HeteroCL: A Multi-Paradigm Programming Infrastructure for Software-Defined Reconfigurable Computing

Yi-Hsiang Lai[1], Yuze Chi[2], Yuwei Hu[1], Jie Wang[2], Cody Hao Yu[2,3], Yuan Zhou[1], Jason Cong[2], Zhiru Zhang[1]

[1]Cornell University
[2]University of California, Los Angeles
[3]Falcon Computing Solutions, Inc.

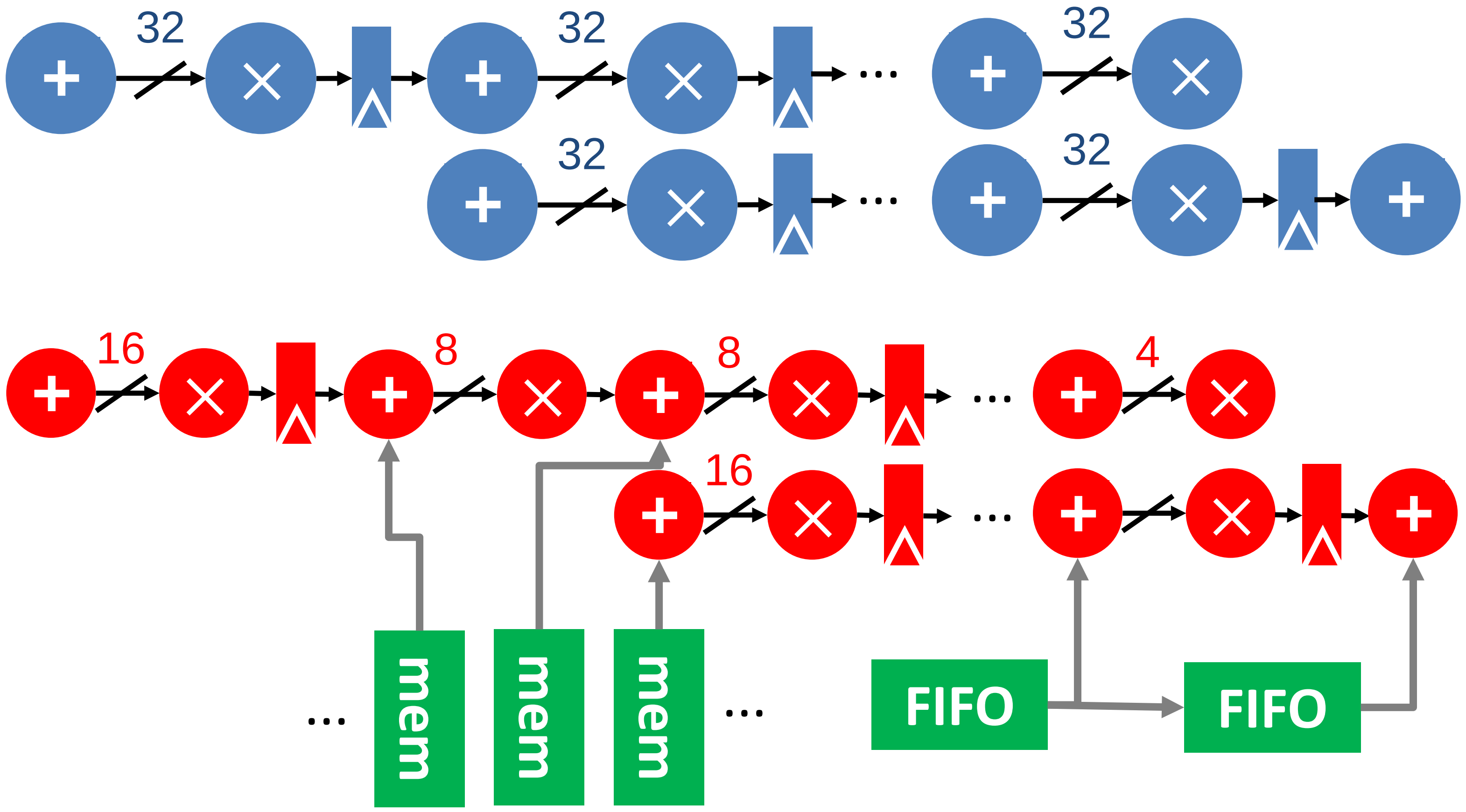
Cornell University

Essential Techniques for Hardware Acceleration

Compute customization
- **Parallelization**
- **Pipelining, etc.**

Data type customization
- **Low-bitwidth integer**
- **Fixed point, etc.**

Memory customization
- **Banking**
- **Data reuse, etc**



There exists interdependence among different customizations

Hardware Customization in High-Level Synthesis

- Driving example: convolutional kernel

```
for (int y = 0; y < N; y++)
 for (int x = 0; x < N; x++)
  for (int r = 0; r < 3; r++)
   for (int c = 0; c < 3; c++)
    out[x, y] += image[x+r, y+c] * kernel[r, c]
```

Algorithm#1
Compute Customization
Algorithm#2
Data Type Customization
Memory Customization
Algorithm#3

