Lecture 6:

Compilers for Accelerators

Sitao Huang

sitaoh@uci.edu

February 15, 2022

Logistics

- Please submit your project proposal ASAP
  - Options: (a) literature review paper or (b) compiler + accelerator project

- Project option (a) or option (b)
  - Reasonable workload for 3 weeks
  - Workload will be considered in evaluation

Tentative Schedule

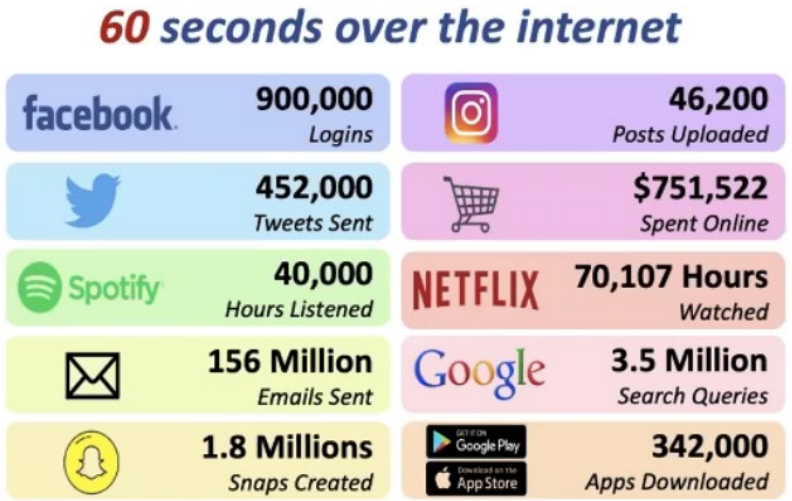
- **Week 1** (1/4, 1/6): Course Introduction
- **Week 2** (1/11, 1/13): Hardware Accelerators
- **Week 3** (1/18, 1/20): Language and Compiler Basics
- **Week 4** (1/25, 1/27): Reconfigurable Accelerators
- **Week 5** (2/1, 1/3): High-Level Synthesis
- **Week 6** (2/8, 2/10): *Midterm*
- **Week 7 (2/15, 2/17): Compilers for Accelerators**
- **Week 8** (2/22, 2/24): Machine Learning Compilers
- **Week 9** (3/1, 3/3): Emerging Architectures and Compilers
- **Week 10** (3/8, 3/10): *Project Presentations*

***PyLog*: An Algorithm-Centric Python-Based FPGA Programming and Synthesis Flow**

https://github.com/hst10/pylog

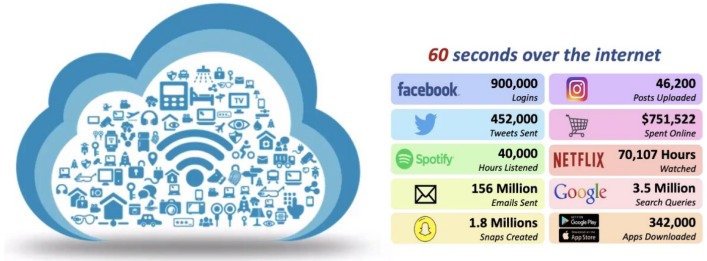
https://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=9591456

Challenges in Modern Computing

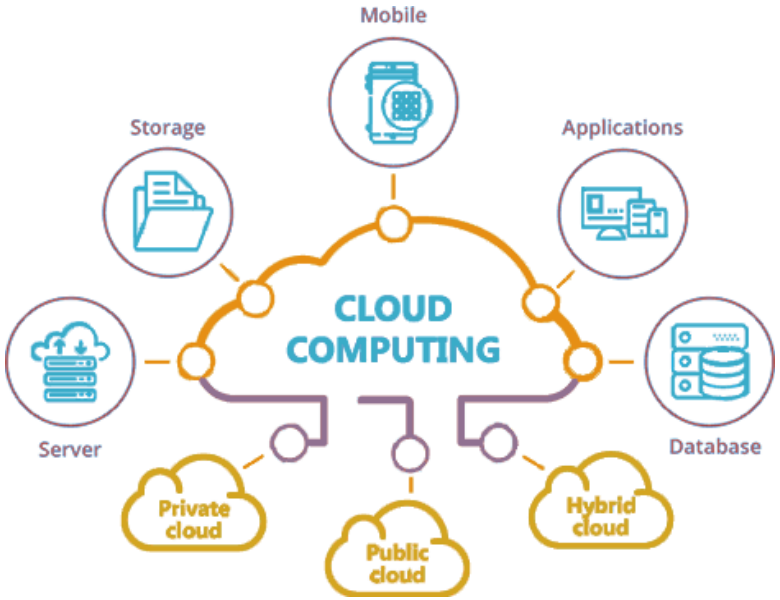


Complexity in Data

Challenges in Modern Computing



Complexity in Data

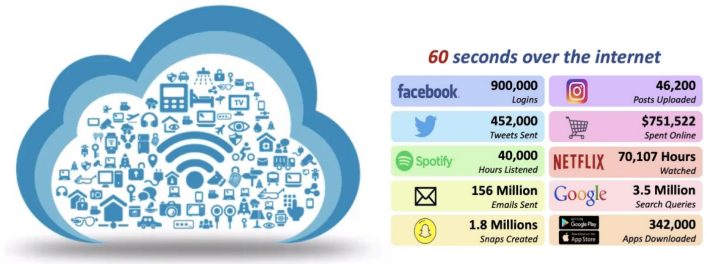


Cloud Computing

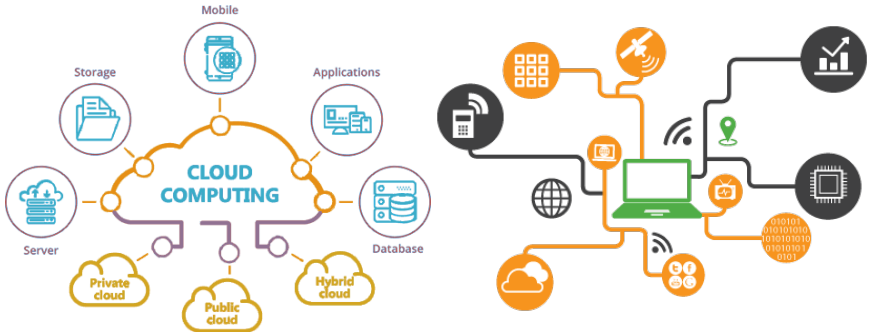
Edge Computing

Complexity in Applications

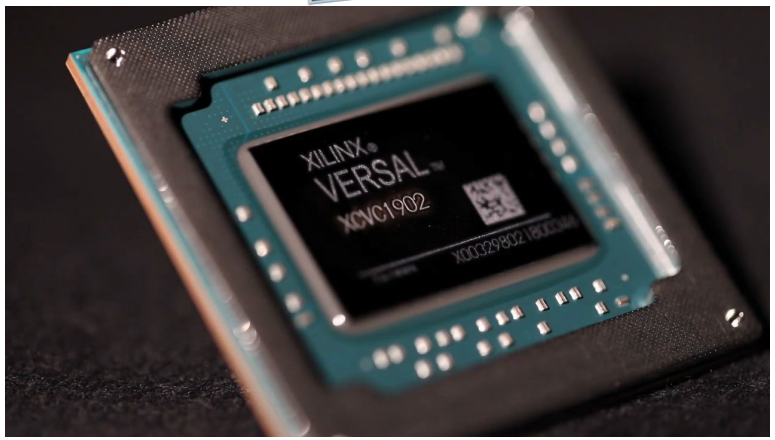
Challenges in Modern Computing

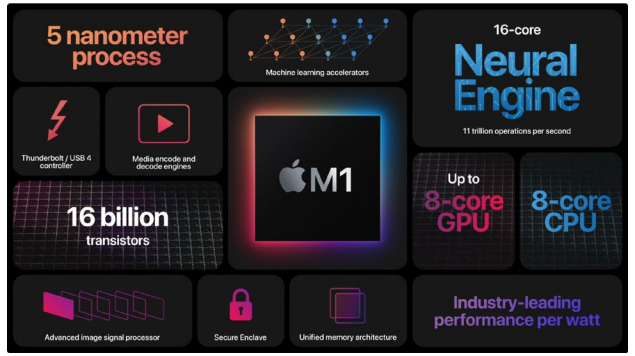


Complexity in Data



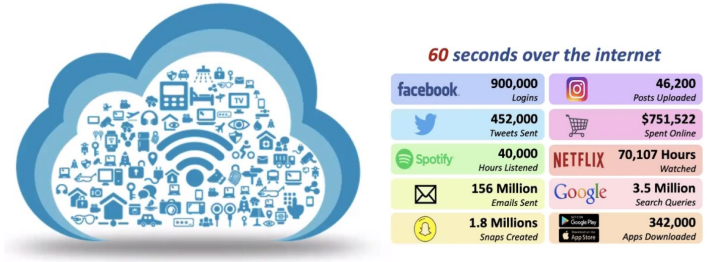
Complexity in Applications



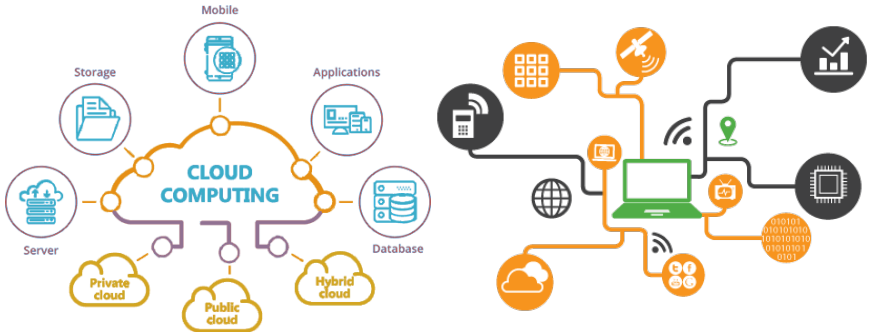


Complexity in Hardware

Challenges in Modern Computing

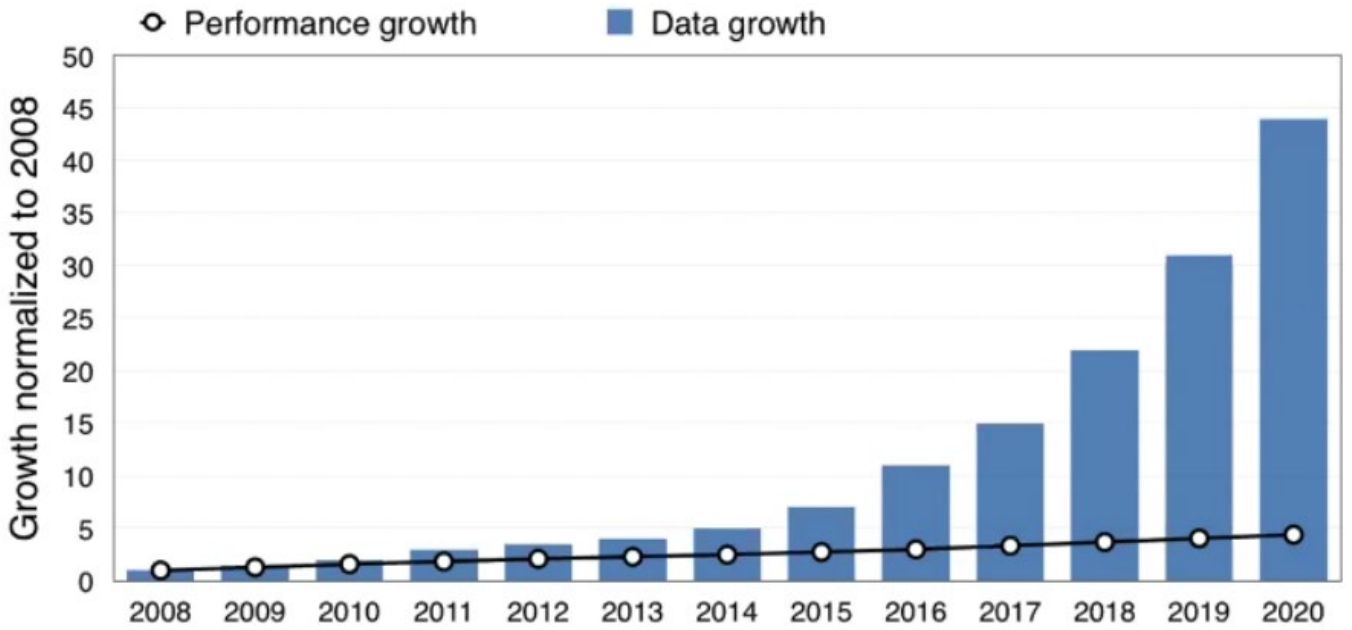


Complexity in Data



Complexity in Applications

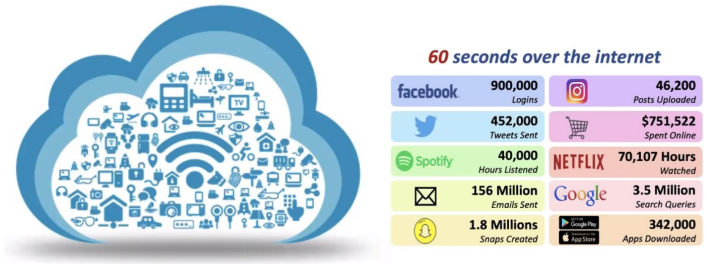
Complexity in Hardware



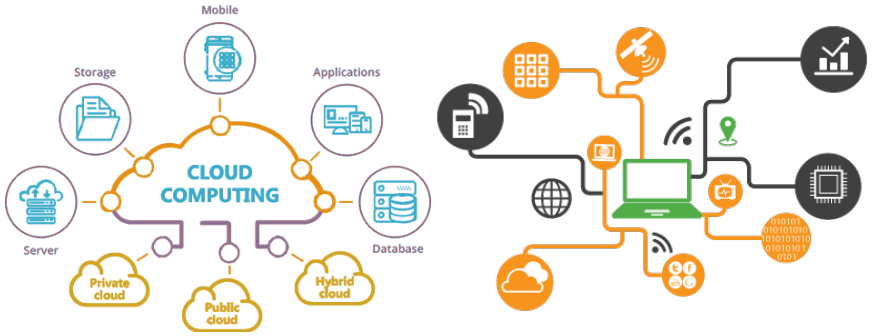
Data growth trends: IDC's Digital Universe Study, December 2012
Performance growth trends: Hadi Esmaeilzadeh et al, "Dark Silicon and the End of Multicore Scaling", ISCA 2011

Huge gap between data growth and processor performance growth!

