Algorithms, Hardware, and Scheduling Co-optimization for Deep Learning Applications

Qijing Jenny Huang
PhD Advisor: John Wawrzynek
University of California, Berkeley

Outline

Motivation

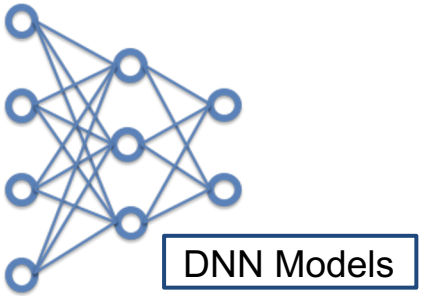
Opportunities & Prior Work

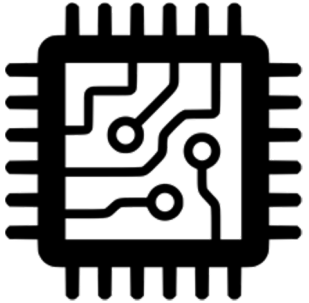
NoC Scheduling

Motivation

Problem Statement

There are gaps among **algorithms**, **software**, and **hardware** which lead to *suboptimal solutions* for accelerating deep learning (DL) tasks

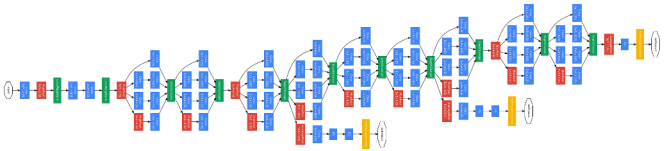




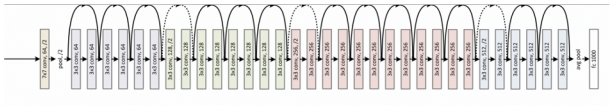
- Algorithm
- Software
- Hardware

How do we design the DNN models, software stack, and hardware accelerators to achieve the highest efficiency for a DL task?

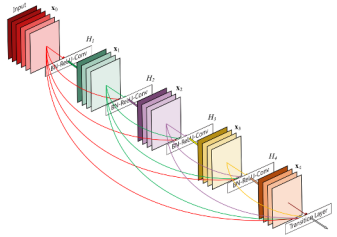
Increasing DNN Model Complexity



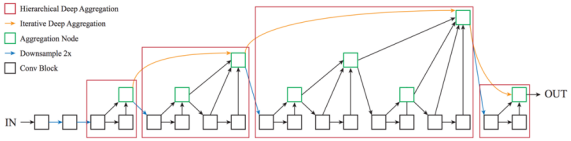
GoogLeNet 2014 (22 layers)



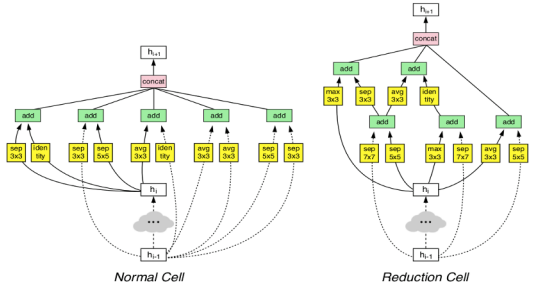
ResNet 2015 (152 layers)



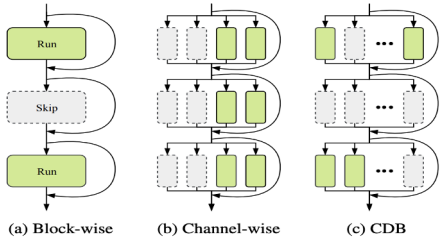
DenseNet 2016 (dense connections)



DLA 2017 (deep aggregation)



NasNet 2017 (NAS design)



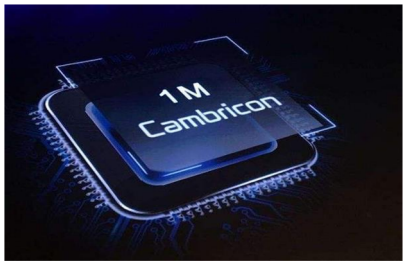
CDB 2018 (dynamic)

Increasing Hardware Diversity

~ 85 AI chip companies worldwide

At the Edge - General (Inference + Training)

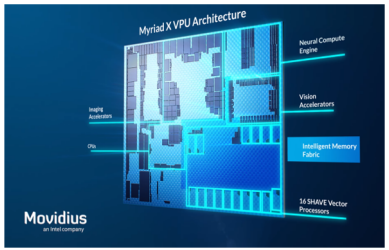
- 10s-1000s GFLOPs
- 100s KB on-chip memory
- 1 - 16 bit precision
- 600 MHz - 1 GHz
- 10-100s mWatts



Cambricon-1M IP

At the Edge - Dedicated (Inference + Training)

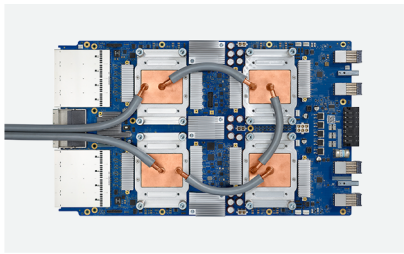
- 100s-1000s GFLOPs
- 100s KB on-chip memory
- 1 - 16 bit precision
- 50 MHz - 400 MHz
- 1-10s Watts



Intel Movidius (4 TFLOP/s)

In the Cloud (Training + Inference)

- 10s TFLOPs
- 10s MB on-chip memory
- 8 - 32 bit precision
- 700 MHz - 1 GHz
- 10-100s Watts

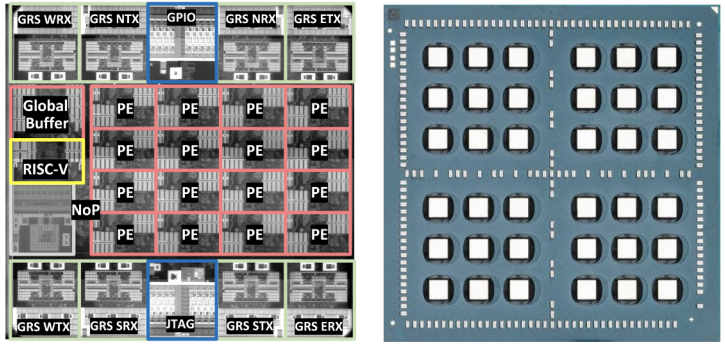


Cloud TPU v3 (45 TFLOP/s)

Increasing Number of Parallel Resources

Network on Chip/Package (NoC/NoP)

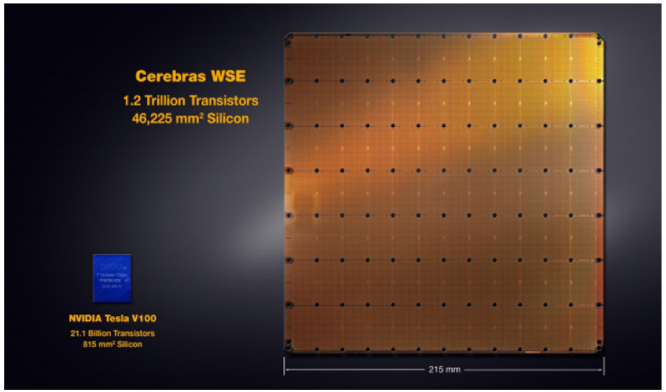
NoC/NoP Chip



(a) Simba chiplet (b) Simba package

[Simba]
16PEs x 36 Chiplets

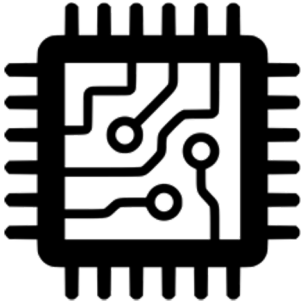
Wafer-scale Chip



Cerebras
84 Interconnected Chips

Scheduling is required everywhere

Scheduling



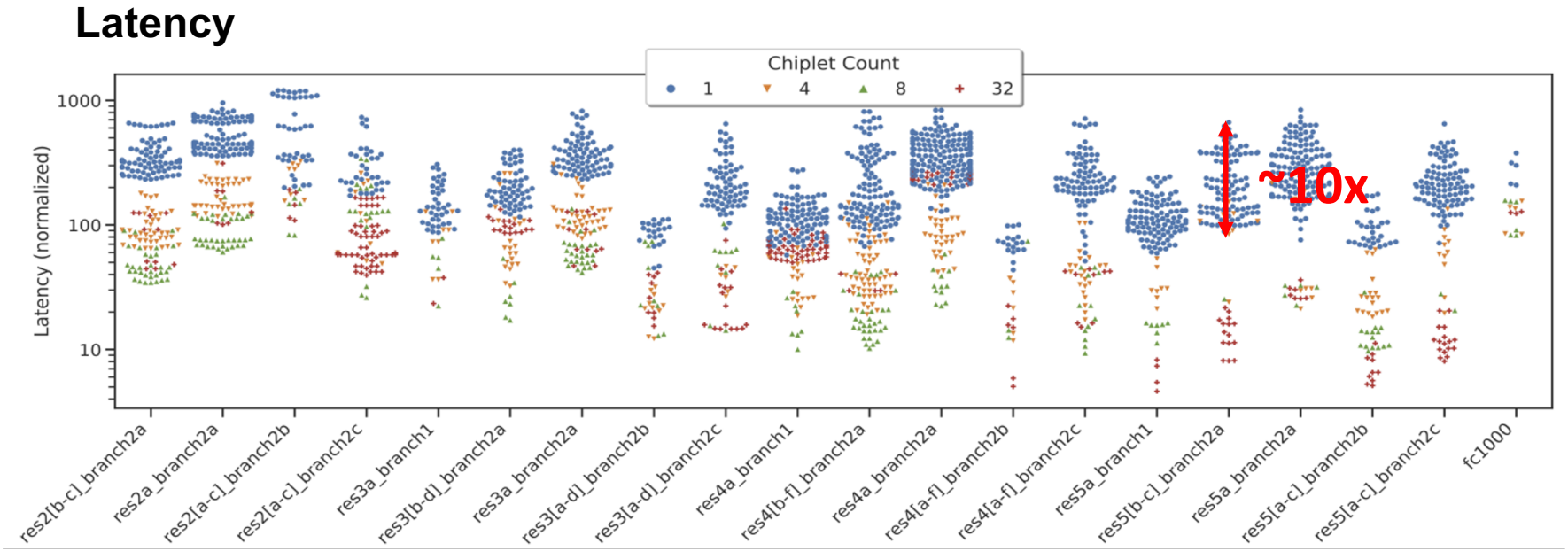
- Algorithm

many architectural states to be run

- Hardware

many hardware resources to be allocated

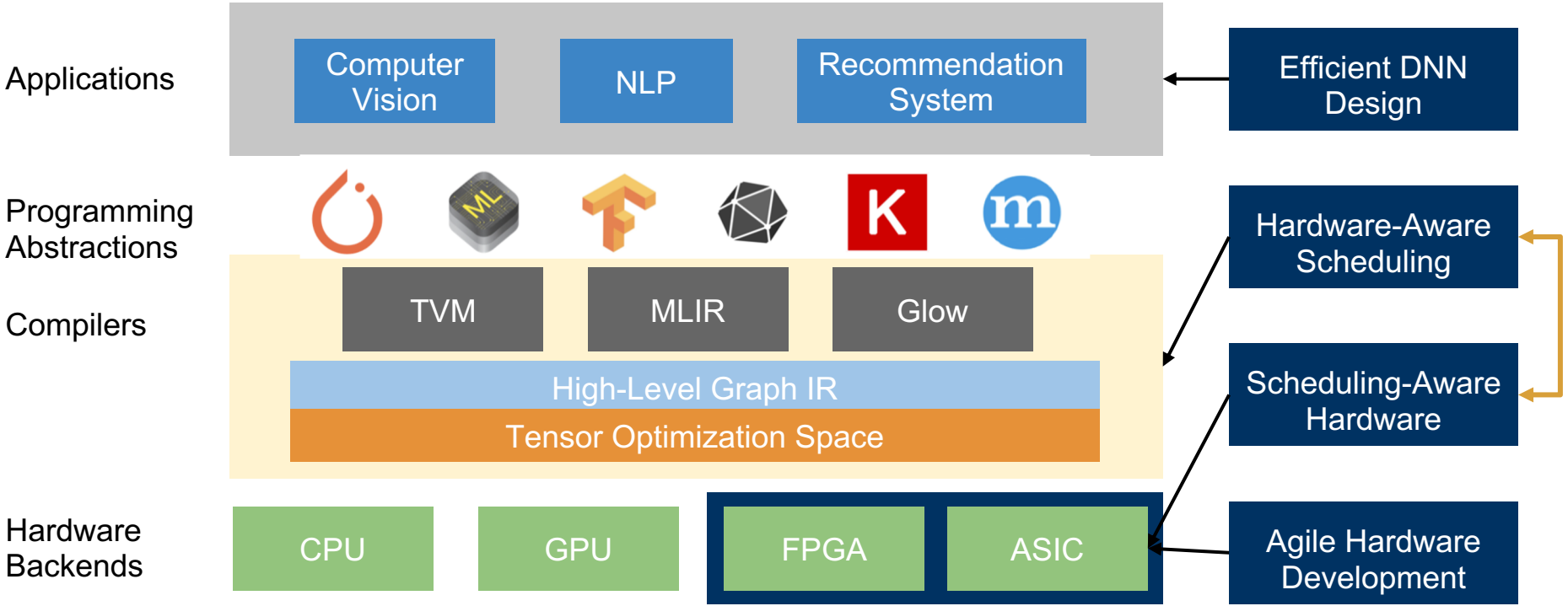
Scheduling significantly affects performance



