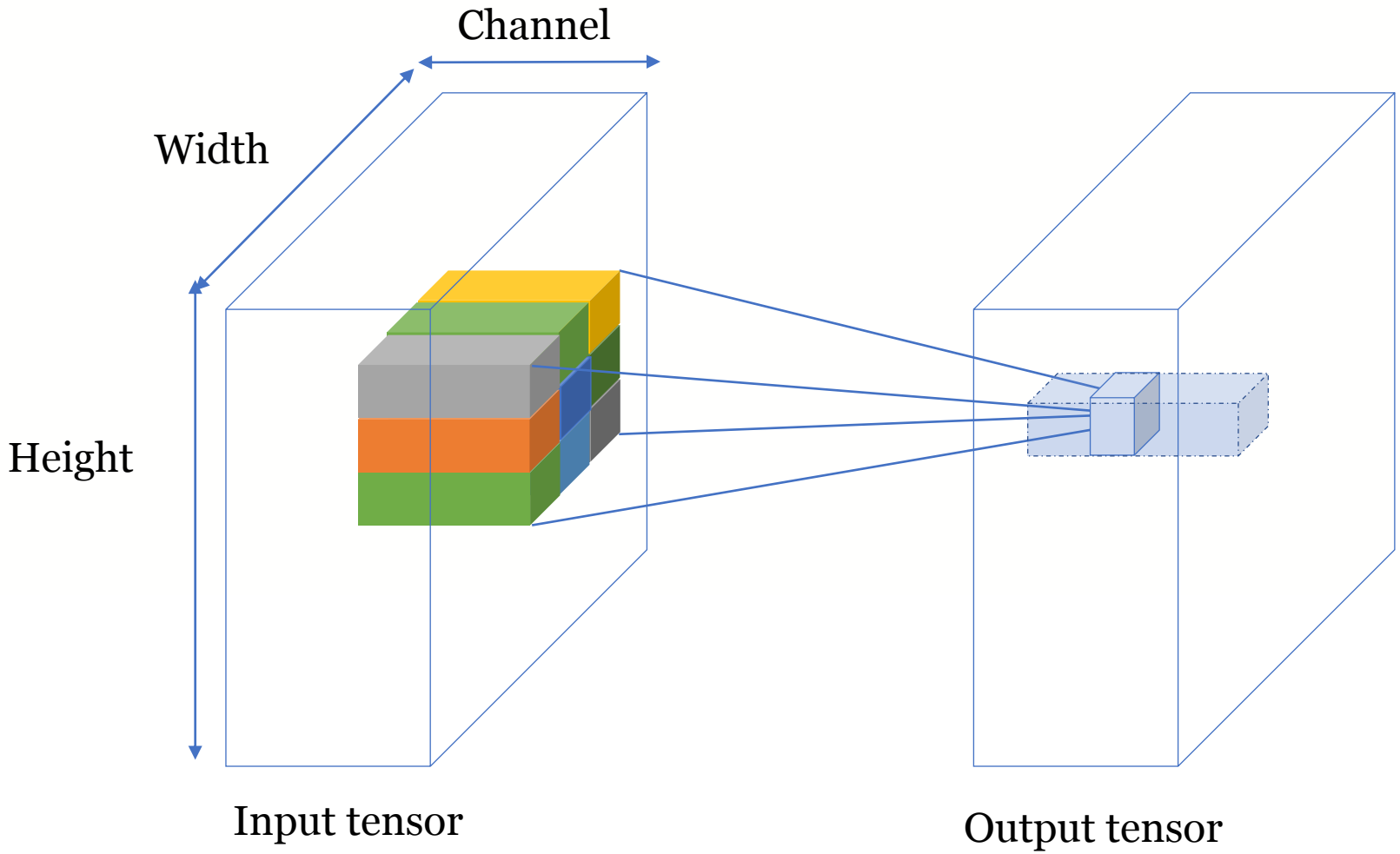
CNN Strategy 2: Simplify operators

- Can we reduce the number of operator types?
- Can we make the operation more hw-friendly?



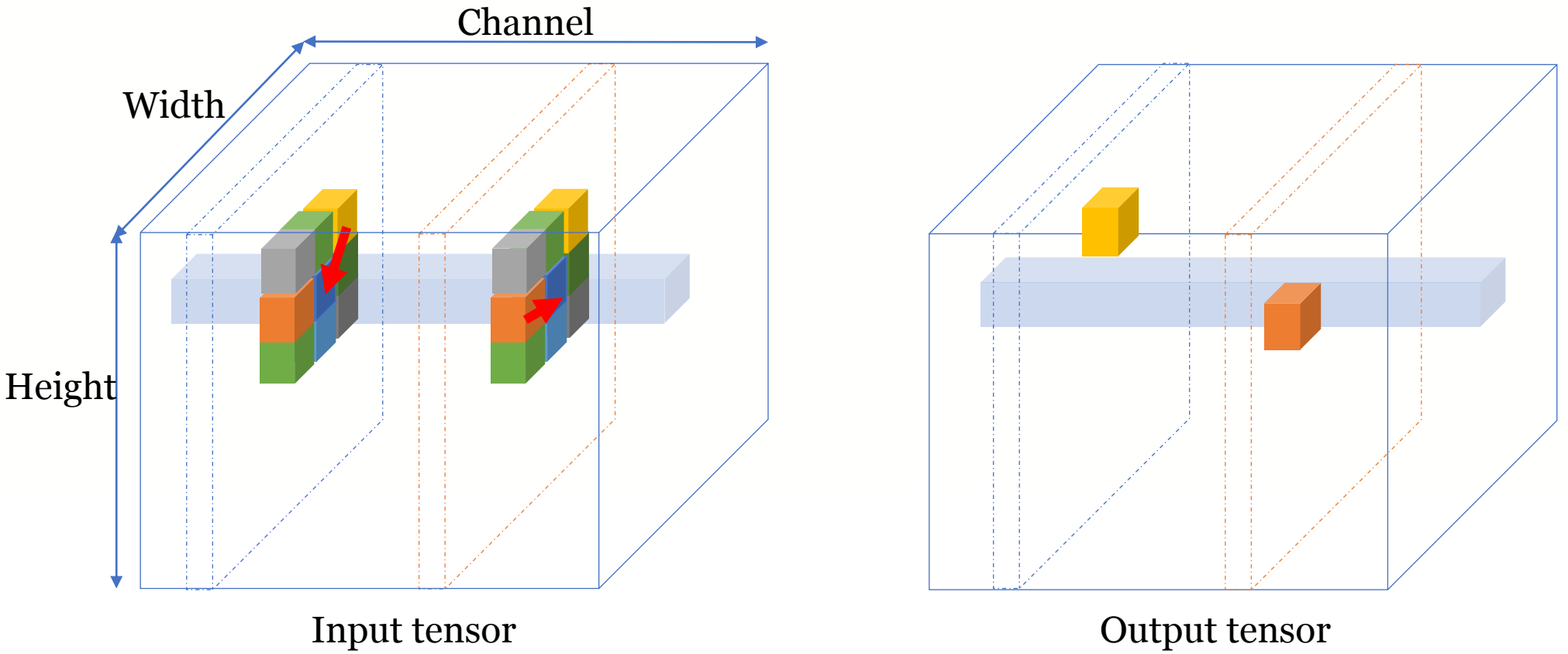
CNN Strategy 2: Simplify operators

- Can we replace 3x3 convolutions?



The Shift Operation

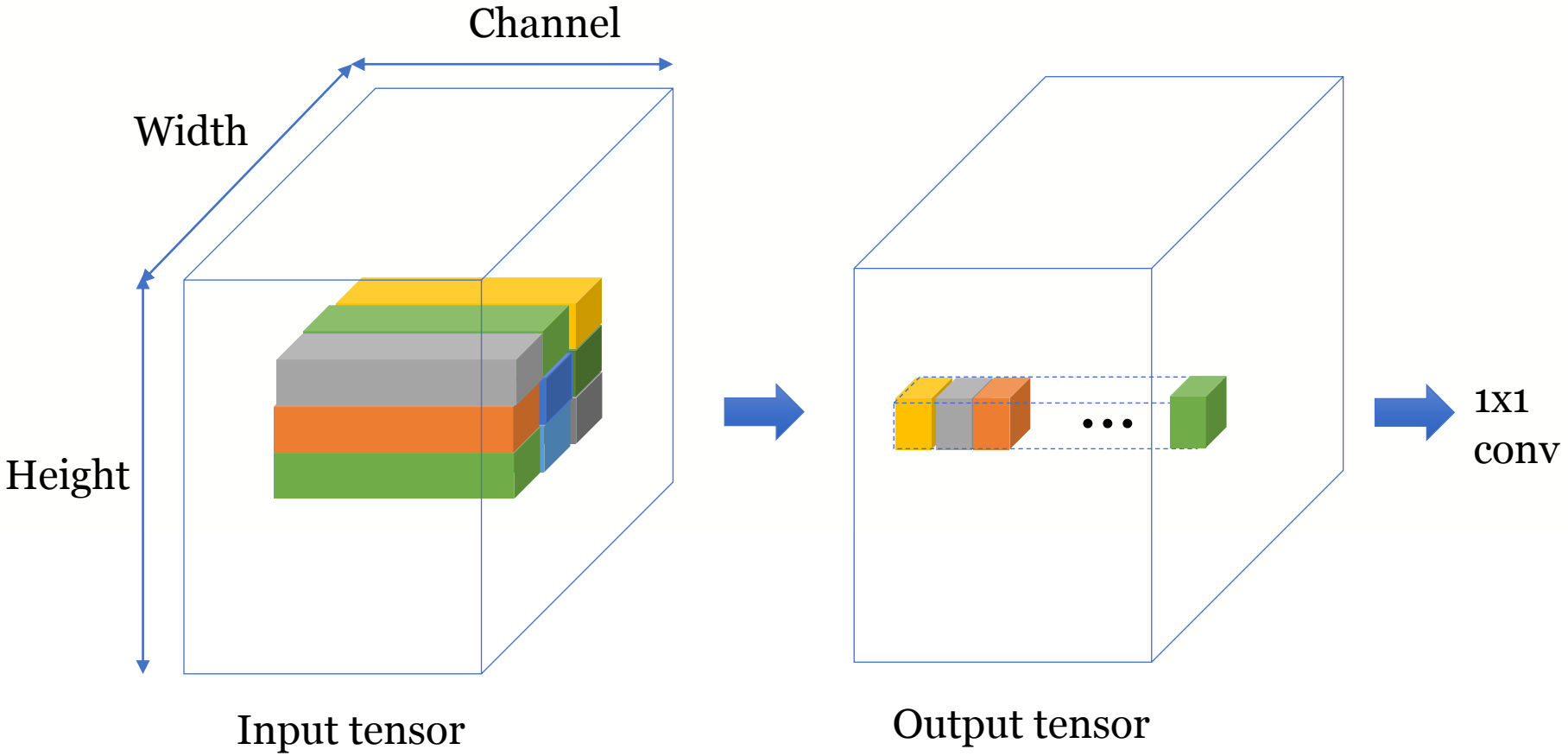
- The shift operation moves a neighboring pixel to the center position



[1] Wu, Bichen, et al. "Shift: A zero flop, zero parameter alternative to spatial convolutions." CVPR 2018.