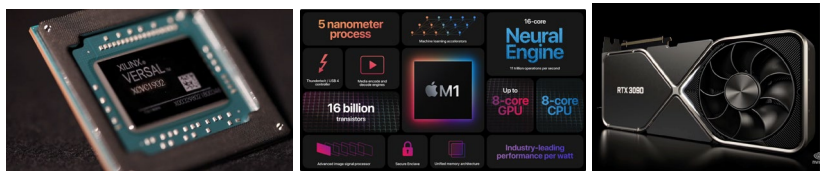
Challenges in Modern Computing



Complexity in Data



Complexity in Applications

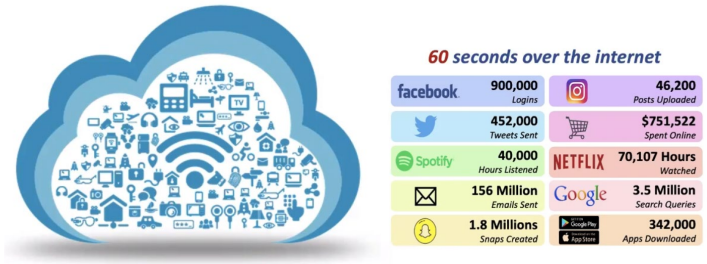


Complexity in Hardware

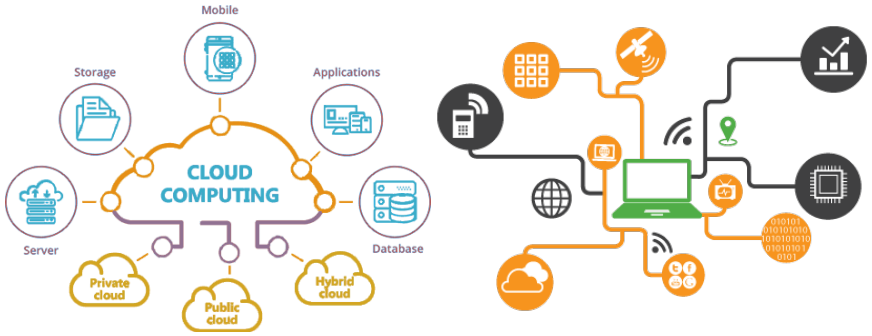
We need efficient acceleration systems!

Data growth trends: IDC's Digital Universe Study, December 2012
Performance growth trends: Hadi Esmaeilzadeh et al, "Dark Silicon and the End of Multicore Scaling", ISCA 2011

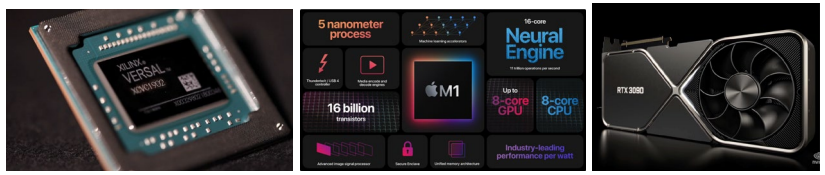
Challenges in Modern Computing



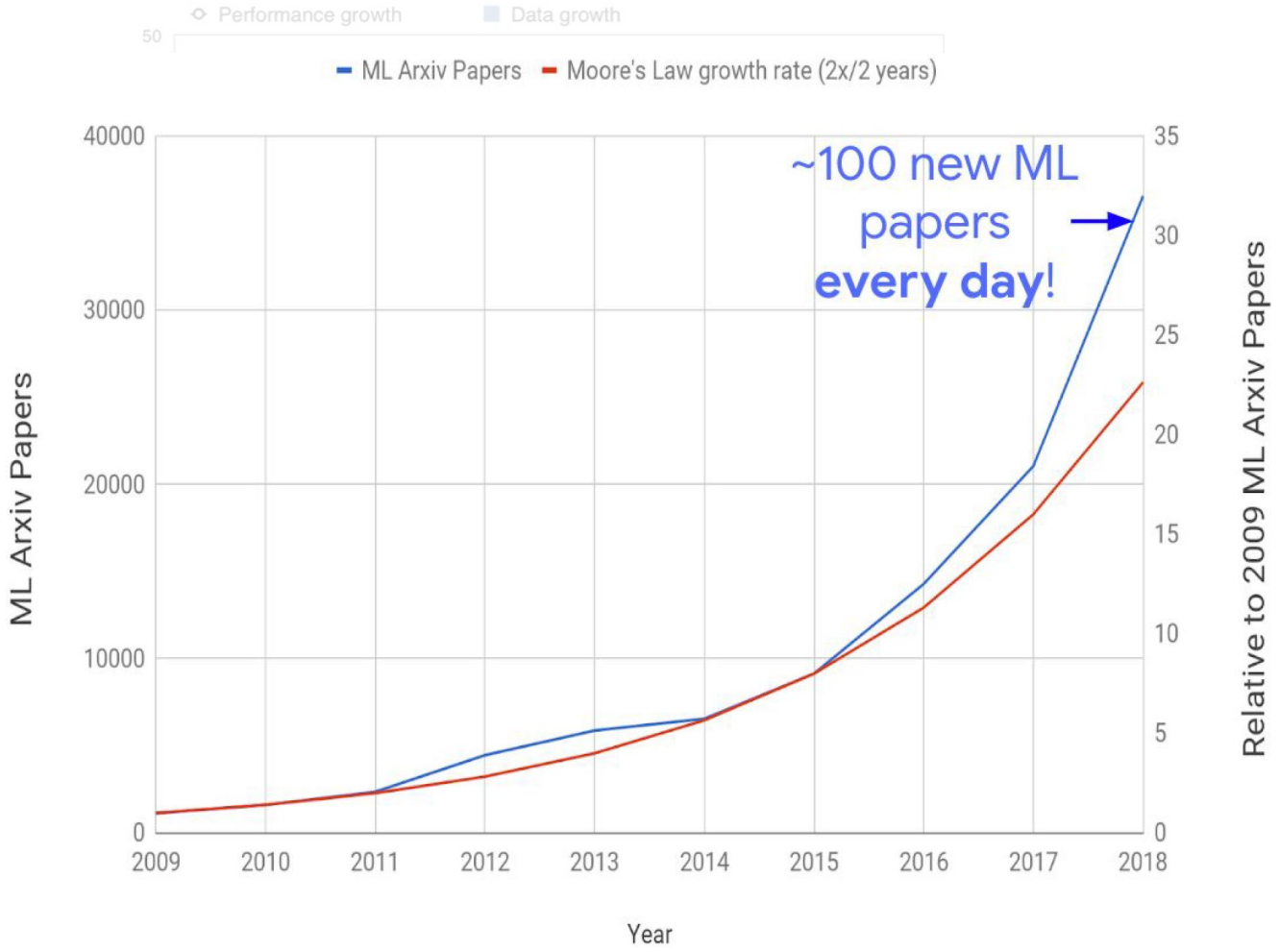
Complexity in Data



Complexity in Applications

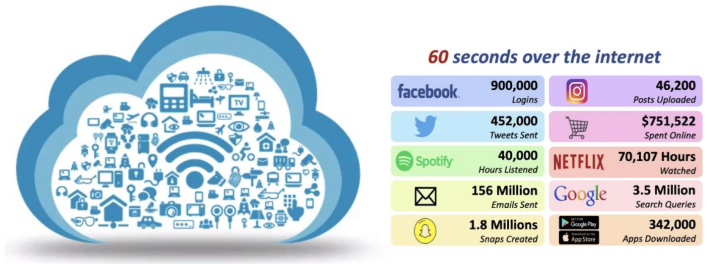


Complexity in Hardware

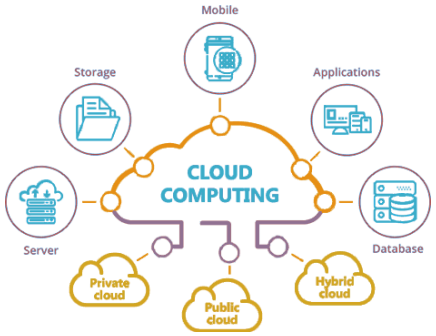


Jeff Dean. The Deep Learning Revolution and Its Implications for Computer Architecture and Chip Design, arXiv 1911.05289.

Challenges in Modern Computing



Complexity in Data

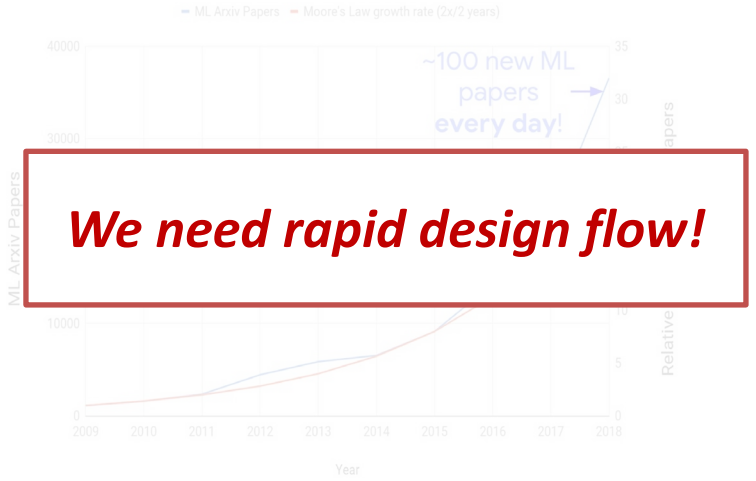


Complexity in Applications

Complexity in Hardware

We need efficient acceleration systems!

Data growth trends: IDC's Digital Universe Study, December 2012
Performance growth trends: Hadi Esmaeilzadeh et al, "Dark Silicon and the End of Multicore Scaling", ISCA 2011

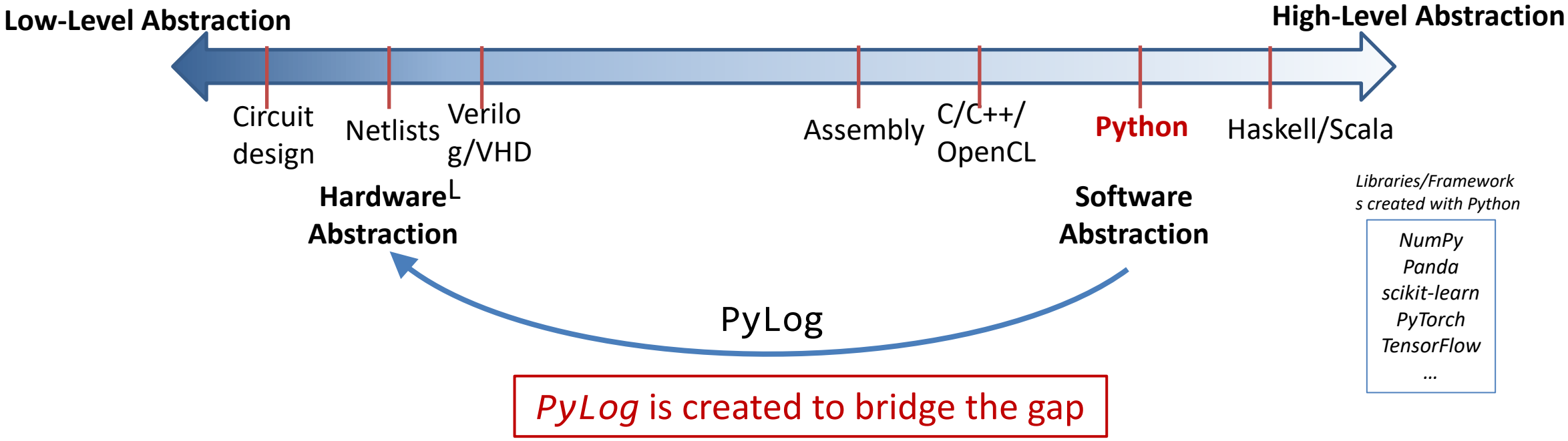


We need rapid design flow!

Jeff Dean. The Deep Learning Revolution and Its Implications for Computer Architecture and Chip Design, arXiv 1911.05289.

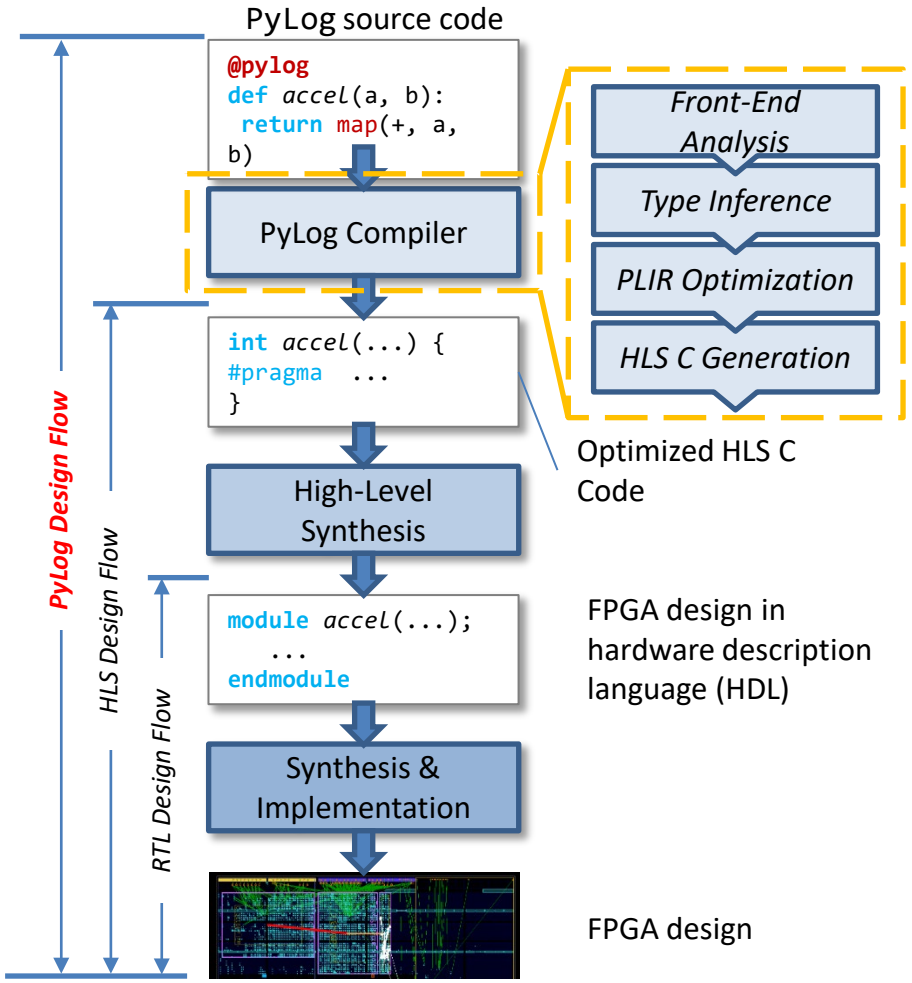
Gap between Hardware and Software Abstraction Levels

- Huge *gap* between *hardware* abstraction levels and *software* abstraction levels
- *Large* number of applications/frameworks are developed with *high-level* languages
- Many applications in high-level abstraction need *hardware acceleration*
- *Very challenging* to accelerate high-level applications due to abstraction gap



PyLog: High-Level Programming and Synthesis Flow for FPGAs

FPGA Programming is moving towards higher abstractions



High-level synthesis (HLS) flow greatly improves design productivity:

- Fewer lines of code (LOC)
- Easier system integration (e.g. cloud/edge FPGAs)
- Easier design space search, explore more alternative designs, achieve better QoR
- Easier design migration
- …

Benefits of using high-level abstraction (C language)

Need high-level programming and synthesis flow!