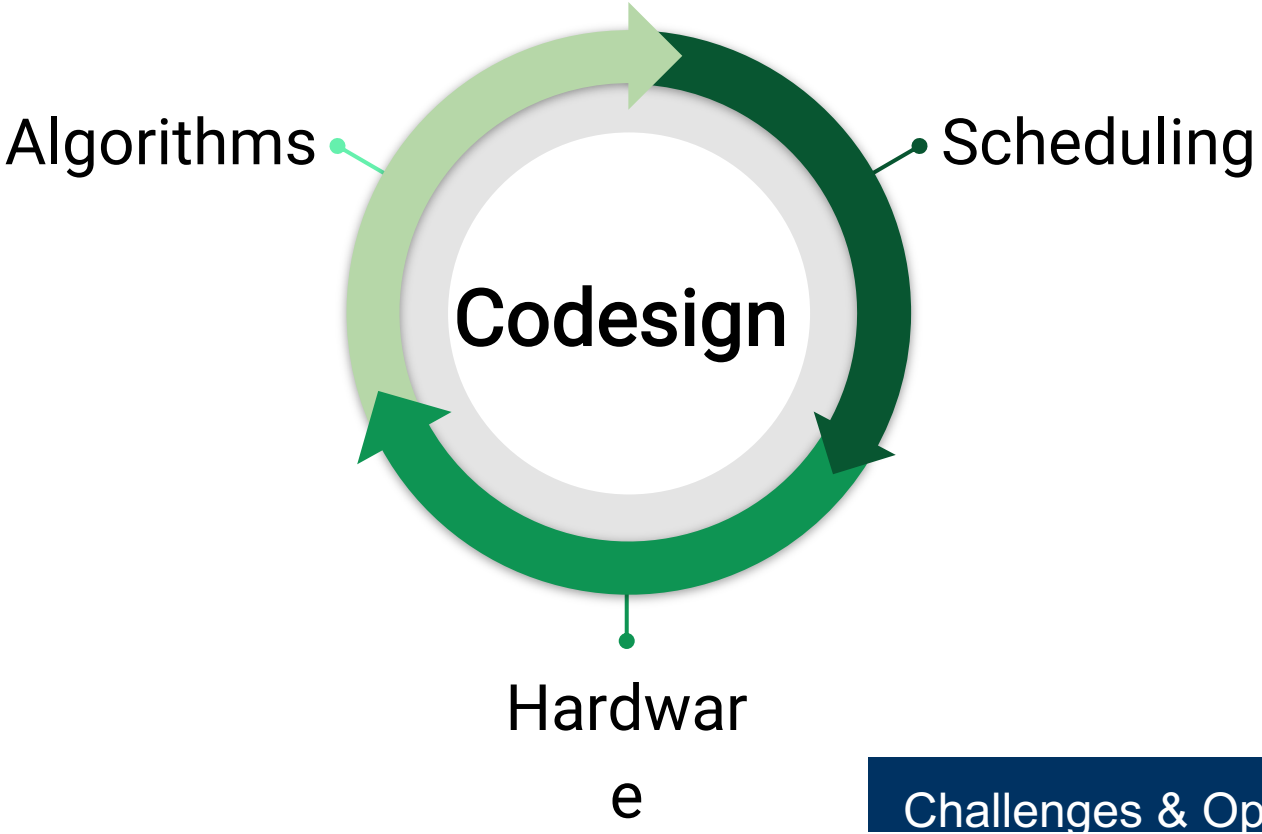
* Shao, Yakun Sophia, et al. "Simba: Scaling Deep-Learning Inference with Multi-Chip-Module-Based Architecture." 2019 MICRO.

Opportunities & Prior Work

DNN Deployment Stack



DNN Acceleration Stack Optimization



Opportunities

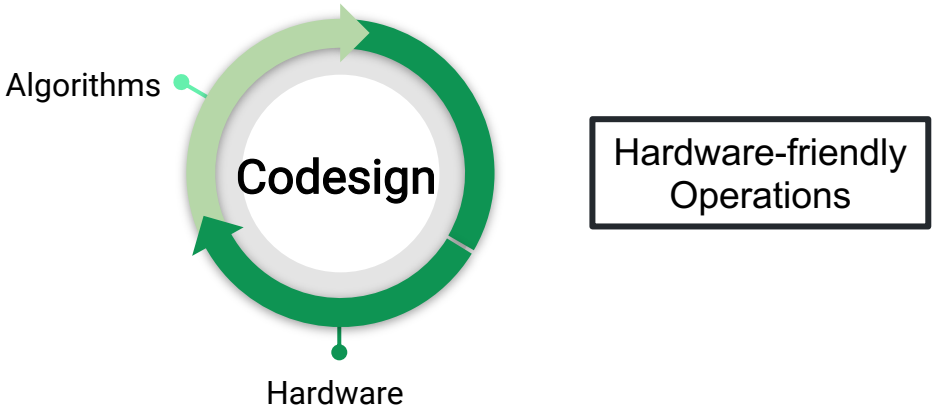
- We see a potential to improve the current designs with:
  - **Efficient Algorithm Design [FPGA'19][EMC2'19]**
    - **Hardware-friendly Operation Codesign**
  - **Agile Hardware Development [ICCAD'19][MLSys'20]**
    - **Hardware Generation from High-Level Description**
  - **Hardware-Aware Scheduling**
    - **Intelligent Scheduler for Heterogeneous Hardware**
  - **Scheduling-Aware Hardware**
    - **Scheduling-Informed Hardware Codesign**

Our Prior Work

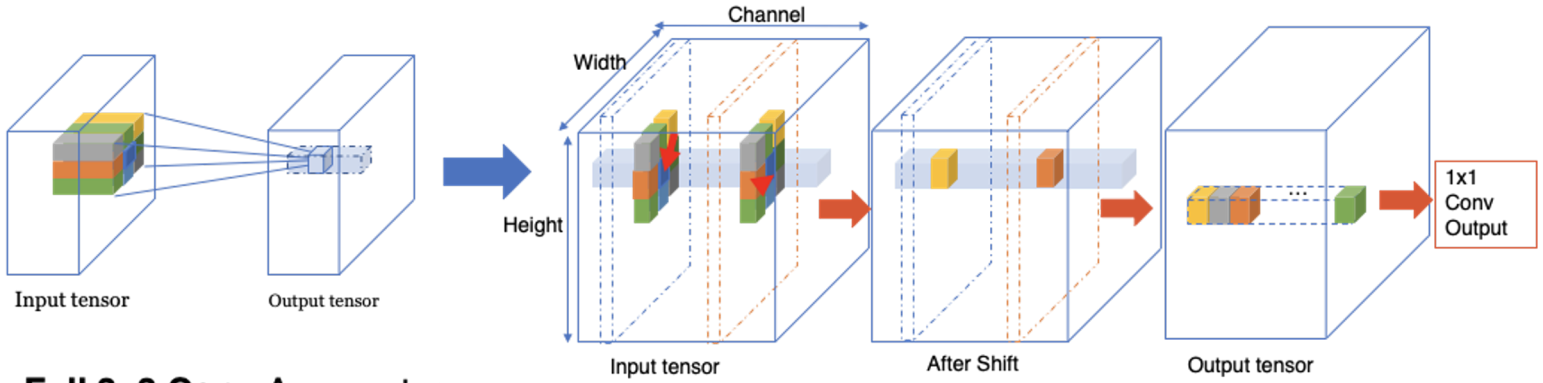
- We see a potential to improve the current designs with:
  - **Efficient Algorithm Design [FPGA'19][EMC2'19]**
    - **Hardware-friendly Operation Codesign**

HW-Algo Codesign

1. Synetgy [FPGA'19]
2. Deformable Convolution Codesign [EMC2'19]



Synetgy: DiracDeltaNet Design



Full 3x3 Conv Aggregates neighboring pixels and mixes channel info

Shift moves a neighboring pixel to the center position

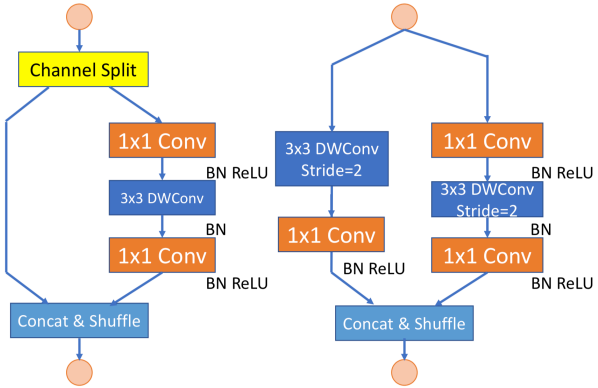
1x1 Conv aggregates spatial info along the channel dimension

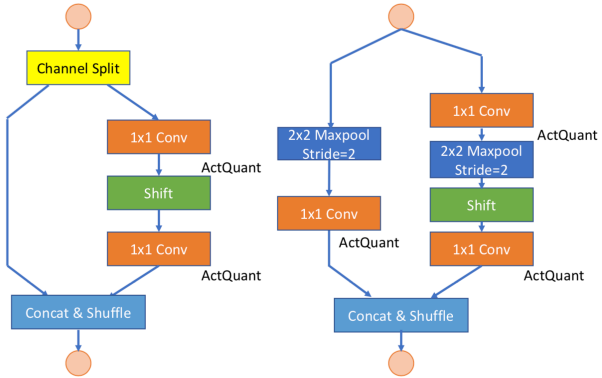
1. Full 3x3 Conv → Shift and 1x1 Conv
2. Depthwise 3x3 Conv w/ Stride 2 → Shift and 2x2 Pooling w/ Stride 2

Spatial 3x3 Convolution Free

Yang, Yifan, **Huang, Qijing**, et al. "Synetgy: Algorithm-hardware co-design for convnet accelerators on embedded fpgas." *Proceedings of the 2019 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays (FPGA)*. ACM, 2019.