The Shift Operation

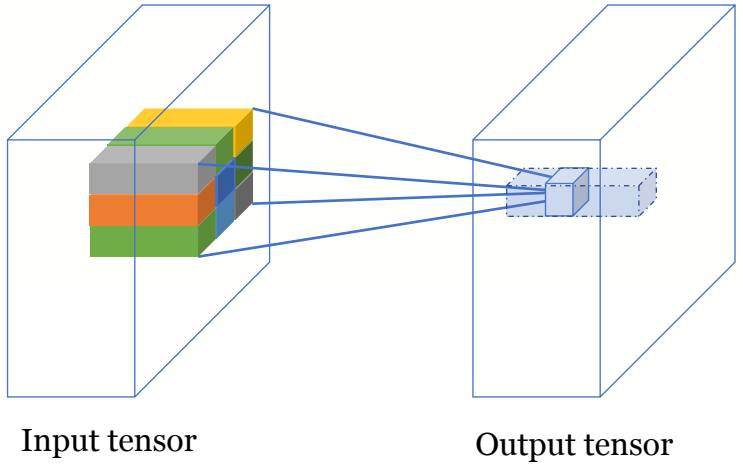
- 1x1 conv aggregates spatial information along the channel dimension

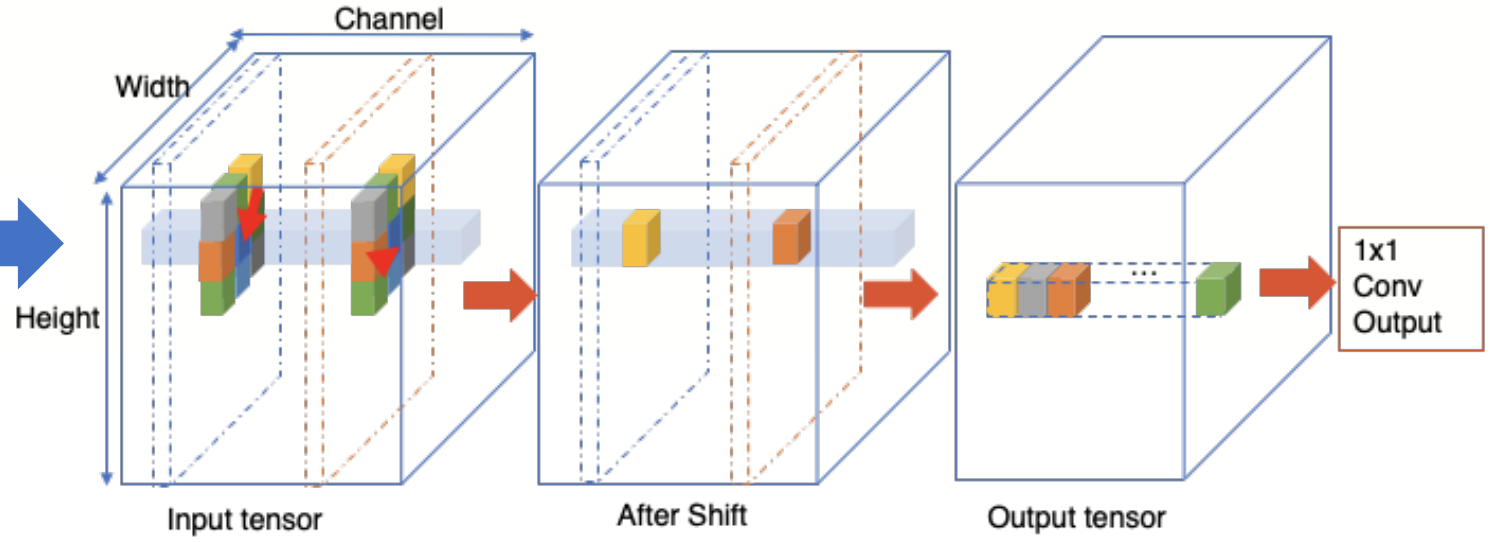


Replace 3 x 3 Conv

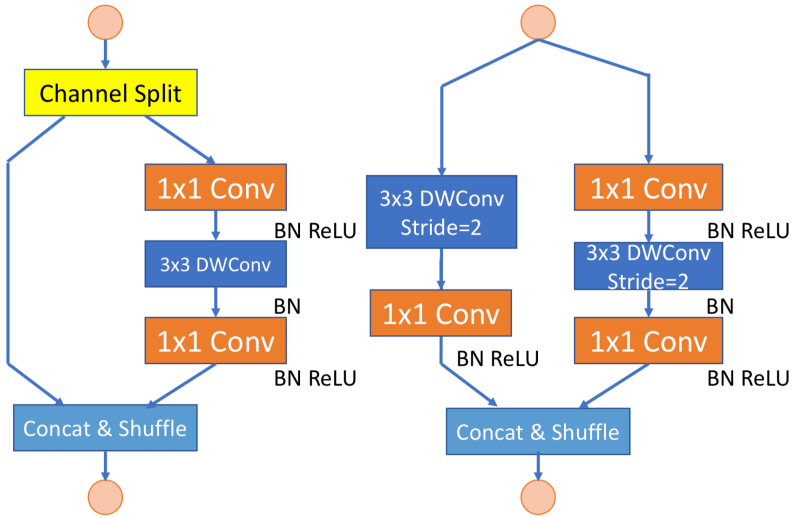
- **3x3 conv:**
  - Aggregates neighboring pixels
  - Mixes channel info

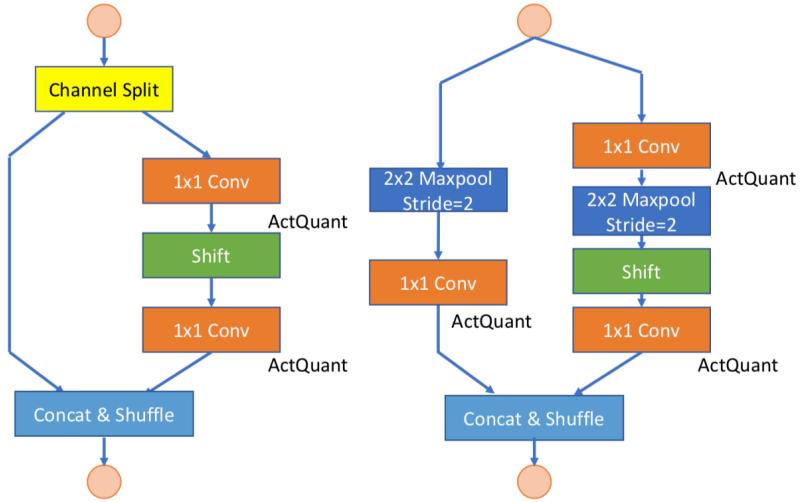
- **shift**: Re-aligns pixels
- **1x1 conv**: Mixes channel info





ShuffleNetV2 -> DiracDeltaNet





- Accuracy (full precision): **69.4%**
- Operators involved:
  - 1x1 convolution
  - 3x3 convolution
  - 3x3 DW convolution
  - 3x3 max pooling
  - Channel split/shuffle/concat

- Accuracy (full precision): **69.7%**
- Operators involved:
  - 1x1 convolution
  - 2x2 max pooling
  - Channel split/shuffle/shift/concat

Berkeley UNIVERSITY OF CALIFORNIA

[1] Ma, N., Zhang, X., Zheng, H. and Sun, J. "ShuffleNet V2: Practical Guidelines for Efficient CNN Architecture Design. " ECCV 2018

CNN Strategy 3: Quantize

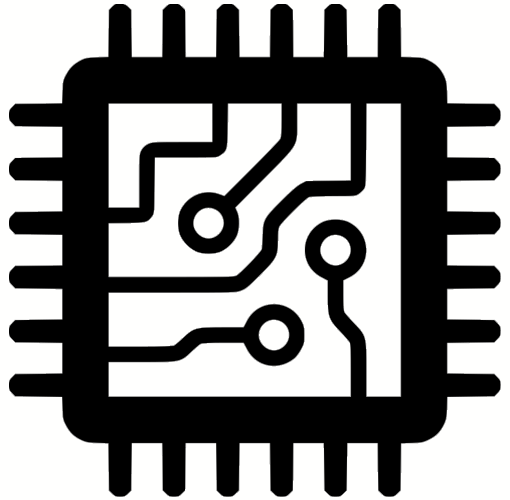
- Quantization has been mostly demonstrated on large networks. Is it effective on the small ones like DiracDeltaNet?
- We used existing quantization methods:
  - DoReFaNet [1] method for weights
  - Modified PACT [2] method for activations
- We achieved 4-bit weight and 4-bit activation precision with competitive accuracy

	Network	Pruning	Precision	Top-1 Acc
[3]	VGG16	Yes	8-8b	67.72%
Ours	DiracDeltaNet	No	4-4b	67.52%

[1] Zhou, S., Wu, Y., Ni, Z., Zhou, X., Wen, H. and Zou, Y. DoReFa-Net: Training Low Bitwidth Convolutional Neural Networks with Low Bitwidth Gradients.
[2] Choi, J., Wang, Z., Venkataramani, S., I-Jen Chuang, P., Srinivasan, V. and Gopalakrishnan, K.PACT: Parameterized Clipping Activation for Quantized Neural Networks.
[3] Guo, K., Han, S., Yao, S., Wang, Y., Xie, Y. and Yang, H. Software-Hardware Codesign for Efficient Neural Network Acceleration. IEEE Micro

Hardware Design Strategies

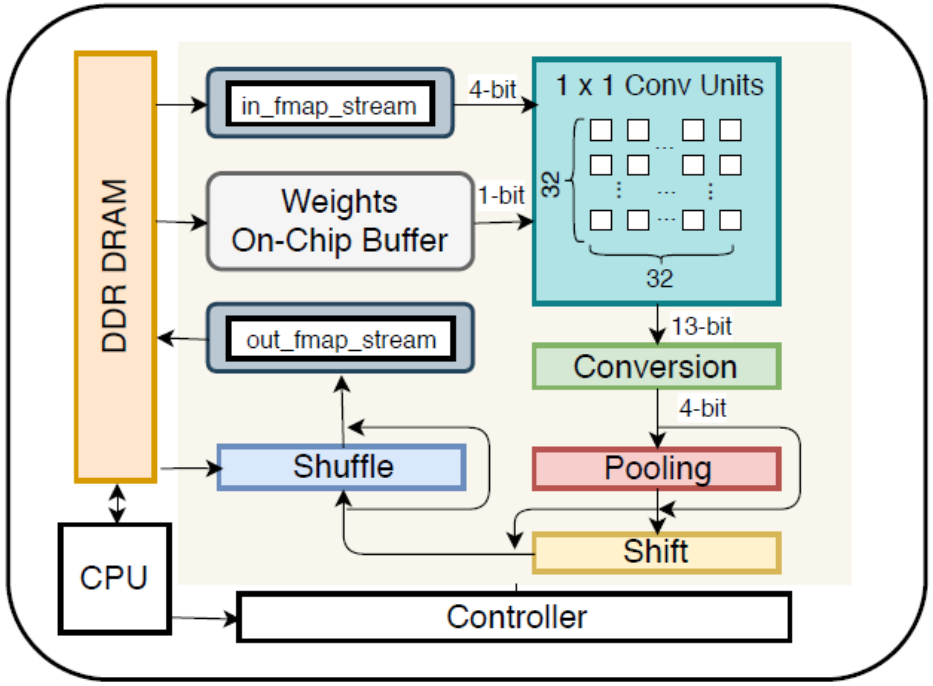
- Strategy 1: Specialize conv engine
- Strategy 2: Use dataflow architecture
- Strategy 3: Merge layers