Some drawbacks:

- Limitations in expressing concurrency with C semantics
- Conservative compiler optimizations due to the lack of high-level computation pattern information
- Highly performant code requires careful tuning and LOC increases quickly (e.g. 8,000 lines of HLS C code for optimized convolution kernel)
- …

Need languages with high-level operators (high-level languages)

Need powerful compiler optimizations

Easier to achieve with high-level languages

PyLog Flow Overview

PyLog Code

```
import numpy as np
from pylog import *
@pylog                                  accelerator
def add(a, b):
  b = a + b
  b = map(lambda x, y: x+y, a, b)

if __name__ == "__main__":
  a = np.array([1, 3, ...])
  b = np.array([8, 9, ...])
  print(add(a, b))                      host
```

vecadd.py

PyLog Code Generator

- Analysis
- Type Inference
- Optimization
- Code Generation

HLS C Code

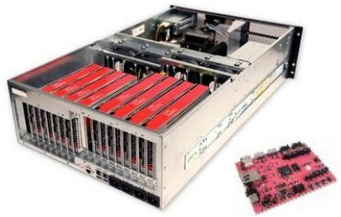
```
void accel(int32* a, int32* b,
int32* c, int32* d, int32* e) {
  for(int i = 0; i < 32; i++) {
    for (ii...)
    #pragma HLS ...
  }
  ...
}
```

PyLog System Generator

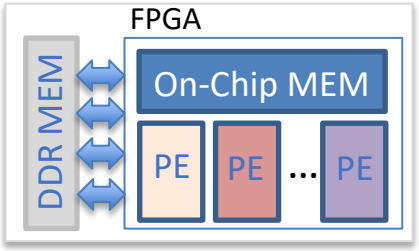
- Tcl Script
- IP Generation
- System Integration
- Design Generation

PyLog Runtime

- FPGA Programming
- Mem Allocation
- Start Accelerator
- Collect Results



Target Systems



System Design

```
>>> python3 vecadd.py
[10, 15, ...]
>>>
```

Execution Results

Synthesis Flow

Deployment Flow

https://github.com/hst10/pylog

PyLog Language and Compiler Highlights

High-level operators, design space exploration

1D Conv:
```
y = map(lambda a: a[-1]+a[0]+a[1], x[1:-1])
```
2D Conv:
```
y = map(lambda a: dot(a[-1:2,-1:2], w), img[1:-1, 1:-1])
```

Multiple possible C code versions from high-level code

```
for (...) {
#pragma HLS
unroll
}
```

```
for (...) {
#pragma HLS unroll factor=4
}
```

```
for (...) {
#pragma HLS
pipeline
}
```

PyLog chooses the optimal implementation given the HW resource of the target platform

Unified FPGA and host programming

```
import numpy as np
from pylog import *

@pylog
def add(a, b):
    return a + b

if __name__ == "__main__":
    a = np.asarray([1, 3, 6, 7])
    b = np.asarray([8, 9, 10,5])

    print(add(a, b))
```

accelerator

host

Python statements, compute customization

```
@pylog
def pl_matmul(a, b, c, d):

  buf = np.empty([16, 16], pl_fixed(8, 3))
  pragma("HLS array_partition variable=buf")

  def matmul(a, b, c):
    for i in range(32):
      for j in range(32).unroll(4):
        tmp = 0.0
        for k in range(32).pipeline():
          tmp += a[i][k] * b[k][j]
        c[i][j] = tmp
```

Type inference and type checking

- No explicit type annotation required
- Top function has NumPy arrays as arguments, which carries input type and shape information
- Type engine infers type and shape of each object in the code

```
y = map(lambda a: dot(a[-1:2,-1:2], w), img)
```

pl_type: PLType(float, 0)
pl_shape: (0,)

pl_type: PLType(float, 2)
pl_shape: (3, 3)

pl_type: PLType(float, 2)
pl_shape: (27, 27)

pl_type: PLType(float, 0)
pl_shape: (0,)

pl_type: PLType(float, 2)
pl_shape: (3, 3)

(known)

(known)

pl_type: PLType(float, 2)
pl_shape: (27, 27)

PyLog High-Level Operations

- map operation: *map(f, x_0, x_1,...)*
  - Repeatedly apply function *f* to each element in x_0, x_1,...
  - Allow access neighbor elements inside function
- Dot operation: dot product of two arrays

Example: vector addition (a and b have same shape)

```
z = map(lambda a, b: a + b, x, y)
```

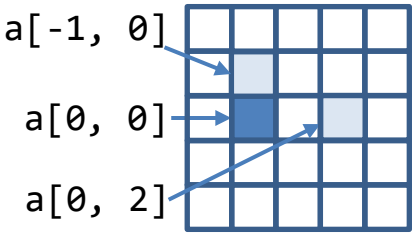
(a) *2D map*

PyLog

```
y = map(lambda a0, a1,…: op(a0[-1, 0], a1[0, 2],…), x0, x1,...)
```

HLS C

```
L1: for(int i1 = 0; i1 < x0.dim(0); i1++) {
L2:   for(int i2 = 0; i2 < x0.dim(1); i2++) {
        y[i1][i2] = op(x0[i1-1][i2], x1[i1][i2+2],...); }}
```



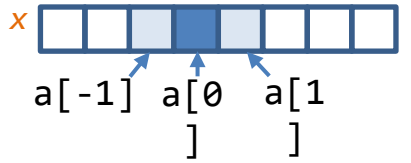
(b) *1D conv*

PyLog

```
y = map(lambda a: a[-1]+a[0]+a[1], x[1:-1])
```

HLS C

```
L1: for(int i1 = 1; i1 < x.dim(0)-1; i1++) {
      y[i1] = x[i1-1] + x[i1] + x[i1+1]; }
```



(c) *2D conv* (with multiple channels)

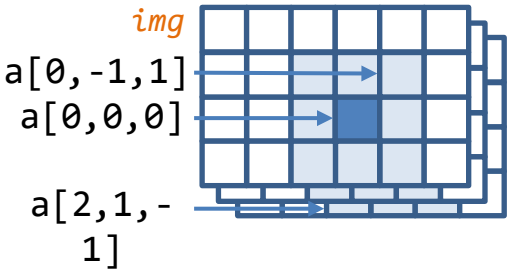
PyLog

```
y = map(lambda a: dot(a[:,-1:2,-1:2], w), img[0,1:-1,1:-1])
```

(HLS C code omitted for simplicity)

3x3 multiplication and reduction across ***all*** *channels*

Image without outermost pixels

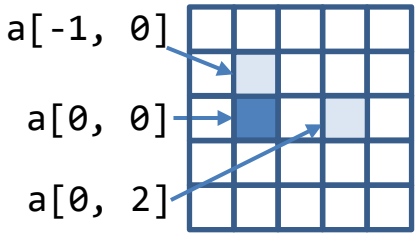


PyLog High-Level Operations

- map operation: *map(lambda a_0, a_1,…: …, x_0, x_1,…)*
  - Repeatedly apply lambda function to each element in x_0, x_1,…
  - Allow access neighbor elements inside lambda function
- Dot operation: dot product of two arrays

PyLog
```
y = map(lambda a0, a1,…: op(a0[-1, 0], a1[0, 2],…), x0, x1,...)
```

HLS C
```
L1: for(int i1 = 0; i1 < x0.dim(0); i1++) {
L2:   for(int i2 = 0; i2 < x0.dim(1); i2++) {
        y[i1][i2] = op(x0[i1-1][i2], x1[i1][i2+2],...); }}
```



Some other examples:

```
1  # Vector add
2  out = map(lambda x, y: x + y, vec_a, vec_b)
3
4  # 1D convolution
5  out = map(lambda x:x[-1]+x[0]+x[1], vec[1:-1])
6
7  # Inner product
8  out_vec[i] = dot(matrix[i, :], in_vec)
9
10 # Square matrix multiplication
11 out = map(lambda x,y: dot(x[0,:],y[:,0]), mat_a, mat_b)
```

PyLog Supported Platforms

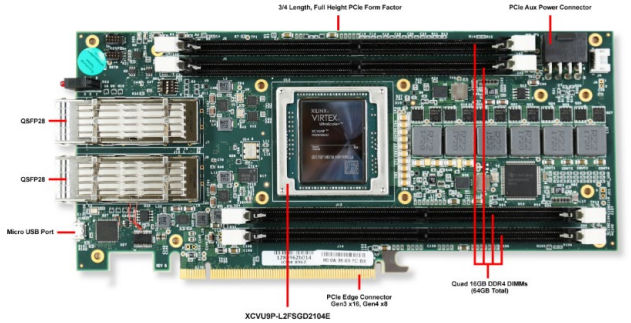
- Platforms supported by PyLog

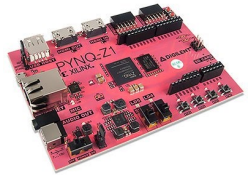
FPGA Platform Type	Platforms	Synthesis Flow	Runtime Library
SoCs and MPSoCs	ZedBoard, PYNQ, Ultra96	Vivado HLS + Vivado	PYNQ
PCIe-Based High-End FPGAs	Amazon EC2 F1 instance (XCVU9p), Alveo series (U200, U250, U280)	Vivado HLS + Vitis	PYNQ

- Design and deploy for different platforms easily
  - Simply change `@pylog` mode, no extra coding needed

Synthesize for AWS F1: `@pylog(mode='hwgen', board='aws_f1')`
Deploy on AWS F1: `@pylog(mode='deploy', board='aws_f1')`

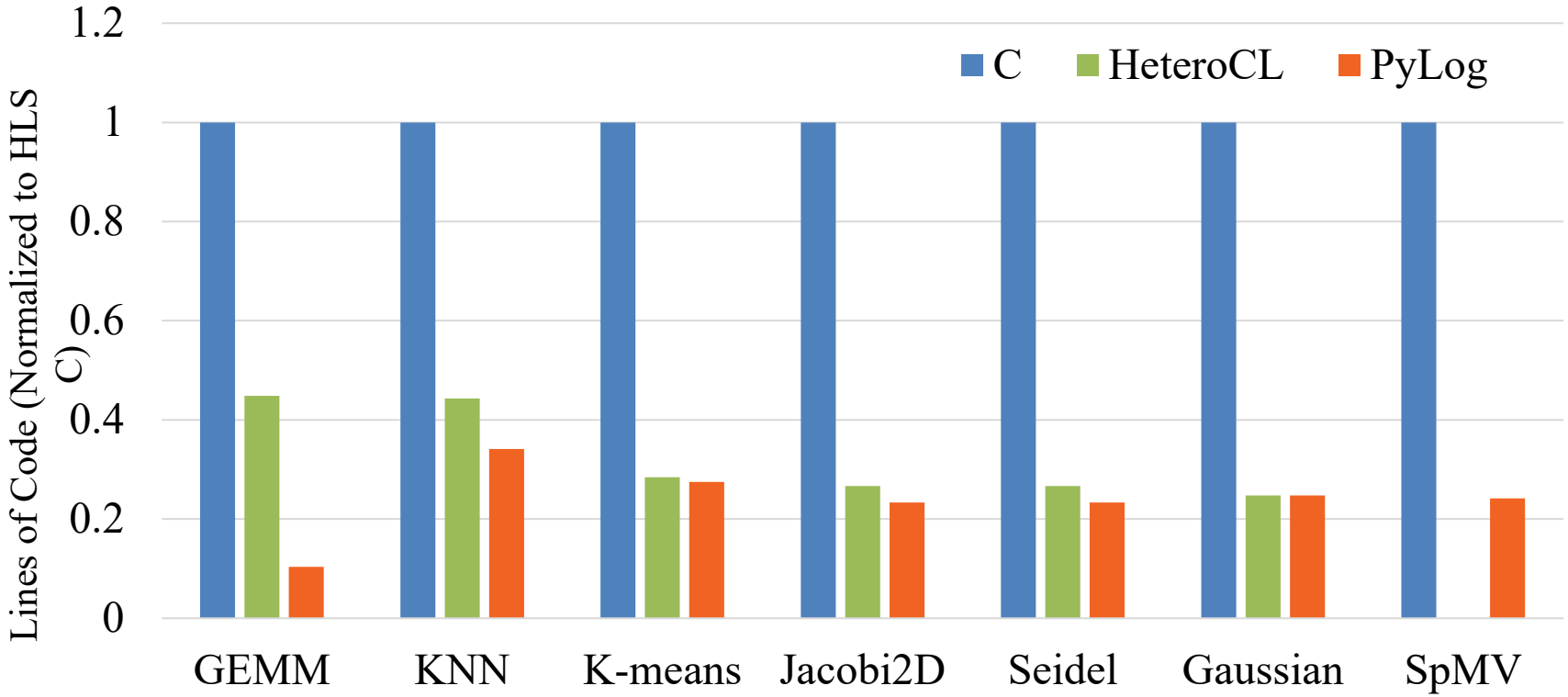
Synthesize for PYNQ: `@pylog(mode='hwgen', board='pynq')`
Deploy on PYNQ: `@pylog(mode='deploy', board='pynq')`





PyLog Evaluation: Expressiveness

- Compare LoC (Lines of Code) of HLS C code vs HeteroCL* code vs PyLog code
  - Normalized to HLS C LOC
  - **70%** reduction compared to HLS C code