Entangled hardware customization and algorithm

- Less portable
- Less maintainable
- Less productive

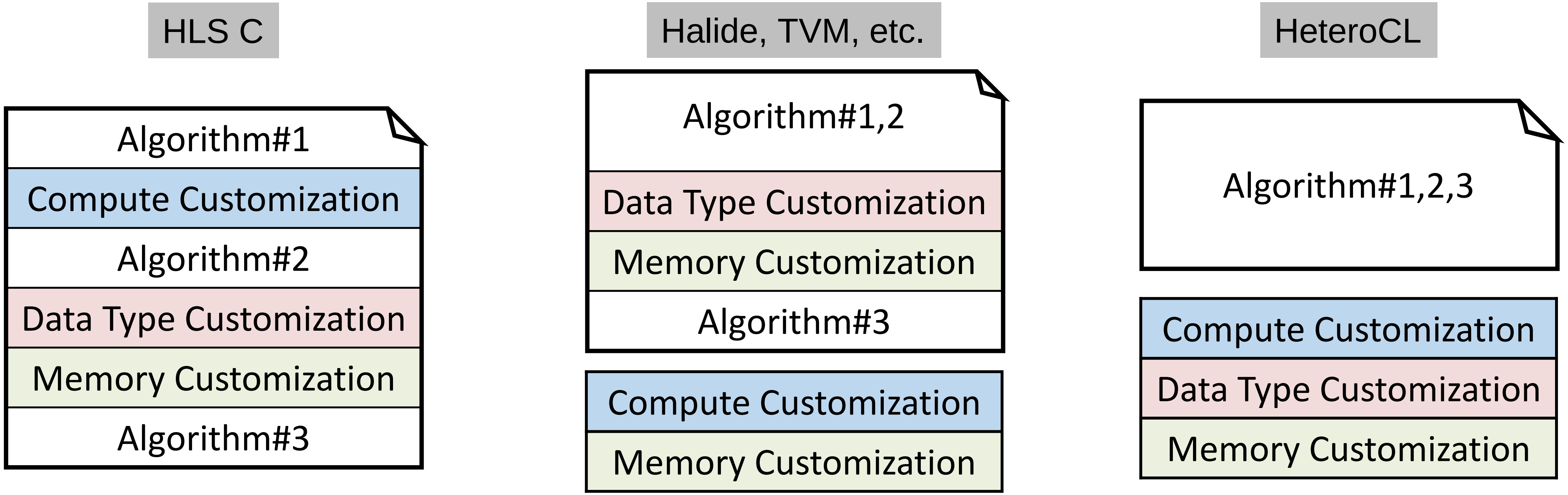
```
#pragma HLS array_partition variable=filter dim=0
 hls::LineBuffer<3, N, ap_fixed<8,4> > buf;
 hls::Window<3, 3, ap_fixed<8,4> > window;
 for(int y = 0; y < N; y++) {
  for(int xo = 0; xo < N/M; xo++) {
#pragma HLS pipeline II=1
   for(int xi = 0; xi < M; xi++) {
    int x = xo*M + xi;
    ap_fixed<8,4> acc = 0;
    ap_fixed<8,4> in = image[y][x];
    buf.shift_up(x);
    buf.insert_top(in, x);
    window.shift_left();
    for(int r = 0; r < 2; r++)
     window.insert(buf.getval(r,x), i, 2);
    window.insert(in, 2, 2);
    if (y >= 2 && x >= 2) {
     for(int r = 0; r < 3; r++) {
      for(int c = 0; c < 3; c++) {
       acc += window.getval(r,c) * kernel[r][c];
     }}
     out[y-2][x-2] = acc;
}}}}
```

Custom compute (Loop tiling)

Custom data type (Quantization)

Custom memory (Reuse buffers)

Decoupling Algorithm from Hardware Customization



Entangled algorithm specification and customization schemes [1,2,3]

[1] Intel HLS
[2] Xilinx Vivado HLS
[3] Canis, et al. FPGA'11

Decoupled temporal schedules [4,5,6,7,8]

[4] Ragan-Kelly, et al. SIGPLAN'13
[5] Baghdadi, et al. arXiv'18
[6] Rong, et al. arXiv'17
[7] Pu, et al. TACO'17
[8] Chen, et al. arXiv'18

Fully decoupled customization schemes +
Clean abstraction capturing the interdependence

Decoupled Compute Customization

Algorithm

HeteroCL code

```
r = hcl.reduce_axis(0, 3)
c = hcl.reduce_axis(0, 3)
out = hcl.compute(N, N),
    lambda y, x:
        hcl.sum(image[x+r, y+c]*kernel[r, c],
            axis=[r, c]))
```

Declarative programming

HLS code

```
for (int y = 0; y < N; y++)
 for (int x = 0; x < N; x++)
  for (int r = 0; r < 3; r++)
   for (int c = 0; c < 3; c++)
    out[x, y] += image[x+r, y+c] * kernel[r, c]
```

Decoupled customization

```
s = hcl.create_schedule()
xo, xi = s[out].split(out.x, factor=M)
s[out].reorder(xi, xo, out.y)
```

Customization primitives

- More productive / less labor-intensive

```
for (int xi = 0; xi < M; xi++)
 for (int xo = 0; xo < N/M; xo++)
  for (int y = 0; y < N; y++)
   for (int r = 0; r < 3; r++)
    for (int c = 0; c < 3; c++)
     out[xi+xo*M, y] +=
      image[xi+xo*M+r, y+c] * kernel[r, c]
```

Tile loop

Reorder loops

Decoupled Memory Customization

- Primitives can be applied with a user-defined sequence

```
r = hcl.reduce_axis(0, 3)
c = hcl.reduce_axis(0, 3)
out = hcl.compute(N, N),
    lambda y, x:
        hcl.sum(image[x+r, y+c]*kernel[r, c],
            axis=[r, c]))
```

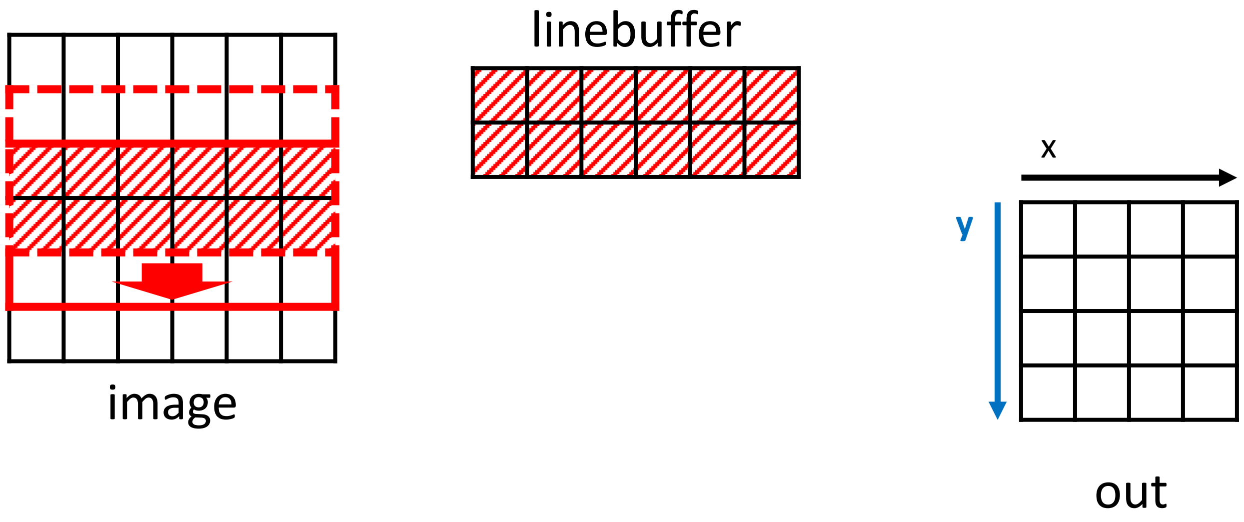
```
for (int y = 0; y < N; y++)
  for (int x = 0; x < N; x++)
    for (int r = 0; r < 3; r++)
      for (int c = 0; c < 3; c++)
        out[x, y] += image[x+r, y+c] * kernel[r, c]
```

Decoupled Memory Customization

- Primitives can be applied with a user-defined sequence

```
r = hcl.reduce_axis(0, 3)
c = hcl.reduce_axis(0, 3)
out = hcl.compute(N, N),
    lambda y, x:
        hcl.sum(image[x+r, y+c]*kernel[r, c],
            axis=[r, c]))

s = hcl.create_schedule()
linebuf = s[image].reuse_at(out, out.y)
```

```
for (int y = 0; y < N; y++)
 for (int x = 0; x < N; x++)
  for (int r = 0; r < 3; r++)
   for (int c = 0; c < 3; c++)
    out[x, y] += image[x+r, y+c] * kernel[r, c]
```

