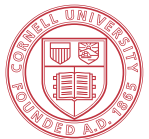


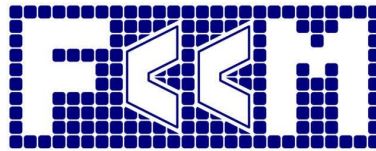
Building High-Performance Systolic Arrays with HeteroCL

Yi-Hsiang Lai, Shaojie Xiang, Zhiru Zhang

05/12/2021

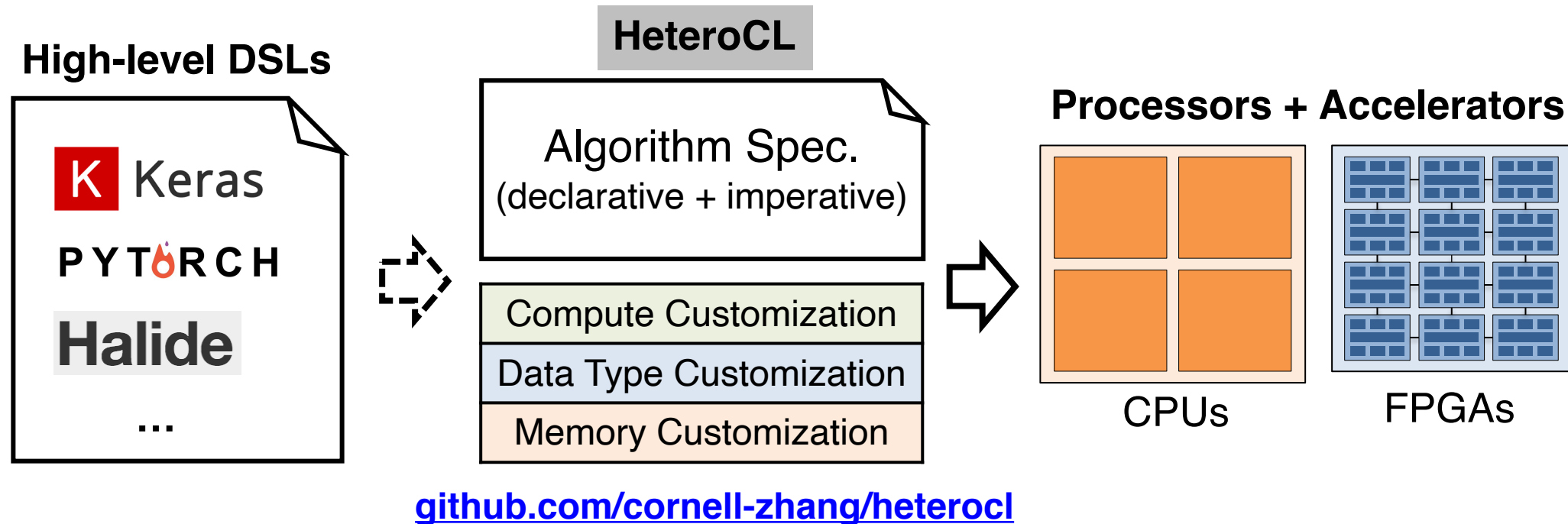


Cornell University



HeteroCL Overview

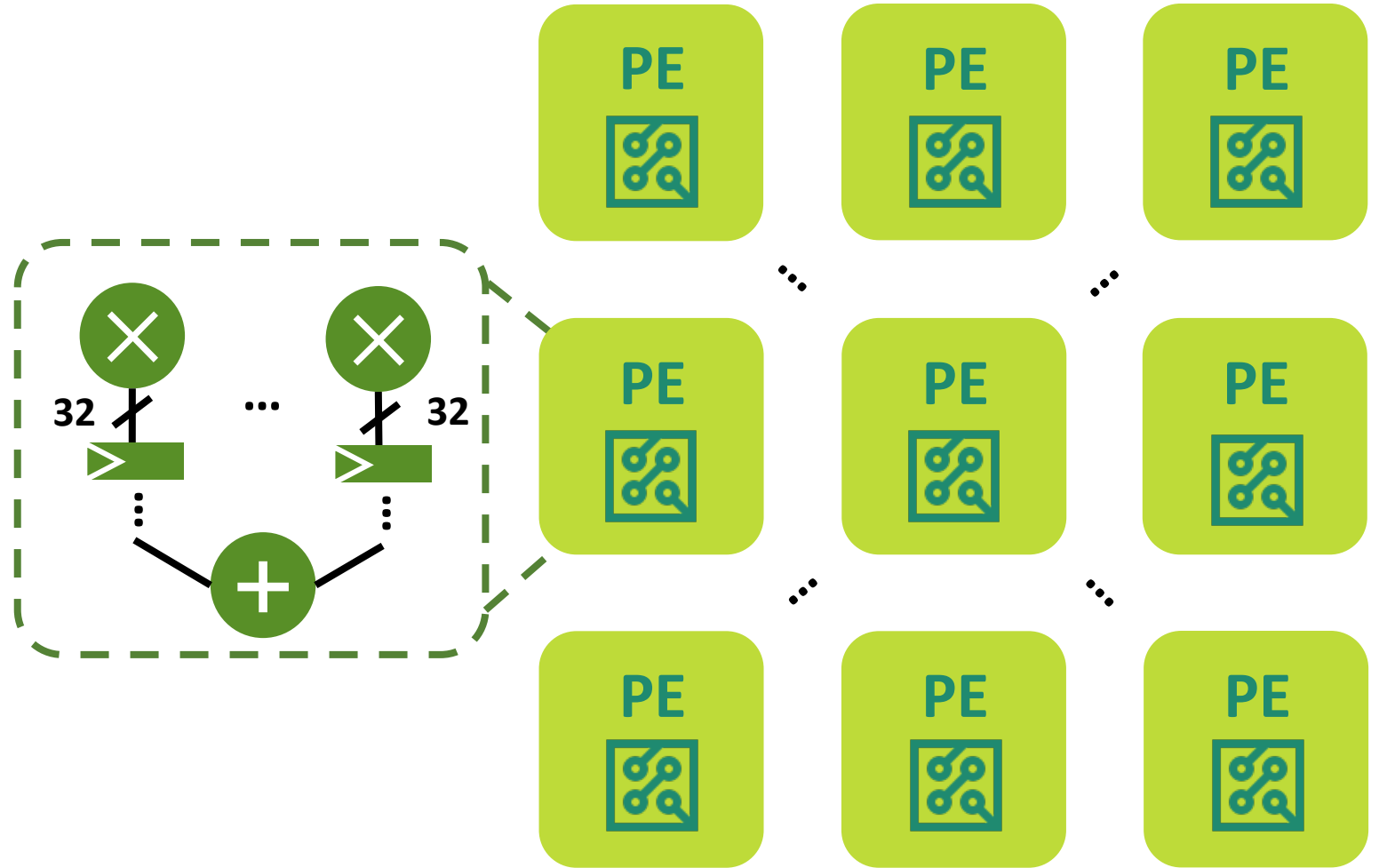
- ▶ A Python-based programming framework for FPGA-targeted compute acceleration
 - **Productive**: Succinct yet flexible programming abstraction
 - **Performant**: Efficient mapping to highly efficient spatial architectures
 - **Portable**: Clean decoupling of algorithm & hardware customizations