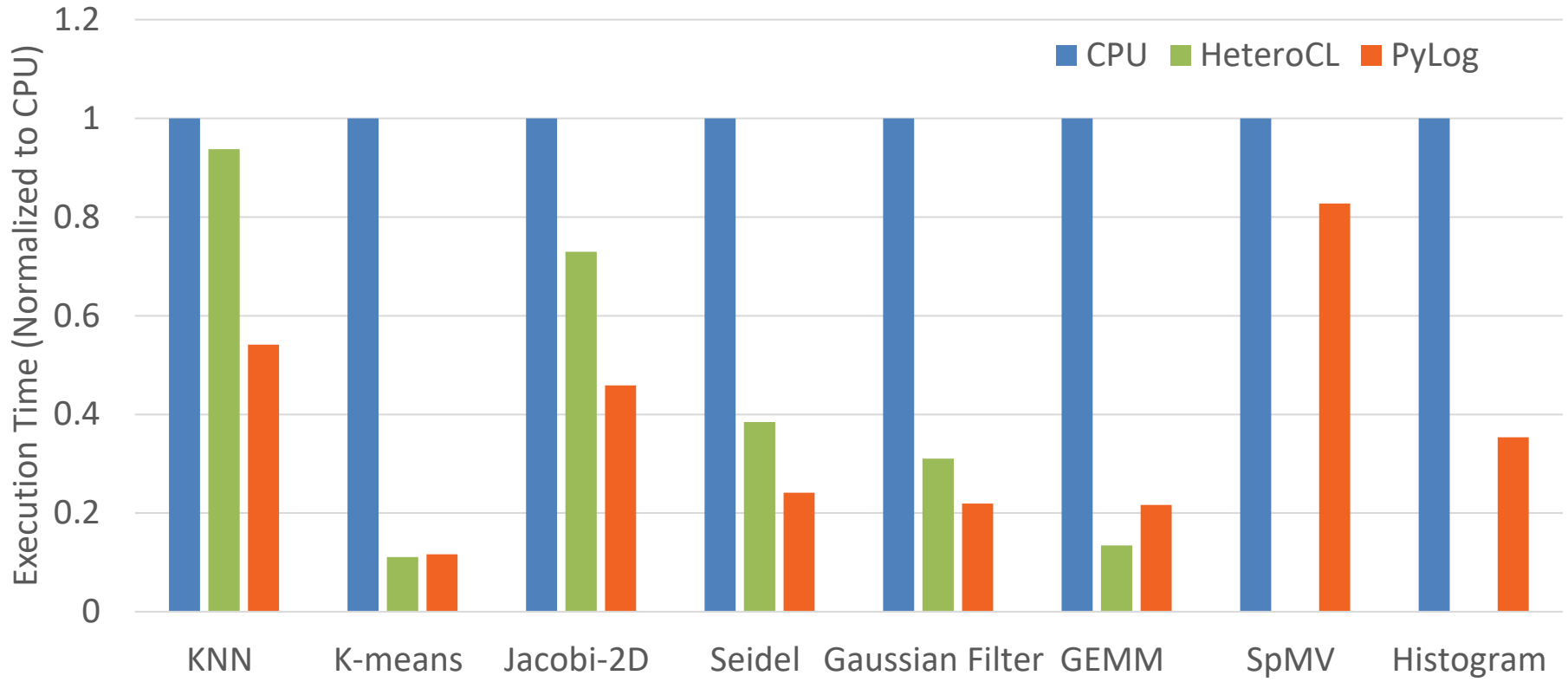


*Yi-Hsiang Lai, Yuze Chi, Yuwei Hu, Jie Wang, Cody Hao Yu, Yuan Zhou, Jason Cong, and Zhiru Zhang. HeteroCL: A Multi-Paradigm Programming Infrastructure for Software-Defined Reconfigurable Computing. FPGA 2019.

PyLog Evaluation: Accelerator Performance

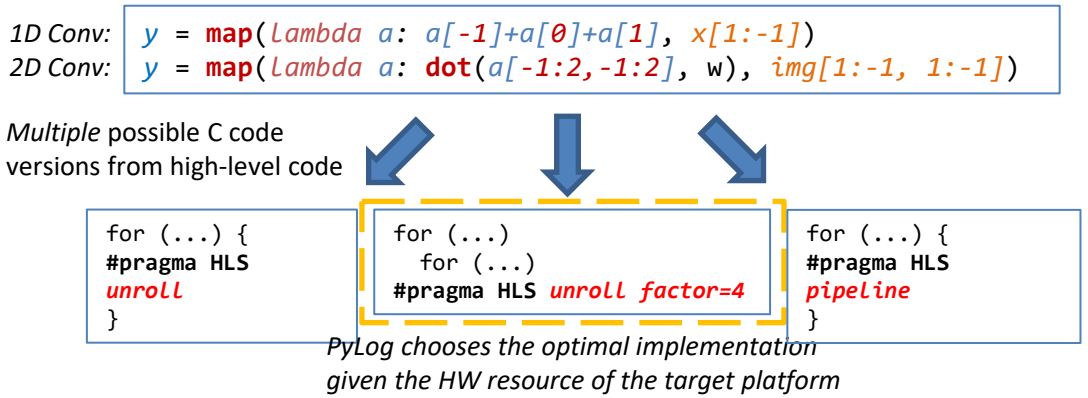
Compare accelerator performance vs CPU (C++) performance and HeteroCL accelerator performance

- Platform: Amazon AWS F1 instance (Xilinx Virtex UltraScale+ XCVU9P)
- **3.17x** and **1.24x** faster than optimized CPU and FPGA accelerators



CPU: Optimized CPU code running on single thread on 8-core Intel Xeon E5-2686 v4 CPU

Operator Implementation Selection



Applied to:

- *N*-dimensional array operations, e.g. c = a + b
- PyLog high-level operations, e.g. map, dot, reduce, etc.

PyLog IR:

- Represent the transformations in IR systematically
- Represent performance and cost of each version

PyLog internal:

- *PyLog schedule: PLSchedule*
- Sequence of transformations applied to loop nests, subscripts, etc.
- Definition:

```
schedule = [ (action1, arg11, arg12, …),
             (action2, arg21, arg22, …) ]
```

- Example:

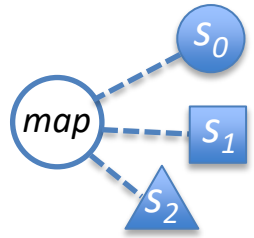
```
schedule = [ ('interchange', 1, 3),
             ('tile', 3, 4) ]
```

- Example (with parameters):

```
schedule = [ ('interchange', 1),
             ('tile', 3) ]
```

- Extensible: action_object are predefined Python functions to apply action to object, e.g. `interchange_PLSubscript`, `tile_list`, etc.

Each high-level operations will have a list of candidate schedules, `schedules = [s0, s1, s2, …]`



Operator Implementation Selection

2D Add:

```
c = a + b
```

Plain

```
for1 (...)
  for2 (...)
```

Interchange 1, 2

```
for2 (...)
  for1 (...)
```

Interchange 1, 2
Tile 2

```
for2 (...)
  for11 (...)
    for12 (...)
```

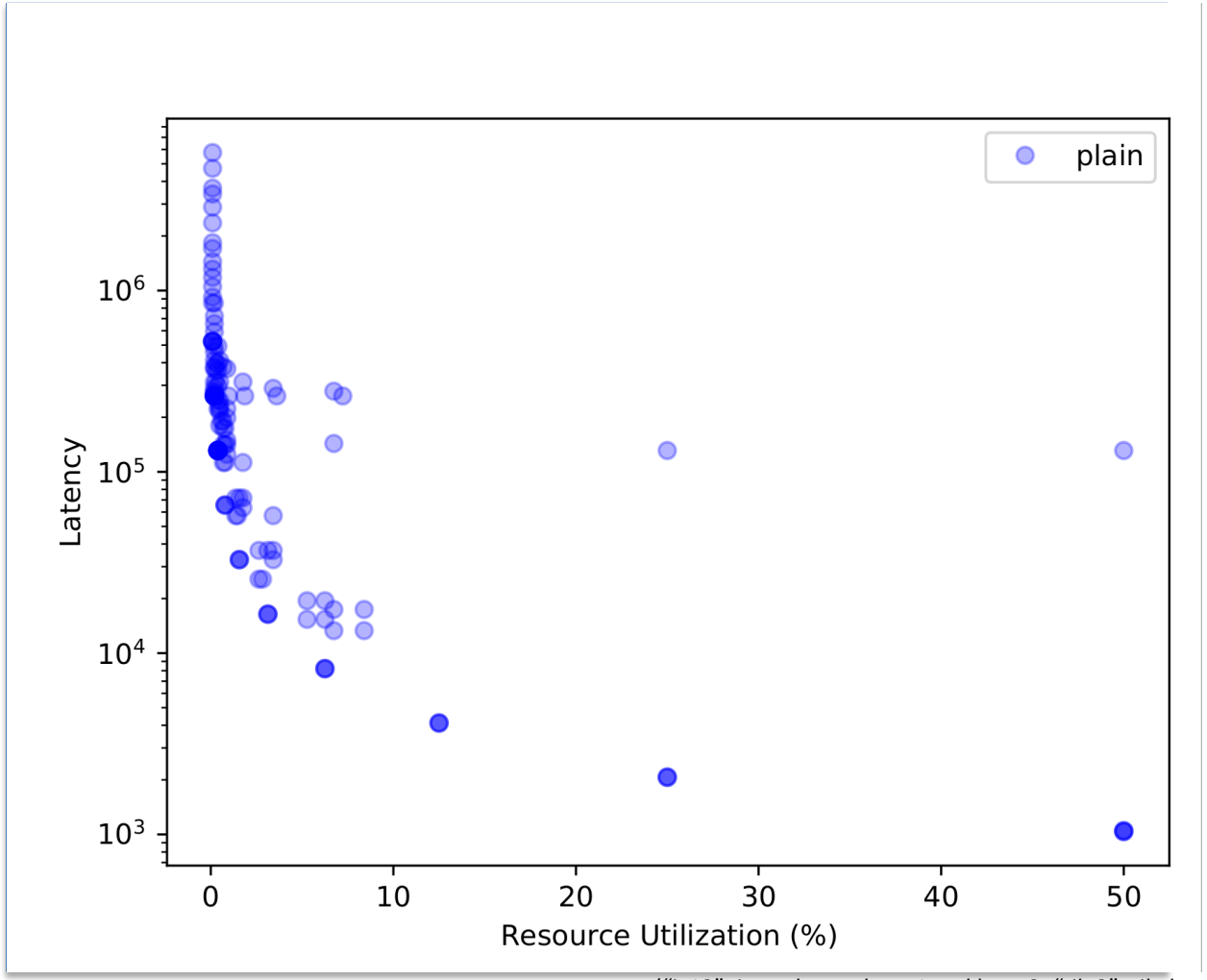
Interchange 1, 2
Tile 2
Tile 1

```
for21 (...)
  for22 (...)
    for11 (...)
      for12 (...)
```

Interchange 1, 2
Tile 2
Tile 1
Interchange 2, 3

```
for21 (...)
  for11 (...)
    for22 (...)
      for12 (...)
```

Design space of various code versions of 2D array addition.



("ic12": interchange loop 1 and loop 2; "tile2": tile loop 2)

Operator Implementation Selection

2D Add:

```
c = a + b
```

Plain

```
for1 (...)
  for2 (...)
```

Interchange 1, 2

```
for2 (...)
  for1 (...)
```

Interchange 1, 2
Tile 2

```
for2 (...)
  for11 (...)
    for12 (...)
```

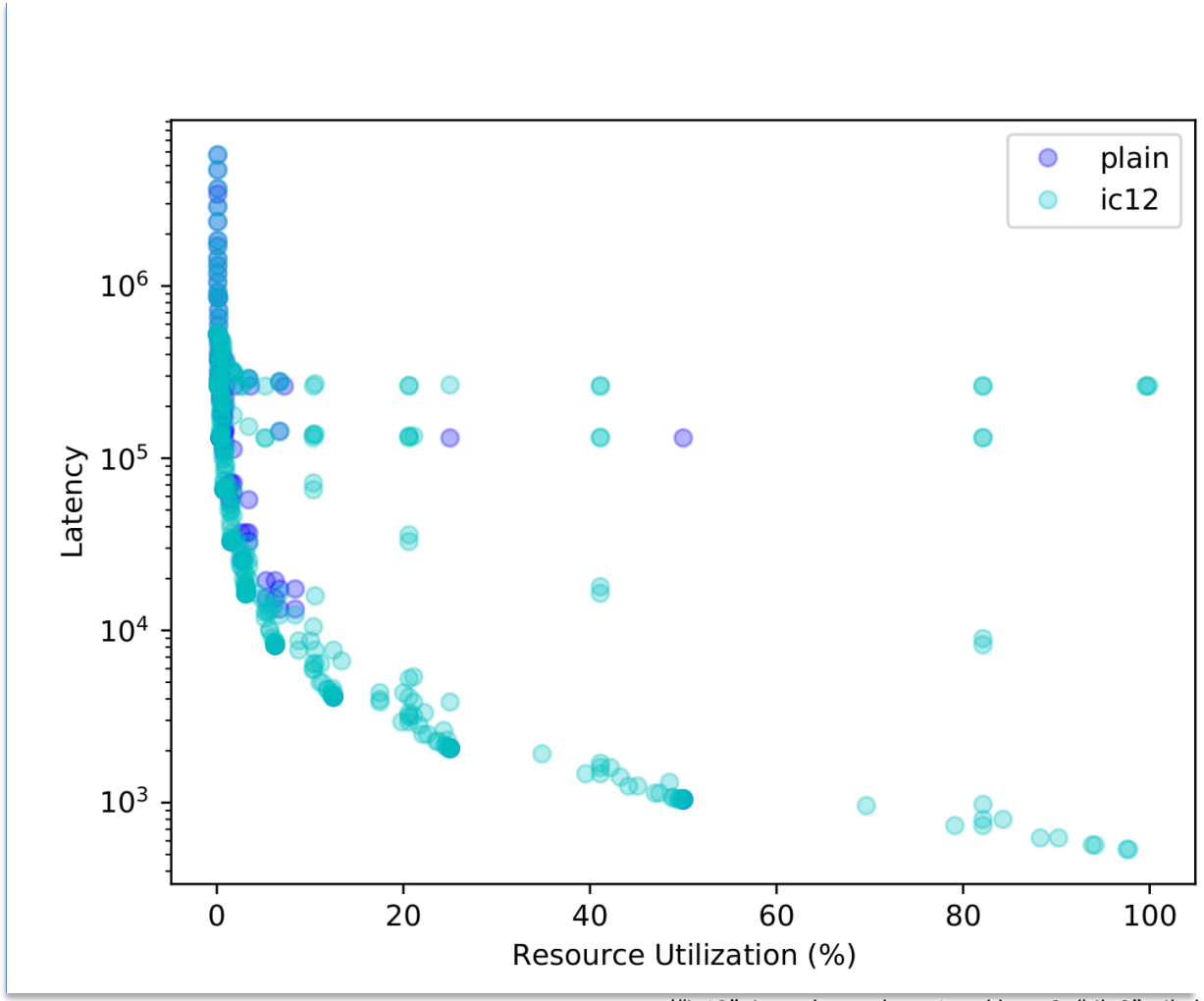
Interchange 1, 2
Tile 2
Tile 1

```
for21 (...)
  for22 (...)
    for11 (...)
      for12 (...)
```

Interchange 1, 2
Tile 2
Tile 1
Interchange 2, 3

```
for21 (...)
  for11 (...)
    for22 (...)
      for12 (...)
```

Design space of various code versions of 2D array addition.