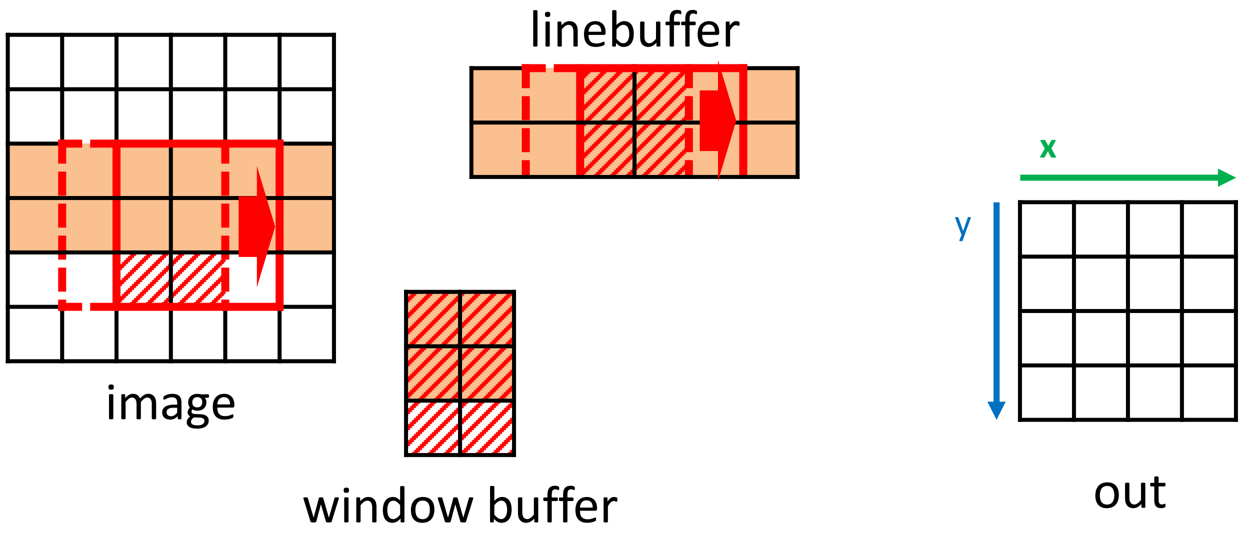


Decoupled Memory Customization

- Primitives can be applied with a user-defined sequence

```
r = hcl.reduce_axis(0, 3)
c = hcl.reduce_axis(0, 3)
out = hcl.compute(N, N),
    lambda y, x:
        hcl.sum(image[x+r, y+c]*kernel[r, c],
            axis=[r, c]))

s = hcl.create_schedule()
linebuf = s[image].reuse_at(out, out.y)
winbuf = s[linebuf].reuse_at(out, out.x)
```

```
for (int y = 0; y < N; y++)
 for (int x = 0; x < N; x++)
  for (int r = 0; r < 3; r++)
   for (int c = 0; c < 3; c++)
    out[x, y] += image[x+r, y+c] * kernel[r, c]
```

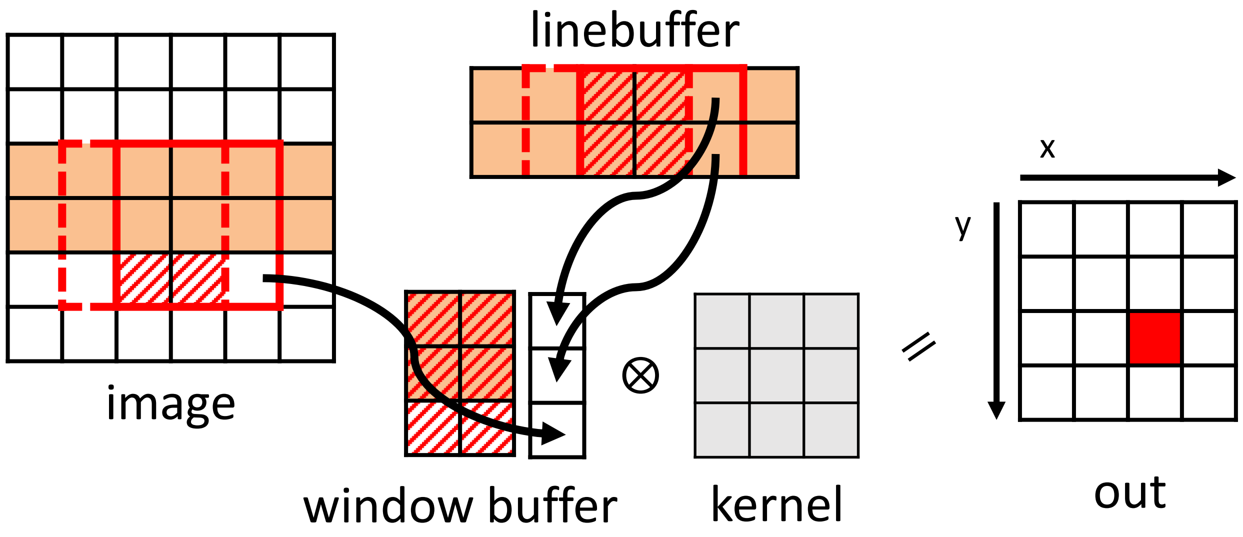


Decoupled Memory Customization

- Primitives can be applied with a user-defined sequence

```
r = hcl.reduce_axis(0, 3)
c = hcl.reduce_axis(0, 3)
out = hcl.compute(N, N),
    lambda y, x:
        hcl.sum(image[x+r, y+c]*kernel[r, c],
            axis=[r, c]))

s = hcl.create_schedule()
linebuf = s[image].reuse_at(out, out.y)
winbuf = s[linebuf].reuse_at(out, out.x)
```

```
for (int y = 0; y < N; y++)
 for (int x = 0; x < N; x++)
  for (int r = 0; r < 3; r++)
   for (int c = 0; c < 3; c++)
    out[x, y] += image[x+r, y+c] * kernel[r, c]
```



Decoupled Data Type Customization

- Bit-accurate data type support (e.g., `Int(15)`, `Fixed(7,4)`)
- Decoupled customization primitives: downsize & quantize

```
r = hcl.reduce_axis(0, 3)
c = hcl.reduce_axis(0, 3)
out = hcl.compute(N, N),
    lambda y, x:
        hcl.sum(image[x+r, y+c]*kernel[r, c],
            axis=[r, c]))
```