Essential Techniques for Hardware Acceleration

Compute customization (Parallelism)

- Pipelining, Unrolling, ...



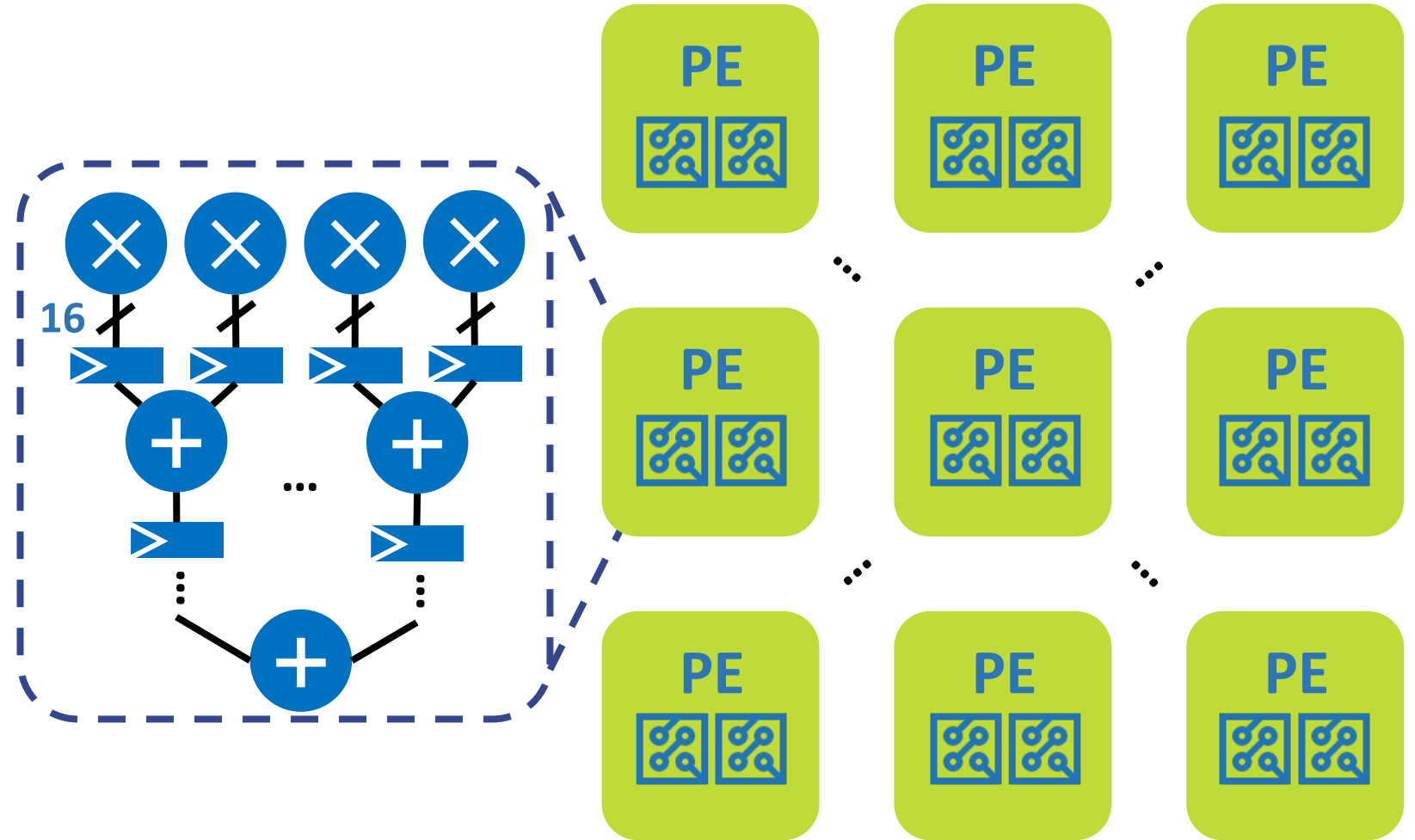
Essential Techniques for Hardware Acceleration

Compute customization (Parallelism)

- Pipelining, Unrolling, ...

Data type customization (Precision)

- Low-bitwidth integer,
Fixed point, ...