("ic12": interchange loop 1 and loop 2; "tile2": tile loop 2)

Operator Implementation Selection

2D Add:

```
c = a + b
```

Plain

```
for1 (...)
  for2 (...)
```

Interchange 1, 2

```
for2 (...)
  for1 (...)
```

Interchange 1, 2
Tile 2

```
for2 (...)
  for11 (...)
    for12 (...)
```

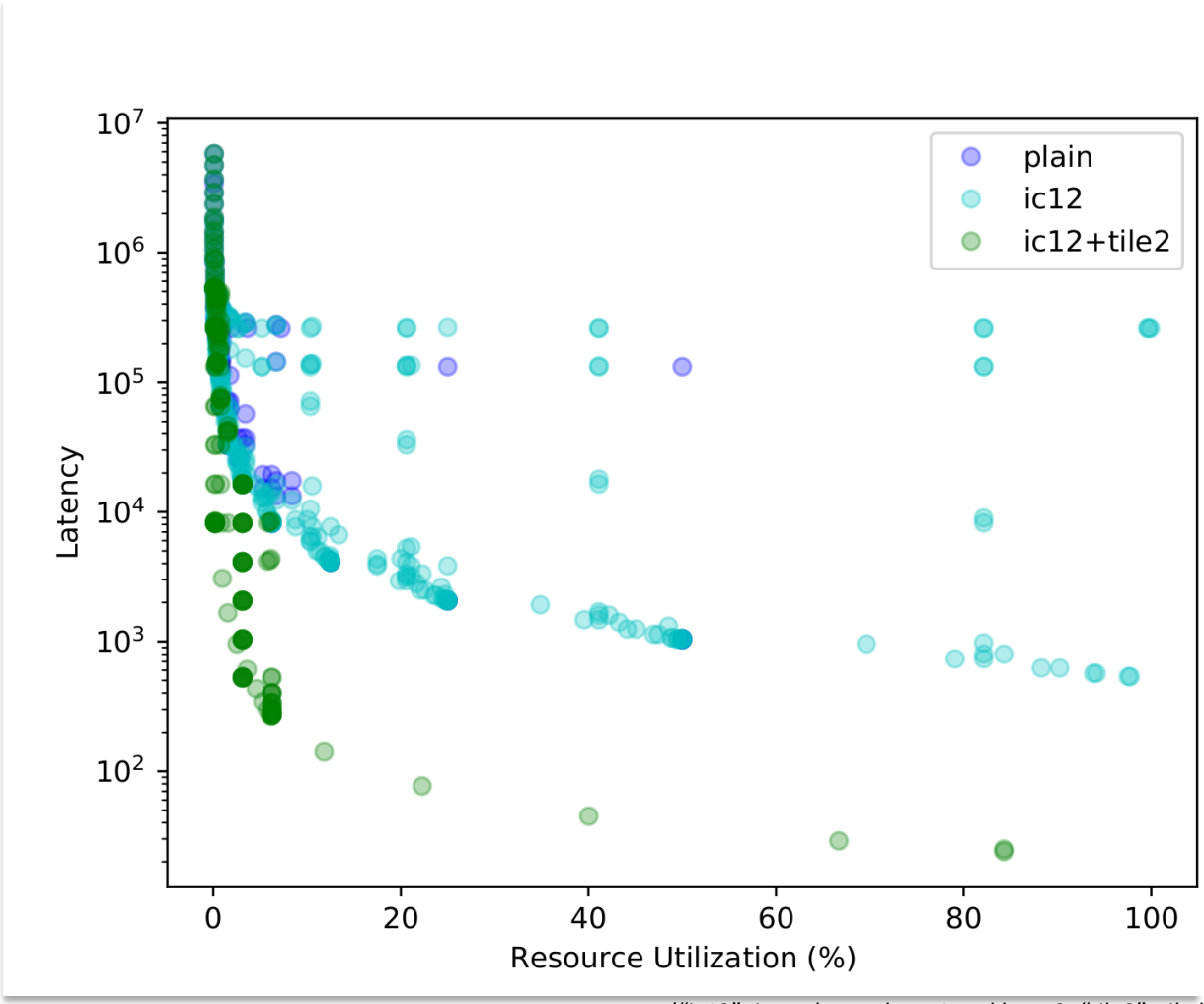
Interchange 1, 2
Tile 2
Tile 1

```
for21 (...)
  for22 (...)
    for11 (...)
      for12 (...)
```

Interchange 1, 2
Tile 2
Tile 1
Interchange 2, 3

```
for21 (...)
  for11 (...)
    for22 (...)
      for12 (...)
```

Design space of various code versions of 2D array addition.



("ic12": interchange loop 1 and loop 2; "tile2": tile loop 2)

Operator Implementation Selection

2D Add:

```
c = a + b
```

Plain

```
for1 (...)
  for2 (...)
```

Interchange 1, 2

```
for2 (...)
  for1 (...)
```

Interchange 1, 2
Tile 2

```
for2 (...)
  for11 (...)
    for12 (...)
```

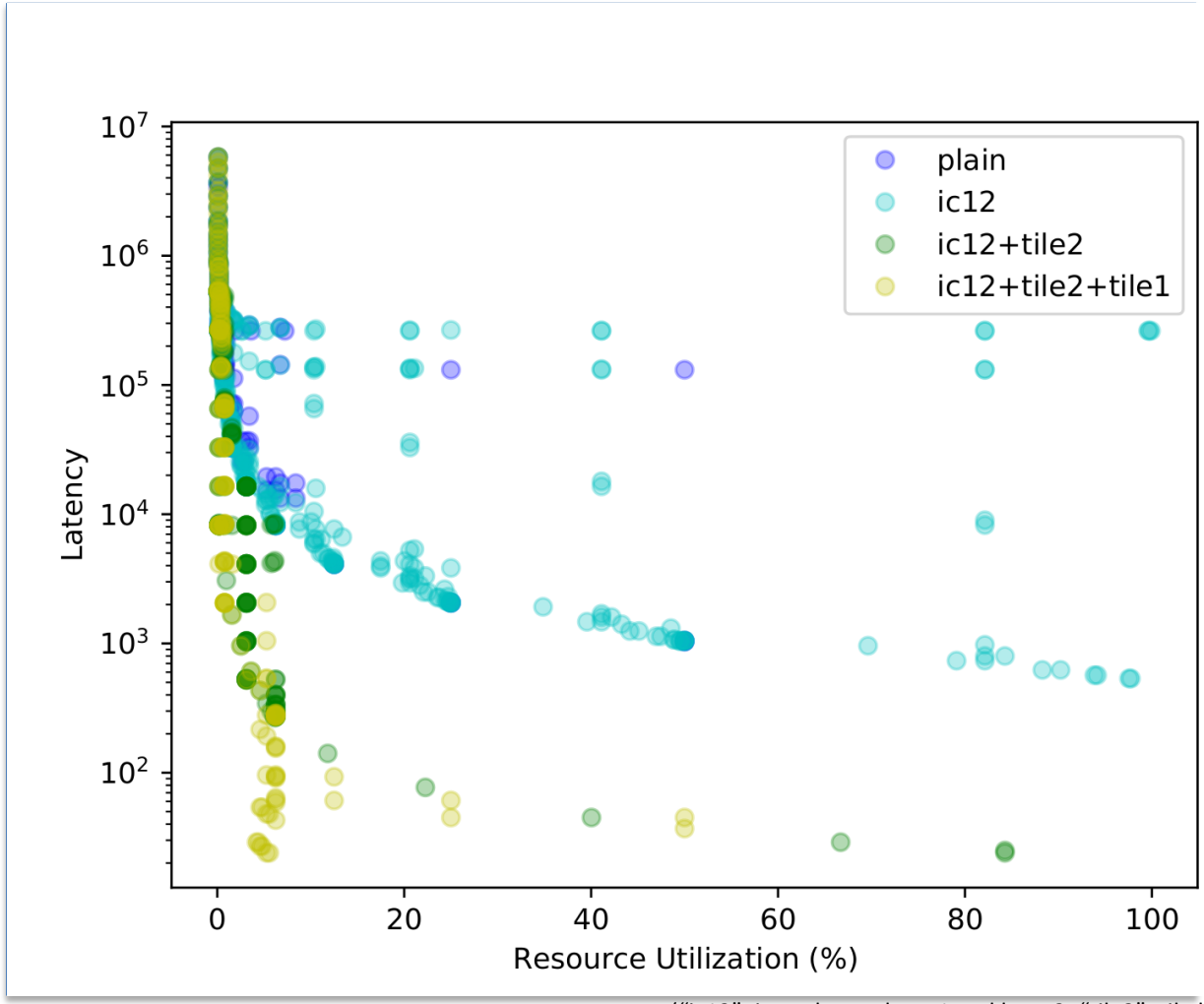
Interchange 1, 2
Tile 2
Tile 1

```
for21 (...)
  for22 (...)
    for11 (...)
      for12 (...)
```

Interchange 1, 2
Tile 2
Tile 1
Interchange 2, 3

```
for21 (...)
  for11 (...)
    for22 (...)
      for12 (...)
```

Design space of various code versions of 2D array addition.



("ic12": interchange loop 1 and loop 2; "tile2": tile loop 2)

Operator Implementation Selection

2D Add: `c = a + b`

Plain

```
for1 (...)
  for2 (...)
```

Interchange 1, 2

```
for2 (...)
  for1 (...)
```

Interchange 1, 2
Tile 2

```
for2 (...)
  for11 (...)
    for12 (...)
```

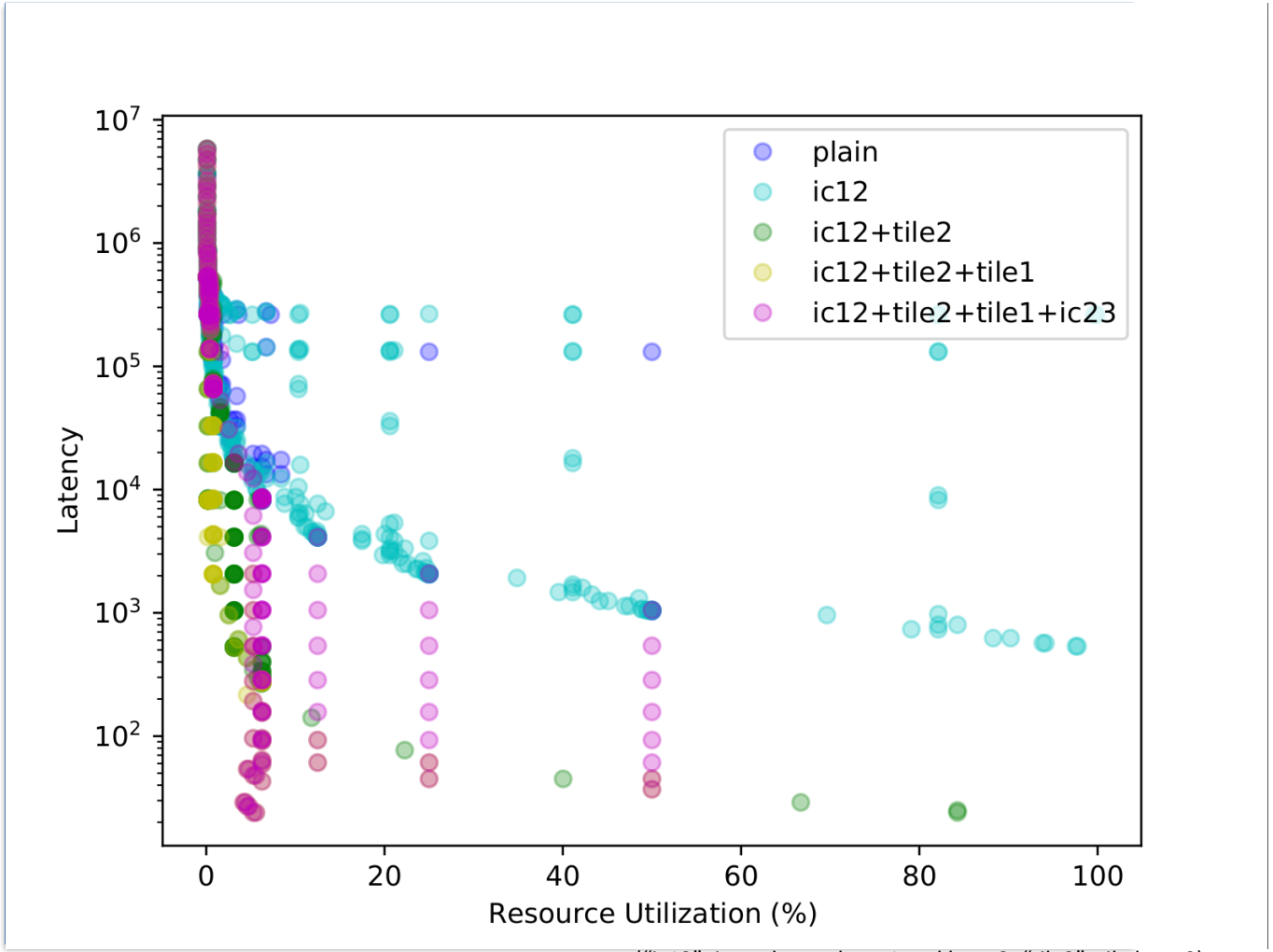
Interchange 1, 2
Tile 2
Tile 1

```
for21 (...)
  for22 (...)
    for11 (...)
      for12 (...)
```

Interchange 1, 2
Tile 2
Tile 1
Interchange 2, 3

```
for21 (...)
  for11 (...)
    for22 (...)
      for12 (...)
```

Design space of various code versions of 2D array addition.