Design better hardware

HW Strategy 1: Specialize conv engine

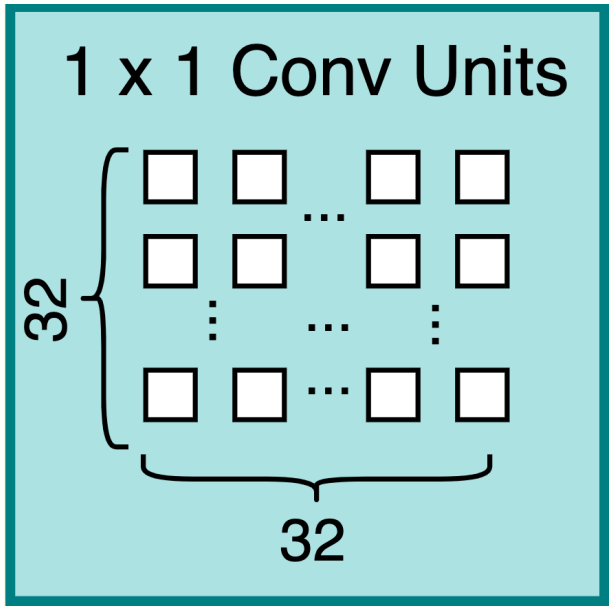
- HW engine supports:
  - 1x1 conv
  - 2x2 max-pooling
  - shift
  - shuffle
- Layer-based design
- Implemented with Vivado HLS and PYNQ



Architecture Diagram

HW Strategy 1: Specialize conv engine

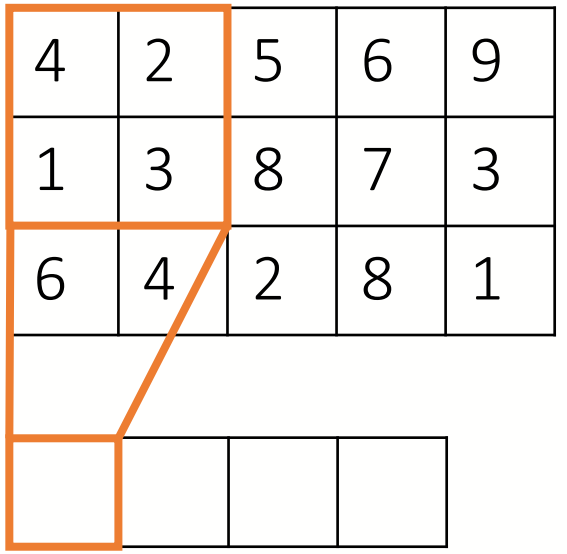
- 1x1 Conv Unit:
  - Supports matrix-vector multiplication
    - 4-bit inputs
    - 4-bit weights
    - 17-bit partial sums
  - Buffers weights and partial sums on-chip
  - Performs 32 x 32 MACs per iteration
  - Each input gets reused *output channel size* times



Strategy 2: Use dataflow architecture

- 1x1 conv
  - No line-buffer
- shift
  - 3x3 sliding window, II=1
- 2x2 max-pooling
  - 2x2 sliding window, II=2

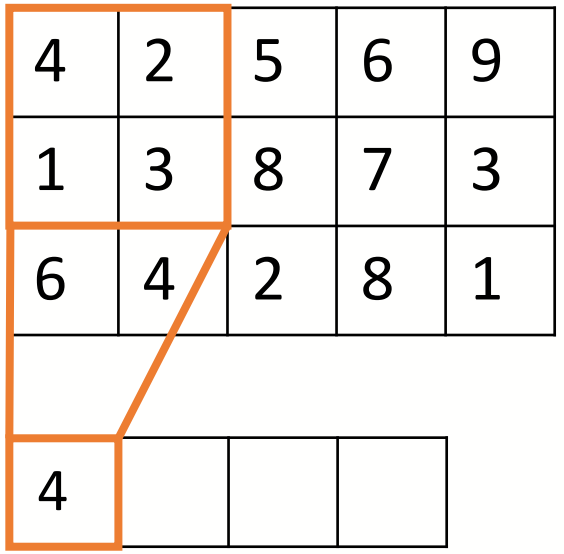
2x2 max-pooling example:



Strategy 2: Use dataflow architecture

- 1x1 conv
  - No line-buffer
- shift
  - 3x3 sliding window, II=1
- 2x2 max-pooling
  - 2x2 sliding window, II=2

2x2 max-pooling example:



Strategy 2: Use dataflow architecture

- 1x1 conv
  - No line-buffer
- shift
  - 3x3 sliding window, II=1
- 2x2 max-pooling
  - 2x2 sliding window, II=2

2x2 max-pooling example:

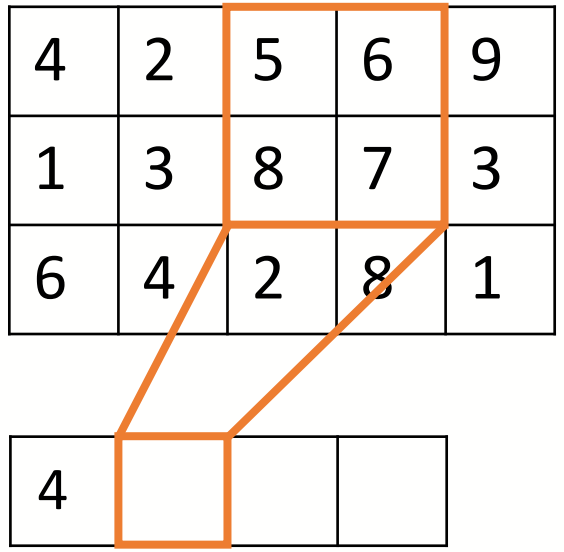
4	2	5	6	9
1	3	8	7	3
6	4	2	8	1

4			

Strategy 2: Use dataflow architecture

- 1x1 conv
  - No line-buffer
- shift
  - 3x3 sliding window, II=1
- 2x2 max-pooling
  - 2x2 sliding window, II=2

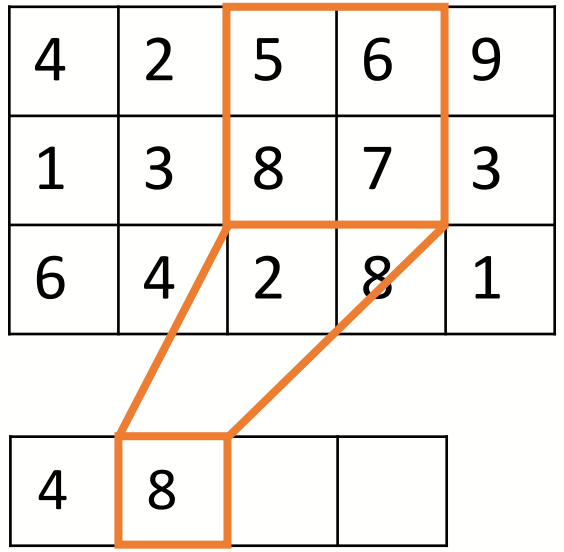
2x2 max-pooling example:



Strategy 2: Use dataflow architecture

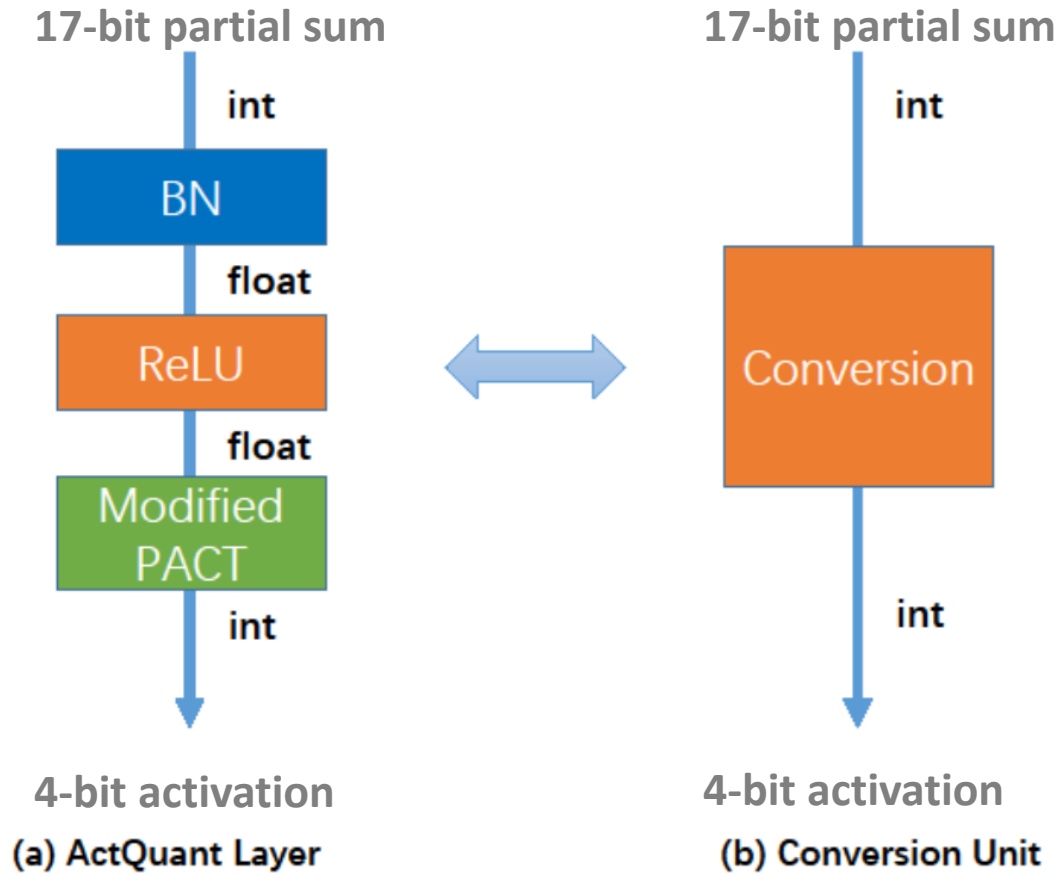
- 1x1 conv
  - No line-buffer
- shift
  - 3x3 sliding window, II=1
- 2x2 max-pooling
  - 2x2 sliding window, II=2

2x2 max-pooling example:



Strategy 3: Merge layers

- Conversion unit includes:
  - Batch Norm
  - ReLU
  - Modified PACT
- It performs 17-bit to 4-bit conversion
- It is implemented with comparators



Comparison with Previous Work

					Energy/
[5]	Stratix V	9.8	68.58%	8-16b	9.626
Ours	Zynq ZU3EG	**66.3**	67.52%	**4-4b**	**0.083**

Algorithm-hardware co-design can achieve both high accuracy (67.5% top-1) and good efficiency (66 FPS) for image classification

- Equal top-1 accuracy
- 11.6x higher framerate
- 6.3x more power efficient

[1] Guo, K., Han, S., Yao, S., Wang, Y., Xie, Y. and Yang, H. Software-Hardware Codesign for Efficient Neural Network Acceleration. IEEE Micro, 37.
[2] Qiu, J., Wang, J., Yao, S., Guo, K., Li, B., Zhou, E., Yu, J., Tang, T., Xu, N., Song, S., Wang, Y. and Yang, H. Going Deeper with Embedded FPGA Platform for Convolutional Neural Network. FPGA 2016.
[3] Suda, N., Chandra, V., Dasika, G., Mohanty, A., Ma, Y., Vrudhula, S.B.K., Seo, J.S. and Cao, Y. "Throughput-Optimized OpenCL-based FPGA Accelerator for Large-Scale Convolutional Neural Networks". FPGA 2016.