Essential Techniques for Hardware Acceleration

Compute customization (Parallelism)

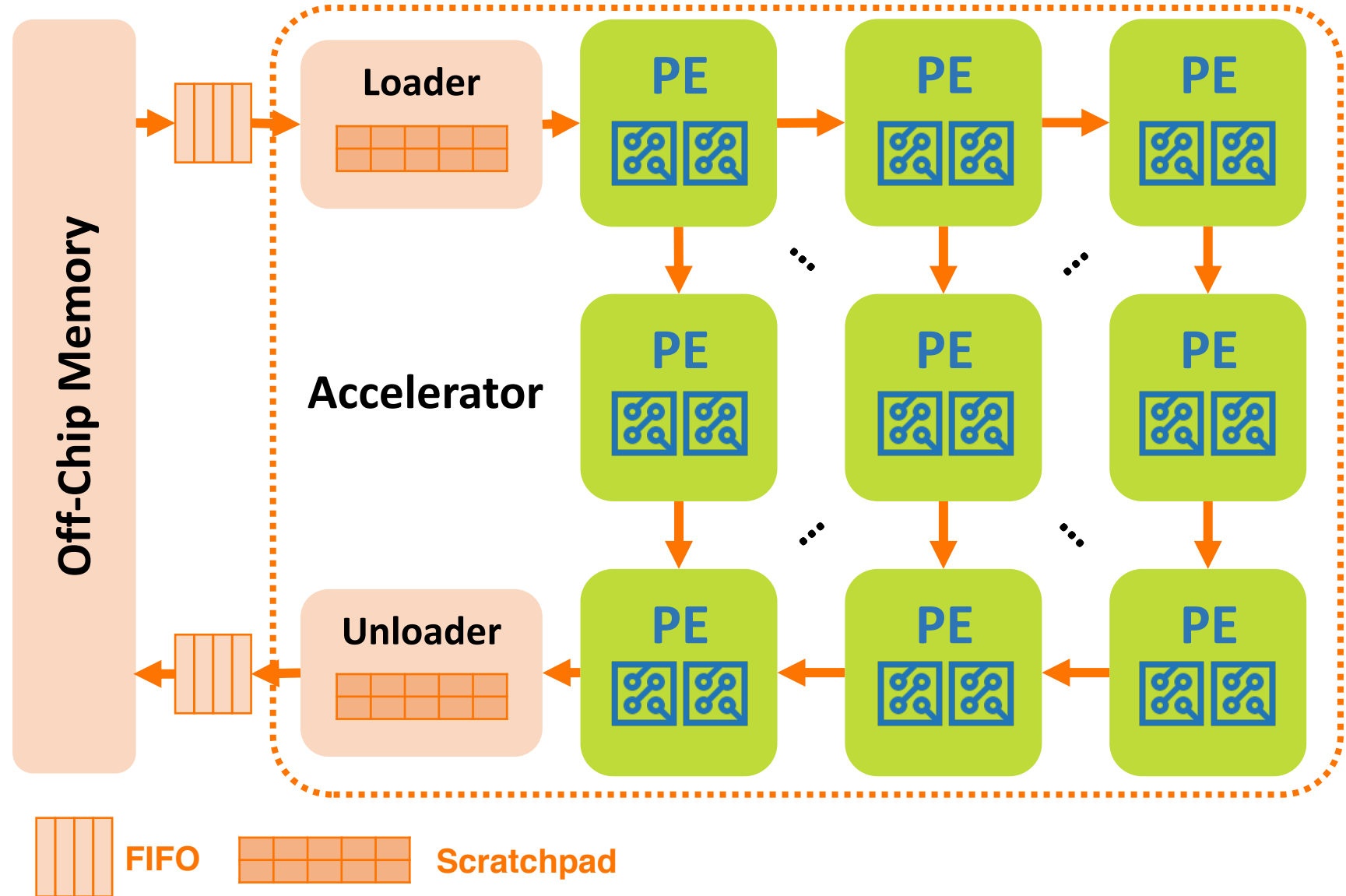
- Pipelining, Unrolling, ...

Data type customization (Precision)

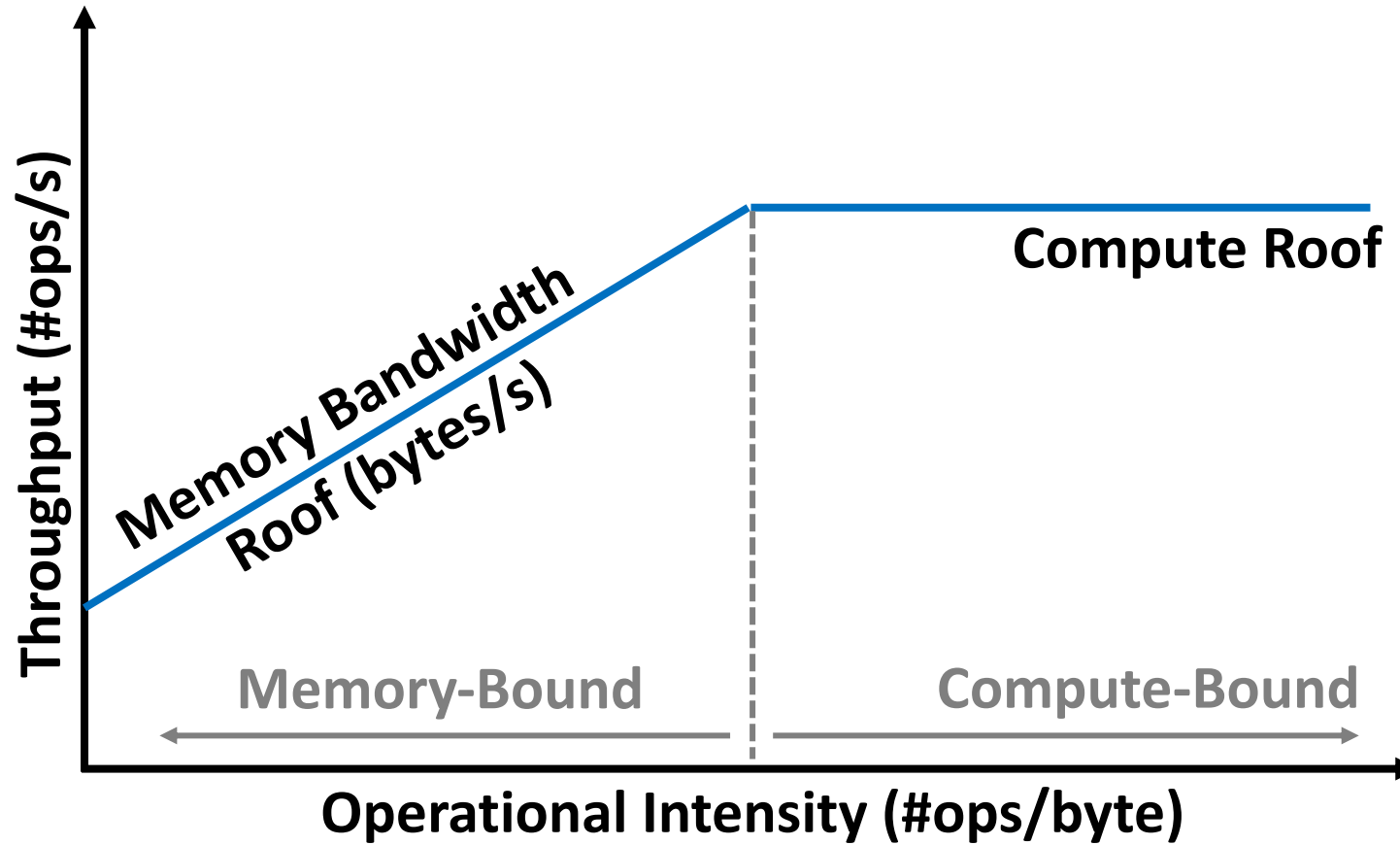
- Low-bitwidth integer,
Fixed point, ...