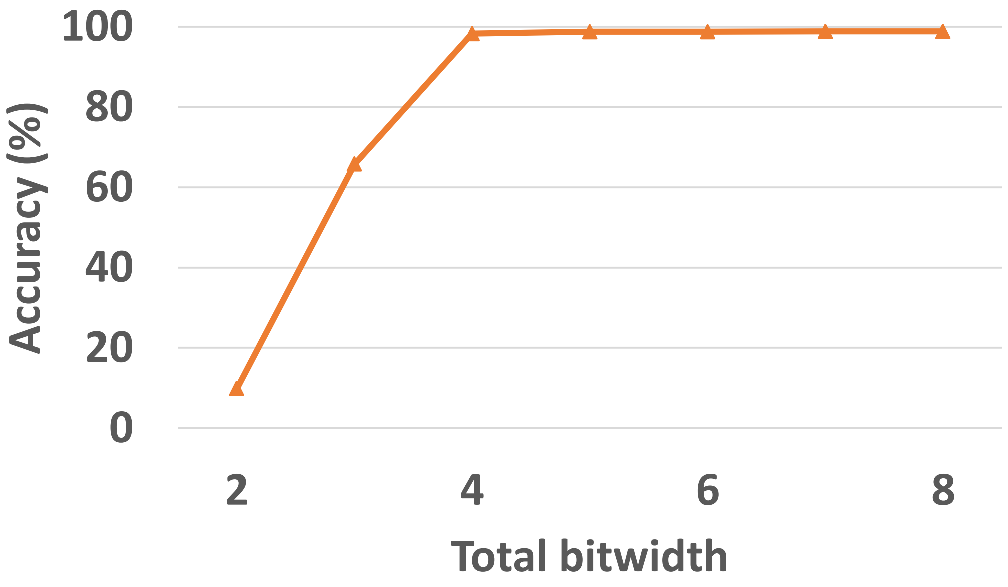
```
s = hcl.create_scheme()
s.quantize([out], Fixed(6, 4))
```

Decoupled Data Type Customization

- Bit-accurate data type support (e.g., `Int(15)`,`Fixed(7,4)`)
- Decoupled customization primitives: downsize & quantize

```
r = hcl.reduce_axis(0, 3)
c = hcl.reduce_axis(0, 3)
out = hcl.compute(N, N),
    lambda y, x:
        hcl.sum(image[x+r, y+c]*kernel[r, c],
            axis=[r, c]))
```

```
for i in range(2, 8):
  s = hcl.create_scheme()
  s.quantize([out], Fixed(i, i-2))
```



Trade-off between accuracy and resource for a neural network

Currently Supported Customization Primitives

Compute customization

Primitive	Description
	Loop transformation
`C.split(i, v)`	Split loop `i` of operation `C` into a two-level nest loop with `v` as the factor of the inner loop.
`C.fuse(i, j)`	Fuse two sub-loops `i` and `j` of operation `C` in the same nest loop into one.
`C.reorder(i, j)`	Switch the order of sub-loops `i` and `j` of operation `C` in the same nest loop.
`P.compute_at(C, i)`	Merge loop `i` of the operation `P` to the corresponding loop level in operation `C`.
	Parallelization
`C.unroll(i, v)`	Unroll loop `i` of operation `C` by factor `v`.
`C.parallel(i)`	Schedule loop `i` of operation `C` in parallel.
`C.pipeline(i, v)`	Schedule loop `i` of operation `C` in pipeline manner with a target initiation interval `v`.

Memory customization

Primitive	Description
`C.partition(i, v)`	Partition dimension `i` of tensor `C` with a factor `v`.
`C.reshape(i, v)`	Pack dimension `i` of tensor `C` into words with a factor `v`.
`memmap(t, m)`	Map a list of tensors `t` with mode `m` to new tensors. The mode `m` can be either vertical or horizontal.
`P.reuse_at(C, i)`	Create a reuse buffer storing the values of tensor `P`, where the values are reused at dimension `i` of operation `C`.

Data type customization

Primitive	Description
`quantize(t, d)`	Quantize a list of tensors `t` from floating to fixed point type `d` in the format defined in Table 2.
`downsize(t, d)`	Downsize a list of tensors `t` from integers with larger bitwidth to integers `d` with smaller bitwidth in the format defined in Table 2.

Macros for spatial architecture templates

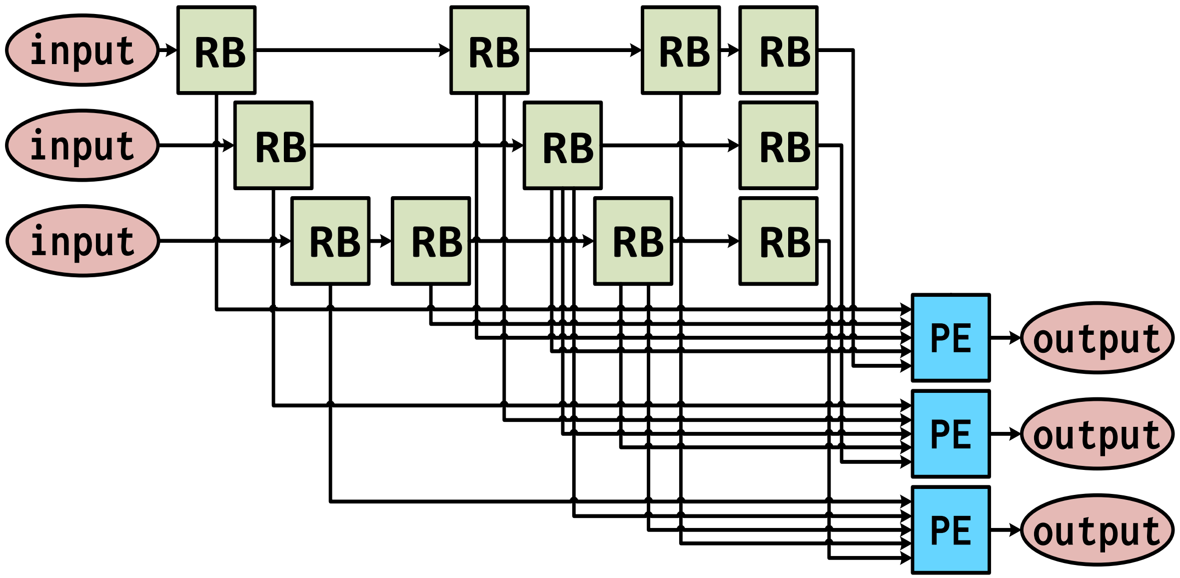
Primitive	Description
`C.stencil()`	Specify operation `C` to be implemented with stencil with dataflow architectures using the SODA framework.
`C.systolic()`	Specify operation `C` to be implemented with systolic arrays using the PolySA framework.