("ic12": interchange loop 1 and loop 2; "tile2": tile loop 2)

Operator Implementation Selection

2D Add:

```
c = a + b
```

Plain

```
for1 (...)
  for2 (...)
```

Interchange 1, 2

```
for2 (...)
  for1 (...)
```

Interchange 1, 2
Tile 2

```
for2 (...)
  for11 (...)
    for12 (...)
```

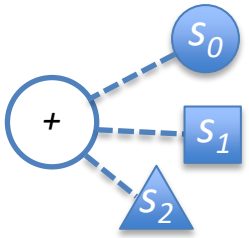
Interchange 1, 2
Tile 2
Tile 1

```
for21 (...)
  for22 (...)
    for11 (...)
      for12 (...)
```

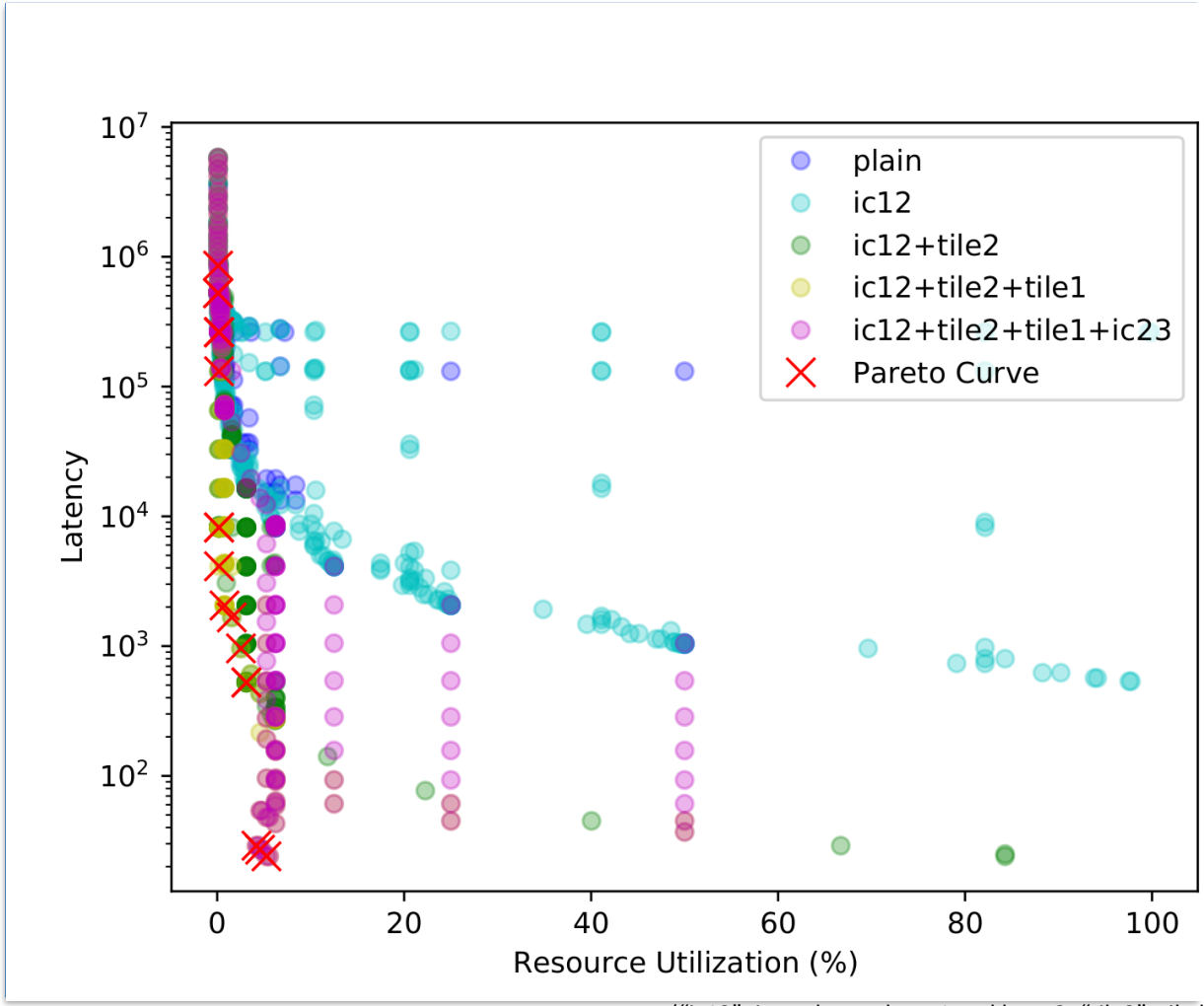
Interchange 1, 2
Tile 2
Tile 1
Interchange 2, 3

```
for21 (...)
  for11 (...)
    for22 (...)
      for12 (...)
```

Each high-level operations will have a list of candidate schedules, `schedules = [s0, s1, s2, …]`



Design space of various code versions of 2D array addition.



("ic12": interchange loop 1 and loop 2; "tile2": tile loop 2)

Operator Implementation Selection

1D Conv:
```
y = map(lambda a: a[-1]+a[0]+a[1], x[1:-1])
```
2D Conv:
```
y = map(lambda a: dot(a[-1:2,-1:2], w), img[1:-1, 1:-1])
```

Multiple possible C code versions from high-level code

```
for (...) {
#pragma HLS
unroll
}
```

```
for (...)
  for (...)
#pragma HLS unroll factor=4
```

```
for (...) {
#pragma HLS
pipeline
}
```

PyLog chooses the optimal implementation given the HW resource of the target platform

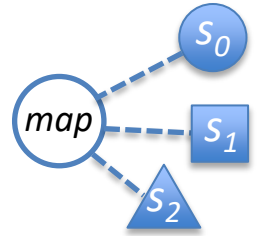
Applied to:

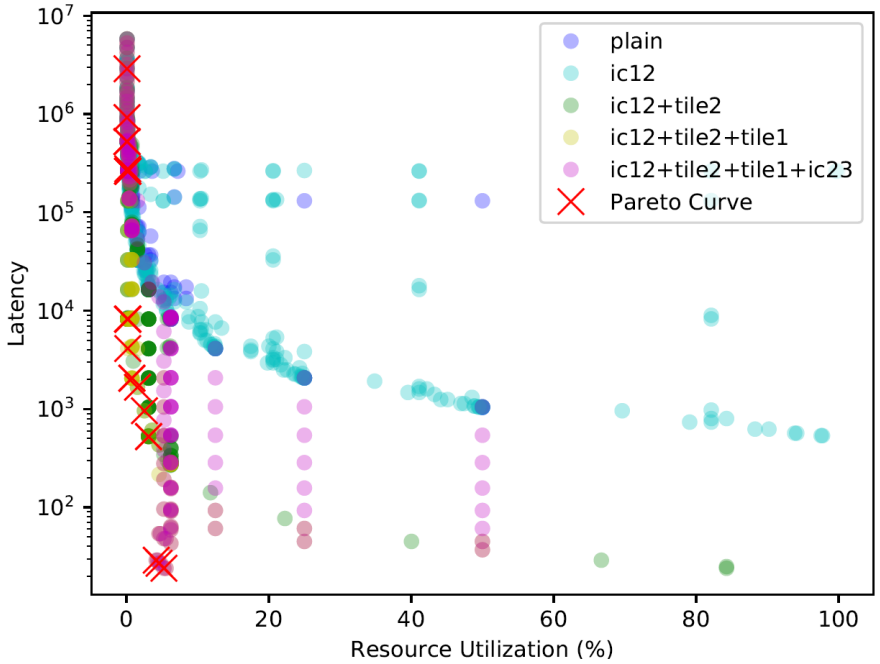
- *N*-dimensional array operations, e.g. c = a + b
- PyLog high-level operations, e.g. map, dot, reduce, etc.

PyLog IR:

- Represent the transformations in IR systematically
- Represent performance and cost of each version

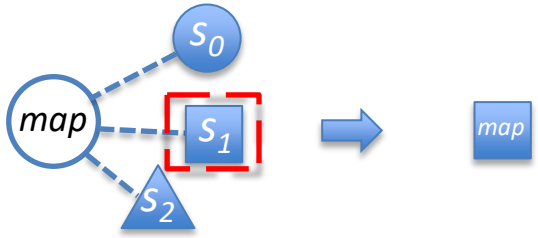
Each high-level operations will have a list of candidate schedules, `schedules = [s0, s1, s2, …]`





Design space of various code versions of 2D array addition. ("ic12": interchange loop 1 and loop 2; "tile2": tile loop 2)

Global Optimization - ILP Formulation



- The j^{th} implementation of i^{th} operator: x_{ij}

$$x_{ij} \in \{0,1\}$$

- Each operator chooses one implementation

$$\forall i, \quad \sum_j x_{ij} = 1$$

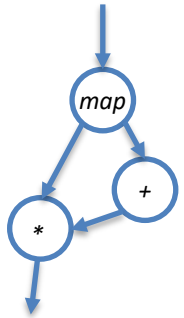
- Latency and area (cost) of the j^{th} implementation of i^{th} operator: c_{ij} (l_{ij} or a_{ij})
- Total cost (assuming no branches):

$$\sum_{i,j} c_{ij} x_{ij}$$

- Total cost (with branches, generic CDFG):

$$\sum_{i,j} p_{ij} c_{ij} x_{ij}$$

where p_{ij} is the probability of operation x_{ij} being called.



Data Movement (Using CPU-FPGA interface as example)

- Where is the data / operator happening?

$$\sum_{j \text{ at FPGA}} x_{ij}$$

- Is there any data copy needed between operations p and q?

$$\sum_{j \text{ at FPGA}} x_{pj} - \sum_{j \text{ at FPGA}} x_{qj}$$

- Cost

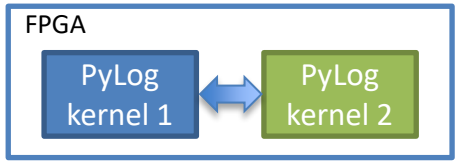
$$c_{F2C}\left(\sum_{j \text{ at FPGA}} x_{pj} - \sum_{j \text{ at FPGA}} x_{qj}\right)$$

- Total cost

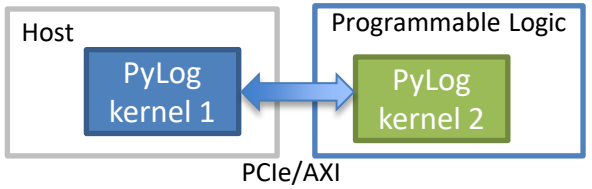
$$c_{F2C} \sum_{p \to q} \left(\sum_{j \text{ at FPGA}} x_{pj} - \sum_{j \text{ at FPGA}} x_{qj}\right)$$

Multiple Kernels

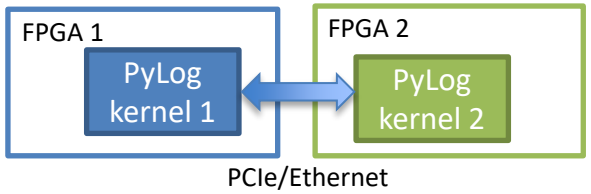
- Within same accelerator



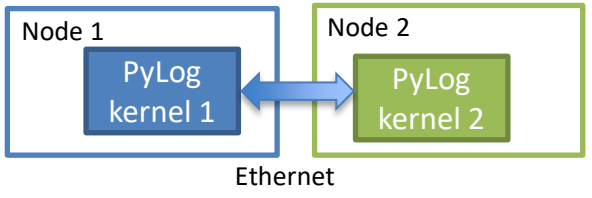
- Between host and accelerator

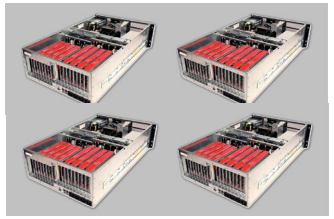


- Across accelerators



- Across nodes





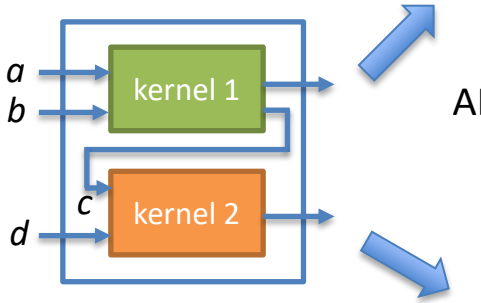
PyLog x Heterogeneous Computing

Multiple Kernels

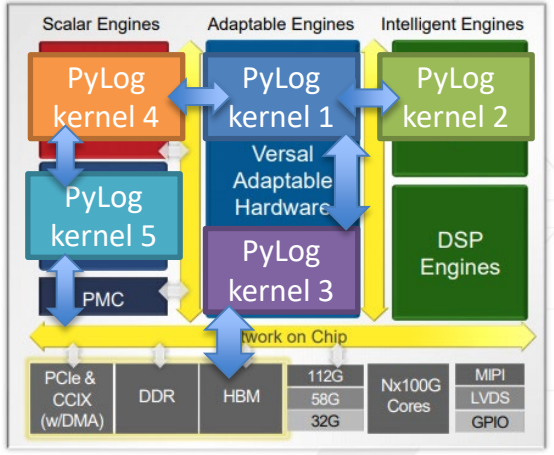
kernel 1
kernel 2
host

```
@pylog
def acc1(a, b):
    ...
@pylog
def acc2(c, d):
    ...
if __name__ == "__main__":
    a = np.array([1, 3, ...])
    b = np.array([8, 9, ...])
    d = np.array([8, 9, ...])
    combine(acc1, acc2)(a, b, d)
```

- Returns a new function that combines kernels
- Connects inputs and outputs with kernels



- New heterogeneous platforms



Xilinx Versal devices

Collaborative computing:

- Across multiple adaptable logic kernels
- Across scalar engines, adaptable engines, and AIEs
- Across multiple ACAP / FPGA devices

Data placement and movement:

- Data sources: PCIe, DDR, HBM, Ethernet, other engines, etc.