Memory customization (Data placement)

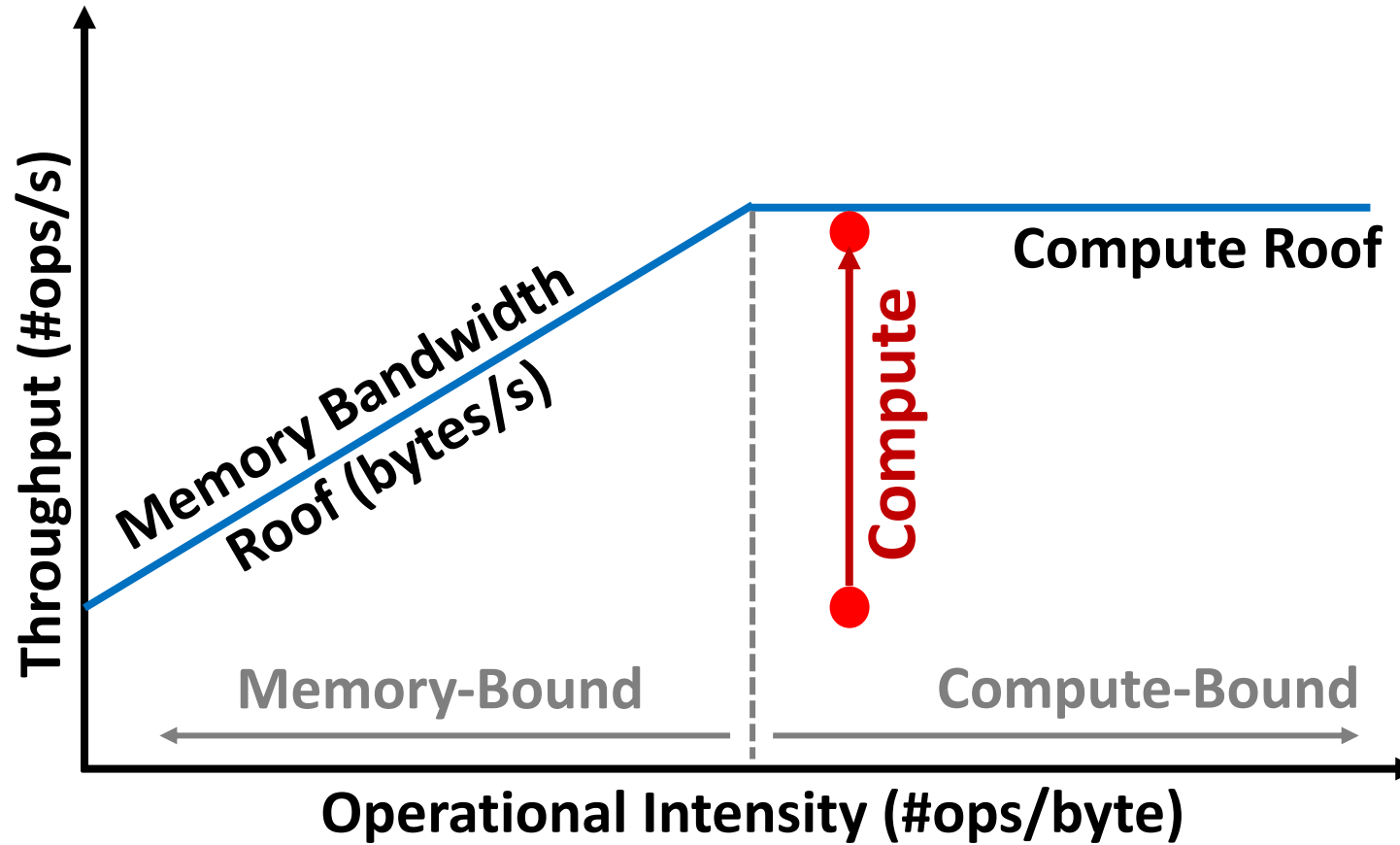
- Banking, Data reuse,
Streaming ...