Outline

- Motivation
- Codesign for Image Classification
  - Shift-based DiracDeltaNet
- C
- Reinforcement Learning for HLS
  - Compiler Phase-Ordering
- Conclusion

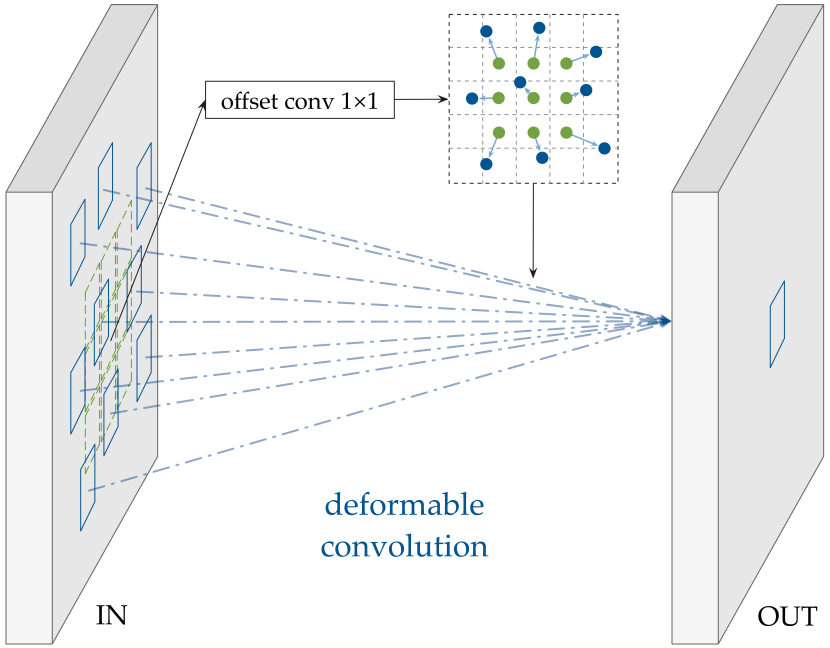
Algorithm-Hardware Co-design for Deformable Convolution

Berkeley
UNIVERSITY OF CALIFORNIA

[1] Huang, Qijing, et al. "Algorithm-hardware Co-design for Deformable Convolution." EMC2: 5th Edition Co-located with NuerIPS'19

Deformable Convolution

- **Deformable Convolution** is an input-adaptive dynamic operation that samples inputs from variable spatial locations



1. Generate offsets
2. Sample from input feature map

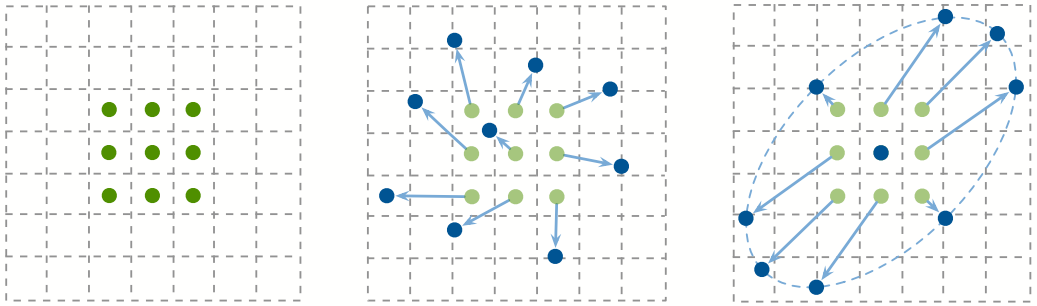
Deformable Convolution

- **Deformable Convolution** is an input-adaptive dynamic operation that samples inputs from variable spatial locations
- Its sampling locations vary with:
  - Different input images
  - Different output pixel locations

Sampling Locations (in red) for Different Output Pixels (in green)

Deformable Convolution

- **Deformable Convolution** is an input-adaptive dynamic operation that samples inputs from variable spatial locations
- Its sampling locations vary with:
  - Different input images
  - Different output pixel locations
- It captures the spatial variance of objects with different:
  - Scales
  - Aspect Ratios
  - Rotation Angles



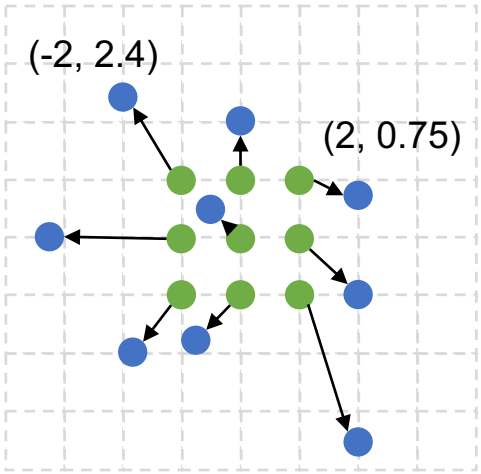
Variable Receptive Fields

Deformable Convolution

- **Deformable Convolution** is an input-adaptive dynamic operation that samples inputs from variable spatial locations
- Its sampling locations vary with:
  - Different input images
  - Different output pixel locations
- It captures the spatial variance of objects with different:
  - Scales
  - Aspect Ratios
  - Rotation Angles
- Challenges:
  - Additional compute and memory requirements for offset generation
  - Irregular input-dependent memory access patterns
    - Not friendly for dataflows that leverage the spatial reuse

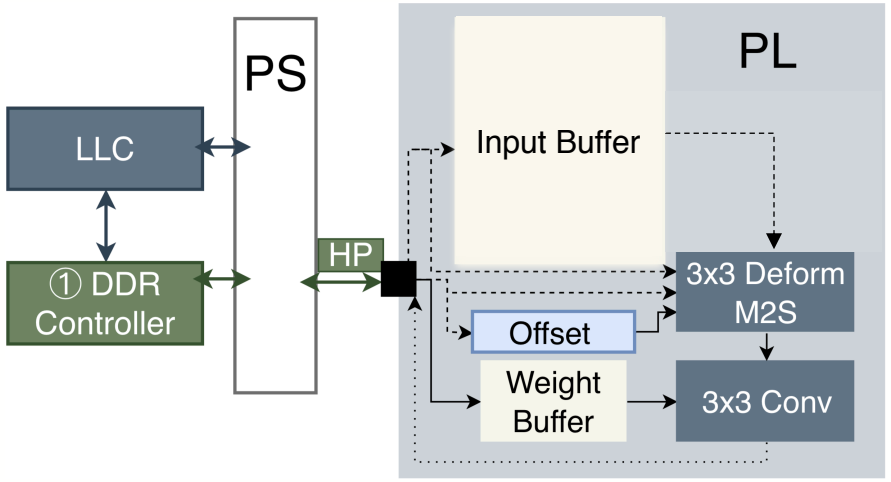
Operation Codesign

Algorithm Modification:



0. Original Deformable

Hardware Optimization:

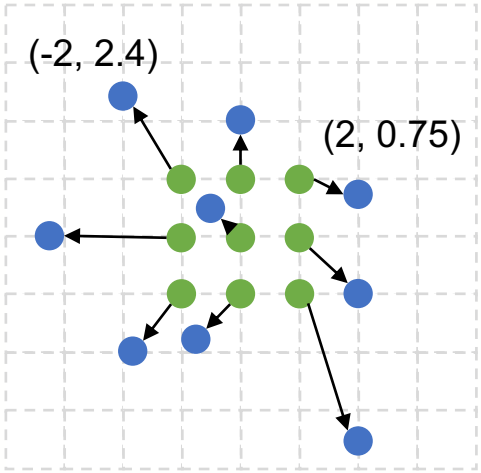


- **Preloads weights to on-chip buffer**
- **Loads input and offsets directly from DRAM**

[1] Accuracy for Object Detection on Pascal VOC

Operation Codesign

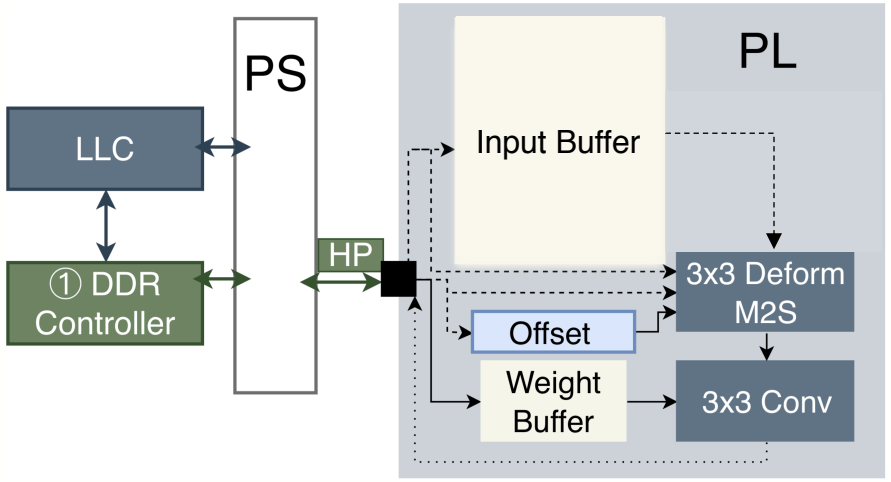
Algorithm Modification:



1. Depthwise Deformable

Accuracy [1](AP): **42.9**

Hardware Optimization:

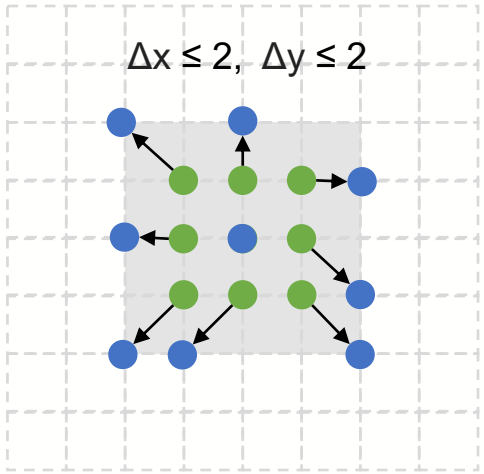


- **Reduce the total MACs**

[1] Accuracy for Object Detection on Pascal VOC

Operation Codesign

Algorithm Modification:

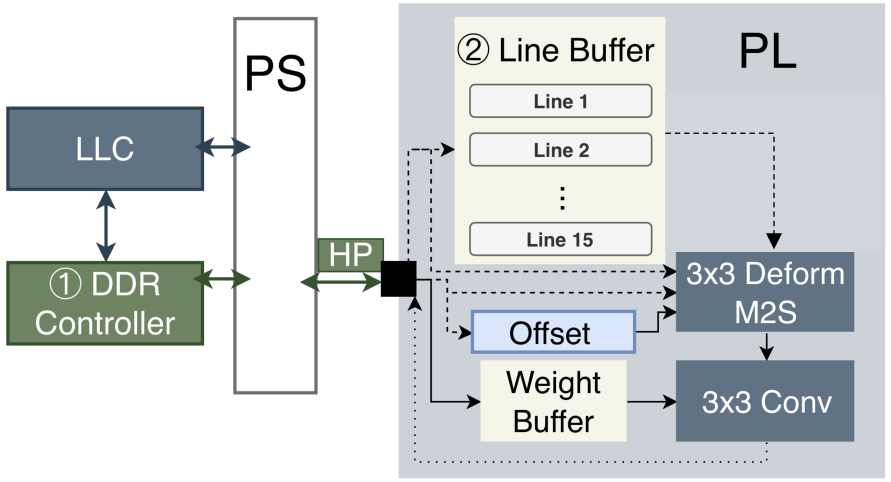


2. Bounded Range

↓ **1.9**

Accuracy [1](AP): **41.0**

Hardware Optimization:

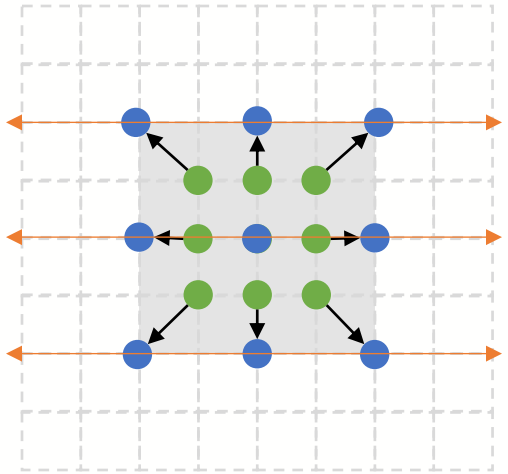


- **Buffers inputs in the on-chip line buffer to allow spatial reuse**

[1] Accuracy for Object Detection on Pascal VOC

Operation Codesign

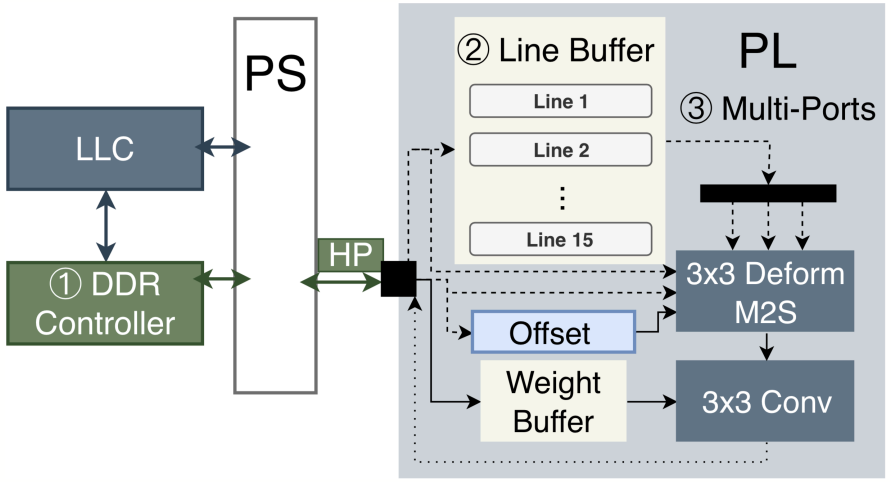
Algorithm Modification:



3. Rectangular Shape

Accuracy [1](AP): **41.1** ↑ **0.1**

Hardware Optimization:



- **Improves on-chip memory bandwidth**

[1] Accuracy for Object Detection on Pascal VOC

Ablation Study

Object Detection Accuracies

Operation	Depthwise	Bound	Square	VOC			COCO					
				AP	AP50	AP75	AP	AP50	AP75	APs	APm	APl
3 × 3				39.2	60.8	41.2	21.4	36.5	21.5	7.3	24.1	33.0
3 × 3	✓			39.1	60.9	40.9	19.8	34.3	19.7	6.3	22.6	31.5
5 × 5	✓			40.6	62.4	42.6	21.3	36.4	21.3	6.7	23.7	34.2
7 × 7	✓			41.9	63.8	43.8	21.7	37.2	21.5	6.9	24.0	35.2
9 × 9	✓			42.3	64.8	44.3						
deform	✓			42.9	64.4	45.7	23.0	38.4	23.3	6.9	24.4	37.8
deform	✓	✓		41.0	63.0	42.9	21.3	36.4	21.1	7.2	23.6	34.4
deform	✓	✓	✓	41.1	63.1	43.7	21.5	36.8	21.5	6.5	23.7	34.8

5x less compute

- 3x3 deformable convolution is more efficient than large convolution kernels
- The accuracy of the codesigned deformable convolution (1.8 AP difference on Pascal VoC and 1.5 AP difference on COCO)

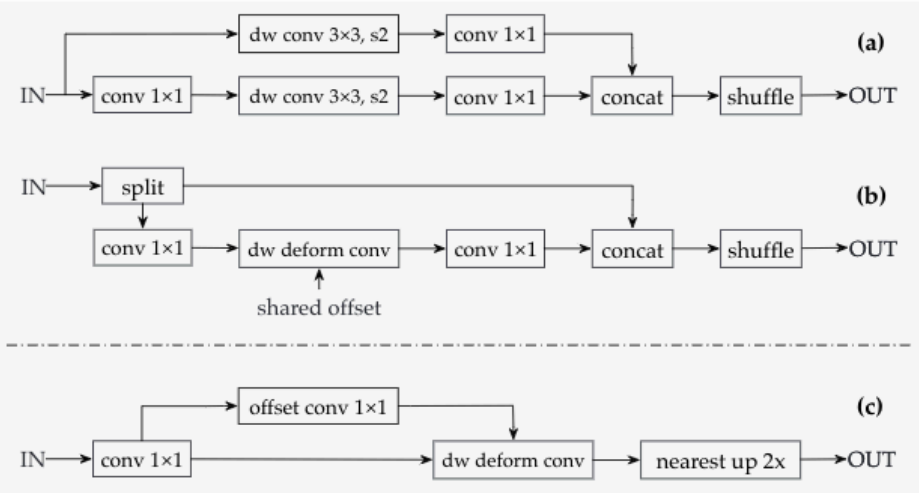
Ablation Study

Hardware Performance

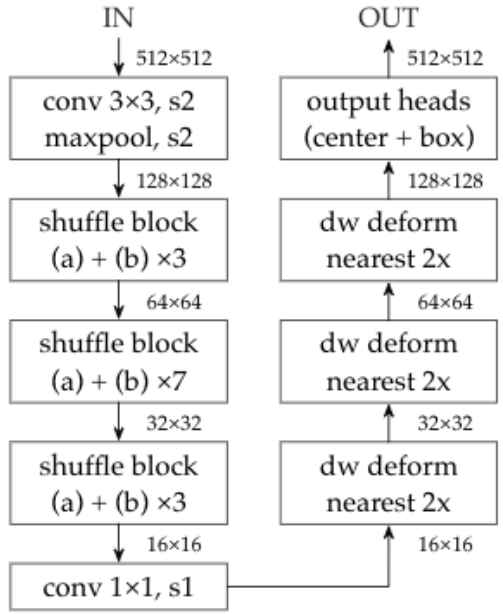
Operation	Original	Deformable	Bound (buffered)	Square (multi-ported)	Without LLC		With LLC	
					Latency (ms)	GOPs	Latency (ms)	GOPs
Full 3×3 Conv	✓				43.1	112.0	41.6	116.2
		✓			59.0	81.8	42.7	113.1
		✓	✓		43.4	111.5	41.8	115.5
		✓	✓	✓	43.4	111.5	41.8	115.6
Depthwise 3×3 Conv	✓				1.9	9.7	2.0	9.6
		✓			20.5	0.9	17.8	1.1
		✓	✓		3.0	6.2	3.4	5.5
		✓	✓	✓	2.1	9.2	2.3	8.2

- Our algorithm-hardware co-design methodology for the deformable convolution achieves a **1.36×** and **9.76×** speedup respectively for the *full* deformable convolution and *depthwise* deformable convolution on FPGA

DNN Architecture



Building Blocks



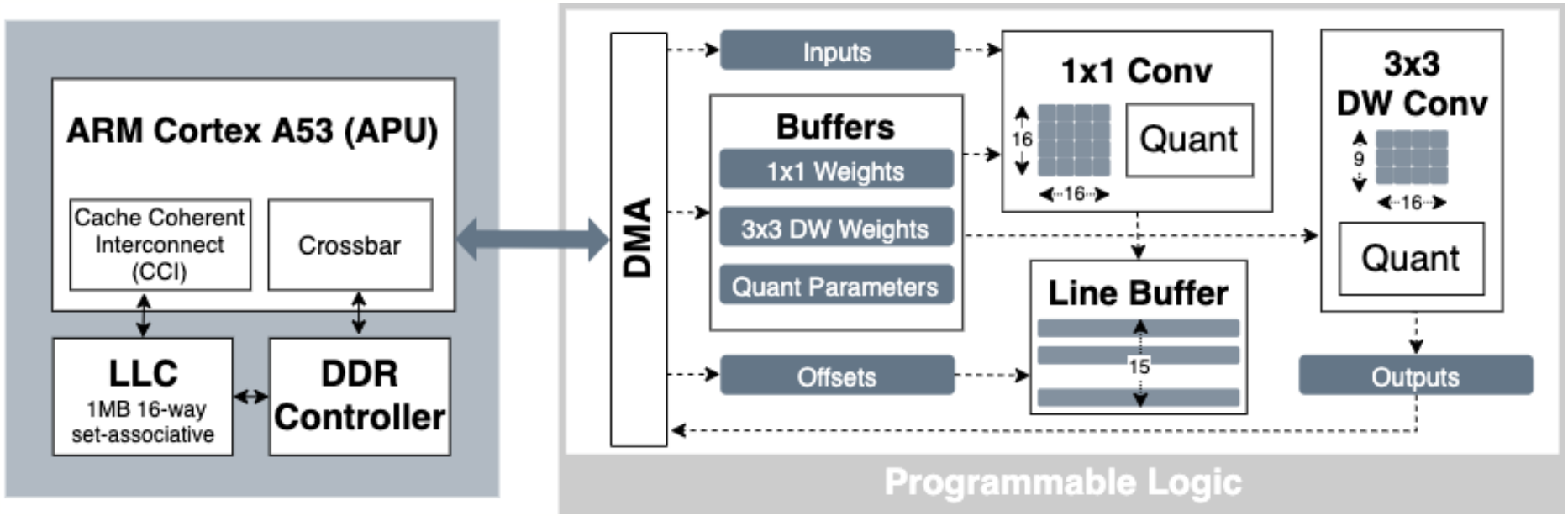
ShuffleNetV2 + CenterNet*

- Simple building blocks to reduce the hardware complexity
- Anchor-free detection system to reduce the postprocessing overhead for Non Maximum Suppression (NMS)

Berkeley
UNIVERSITY OF CALIFORNIA

[1] Zhou, Xingyi, Dequan Wang, and Philipp Krähenbühl. "Objects as points." *arXiv preprint arXiv:1904.07850* (2019).