Macro for Stencil with Dataflow Architecture

- A sliding window applied on a tensor
- For applications where data elements are updated with some fixed, local patterns
- Incorporate with SODA [Y. Chi, et al. ICCAD'18]
  - Scalable reuse buffers with minimum buffer size that achieve highest throughput

```
r = hcl.reduce_axis(0, 3)
c = hcl.reduce_axis(0, 3)
out = hcl.compute(N, N),
   lambda y, x:
       hcl.sum(image[x+r, y+c]*kernel[r, c],
          axis=[r, c]))

s = hcl.create_schedule()
s[out].stencil()
```

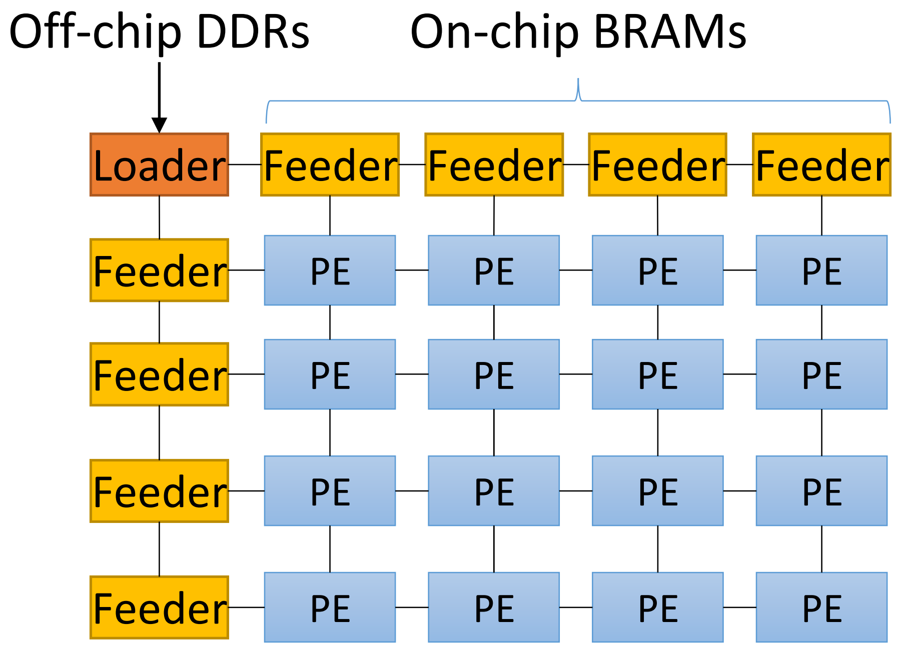
Pipelined reuse buffers (RB)



Macro for Systolic Array

- A group of PEs locally connected to each other
- For applications having perfectly nested loops with uniform dependency
- Incorporate with PolySA [J. Cong, et al. ICCAD'18]
  - Systematic and efficient design space exploration => Comparable performance to manual designs within hours

```
r = hcl.reduce_axis(0, 3)
c = hcl.reduce_axis(0, 3)
out = hcl.compute(N, N),
   lambda y, x:
      hcl.sum(image[x+r, y+c]*kernel[r, c],
         axis=[r, c]))

s = hcl.create_schedule()
s[out].systolic()
```



[X. Wei, et al. DAC'17]

Imperative Programming in HeteroCL

- HeteroCL further provides an embedded imperative DSL
  - Not all algorithms can be described using declarative code
- Unified interface for applying hardware customization to both imperative and declarative codes

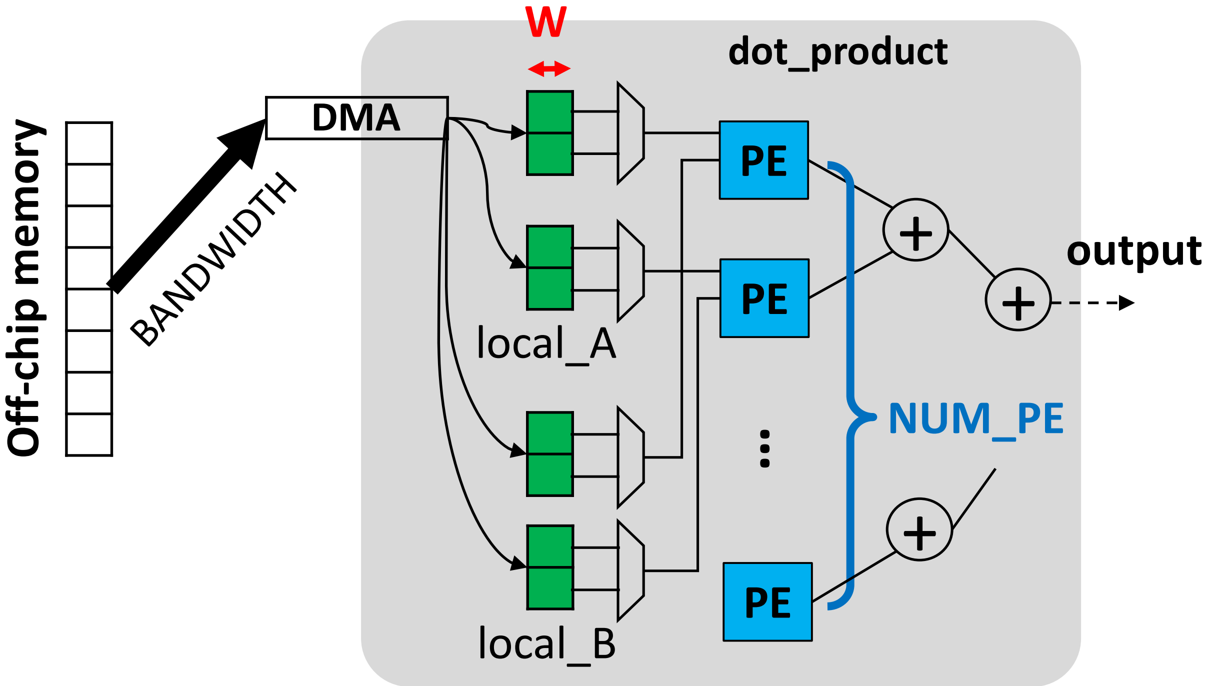
```
with hcl.for_(0, N) as y:
  with hcl.for_(0, N) as x:
    with hcl.for_(0, 3) as r:
      with hcl.for_(0, 3) as c:
        out[x, y] += image[x+r, y+c] * kernel[r, c]

s = hcl.create_schedule()
s[out].split(out.x, M)
linebuf = s[image].reuse_at(out, out.y)
s.quantize([out], Fixed(6, 4))
# ...
```

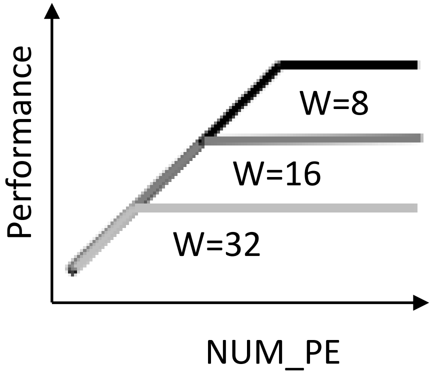
We need DSL because normal Python is too flexible (i.e., not all semantics are synthesizable)