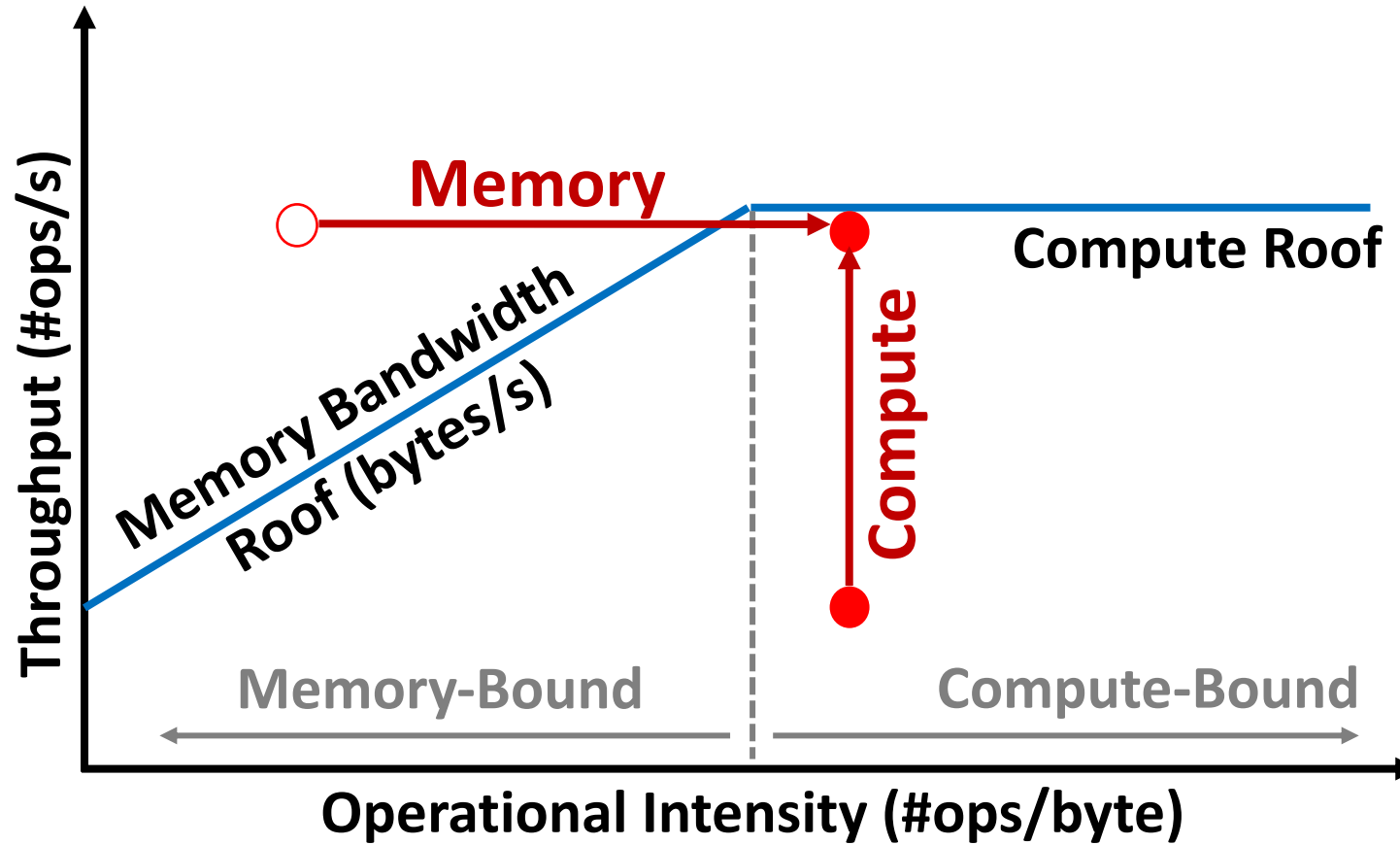
Customizations in Roofline Diagram



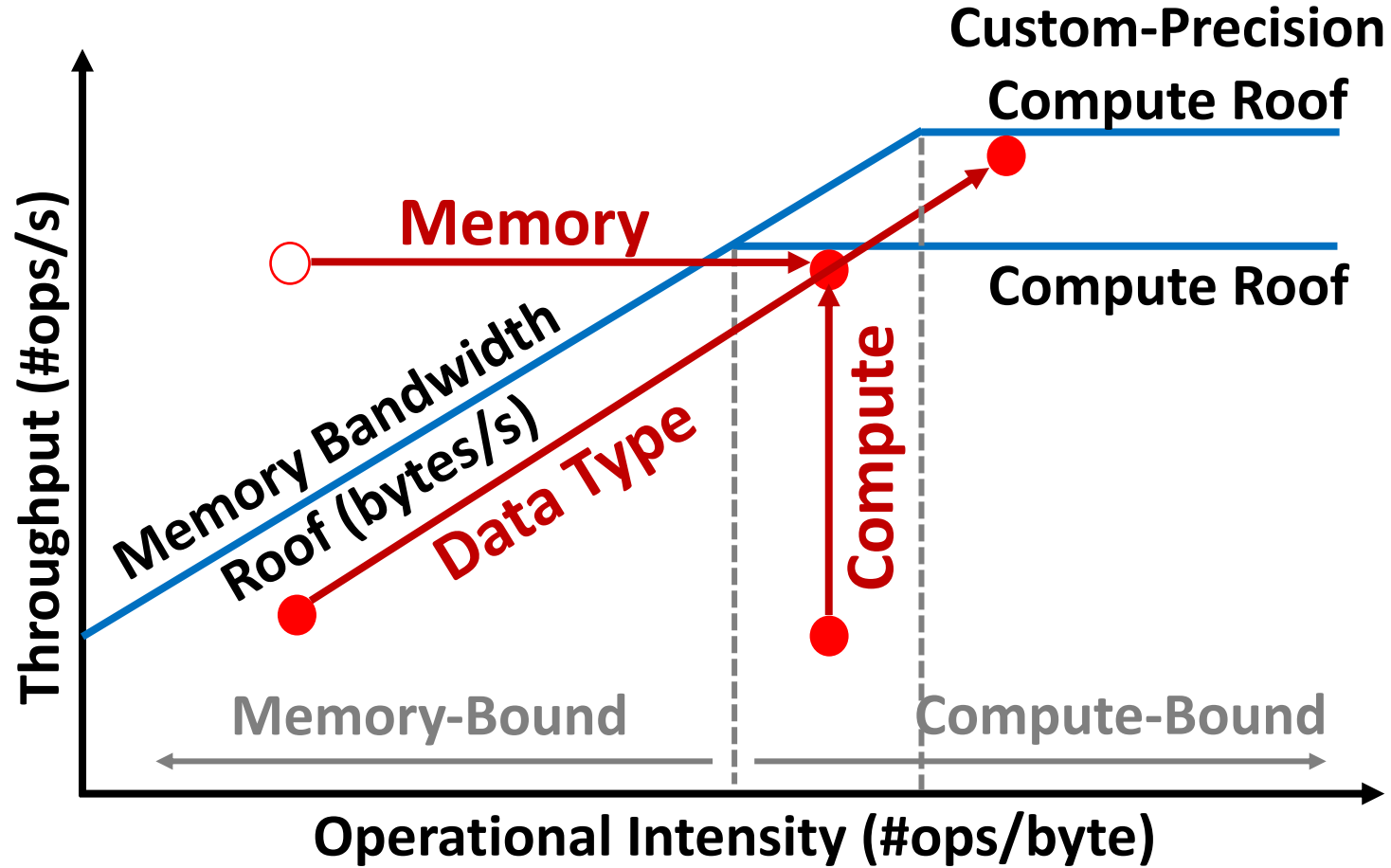
Customizations in Roofline Diagram



Customizations in Roofline Diagram



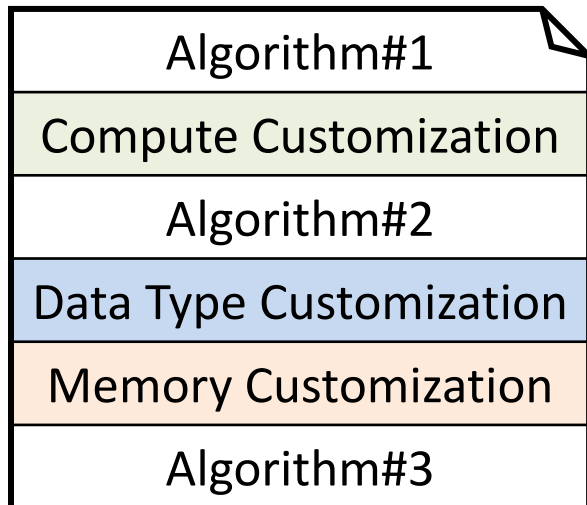
Customizations in Roofline Diagram



FPGA Programming with HLS

► Example: convolution

```
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] * filter[r, c]
```



Entangled hardware customization and algorithm

- Less portable
- Less readable
- 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) * filter[r][c];
```

```
          }