Hardware Accelerator



- Specialized engine for efficiency
- Weight buffers and line buffers for maximum data reuse

Results

Object Detection Accuracies

Detector	Weights	Activations	Model Size	MACs	AP50
Tiny-YOLO	32-bit	32-bit	60.5 MB	3.49 G	57.1
CoDeformNet 1×	32-bit	32-bit	6.06 MB	1.14 G	64.6

Codesign achieves both high accuracy (55.1 mAP) and real-time inference (26.9 FPS) for object detection on embedded FPGAs

Ha

	Platform	Framerate (fps)	Test Dataset	Precision	Accuracy
Finn-R [2] [26]	Zynq XCZU3EG (Ultra96)	16	VOC07	1-3b	mAP(50.1)
Ours (256 × 256)	Zynq XCZU3EG (Ultra96)	26.9	VOC07	4-8b	mAP(55.1)

- Our design achieves both higher framerate and higher accuracy

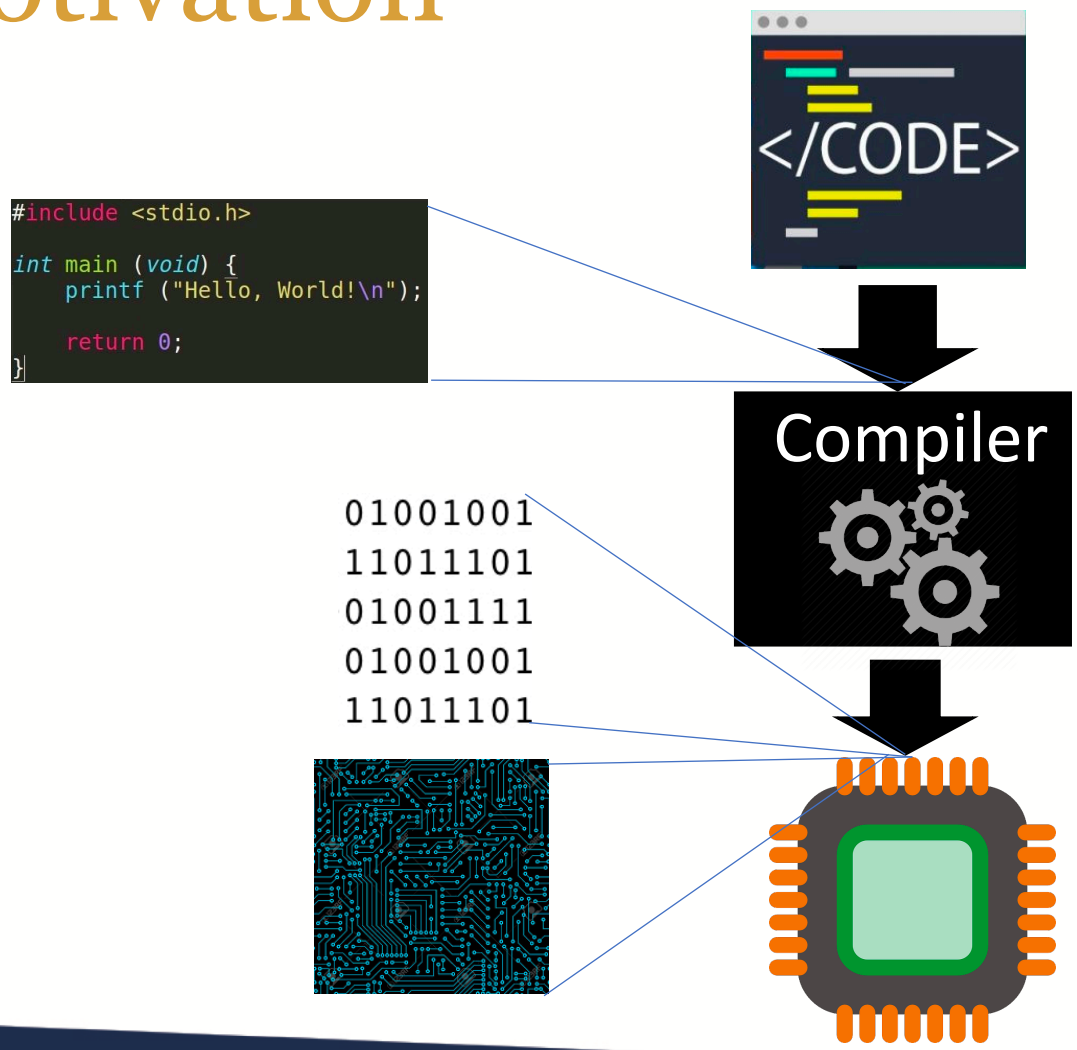
Outline

- Motivation
- Codesign for Image Classification
  - Shift-based DiracDeltaNet
- Co
- Reinforcement Learning for HLS
  - Compiler Phase-Ordering
- Conclusion

AutoPhase: Juggling HLS Phase Orderings in Random Forests with Deep Reinforcement Learning

[1] Huang, Qijing*, Ameer Haj-Ali*. "AutoPhase: Juggling HLS Phase Orderings in Random Forests with Deep Reinforcement Learning". MLSys 2020

Motivation



For years, compiler optimizations rely on **heuristics** and **hand engineering**

Compiler Phase Ordering

clang program.c -flag1 -flag2 …

Which passes?
In which order?
More than 2^{247} possibilities …

NP-Hard

[1] Touati et al., "On the decidability of phase ordering problem in optimizing compilation," *CD-ROM* 2006

Compiler Phase Ordering

Example:

Normalizing a Vector

```
for(int i=0; i<n; i++) {
    out[i] = in[i] / mag(in, n);
}
```

Pass #1: -licm (loop-invariant code motion)

```
double precompute = mag(in, n);
for(int i=0; i<n; i++) {
    out[i] = in[i] / precompute;
}
```

Pass #2: -inline

```
double precompute, sum = 0;
for(int i=0; i<n; i++){
    sum += A[i] * A[i];
}
precompute = sqrt(sum);
for(int i=0; i<n; i++) {
    out[i] = in[i] / precompute;
}
```

$\Theta(n)$

Compiler Phase Ordering

Example:

Normalizing a Vector

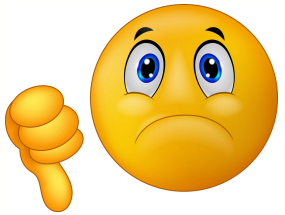
```
for(int i=0; i<n; i++) {
    out[i] = in[i] / mag(in, n);
}
```

Pass #1: -inline

```
for(int i=0; i<n; i++) {
    double sum = 0;
    for(int j=0; j<n; j++){
        sum += A[j] * A[j];
    }
    out[i] = in[i] / sqrt(sum);
```

Pass #2: -licm (loop-invariant code motion)

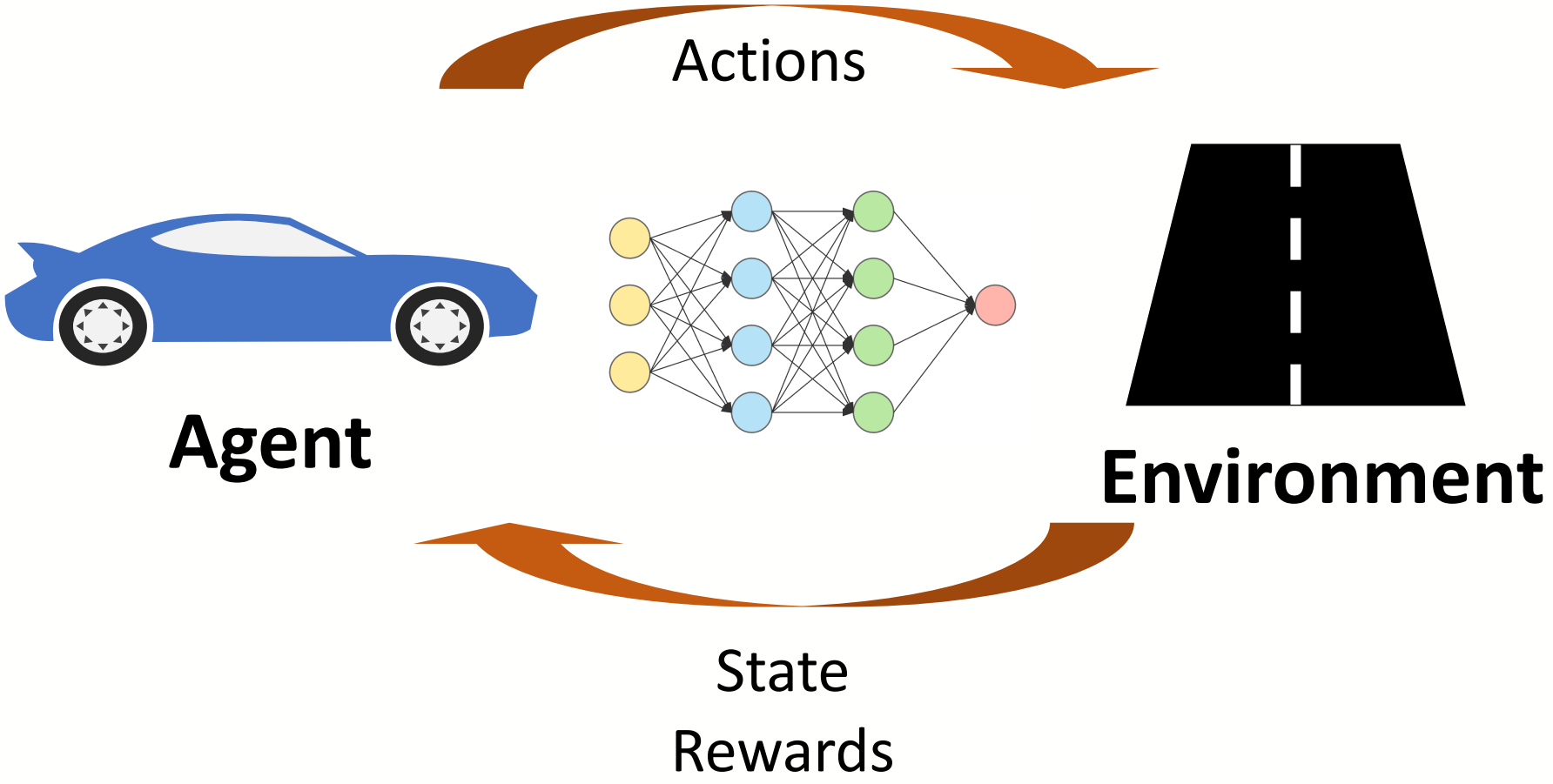
```
double sum;
for(int i=0; i<n; i++) {
    sum = 0;
    for(int j=0; j<n; j++){
        sum += A[j] * A[j];
    }
    out[i] = in[i] / sqrt(sum);}
```