Synetgy: DiracDeltaNet Design





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

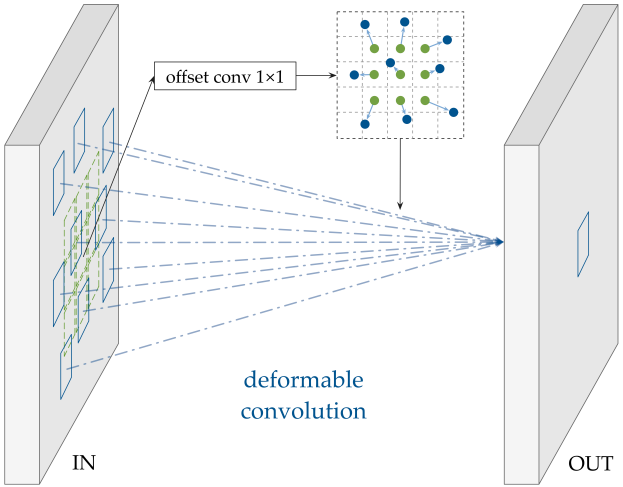
- Accuracy (full precision): **69.7%**
- Operators involved:
  - 1x1 convolution
  - 2x2 max pooling
  - /shuffle/shift/concat

11.6x higher framerate
6.3x more power efficient

Yang, Yifan, **Huang, Qijing**, et al. "Synetgy: Algorithm-hardware co-design for convnet accelerators on embedded fpgas." *Proceedings of the 2019 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays (FPGA)*. ACM, 2019.

Deformable Convolution Codesign

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



1. Generate offsets
2. Sample from input feature map

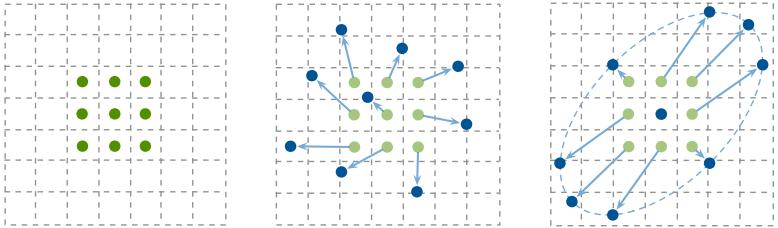
Deformable Convolution Codesign

- **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 Codesign

- **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



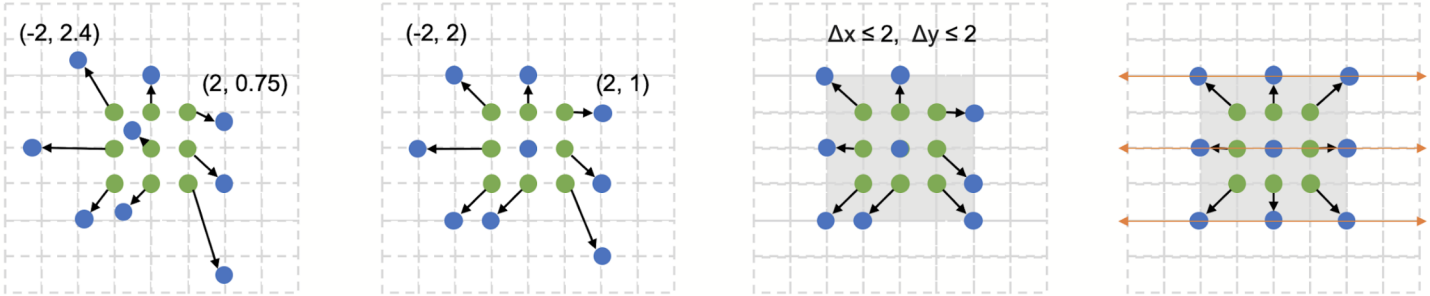
Variable Receptive Fields

Deformable Convolution Codesign

- **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

Deformable Convolution Codesign

1. Algorithm Modifications



0. Original Deformable **1. Rounded Offsets** **2. Bounded Range** **3. Rectangular Shape**

a. **Deformable Convolution** samples inputs from variable offsets generated based on the input feature

b. **Rounded Offsets** rounds the fractional offsets to integer

c. **Bounded Range** restricts the range of offsets

d. **Rectangle Shape** limits the geometry to a rectangle shape

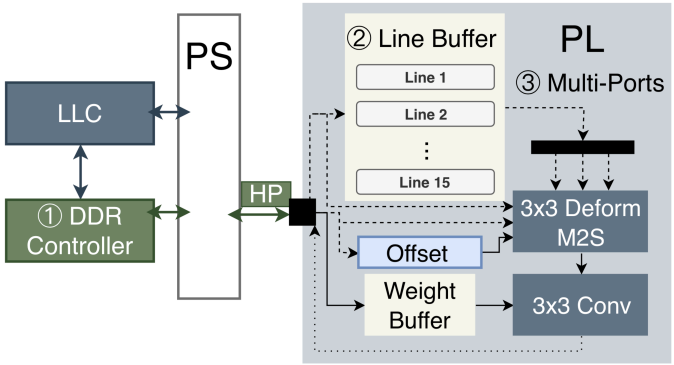
e. **Depthwise** replaces full conv with 3x3 dw

More regular access patterns
More efficient operators

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

Deformable Convolution Codesign

2. Hardware Optimizations



a. **Baseline** loads input features with dynamic offsets from DRAM directly

b. **Caching** adds LLC to leverage temporal and spatial locality

c. **Buffering** uses on-chip BRAM to buffer all inputs from limited range

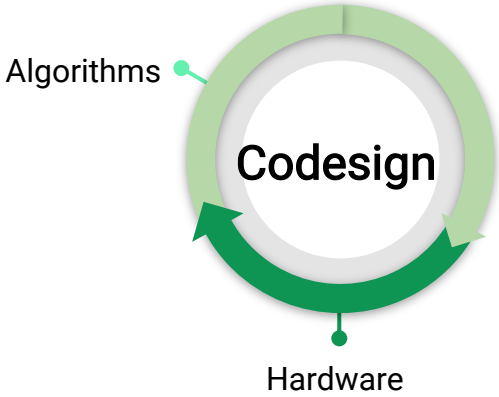
1.36x and **9.76x** speedup respectively for the full and depthwise deformable convolution on FPGA

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

Our Prior Work

- We see a potential to improve the current designs with:
  - **Efficient Algorithm Design [FPGA'19][EMC2'19]**
    - **Hardware-friendly Operation Codesign**
  - **Agile Hardware Development [ICCAD'19][MLSys'20]**
    - **Hardware Generation from High-Level Description**

Agile HW Development
1. Centrifuge [ICCAD'19] 2. AutoPhase [MLSys'20]



Automatic HW Generation and Optimization

Centrifuge: Accelerator-SoC Generation with HLS

1. Workload Characterization
2. Accelerator Modeling
    - Analytical
    - Transaction-based
3. RTL Development
4. Emulation and Verification
5. Chip Tapeout
6. Software Development

Proposal 1: Native Simulation

Run full-stack software with target SoC simulation

Proposal 2: Rapid Prototyping

Combine 2 and 3 with one high-level abstraction

Proposal 3: Agile Development

Improve software stack concurrently with hardware development

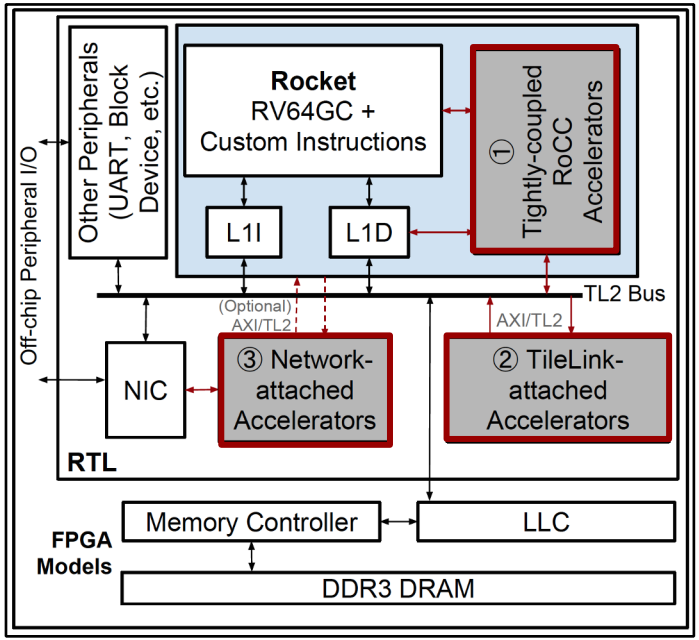
Huang, Qijing, et al. "Centrifuge: Evaluating full-system HLS-generated heterogenous-accelerator SoCs using FPGA-Acceleration." *2019 ICCAD*.

Centrifuge: Accelerator-SoC Generation with HLS

Goal: Enable rapid accelerator SoCs generation and evaluation with High-level Synthesis (HLS)

Centrifuge provides the user with:

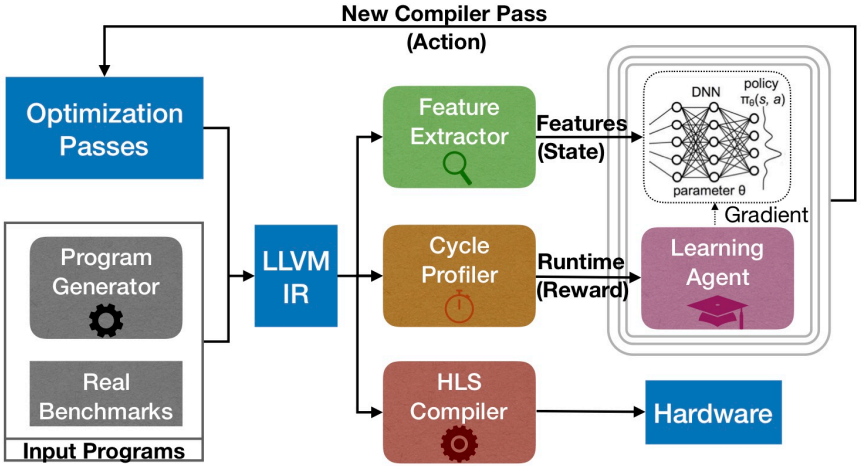
1. Full-system evaluation of the target workload
2. Fast development and verification cycle for both HW/SW
3. Large design space for rapid algorithm-hardware exploration
    - Hardware Integration
    - Architectural Design Variation
    - Software Integration



System Diagram

Huang, Qijing, et al. "Centrifuge: Evaluating full-system HLS-generated heterogenous-accelerator SoCs using FPGA-Acceleration." *2019 ICCAD*.

AutoPhase: RL for HLS Phase-Ordering

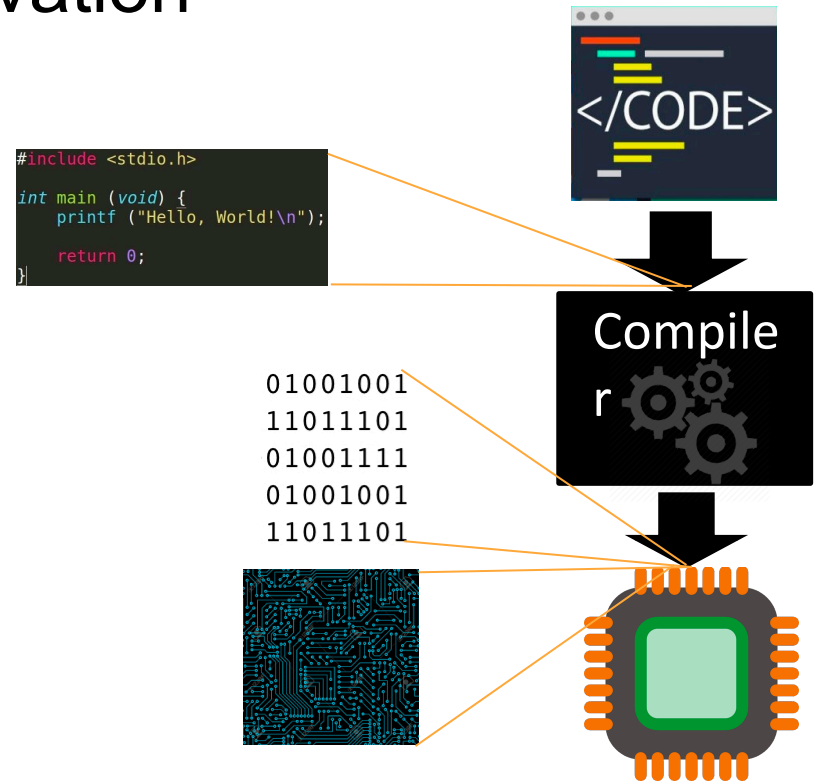


NP-Hard

- **HLS Phase-Ordering:** Choosing a good order to apply the optimizations
- **Reinforcement Learning (RL)**: A machine learning approach in which an agent continually interacts with the environment to learn a policy that optimizes long-term rewards