```

```
        out[y-2][x-2] = acc;
```

```
      }  
    }  
  }  
}
```

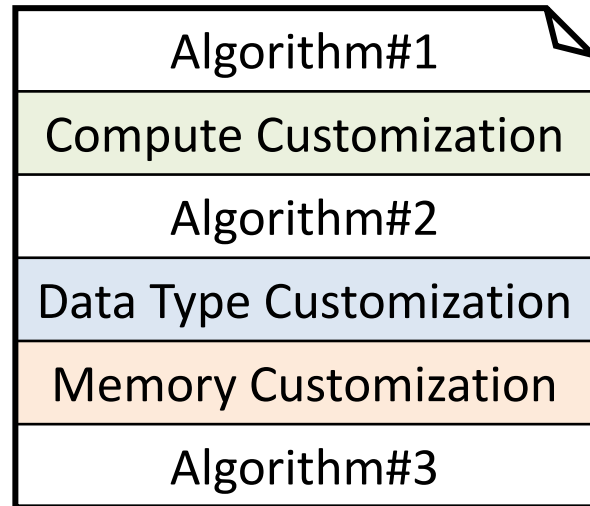
**Custom compute
(Loop tiling)**

**Custom data type
(Quantization)**

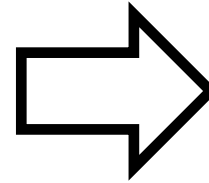
**Custom memory
(Reuse buffers)**

Decoupling Algorithm from Hardware Customizations

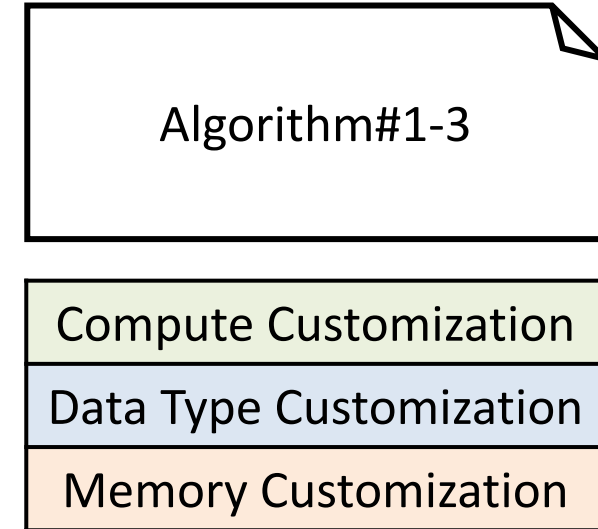
HLS C/C++



Entangled algorithm specification and customization schemes



HeteroCL



Fully decoupled customization schemes
(More Productive & Portable)

Hardware Customizations in HeteroCL

HeteroCL code

Algorithm {