Generic programmable logic: Vitis connectivity configuration

Creating Multiple Instances of a Kernel

```
[connectivity]
#nk=<kernel name>:<number>:<cu_name>.<cu_name>...
nk=vadd:3:vadd_X.vadd_Y.vadd_Z
```

Mapping Kernel Ports to Global Memory

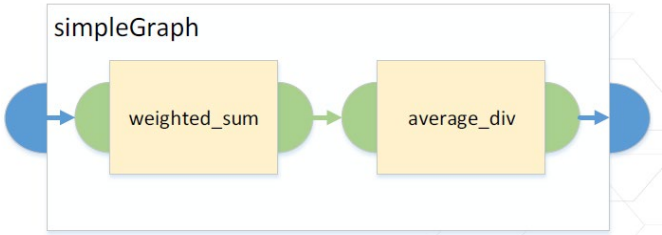
```
#sp=<compute_unit_name>.<interface_name>:<bank name>
sp=cnn_1.m_axi_gmem:DDR[0]
```

Specify streaming connections

```
#stream_connect=<cu_name>.<output_port>:<cu_name>.<input_port>
stream_connect=vadd_1.stream_out:vadd_2.stream_in
```

AI Engines: Dataflow model (on-going)

Connecting Kernels in graph.h



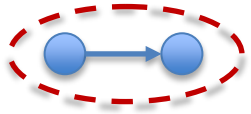

```
simpleGraph () {
  // Bind a function to each of the declared kernels
  k1  = kernel::create(weighted_sum);
  k2  = kernel::create(average_div);

  // create nets to connect kernels and IO ports
  connect<window<128> > net0 (in, k1.in[0]);
  connect<window<128> > net1 (k1.out[0], k2.in[0]);
  connect<window<128> > net2 (k2.out[0], out);
};
```

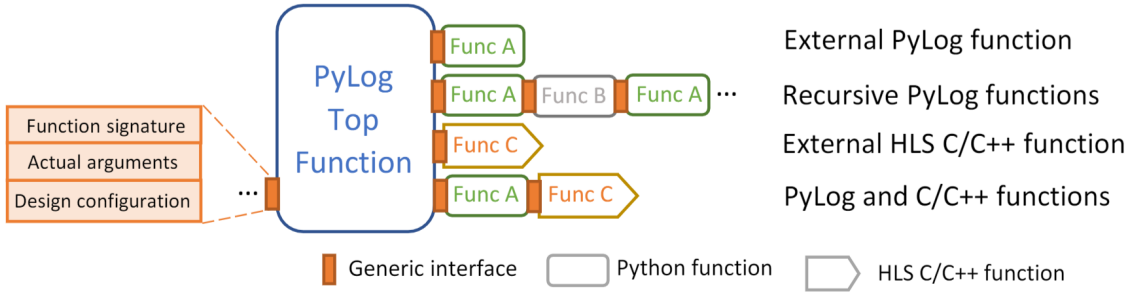
Multiple Kernels

1. Generic programmable logic: Inside IP (inside HLS)

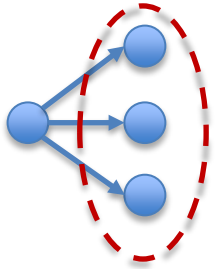
Types of connections

- Function calls (A calls B)
  - **Shared buffer**
  - **HLS stream**





- List of function calls connected with shared data structures
  - **Ping-pong buffers**
  - **FIFOs**



```
void top ( ... ) {
#pragma HLS dataflow
 int A[1024];
#pragma HLS stream off variable=A depth=3
 producer(A, B, …); // producer writes A and B
 middle(B, C, ...); // middle reads B and writes C
 consumer(A, C, …); // consumer reads A and C }
```

2. Generic programmable logic: IP level (system integration)

- Shared DDR memory space
- AXI-stream channel

Creating Multiple Instances of a Kernel

```
[connectivity]
#nk=<kernel name>:<number>:<cu_name>.<cu_name>...
nk=vadd:3:vadd_X.vadd_Y.vadd_Z
```

Mapping Kernel Ports to Global Memory

```
#sp=<compute_unit_name>.<interface_name>:<bank name>
sp=cnn_1.m_axi_gmem:DDR[0]
```

Specify streaming connections

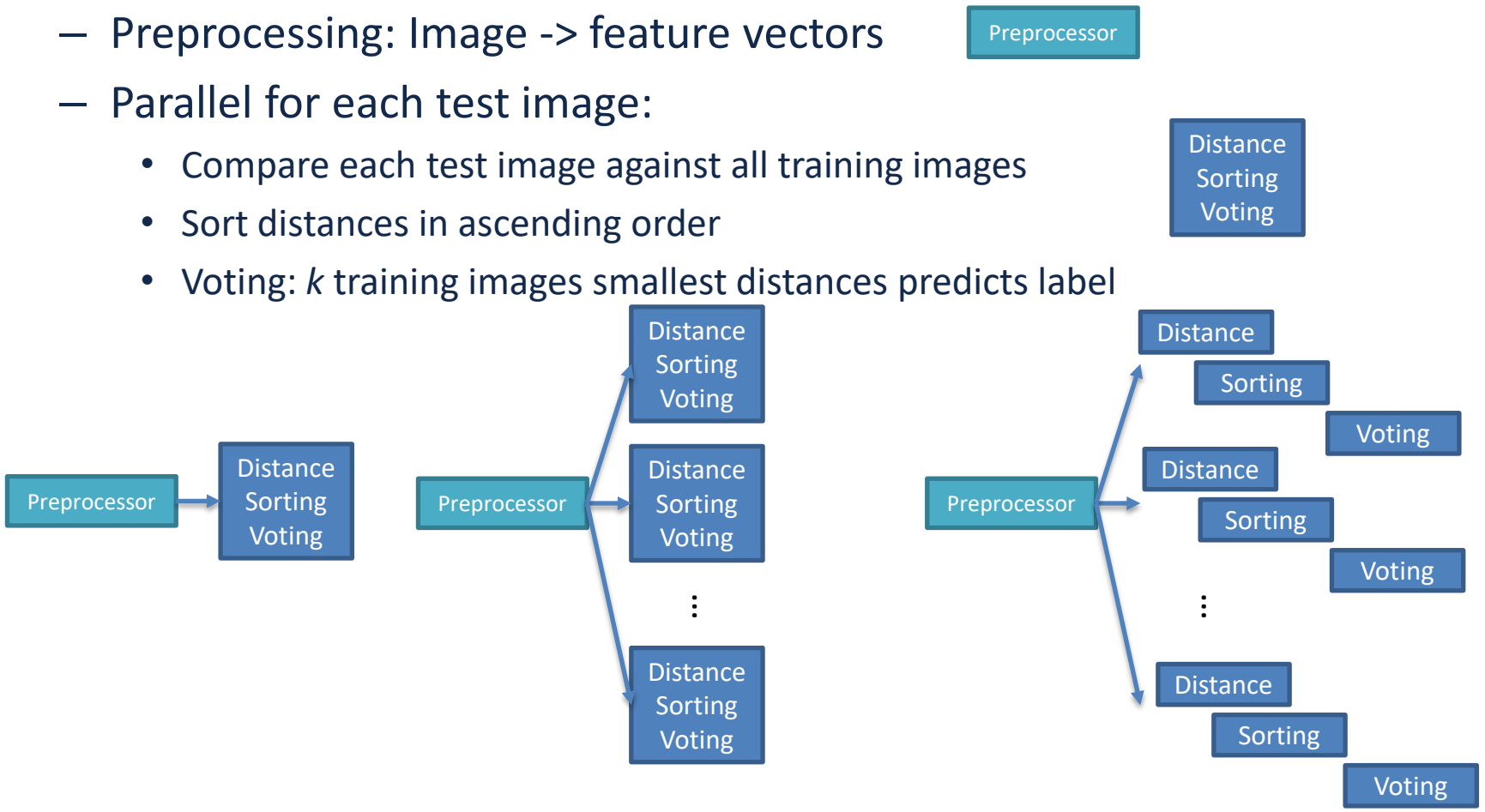
```
#stream_connect=<cu_name>.<output_port>:<cu_name>.<input_port>
stream_connect=vadd_1.stream_out:vadd_2.stream_in
```

3. Device level (CPU-accelerator)

- Shared virtual memory (AXI bus, memory mapped)
- Streaming port (AXI bus, streaming)
- Persistent memory?

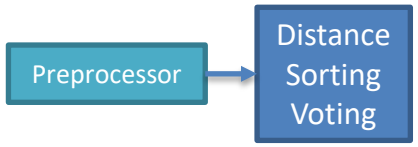
KNN Example

- Digit recognition KNN example
- Steps
  - Preprocessing: Image -> feature vectors
  - Parallel for each test image:
    - Compare each test image against all training images
    - Sort distances in ascending order
    - Voting: *k* training images smallest distances predicts label



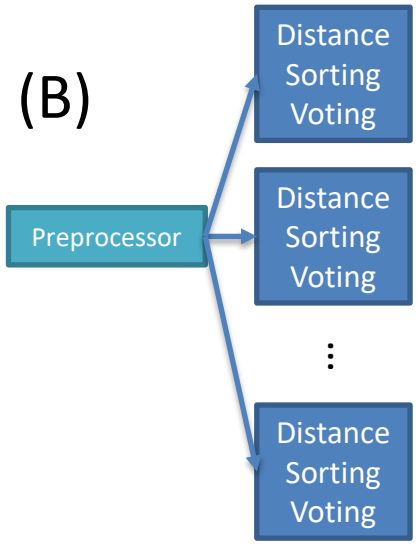
KNN Example

(A)



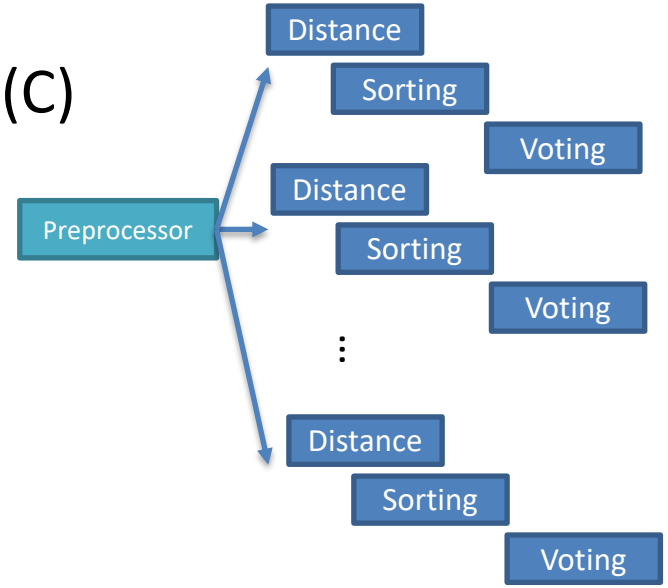

```
@pylog
def prep(a, b):
    ...
@pylog
def dist_sort_vote(c, d):
    ...
```

(B)



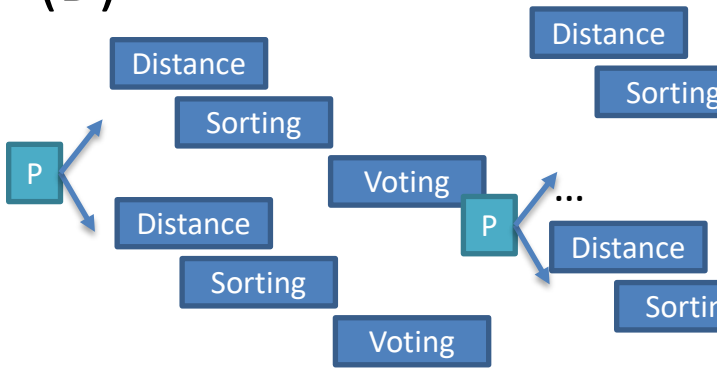

```
@pylog
def prep(a, b):
    ...
@pylog
@config(dup = N)
def dist_sort_vote(c, d):
    ...
```

(C)



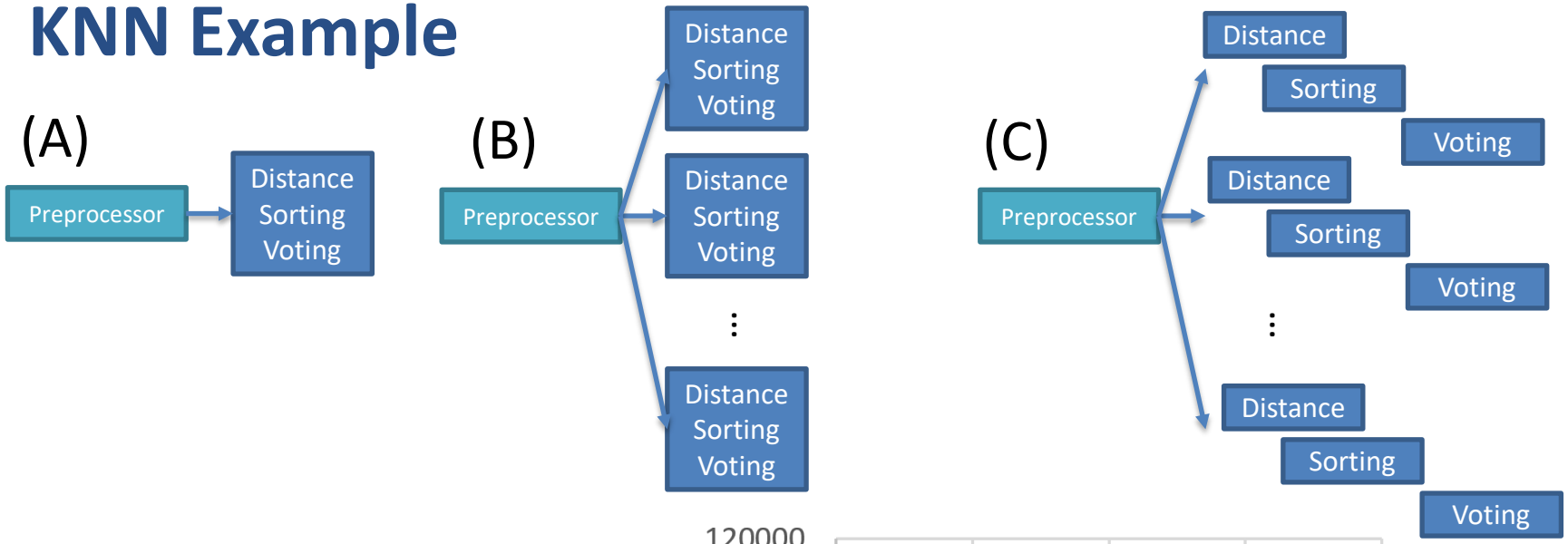

```
@pylog
def prep(a, b):
    ...
@config(dup = N)
def dist_sort_vote(c, d):
    @pylog
    def dist(c, d):

    @pylog
    def sort(c, d):
    ...
```

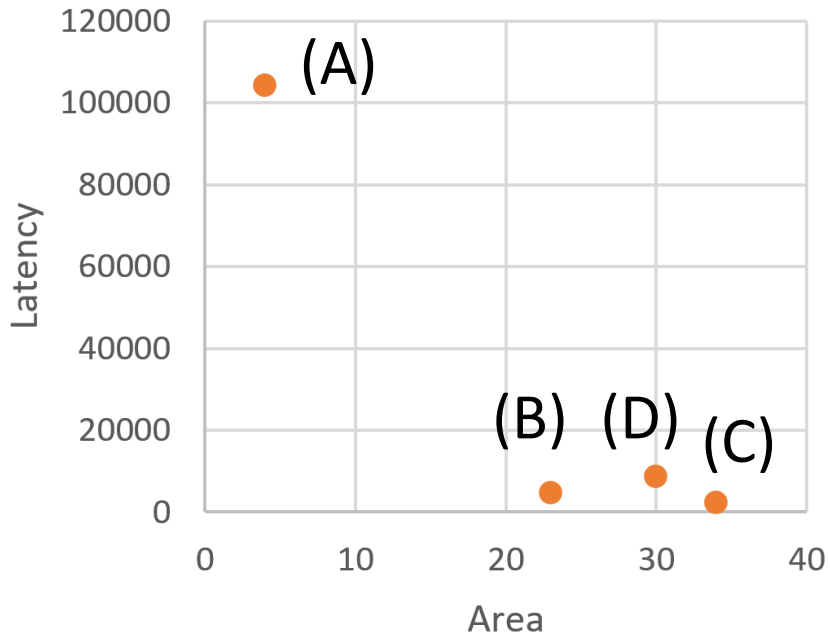
(D)