Explore the Interdependence: Dot Product

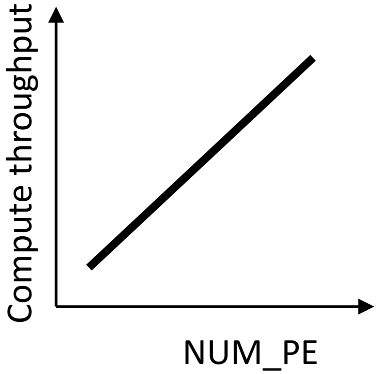
```
i = hcl.reduce_axis(0, N)
return hcl.compute((1,),
  lambda x: hcl.sum(local_A[i] * local_B[i],
    axis=i))
```

```
          for W in [4, 8, 16, 32]:
            NUM_PE = BANDWIDTH / W
            xo, xi = s[psum].split(x, NUM_PE)
Compute     s[psum].unroll(xi)
Data type   s.quantize(local_A, hcl.Fixed(W))
Memory      s[local_A].partition(NUM_PE)
```

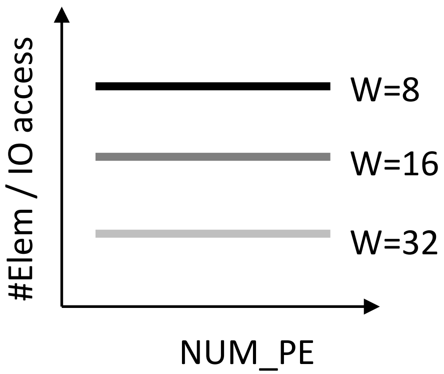




= min(

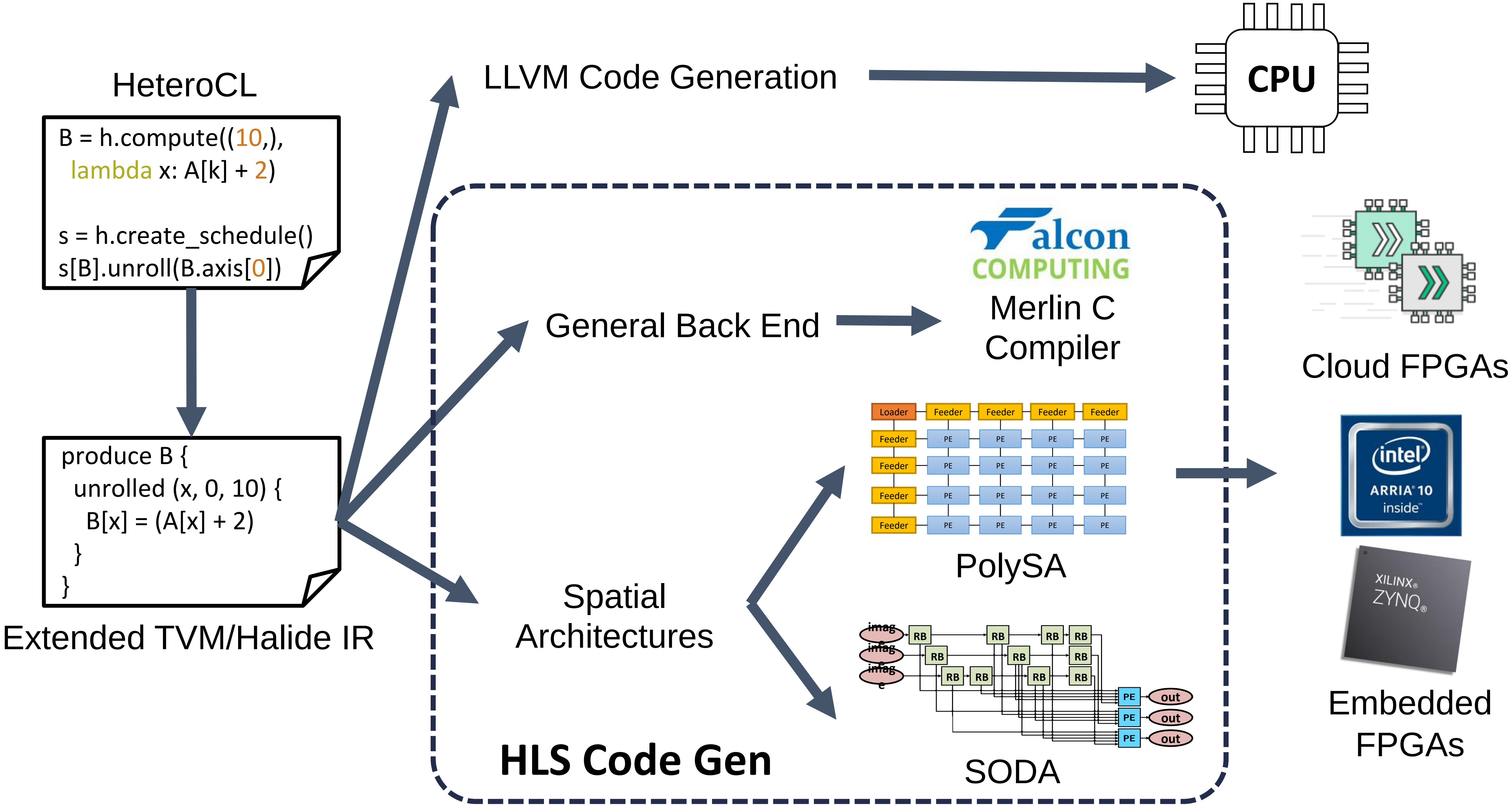


,



)

HeteroCL Compilation Flow



Evaluation with Amazon AWS f1

Stencil

Benchmark	Application Field	+ stencil	+ unroll	+ quantize	Theoretical (limited by memory bandwidth)
Seidel	Image processing	0.2	2.9	5.9	6.8
Gaussian	Image processing	1.1	6.7	13.2	15.6
Jacobi	Linear algebra	0.4	2.3	5.0	5.4

Systolic

Benchmark	Application Field	Back end	Data type	Performance (GOPs)	Speedup
GEMM	Matrix multiplication	CPU (Intel MKL)	float32	76.0	1.0
		FPGA	float32	245.9	3.2
			fixed16	807.6	10.6
LeNet	Convolutional neural network	CPU (TVM TOPI)	float32	15.4	1.0
		FPGA	float32	79.8	5.2
			fixed16	137.8	8.9