```

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 code
(based on TVM)

Custom Compute

```

s = hcl.create_schedule()
s[out].unroll([r,c])
    
```

Custom Data Type

```

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

Custom Memory

```

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

Corresponding C 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]
    
```

Unroll inner loops

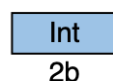
32-bit Floating-point



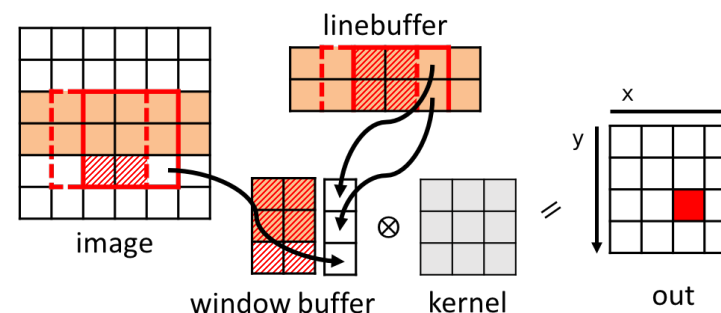
8-bit Fixed-point



2-bit Integer



Quantize/downsize

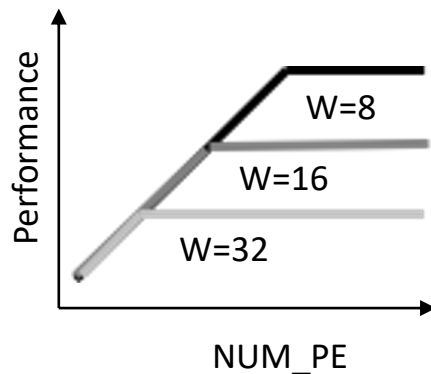
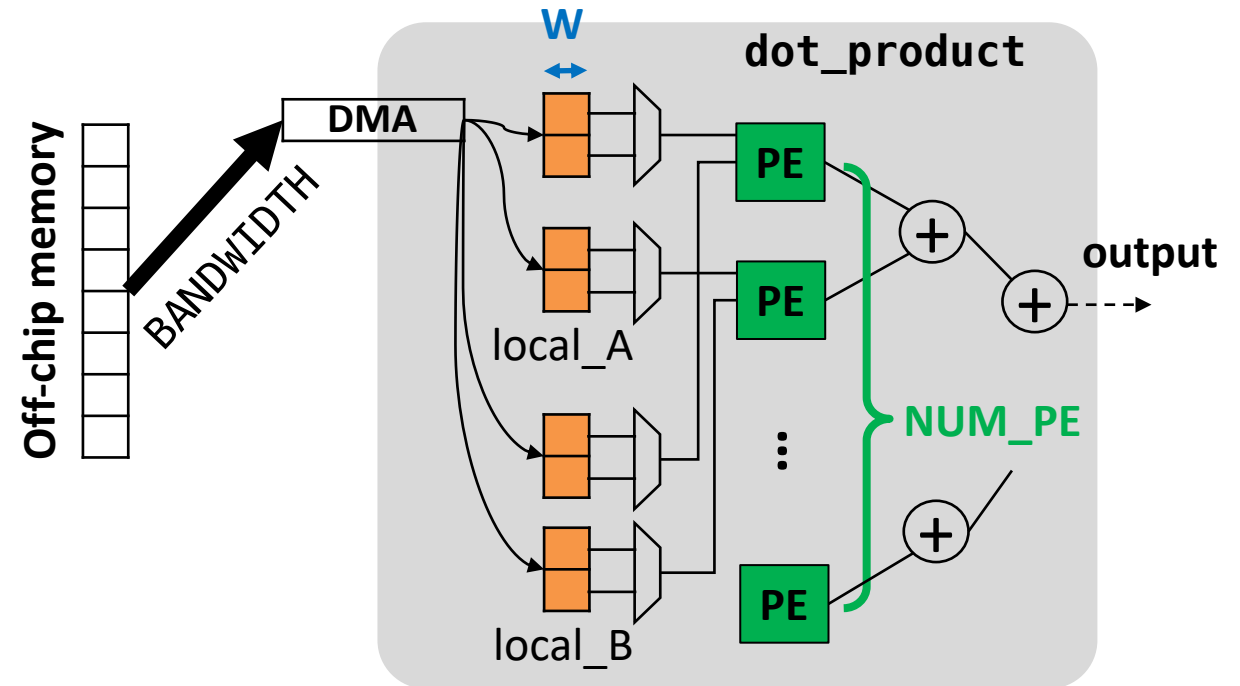


Exploring the Interdependence amongst Customizations

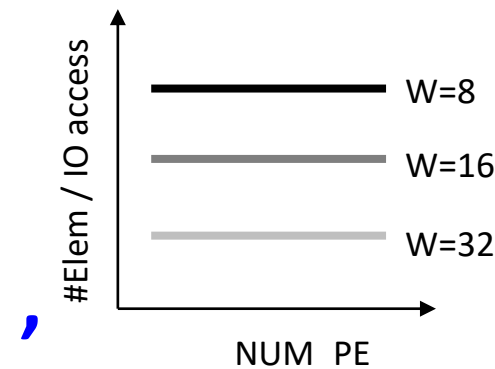
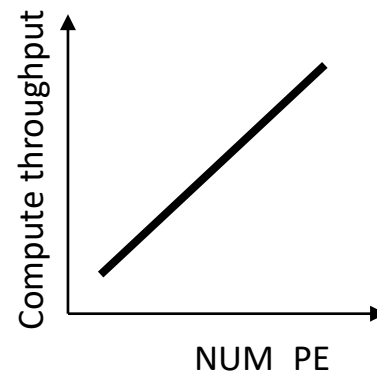
```
i = hcl.reduce_axis(0, N)
return hcl.compute((1,),
    lambda x: hcl.sum(local_A[i] * local_B[i],
        axis=i))
```

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

Compute
Data type
Memory



= min(



)

Decoupled Compute Customization

HeteroCL code

Algorithm