Haj-Ali, Ameer*, **Huang, Qijing*,** et al. "AutoPhase: Juggling HLS Phase Orderings in Random Forests with Deep Reinforcement Learning." Third Conference on Machine Learning and Systems (ML-Sys). 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*

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);
```

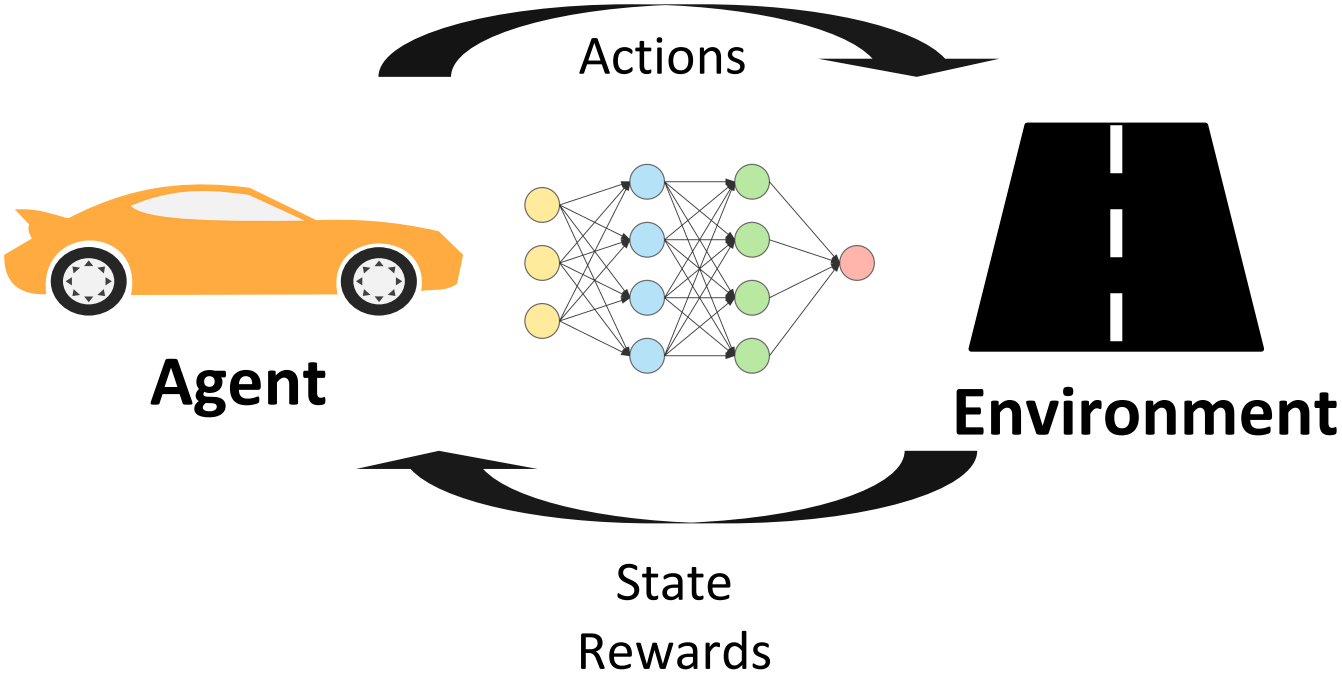
Pass #2: -licm (loop-invariant code motion)

```
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

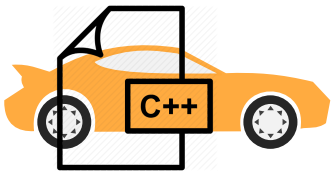
A Tree

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

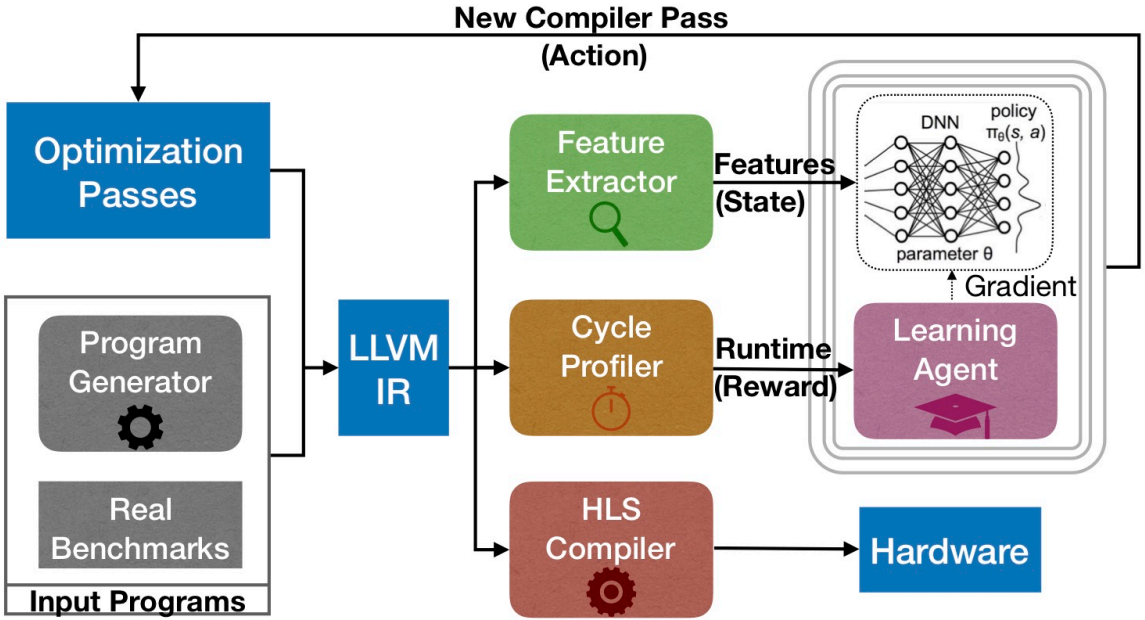
- Reward definition:

  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

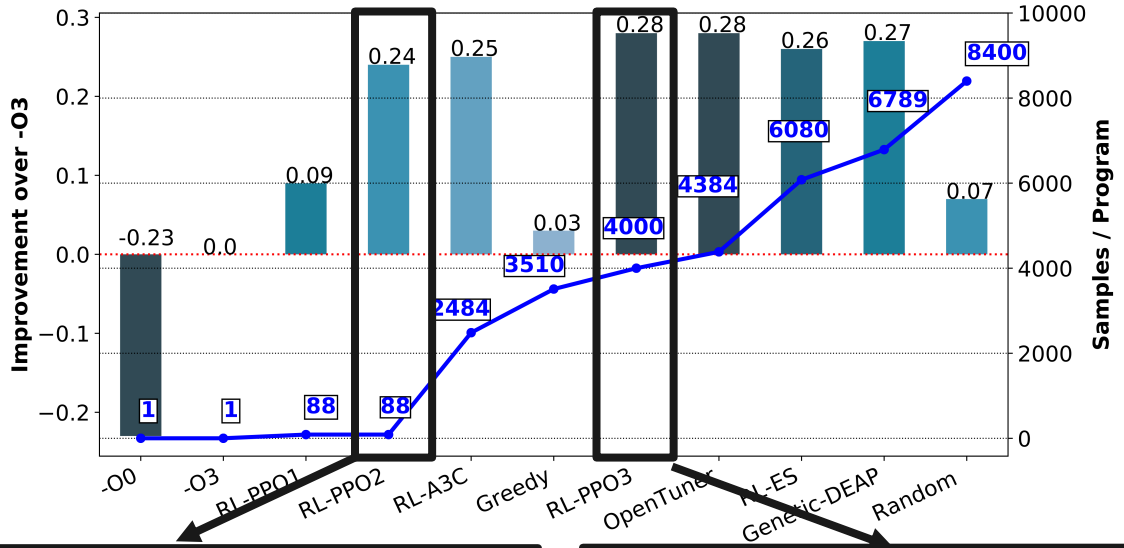
AutoPhase Framework



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)



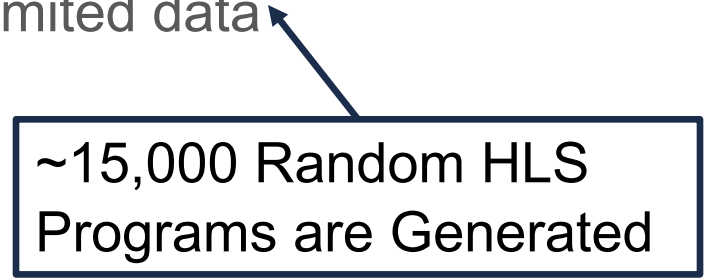
- Uses only **Histogram of Applied Passes**
- Compiles and runs **once** per trajectory

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

- 28% improvement over -O3

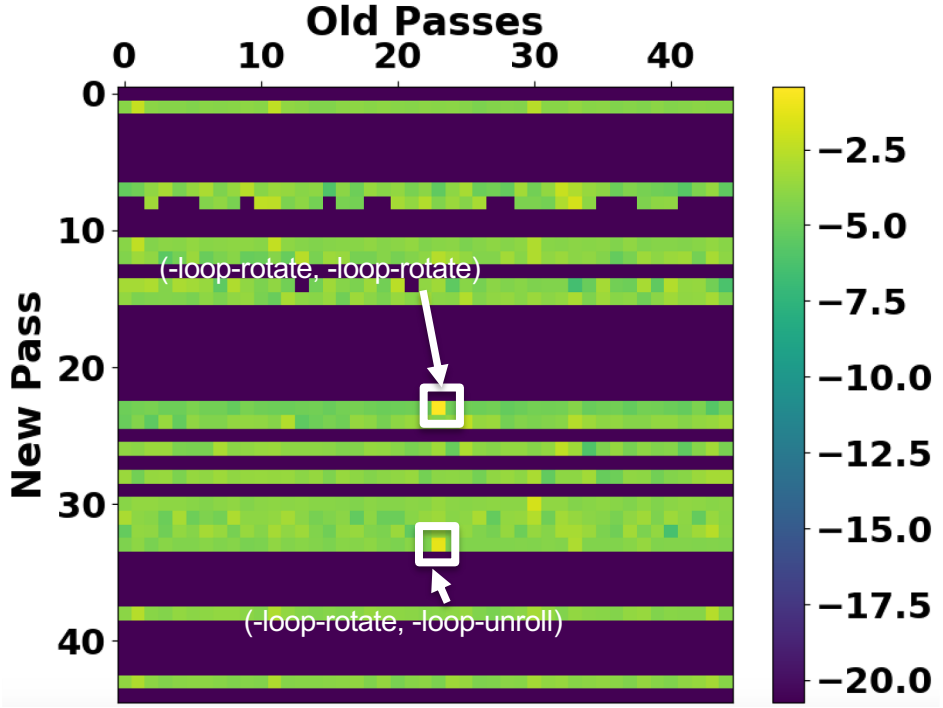
RL Generalization

- 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 **Very Difficult!**
  - Limited observations
  - Limited data



[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

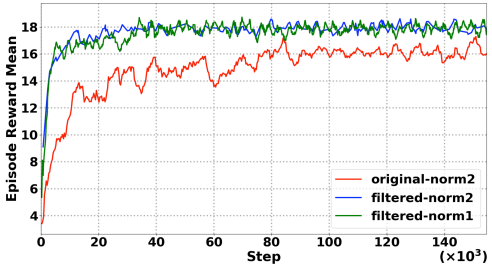
Generalization Results

Train on:

- 100 random programs

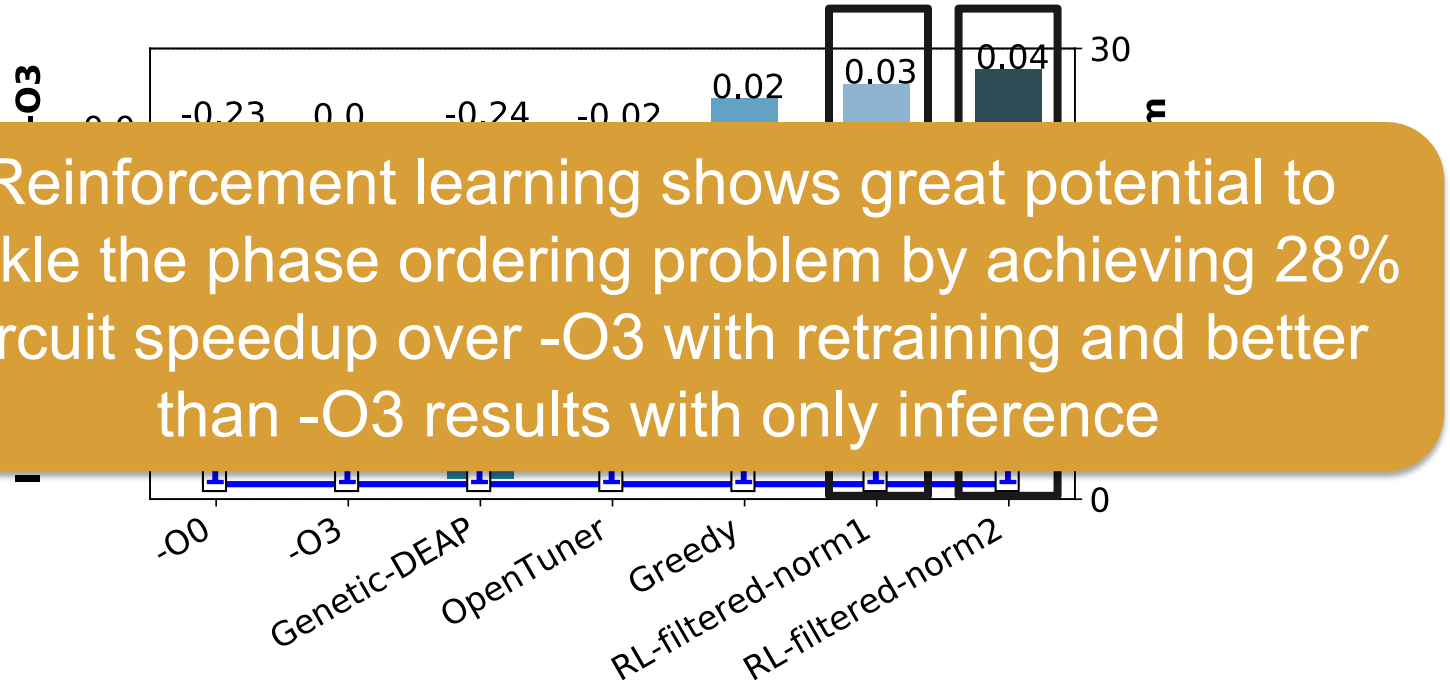
Inference-only on:

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



Generalization Results

- ~4% improvement over -O3 on CHStone



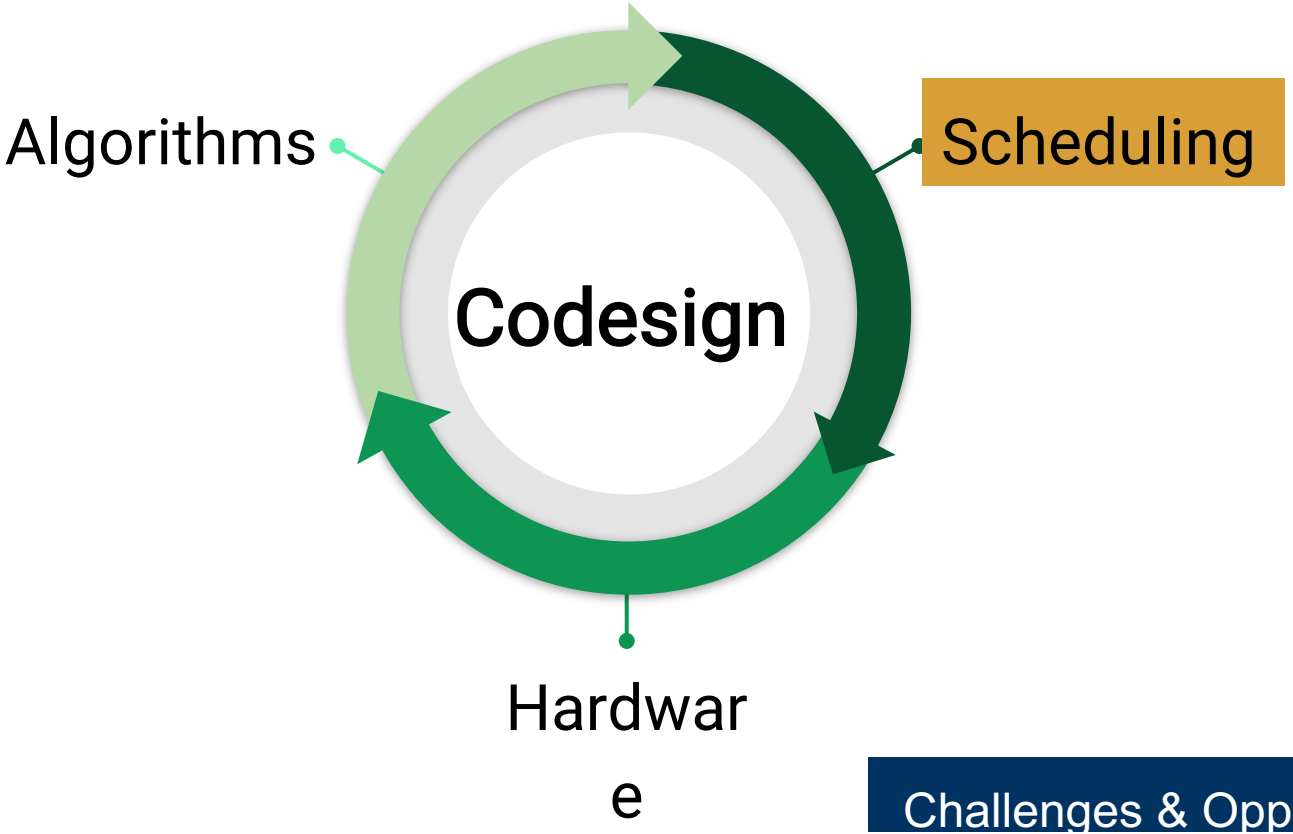
Reinforcement learning shows great potential to tackle the phase ordering problem by achieving 28% circuit speedup over -O3 with retraining and better than -O3 results with only inference

Opportunities

- We see a potential to improve the current designs with:
  - **Efficient Algorithm Design [FPGA'19][EMC2'19]**
    - **Hardware-friendly Operation Codesign**
  - **Agile Hardware Development [ICCAD'19][MLSys'20]**
    - **Hardware Generation from High-Level Description**
  - **Hardware-Aware Scheduling**
    - **Intelligent Scheduler for Heterogeneous Hardware**
  - **Scheduling-Aware Hardware**
    - **Scheduling-Informed Hardware Codesign**

Proposed Thesis Work

Proposed Thesis Work



NoC Scheduling

NoC Scheduling for DNNs

1. Motivation
2. Problem Definition
3. Infrastructure
4. Scheduling Algorithms
5. Future Work

Hardware-Oriented Scheduling is Needed

Existing software-oriented compiler frameworks for scheduling DNNs:

- Assume a managed memory system
  - Data movement is implicit
- Treat the hardware backend as a black box
  - Feedback is crucial for the autoscheduling
- Lack of support for spatial organization
  - Allocation & placement are not handled

Explicit Data Movement

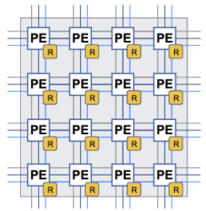
Known Constraints

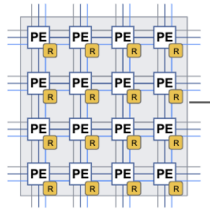
Spatial Organization

NoC Scheduling for DNNs

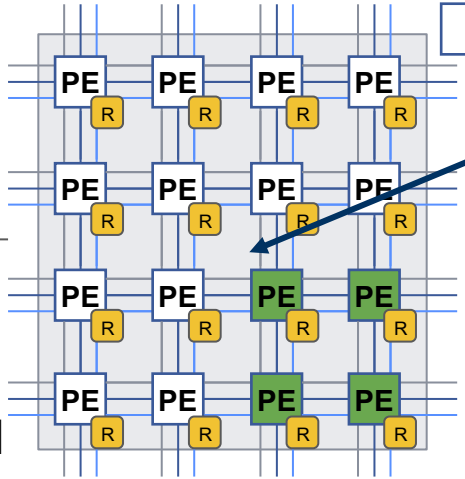
1. Motivation
2. Problem Definition
3. Infrastructure
4. Scheduling Algorithms
5. Future Work

Scheduling for Heterogeneous Hardware Systems





Multi-level Network



NoC

CPU

Tensor Core

DNN Graphs

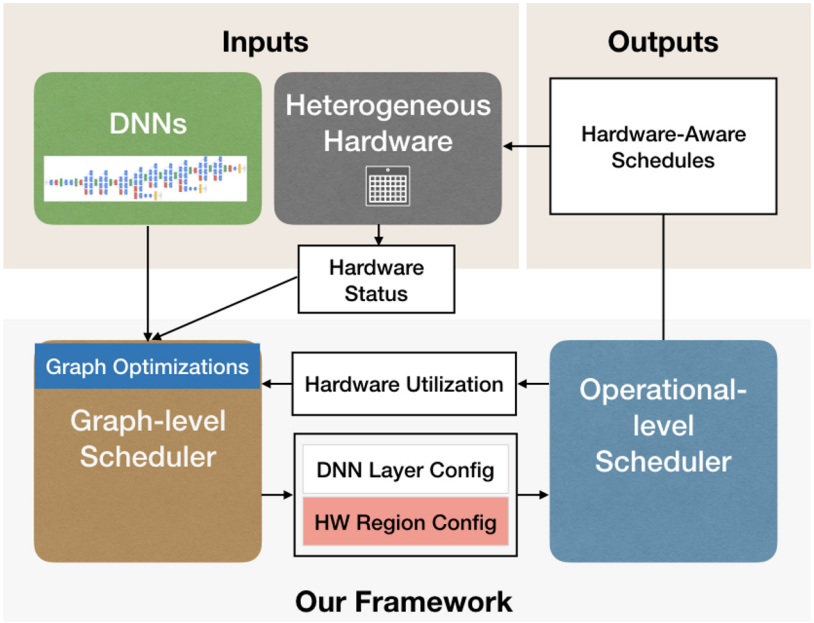
Scheduler

Scheduler explicitly specifies:
1. Temporal Ordering
2. Data Movement among
   a. Spatial PEs
   b. Multi-level Memories

Target workload types:
1. Single-layer
2. Multiple-layers
3. Multiple-networks

PE carries out arithmetic and logic operations
NoC is a router-based packet switching network

Two-Level Scheduling



1. **Operation-level:**
   - Tiling Factors
   - Spatial Mapping
   - Loop Permutation
2. **Co-scheduling:**
   - Ordering
   - Allocation
   - Placement

This talk

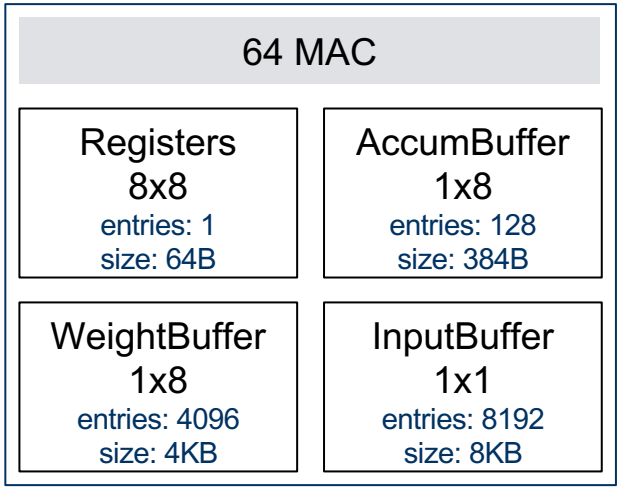
Exhaustive search is not tractable

Operation-level Scheduling

- **Inputs Constraints:**

Problem: ----- 7 nested loops
R=3, S=3, P=28, Q=28, C=128, K=128, N=1

Architecture: ----- 5 levels of memory



- **Output Schedule:**

```
DRAM [ Weights:147456 Inputs:115200 Outputs:100352 ]
-----------------------------------------------------
| for P in [0:4)
|   for S in [0:3)
|     for C in [0:16) (Spatial-X)
InputBuffer [ Inputs:2016 ]
---------------------------
|       for N in [0:1)
|         for R in [0:3) (Spatial-X)
WeightBuffer [ Weights:1024 ]
-----------------------------
|           for Q in [0:28)
|             for P in [0:7)
AccumulationBuffer [ Outputs:128 ]
-----------------------------------
|               for K in [0:128)
|                 for C in [0:8)
Registers [ Weights:1 ]
-----------------------
|                   for N in [0:1)
```