General

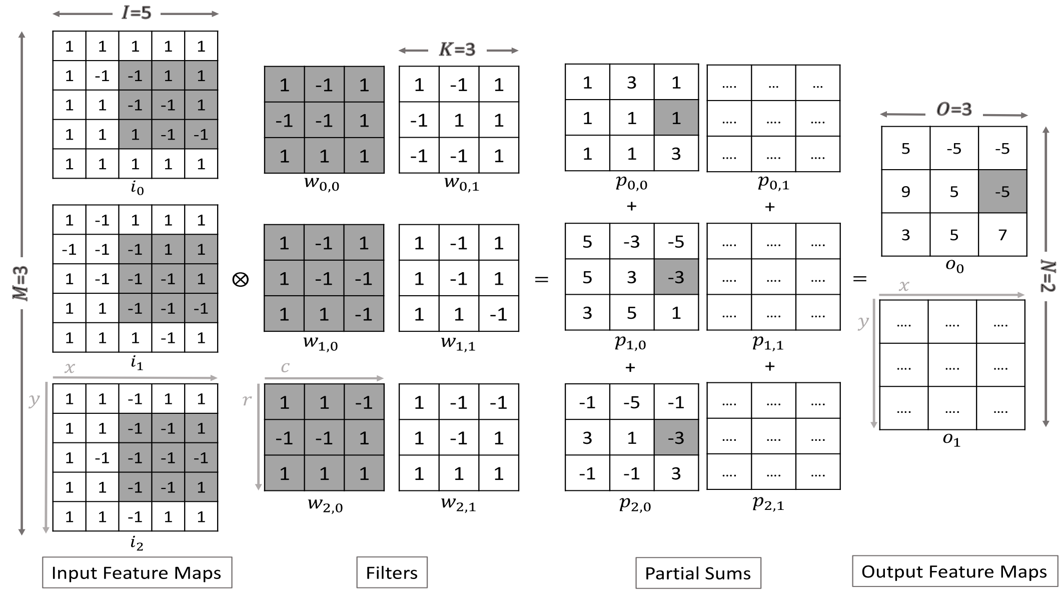
Benchmark	Application Field	Speedup
KNN Digit Recognition	Image classification	12.5
K-Means	Clustering	16.0
Smith-Waterman	Genomic sequencing	20.9

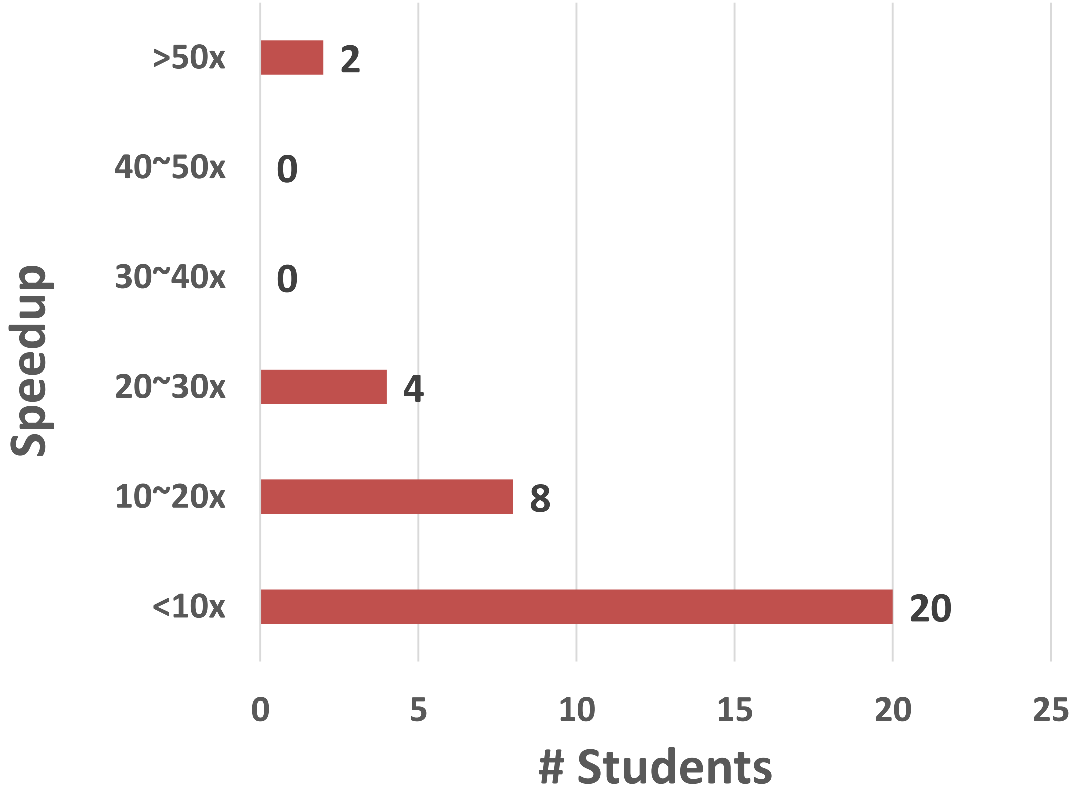
Rapidly achieve good speedup for a rich set of applications

Case Study: Binarized Neural Network (BNN)

- ECE 5775 (high-level digital design automation) at Cornell [1]
  - 34 students: graduates and senior undergrads

- In-class competition: higher speedup => higher score
  - Baseline: unoptimized BNN on ARM (Zynq)
  - Time: two weeks





[1] https://www.csl.cornell.edu/courses/ece5775/

Optimized BNN in HLS C

```
template<int M, int N, int I, int L>
void conv(ap_int<32> input[MAX_FMAP_PACK_SIZE],
        ap_int<32> output[MAX_FMAP_PACK_SIZE],
        const ap_int<8> threshold[MAX_FMAP],
        hls::LineBuffer<F, I, bit> buf[M]) {
 int O = I - F + 1, ifmap_size = I * I, ofmap_size = O * O;
 hls::Window<F, F, bit> window[M];
 for (int y = 0; y < O; y++) {
  for (int m = 0; m < M; m++) {
  #pragma HLS pipeline
   for( int x = 0; x < F - 1; x++) {
    int i_index = x + (y + F - 1) * I + m * ifmap_size;
    bit newBit = GET_BIT(input, i_index, PACK_WIDTH_LOG);
    fillBuffer<F, I>(window[m], buf[m], x, newBit);
  }}
  for (int x = 0; x < O; x++) {
   for (int m = 0; m < M; m++) {
    int i_index = x + F - 1 + (y + F - 1) * I + m * ifmap_size;
    bit newBit = GET_BIT(input, i_index, PACK_WIDTH_LOG);
    fillBuffer<F, I>(window[m], buf[m], x + F - 1, newBit);
   }
   for (int n = 0; n < N; n++) {
   #pragma HLS pipeline
    int sum = 0;
    int o_index = x + y * O + n * ofmap_size;
    for (int m = 0; m < M; m++) {
     int one_out = 0, mac_num = 0;
     for (int c = 0; c < F; c++) {
      for (int r = 0; r < F; r++) {
       if (if_mac(x + c, y + r, I)) { //neglect padding pixels in mac
        int i_index = x + c + (y + r) * I + m * ifmap_size;
        int w_index = c + r * F + (n + m * N) * FILTER_SIZE;
        if (L == 0) one_out += window[m].getval(r, c) == w_conv1[w_index];
        else      one_out += window[m].getval(r, c) == w_conv2[w_index];
        mac_num++;
     }}}
     sum += (one_out << 1) - mac_num;
    }
    SET_BIT(output, o_index, PACK_WIDTH_LOG, sum > threshold[o_index] ? 1 : 0);
}}}}
```