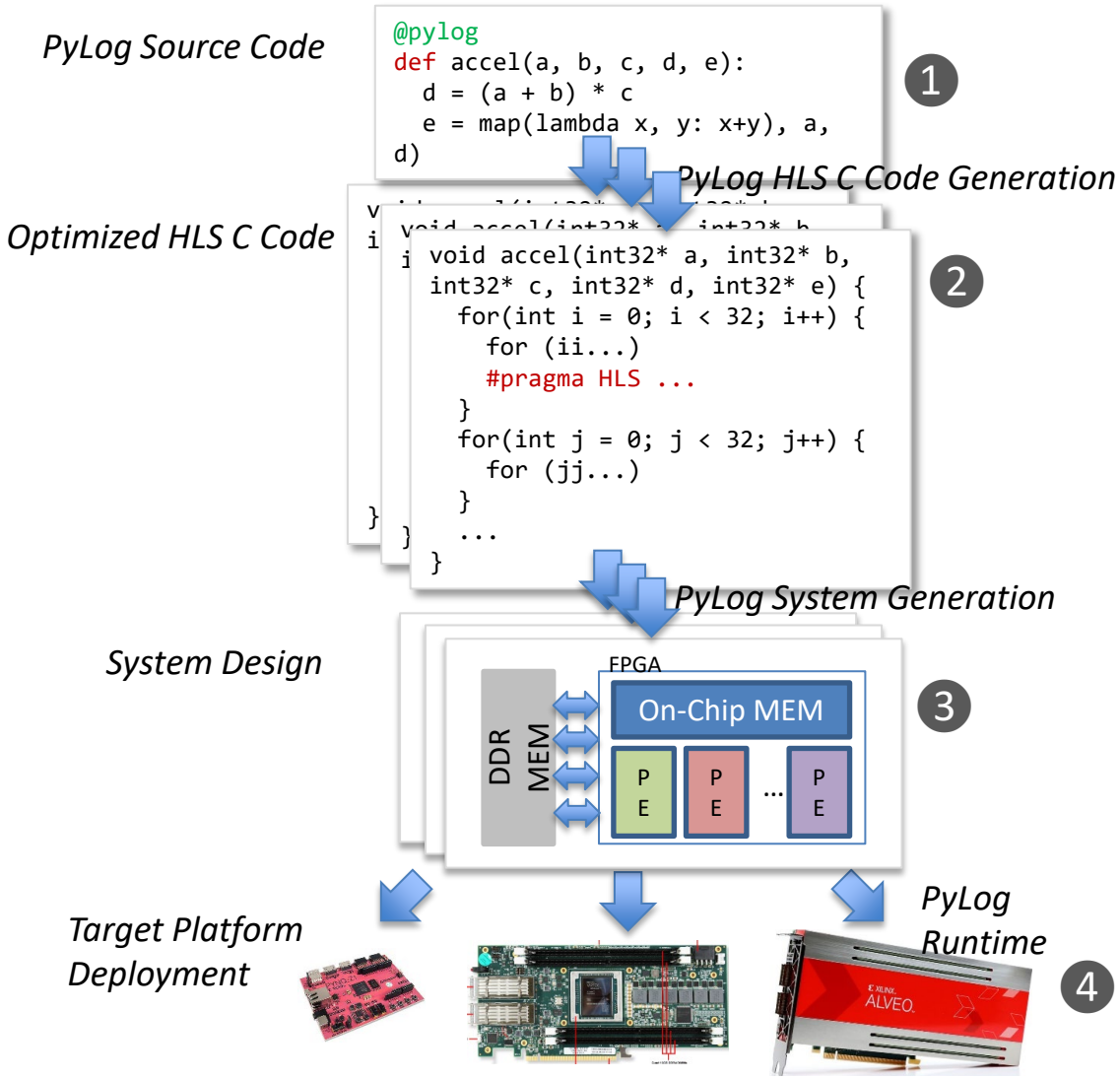

```
@pylog
@config(dup = N)
def prep(a, b):
    ...
@config(dup = N)
def dist_sort_vote(c, d):
    @pylog
    def dist(c, d):

    @pylog
    def sort(c, d):
    ...
```

KNN Example


(A)
Preprocessor
Distance
Sorting
Voting
(B)
Preprocessor
Distance
Sorting
Voting
Distance
Sorting
Voting
Distance
Sorting
Voting
(C)
Preprocessor
Distance
Sorting
Voting
Distance
Sorting
Voting
Distance
Sorting
Voting
(D)
P
Distance
Sorting
Voting
Distance
Sorting
Voting
P
...
Distance
Sorting
Distance
Sorting


120000
100000
80000
60000
40000
20000
0
Latency
0
10
20
30
40
Area
(A)
(B)
(D)
(C)

Verification in PyLog



UIUC ECE 527 - SoC Design (Fall 2020): Final Project by ***Anjana S Kumar*** and ***Christopher Baldwin***

1. PyLog code:
   - functional correctness
   - PyLog simulation: PySim

   → PyModel testing
2. ***Generated HLS C code***
   - Equivalence to PyLog code
   - Test with PyLog input

   → PyLog verification flow: `@pylog(mode='verify')`
3. RTL code
   - Equivalence to HLS C code
   - Vivado HLS RTL co-simulation

   → Vendor's simulation tools
4. On-board testing correctness
   - PyLog runtime tests

   → PyLog runtime tests

- Input: randomly generated test vectors, with various data types, input lengths, corner cases
- Check:
  - Generate and insert asserts into code
  - Compare outputs from PyLog code and outputs from C code

Verification in PyLog

Anjana S Kumar, Christopher Baldwin

Steps

- Clean up input PyLog file
- Identifies input and output arguments
- Compiles PyLog generated C file to a shared library
- Generates test vectors
- Run PyLog code with PySim
- Run top C function (from Python)
- Compare outputs from C and Python

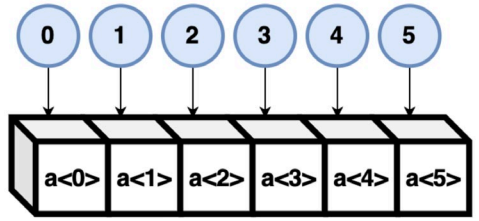
High-Level Descriptive Operators

All to help express spatial structures to the compiler

Parallelism Generation

gen.for

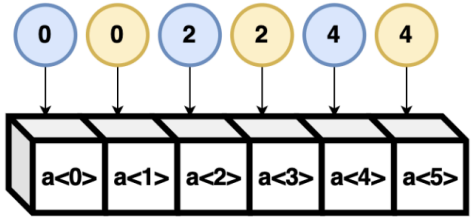
```
1 gen.for I in range(5):
2     a<I> = I
```

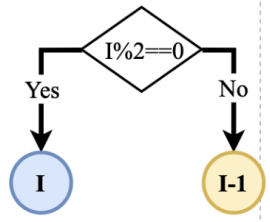


Spatial Irregularity Generation

gen.if gen.elif gen.else

```
1 gen.for I in range(5):
2     gen.if I % 2 == 0:
3         a<I> = I
4     gen.else :
5         a<I> = I
```

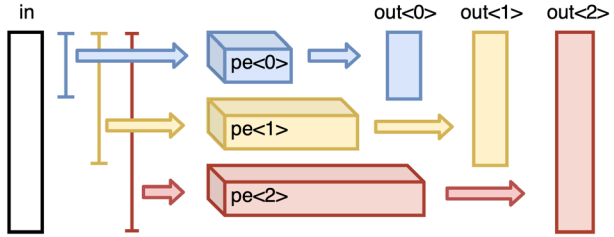




Configurable Kernel Generation

.config() attribute

```
1 @config(batch)
2 def pe(in,out):
3     ...do work...
4
5 gen.for I in range(3):
6     pe<I>(in,out<I>).config(I*100)
```



High-Level Descriptive Operators

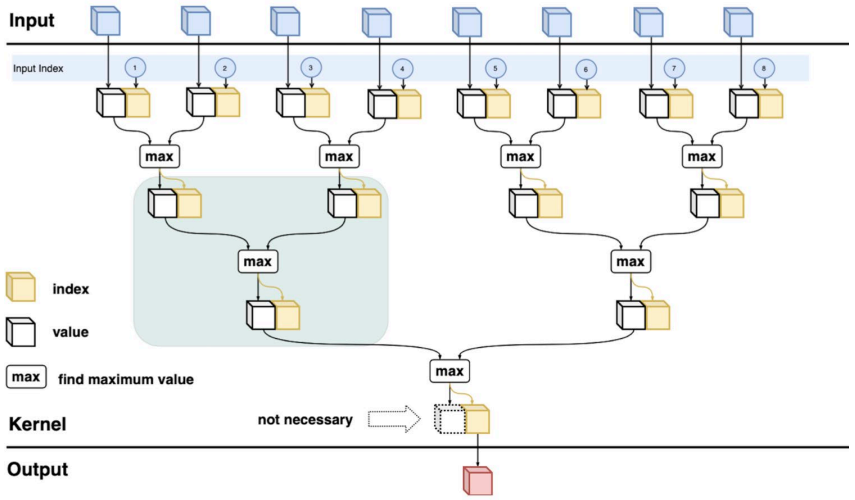
```
@config(dict = [0:4, 1:2, 3:1])
def argmax(b) :

    gen.for I in range(3):
        gen.for J in range(dict[I]):
            gen.if ( I == 2 ):
     ## loop body dedicated to the last loop ##
            gen.else:
                if (  b<I-1,J*2> > b<I-1,J*2+1> ):
                    b<I,J>       = b<I-1,J*2>
                    index_b<I,J> = index_b<I-1,J>
                else:
                    b<I,J>       = b<I-1,J*2+1>
                    index_b<I,J> = index_b<I-1,J+1>

    return index_b<2,0>
```

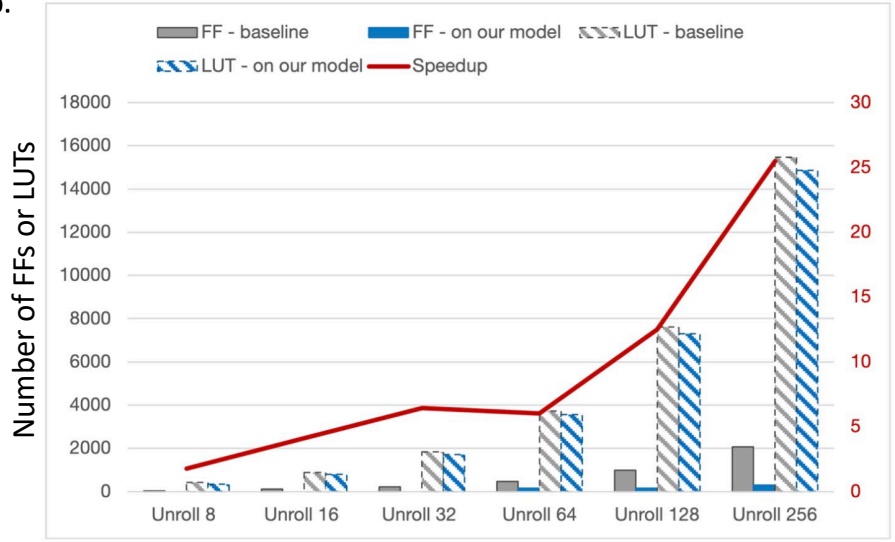
Code 4. `argmax` Example.

```
int argmax(int input[SIZE]){
#pragma HLS array_partition variable=input complete
    int max_v = 0;
    int max_id  = 0;
    for (int i=0;  i<SIZE; i++){
#pragma HLS unroll
        if (input[i] > max_v){
            max_id = i;
            max_v = input[i];
        }
    }
    return max_id;

}
```

Code 6. Baseline Algorithm Argmax Using Regular for Loop.

The implementation graph of the code using new syntax (unroll factor = 8)





Future Directions

Compilation

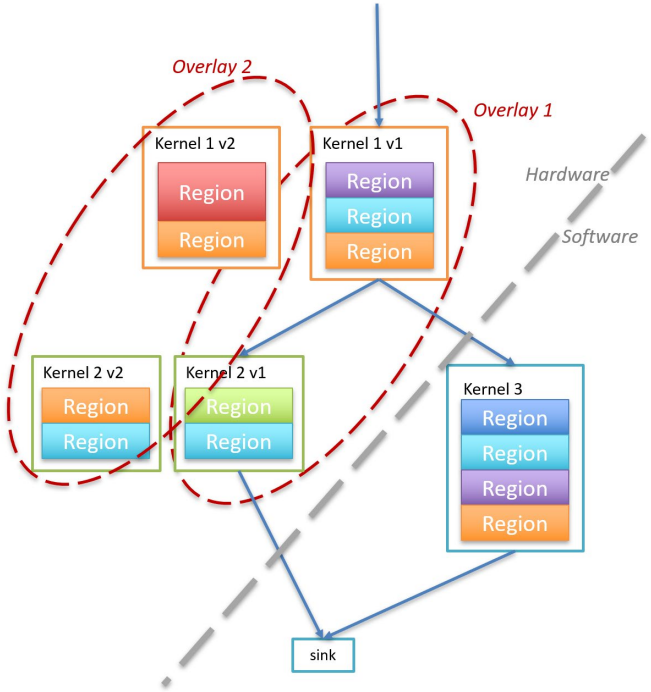
- Better support on Python/NumPy/Panda ops
- HW/SW partitioning and collaboration
- Fine-grained synthesis (op level)
- Compilation error handling – falling back to CPU
- GPU kernel generation
- Targeting new architectures like AI engines

Applications

- Deep learning
- Graph processing
- Data table operations
- Lightweight Cryptography

Use/Deployment Modes

- Better integration in the cloud/distributed environment
  - Refs: Dask for GPU, Numba for CPUs
- Closer integration with PYNQ environment
  - Utilize latest features provided by PYNQ
  - Pipeline templates, partial reconfiguration, etc.
  - Refs: Raspberry Pi flows for Arm CPUs



Reading Materials

- PyLog: An Algorithm-Centric FPGA Programming and Synthesis Flow. [paper]
- HeteroCL: A Multi-Paradigm Programming Infrastructure for Software-Defined Reconfigurable Computing. [paper][slides]
- ScaleHLS: A New Scalable High-Level Synthesis Framework on Multi-Level Intermediate Representation. [paper]
- Predictable Accelerator Design with Time-Sensitive Affine Types. [paper][demo]
- Chisel/FIRRTL Hardware Compiler Framework. [webpage]
- TVM: An End to End Machine Learning Compiler Framework for CPUs, GPUs and accelerators. [webpage]
- The Versatile Tensor Accelerator (VTA). [webpage]
- TACO: The Tensor Algebra Compiler. [webpage]

EECS 221:
Languages and Compilers for Hardware Accelerators

(Winter 2022)

Sitao Huang

sitaoh@uci.edu