Loop Permutation

Spatial Mapping

Temporal Mapping

Tiling Factors

Operation-level Scheduling Complexity

Given a 7D CNN on a 5-tiling-level architecture:

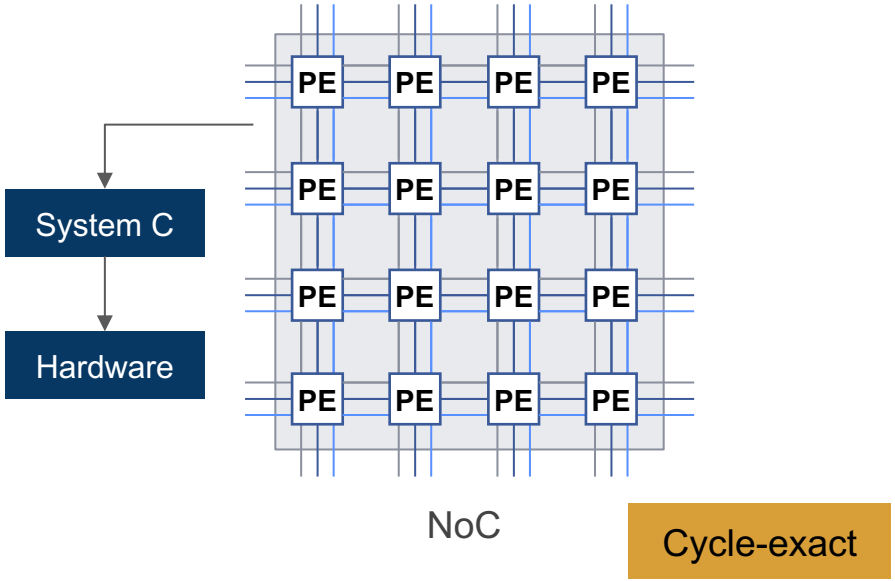
1. Temporal or Spatial Mapping: 2
2. Loop Permutation: (7!) ^ 5
3. Factors: ((5 x 2) ^ (total # of prime factors for each loop bound)) ^ 7
4. Total: **Permutations x Factors**

The search space is large

NoC Scheduling

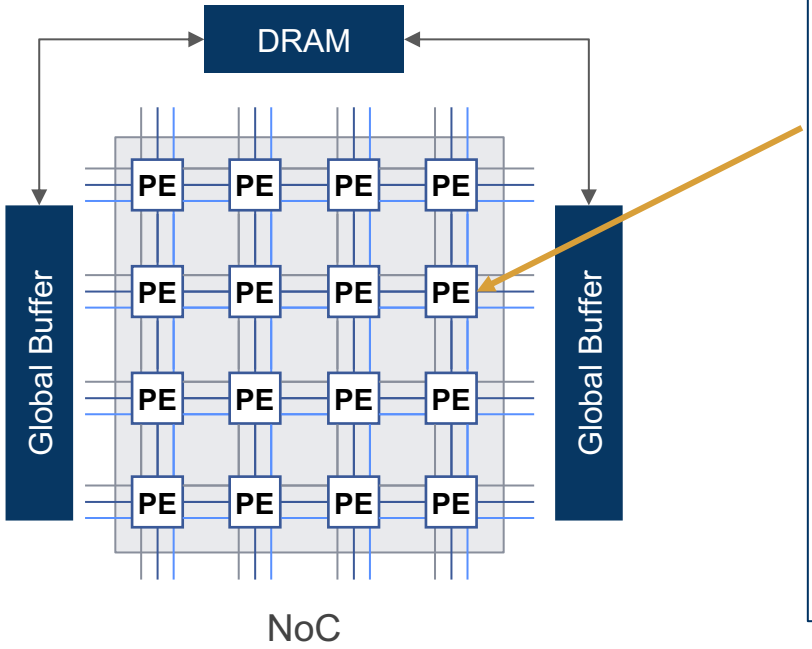
1. Motivation
2. Problem Definition
3. Infrastructure
4. Scheduling Algorithms
5. Future Work

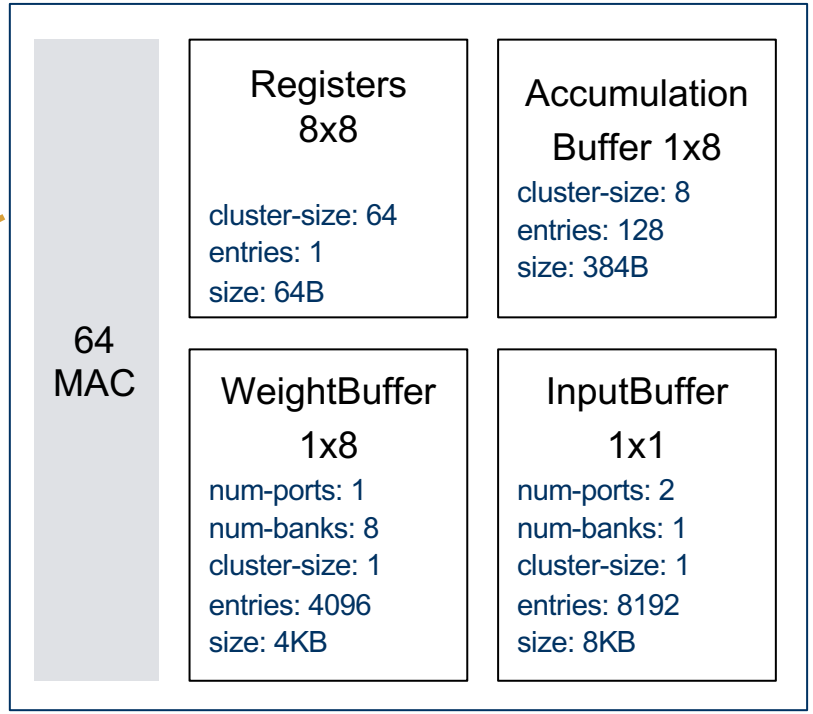
NoC Implementation



- Uses router design in System C HLS from NVLAB MatchLib
- XY Routing
- Supports:
  - Unicast (Wormhole routing)
  - Multicast (Cut-through routing)
  - Reduction
  - Scatter gather
  - Barriers

Memory Modeling

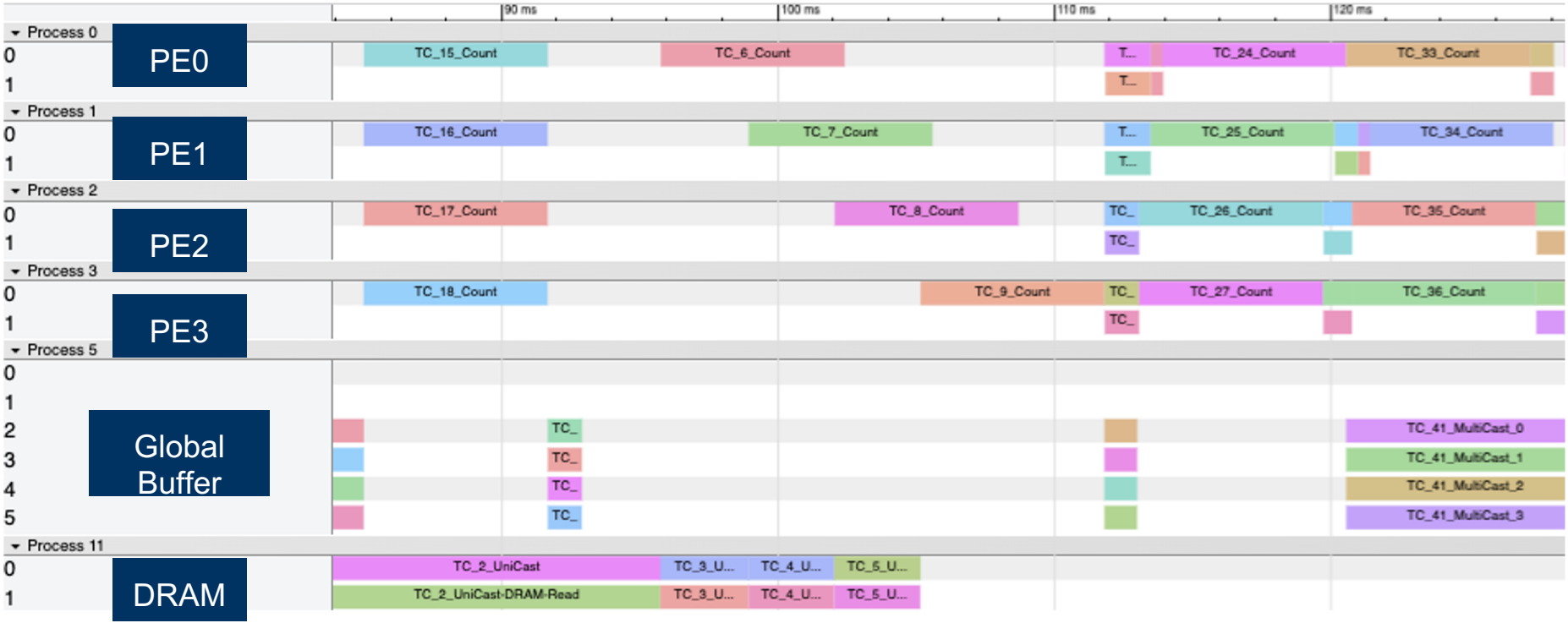




Transaction-based Model:

1. DRAMSim2 for DRAM simulation
2. 1 Cycle latency for on-chip buffers

Execution Traces



NoC Scheduling

1. Motivation
2. Problem Definition
3. Infrastructure
4. Scheduling Algorithms
5. Future Work

Operation-level:

A. Tiling Factors ✓
B. Spatial Mapping
C. Loop Permutation

A. Tiling Factors

Intuition 1:

Higher buffer utilization

⟶ better performance

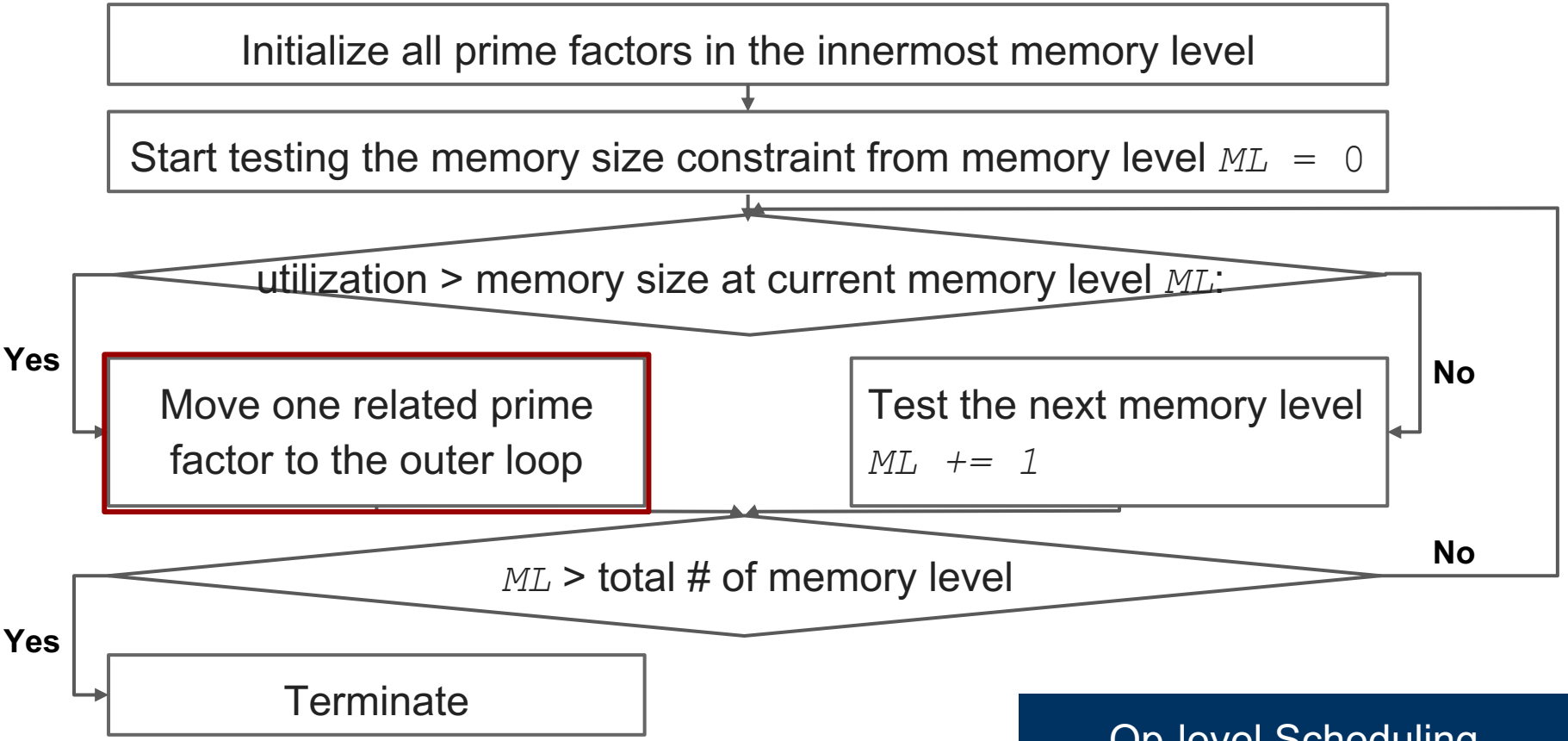
Naive Solution: Greedy Algorithm

Schedule:

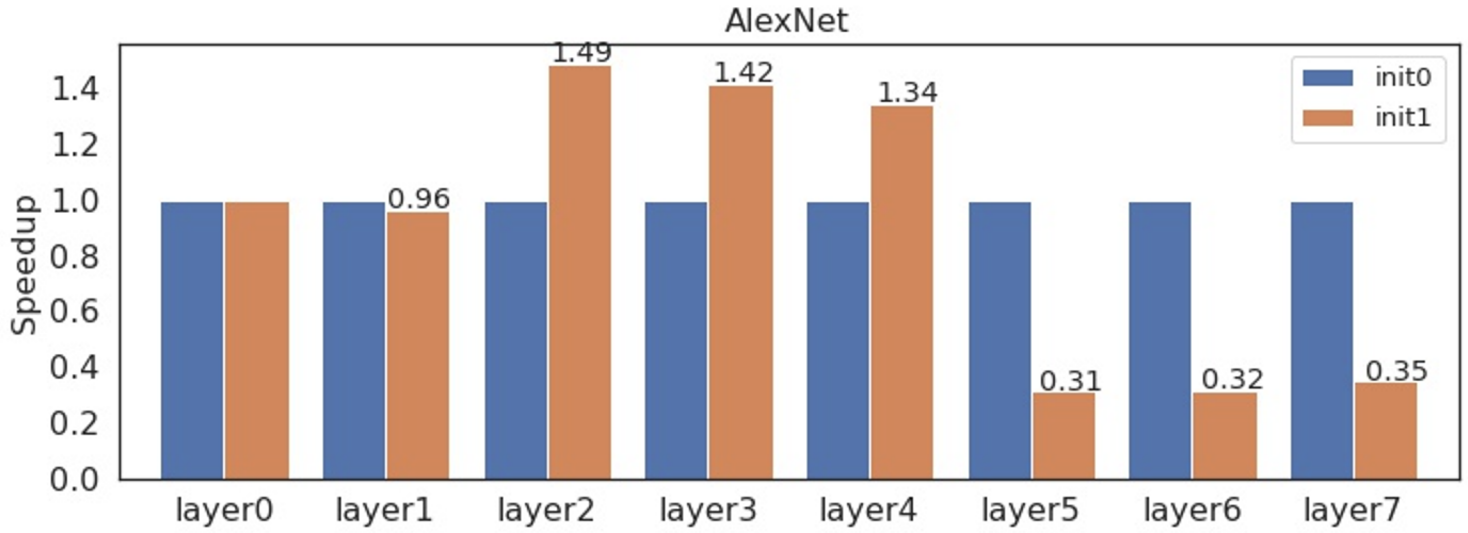
```
DRAM [ Weights:147456 Inputs:115200 Outputs:100352 ]
-----------------------------------------------------
| for P in [0:4)
|   for S in [0:3)
|     for C in [0:16) (Spatial-X)
InputBuffer [ Inputs:2016 ]
---------------------------
|       for N in [0:1)
|         for R in [0:3) (Spatial-X)
WeightBuffer [ Weights:1024 ]
-----------------------------
|           for Q in [0:28)
|             for P in [0:7)
AccumulationBuffer [ Outputs:128 ]
-----------------------------------
|               for K in [0:128)
|                 for C in [0:8)
Registers [ Weights:1 ]
-----------------------
|                   for N in [0:1)
```

Tiling Factors

A. Tiling Factors: Greedy Algorithm



Observation 1: Greedy Algorithm Initialization Matters



Up to 49% Performance Difference

A. Tiling Factors

Observation 1:

The initial order to exclude the prime factors affects the greedy results

Solution:

1. Mixed Integer Linear Programming (MILP)
    a. Communication-optimal Algo*
    b. Prob-driven Algo

Schedule:

```
DRAM [ Weights:147456 Inputs:115200 Outputs:100352 ]
-----------------------------------------------------
| for P in [0:4)
|   for S in [0:3)
|     for C in [0:16) (Spatial-X)
InputBuffer [ Inputs:2016 ]
---------------------------
|       for N in [0:1)
|         for R in [0:3) (Spatial-X)
WeightBuffer [ Weights:1024 ]
-----------------------------
|           for Q in [0:28)
|             for P in [0:7)
AccumulationBuffer [ Outputs:128 ]
-----------------------------------
|               for K in [0:128)
|                 for C in [0:8)
Registers [ Weights:1 ]
-----------------------
|                   for N in [0:1)
```

Tiling Factors

* Demmel, James, and Grace Dinh. "Communication-optimal convolutional neural nets." *arXiv preprint arXiv:1802.06905* (2018).

MILP: Problem-driven Formulation

Problem: R=3, S=3, P=28, Q=28, C=16, K=16, N=1

Prime Factors: [3], [3], [2, 2, 7], [2, 2, 7], [2, 2, 2, 2], [2, 2, 2, 2], [1]

Connection Variables: c_{ji}

Memory Levels: Reg | AccBuffer | WBuffer | InBuffer | GlobalBuffer | DRAM

Tiling Factors: 1 | 6 | 28 | 24 | 8 | 28

Objective: Maximize the overall utilization of the buffers

Op-level Scheduling

Tiling Factors: MILP

1. Setup:
   - 3 variables: weights, inputs, outputs. $v = 1, \ldots, 3$. v is the index to each variable.
   - 5 levels of memory hierarchies, each has $M_{i,v}$ bytes for each variable v, $i = 1, \ldots, 5$, $v = 1, \ldots, 3$.
   - 7 levels of loop nests to map to the memories, each has a loop bound B_j, $j = 1, \ldots, 7$.
   - Each of the loop bounds can be factorized into N_j prime factors $f_{j,n}$, $n = 1, \ldots, N_j$.
   - Loop level and variable correlation factors: $a_{j,v}$, $j = 1, \ldots, 7$, $v = 1, \ldots, 3$.
   - Variable and memory bypass factors: $b_{v,i}$, $v = 1, \ldots, 3$, $i = 1, \ldots, 5$. 0 is bypass, 1 is not bypass.
   - Prime factor and memory correlation factors: $c_{(j,n),i}$, $j = 1, \ldots, 7$, $n = 1, \ldots, N_j$, $i = 1, \ldots, 5$. 0 is correlated, 1 is not correlated.
   - Importance of each memory level utilization I_i, the higher the more important the utilization of memory i is.
1. Constraints:
   - Each prime factor $c_{(j,n),i}$ can only be put in one memory level
   - The memory utilization at each level i for variable v cannot exceed its capacity $M_{i,v}$.
2. Objectives:
   - Maximize the utilization of each memory level $\hat{U}$.

MILP: Problem-driven Formulation

4. Formulation:

$$\max \hat{U}$$

$$\text{s.t.} \quad \sum_{i=0}^{4} c_{(j,n),i} \leq 5, 0 \leq c_{(j,n),i}, \leq 1$$

$$U_{I,v} = \prod_{i=0}^{I-1} \prod_{j=0}^{6} \begin{cases} f_{j,n}, & c_{(j,n),i} a_{j,v} b_{v,i} = 1 \\ 1, & \text{otherwise} \end{cases}$$

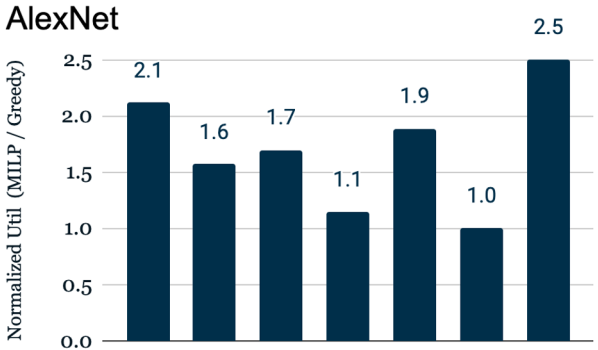
$$U_{I,v} \leq M_{I,v}$$

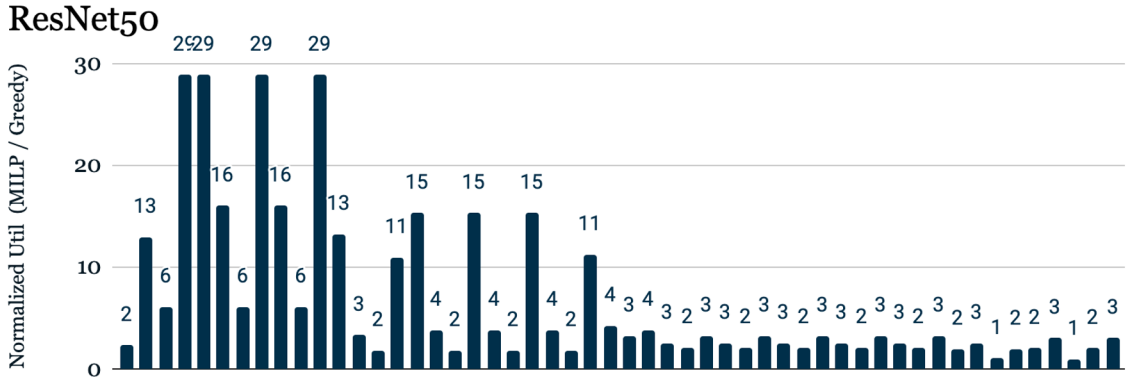
$$\hat{U} = \prod_{i=0}^{4} \prod_{v=0}^{2} U_{i,v} I_i$$

Take the log terms

Solve for variables: $c_{(j,n),i} \log(f_{j,n})$

MILP Attains Optimal Utilization





MILP achieves 1.3x and 4.2x better buffer utilization compared to greedy

Op-level Scheduling

B. Spatial Mapping

Observation 1:

Different NoC-level partitioning

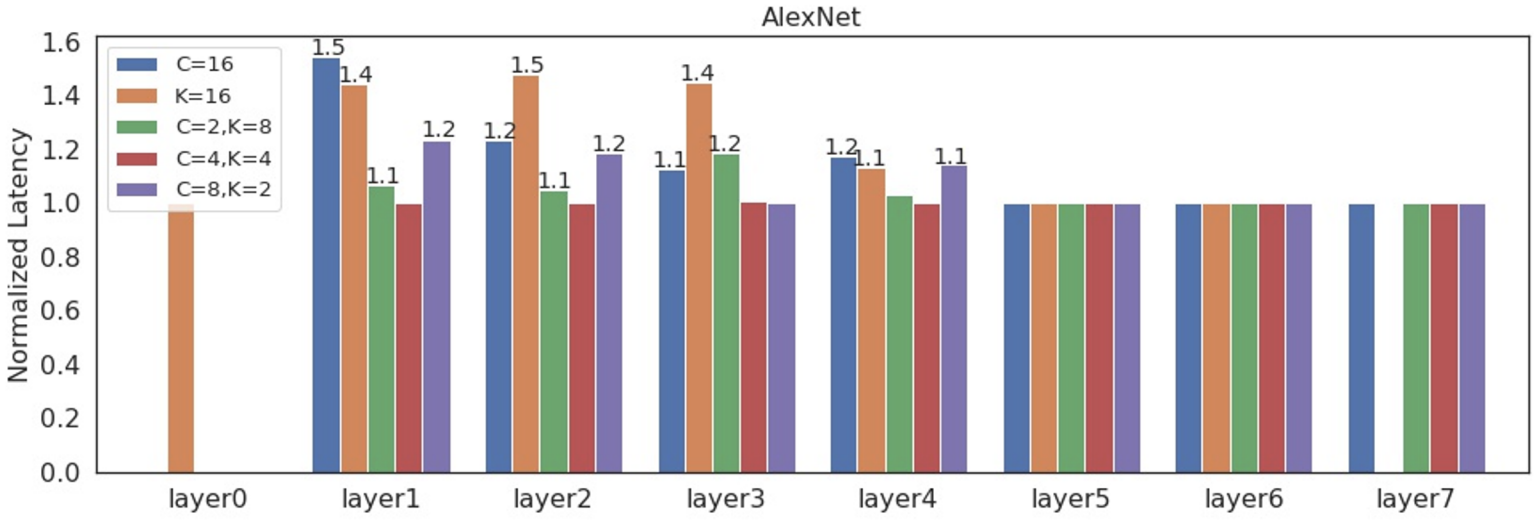
⟶ different performance

Output Schedule:

```
DRAM [ Weights:147456 Inputs:115200 Outputs:100352 ]
-----------------------------------------------------
| for P in [0:4)
|   for S in [0:3)
|     for C in [0:16) (Spatial-X)
InputBuffer [ Inputs:2016 ]
---------------------------
|       for N in [0:1)
|         for R in [0:3) (Spatial-X)
WeightBuffer [ Weights:1024 ]
-----------------------------
|           for Q in [0:28)
|             for P in [0:7)
AccumulationBuffer [ Outputs:128 ]
----------------------------------
|               for K in [0:128)
|                 for C in [0:8)
Registers [ Weights:1 ]
----------------------
|                   for N in [0:1)
```

Spatial Mapping

NoC-Level Spatial Partitioning Results



Performance for Different Spatial Partition + Greedy Algorithm

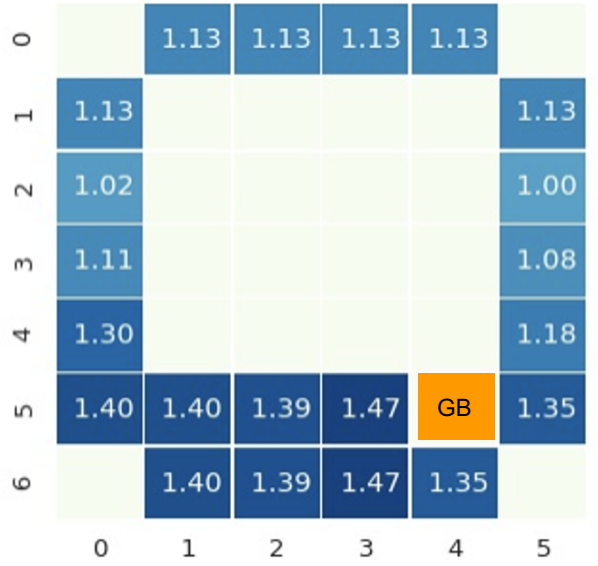
Up to 50% Performance Difference

B. Spatial Mapping

Observation 2:

Different DRAM locations

⟶ different performance



Up to 50% difference

DRAM location

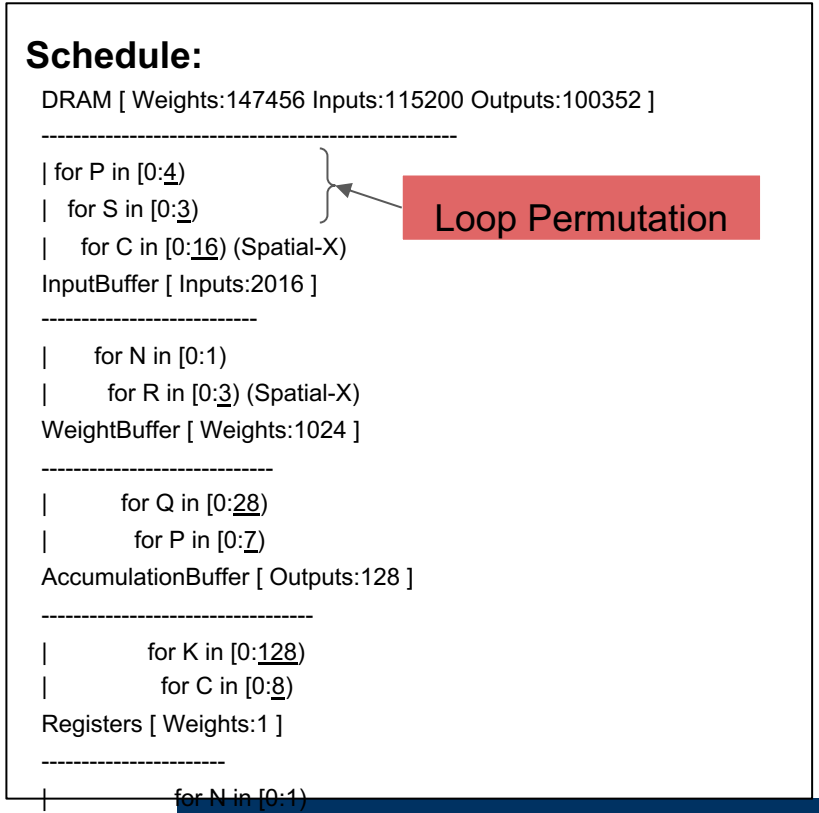
C. Loop Permutation

Observation 1:

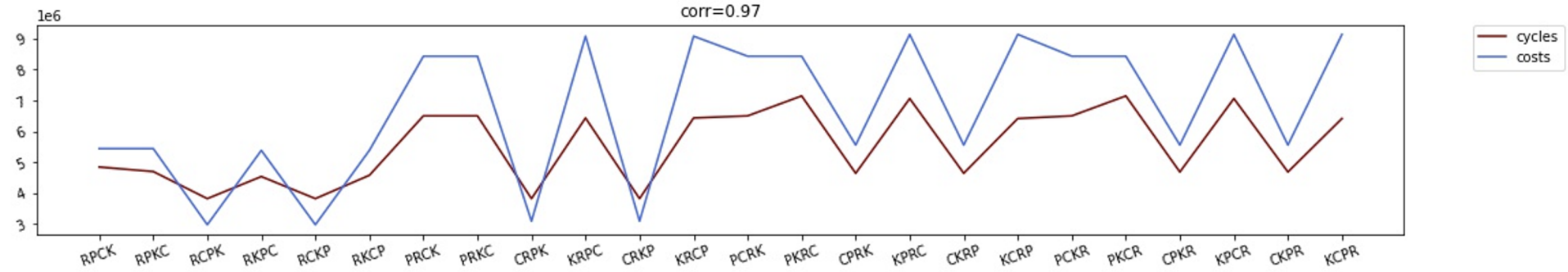
With A and B fixed,

Less NoC Traffic

⟶ better Performance



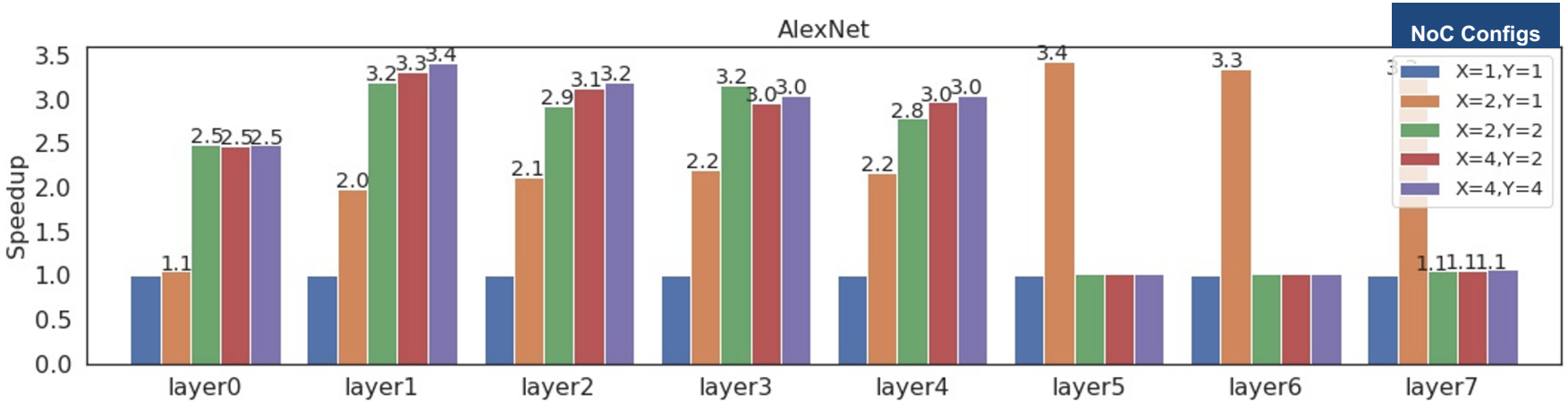
Observation 1: Latency is correlated to NoC Traffic



Different temporal permutation

High Correlation between Latency and NoC Traffic Cost

Observation: Strong Scaling Performance



Configurations:

- **Different # of PEs in different shape**

Schedul

A. Tiling Factors: Greedy
B. Spatial Mapping: Exhaustive Search
C. Loop Permutation:
Exhaustive Search with Cost Model

Not more PEs leads to better performance

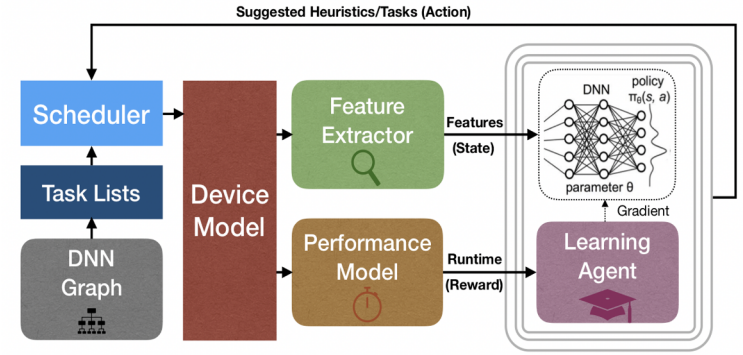
Op-level Scheduling

NoC Scheduling for DNNs

1. Motivation
2. Problem Definition
3. Infrastructure
4. Scheduling Algorithms
5. Future Work

Future Work

- Operational-Level Scheduling
  - Spatial Mapping
  - Loop Permutation
- Coscheduling
  - Multi-layer
  - Multi-network
- More Dynamic DNN Workloads
- Optimization + RL/ML



Questions?

Email: qijing.huang@berkeley.edu

Thanks

- Fellow Researchers:
  - Josh Kang (Timeloop)
  - Aravind Kalaiah (NoC)
  - Thomas Norell (Simulation Parallelization)
  - Grace Dinh (MILP)
  - Prof. James Demmel (MILP)

Thanks!