Applied customization techniques

- **Compute:** tiling, pipelining, reordering
- **Data type:** bit packing
- **Memory:** partitioning, line buffer, window buffer

Compute customization
Data type customization
Memory customization

Optimized BNN in HeteroCL

- Development time: < 3 days
- Final speedup: 63x

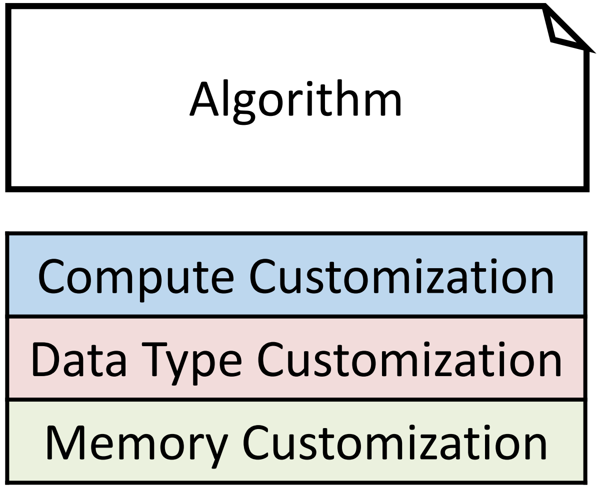
```
rc = hcl.reduce_axis(0, in_fmaps)
ry = hcl.reduce_axis(0, F)
rx = hcl.reduce_axis(0, F)
C = hcl.compute((1, out_fmaps, O, O),
  lambda nn, ff, yy, xx:
  hcl.select(
      hcl.sum(A[nn,rc,yy+ry,xx+rx] * B[ff,rc,ry,rx], axis=[rc,ry,rx]) >
         threshold[nn,ff,yy,xx], 1, 0 ),
  dtype=hcl.UInt(1))

s.quantize(C, hcl.UInt(32))
s[C].split(C.axis[1], factor=5)
s[C].unroll(C.axis[2], factor=5)
s[C].pipeline(C.axis[3])
lb = s[A].reuse_at(C, C.axis[0])
wb = s[lb].reuse_at(C, C.axis[1])
```

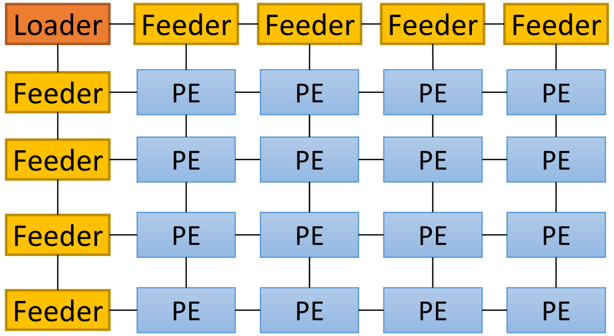
✓ More productive

✓ More maintainable

Conclusions

- HeteroCL is a multi-paradigm programming infrastructure
  - Decouples algorithm from compute, data type, and memory customization
  - Provides an abstraction capturing the interdependence and trade-offs
- Maps to spatial architecture templates with macros
  - Stencil with dataflow architecture
  - Systolic array
- Validated against a rich set of benchmarks from multiple domains
  - Image processing, linear algebra, deep learning, etc.





Next Stop

- Connecting with front-end DSLs
  - E.g., PyTorch & MXNet

- Open source release of HeteroCL is coming soon!

heterocl
0.1
Search docs

HeteroCL Tutorials
samples
digirect example
⊟ Python API
heterocl.api
heterocl.compute_api
heterocl.dsl
heterocl.nparray
heterocl.schedule
heterocl.scheme
heterocl.tensor
heterocl.types
Developer Reference

heterocl.api

`heterocl.init` ([init_dtype])	Initialze a HeteroCL environment with configurations.
`heterocl.placeholder` (shape[, name, dtype])	Construct a HeteroCL placeholder for inputs/outputs.
`heterocl.create_scheme` (inputs, func)	Create a quantization scheme.
`heterocl.create_schedule` (inputs[, func])	Create a schedule for compute optimizations.
`heterocl.create_schedule_from_scheme` (scheme)	Create a schedule from a scheme.
`heterocl.lower` (schedule)	Get the generated IR of a given schedule.
`heterocl.cast` (dtype, expr)	Cast an expression to specified data type.

`init`(*init_dtype='int32'*) [source]

Initialze a HeteroCL environment with configurations.

This API must be called each time the users write an application. Within the same HeteroCL environment, users can try different combinations of customization primitives.

Parameters: **init_dtype** (*Type, optional*) – The default data type for each variables

Examples

```
# app 1
hcl.init()
A = hcl.placeholder(...)
B = hcl.placeholder(...)
def app1(A, B):
    # define the algorithm for app 1
s = hcl.create_scheme([A, B], app1)
# apply customization primitives
f1 = hcl.build(s)
# execute f1
```

heterocl
0.1
Search docs

⊟ HeteroCL Tutorials
⊟ Getting Started
Import HeteroCL
Initialize the Environment
Algorithm Definition
Inputs/Outpus Definition
Apply Hardware Customization
Inspect the Intermediate Representation (IR)
Create the Executable
Prepare the Inputs/Outputs for the Executable
Run the Executable
View the Results
Imperative Programming
HeteroCL Compute APIs
Compute Customization
Data Type Customization
Custom Module Definition
samples
digirect example
Python API

Docs » HeteroCL Tutorials » Getting Started   View page source

Note

Click here to download the full example code

Getting Started

Author: Yi-Hsiang Lai (seanlatias@github)

In this tutorial, we demonstrate the basic usage of HeteroCL.

Import HeteroCL

We usually use `hcl` as the acronym of HeteroCL.

```
import heterocl as hcl
```

Initialize the Environment

We need to initialize the environment for each HeteroCL application. We can do this by calling the API `hcl.init()`. We can also set the default data type for every computation via this API. The default data type is **32-bit** integers.

Note

For more information on the data types, please see Data Type Customization.

HeteroCL

Yi-Hsiang Lai[1], Yuze Chi[2], Yuwei Hu[1], Jie Wang[2], Cody Hao Yu[2,3], Yuan Zhou[1], Jason Cong[2], Zhiru Zhang[1]

[1]Cornell University, [2]University of California, Los Angeles, [3]Falcon Computing Solutions, Inc.

Questions?