$\cancel{\Theta(n)}$

$\Theta(n^2)$

Deep Reinforcement Learning



Deep Reinforcement Learning

Decision(**Action**): Gas

Decision(**Action**): Brake

Deep Reinforcement Learning

Decision (**Action**):
- Gas
- Brake
- Steering

Destination

Fee (**Reward**): $100

GPS Location (**State**):
37.87631055, -122.2388

Deep Reinforcement Learning

Decision (**Action**):
- Gas
- Brake
- Steering

Destination

Fee (**Reward**): $100

GPS Location (**State**):
37.87631055, -122.2388

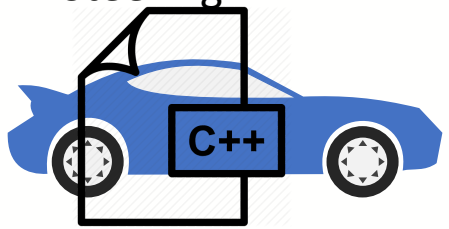
A Tree

Fee (**Reward**): -$500

GPS Location (**State**):
-30.43131384, 73.1693

Deep Reinforcement Learning

Decision (**Action**):
- Gas
- Brake
- Steering



Source Code

Action: Optimization Pass #
(e.g. -licm, -inline, -sroa, mem2reg)

Destination

Fast Code

Fee (**Reward**): $100

GPS Location (**State**):
37.87631055, -122.2388

Reward:
Cycle Count Improvement

State: a) Program Features
b) Applied Passes

Slow Code
A Tree

Fee (**Reward**): -$500

GPS Location (**State**):
-30.43131384, 73.1693

Generating the States

1. **Program Features:**
   - 56 static features (# of instructions, # of loops, ...)
   - An LLVM analysis pass is built

2. **Histogram of Applied Passes:**
   - 45 optimization passes
   - Once pass ***i*** is applied, ***Histogram[i]++***
     (***Histogram:*** a vector of length 45, ***i:*** index of a pass)

Generating the Rewards

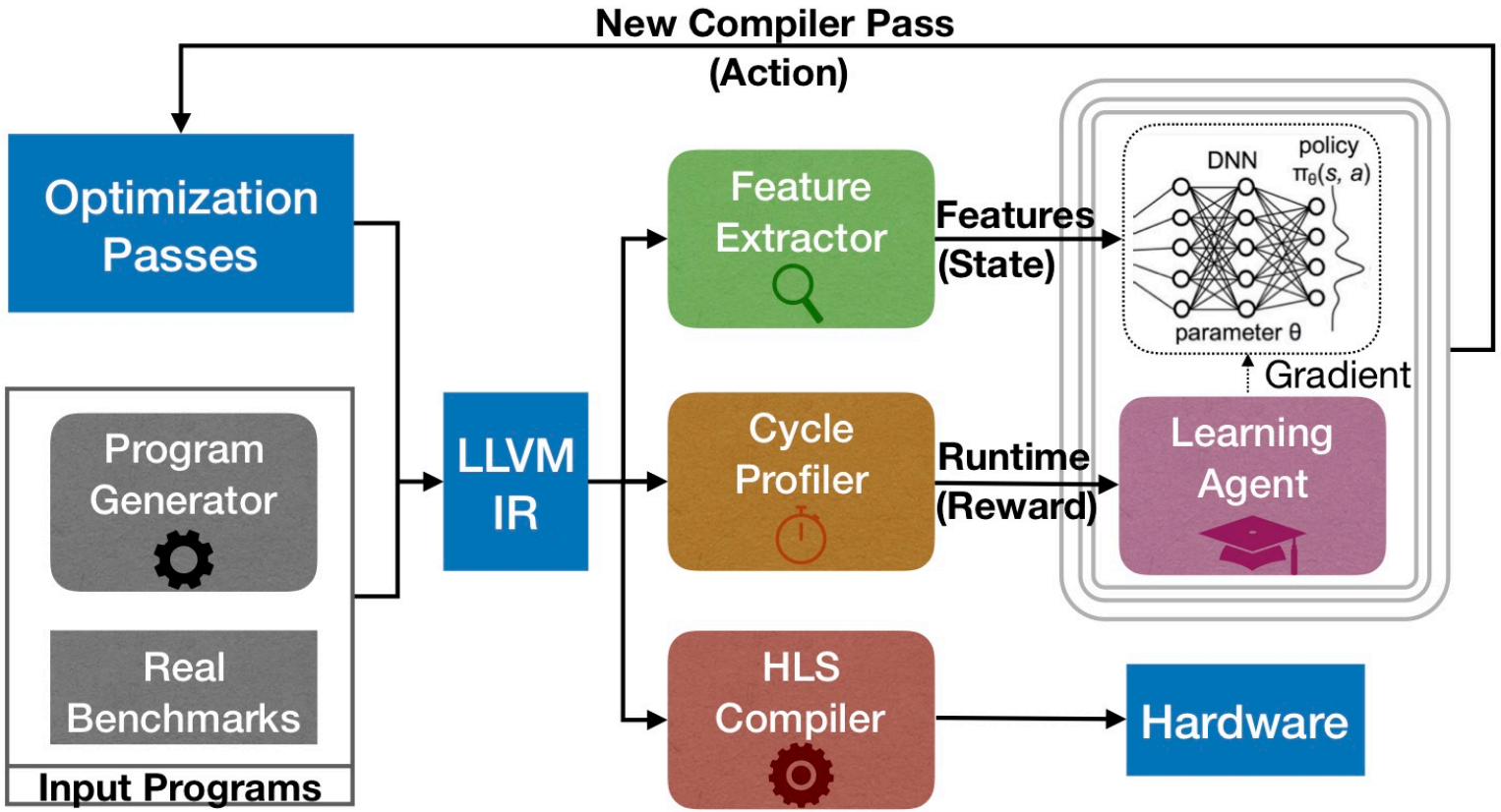
- Reward definition:

  cycles before a pass is applied

  − cycles after a pass is applied

- LegUp estimates the circuit cycles from C
  - Leveraging SW run information
  - 20× faster than RTL simulation
  - 0.48% error rate on our benchmarks

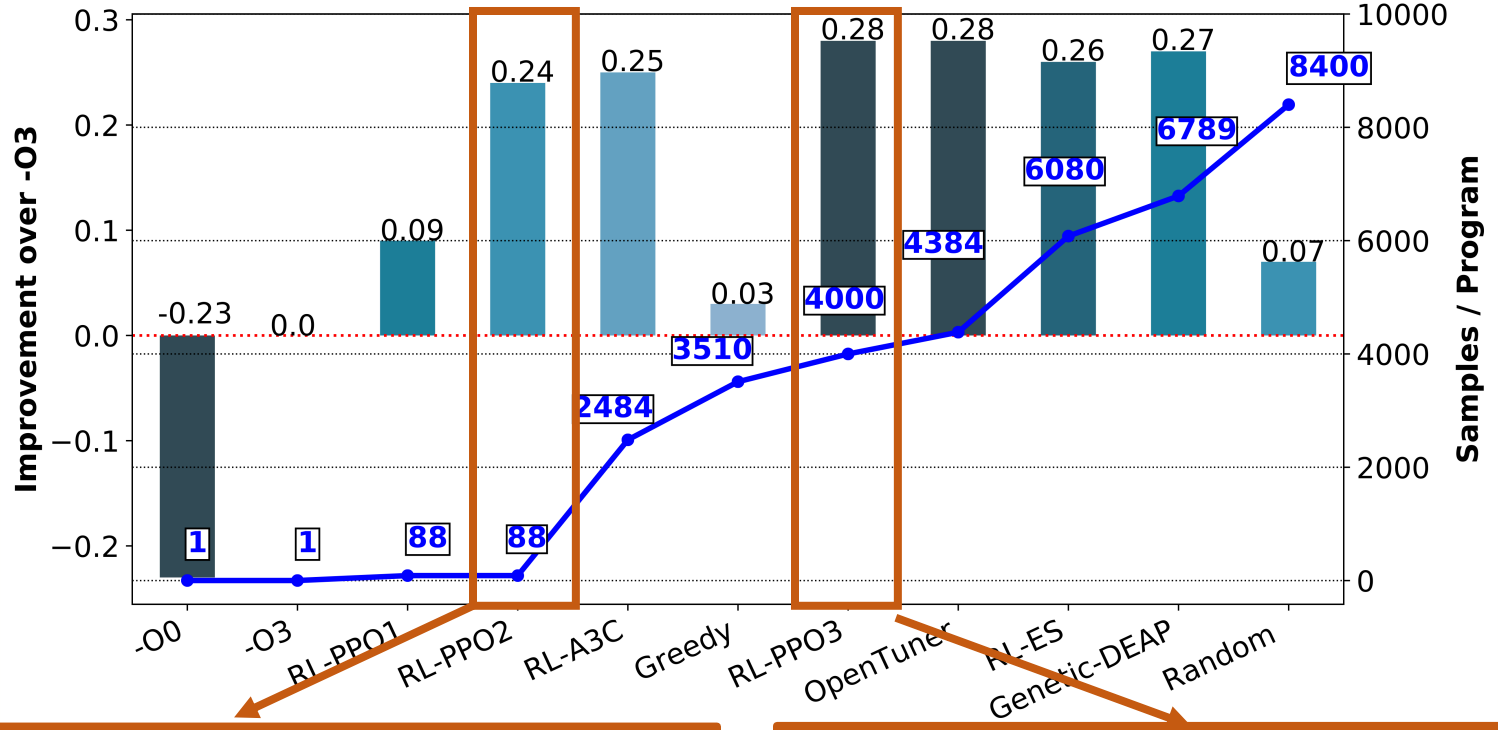
AutoPhase Framework



Evaluation

- Bencmarks:
  - 9 benchmarks from CHStone
- Algorithms:
  - Random Search
  - Genetic Algorithms
  - OpenTuner – Multiarmed Bandit/PSO
  - Greedy Algorithms/Beam Search
  - Deep Reinforcement Learning
    - Proximal Policy Optimization (PPO)
    - Asynchronous Actor-Critic Agents (A3C)
    - Evolutionary Strategies (ES)

Performance (Execution Time)



- Uses only **Histogram of Applied Passes**
- Compiles and runs **once** per trajectory

- Uses **Program Features** and **Histogram of Applied Passes**

- Fewer samples required
- 28% improvement over -O3

RL Generalization

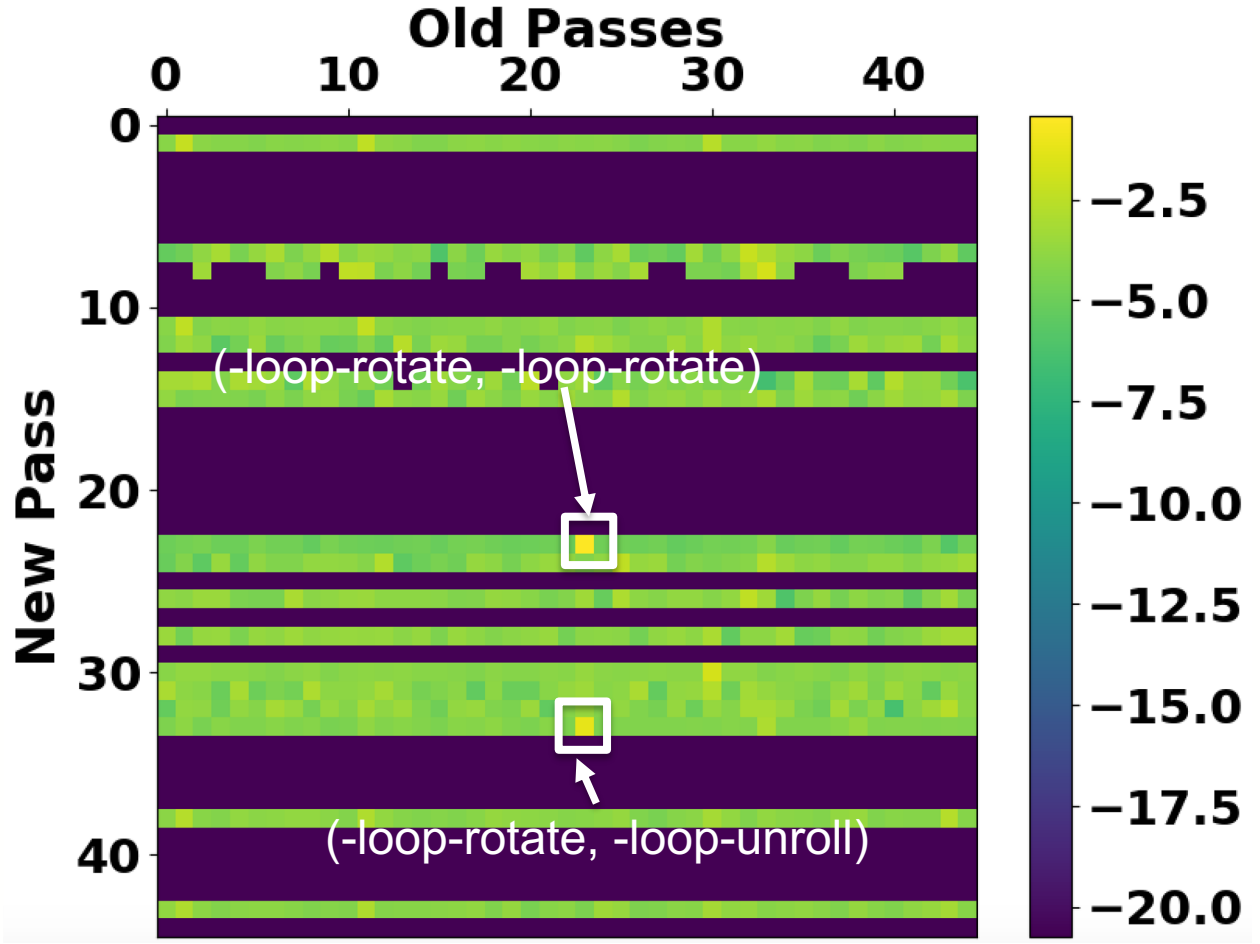
- Can we train a network and use it to predict the optimal sequence of any program in a single compilation?
  - More than 45^{45} possible phase orderings
  - Limited observations
  - Limited data

Very Difficult!

~15,000 Random HLS Programs are Generated

[1] Yang, Xuejun, et al. "Finding and understanding bugs in C compilers." *Proceedings of the 32nd ACM SIGPLAN conference on Programming language design and implementation*. 2011.

Importance Analysis



Normalization Schemes

1. Logarithm of program features and rewards:
   - Reduces the magnitude of features/rewards
   - The neural network learns to correlate the products of features instead of a linear combination of them
     - $w_1 \log(o_{f_1}) + w_2 \log(o_{f_2}) = \log(o_{f_1}^{w_1} \cdot o_{f_2}^{w_2})$

2. Dividing the program feature values by the total number of instructions

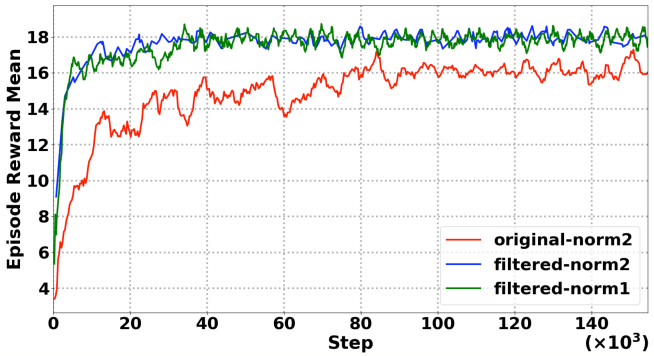
Generalization Results

Train on:

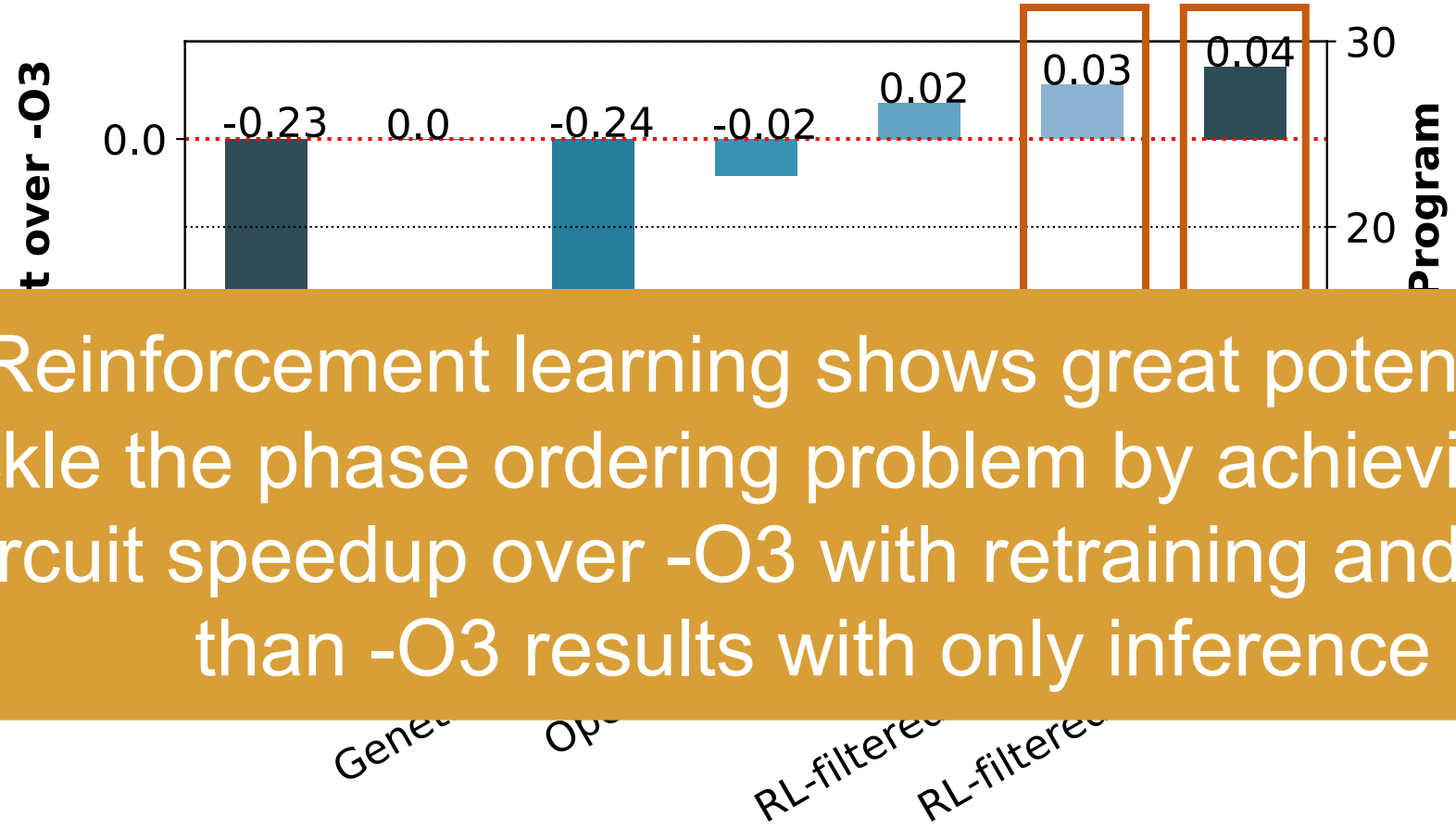
- 100 random programs

Inference-only on:

- 13,000 randomly-generated programs
  - 6% better than –O3
- CHStone benchmarks
  - different from the randomly-generated programs



Generalization Results



Reinforcement learning shows great potential to tackle the phase ordering problem by achieving 28% circuit speedup over -O3 with retraining and better than -O3 results with only inference

- ~4% improvement over -O3 on CHStone

Outline

- Motivation
- Codesign for Image Classification
  - Shift-based DiracDeltaNet
- Codesign for Object Detection
  - Deformable Convolution
- Reinforcement Learning for HLS
  - Compiler Phase-Ordering
- Conclusion

Conclusion

- *Algorithm-hardware co-design* can achieve both high accuracy and good efficiency for embedded CV applications
- *Reinforcement learning* is a promising approach for addressing the hardware design problem with intractable search space

Questions?

Email: qijing.huang@berkeley.edu

Special Thanks

Fellow Researchers:

- Yifan Yang
- Bichen Wu
- Tianjun Zhang
- Liang Ma
- Giulio Gambardella
- Michaela Blott
- Luciano Lavagno
- Kees Vissers
- Prof. Kurt Keutzer
- Dequan Wang
- Zhen Dong
- Yizhao Gao
- Ameer Haj-Ali
- William Moses
- John Xiang
- Prof. Krste Asanovic
- Prof. Ion Stoica
- Prof. John Wawryznek

Thanks!