```
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]))
```

Decoupled customization

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

Customization primitives

- Portable, less error-prone

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]
```

```
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 Data Type Customization

- ▶ Bit-accurate data type support (e.g., Int(15), Fixed(7,4))
 - WIP: custom floating-point types (e.g., bfloat16)
- ▶ Decoupled customization primitives: **downsize** & **quantize**

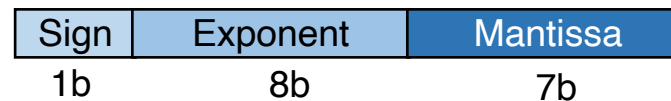
```
r = hcl.reduce_axis(0, 3)
c = hcl.reduce_axis(0, 3)
out = hcl.compute(N, N),
    lambda y, x:
        hcl.sum(image[x+r, y+c]*kernel[r, c],
                axis=[r, c]))
```

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

32-bit Floating-point



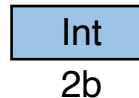
16-bit Brain Floating-point (bfloat)



8-bit Fixed-point Fixed(8, 6)



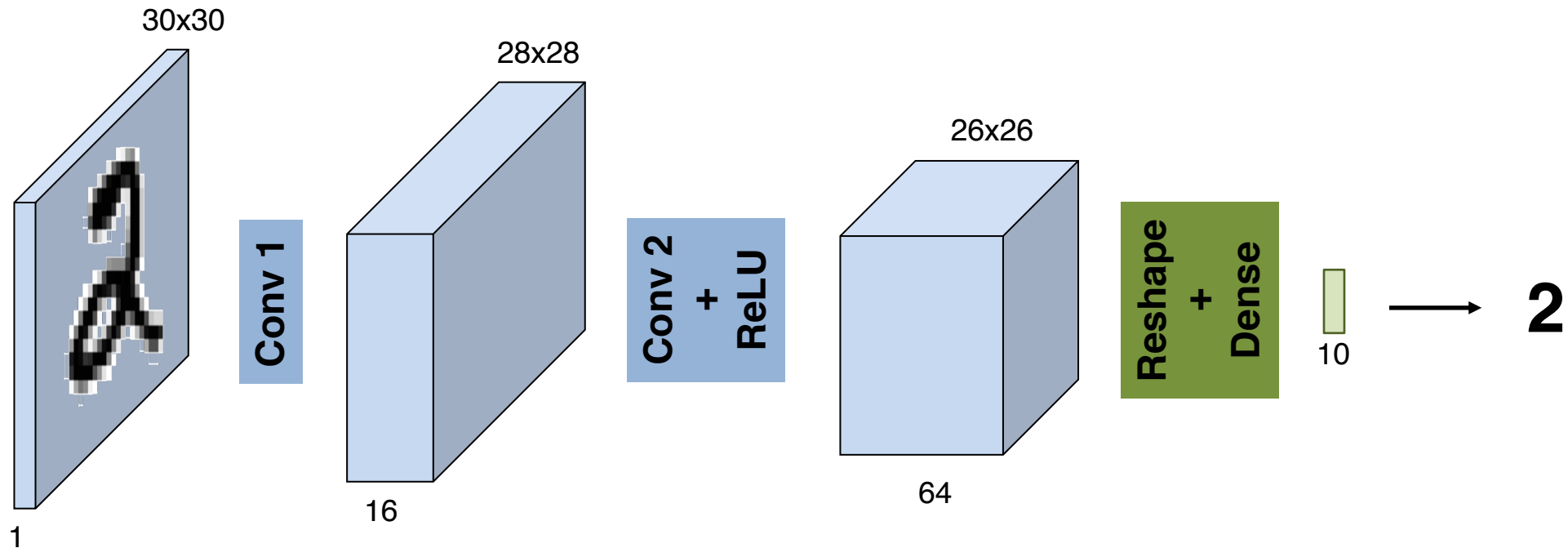
2-bit Integer Int(2)



Quantize/downsize

Case Study: Simple Convolutional Neural Network

- ▶ Digit recognition with MNIST dataset
 - Train the model in Keras
 - Run the inference in HeteroCL
- ▶ Goal: (Partially) deploy the model to FPGA for acceleration



Imperative Programming in HeteroCL

- ▶ HeteroCL further provides an embedded imperative DSL
 - Not all algorithms can be described in declarative tensor-style code
- ▶ Imperative & declarative programs share a unified interface for customization primitives

```
def reshape():  
  with hcl.for_(0, I) as i:  
    with hcl.for_(0, J) as j:  
      with hcl.for_(0, K) as k:  
        B[(i * J + j) * K + k] = A[i, j, k]
```

```
s = hcl.create_schedule()  
s[reshape].unroll(reshape.k)  
s.quantize([reshape.B], Fixed(6, 4))
```

Demo 1: Syntax & Data Quantization

► Basic components of a HeteroCL program

1. Placeholder: tensors served as inputs/outputs
2. Algorithm definition
3. Hardware customization
4. Function implementing #2 and #3
5. Input data in NumPy arrays

► Main program can contain both imperative and declarative code

```
import heterocl as hcl
import numpy as np
```

```
1 img = hcl.placeholder(input_size)
  conv_w1 = hcl.placeholder((16,1,3,3))
  conv_w2 = hcl.placeholder((64,16,3,3))
  dense_w = hcl.placeholder((64*26*26,10))
```

```
2 def top(img, conv_w1, conv_w2, dense_w):
  output1 = conv2d(img, conv_w1)
  output2 = conv2d(output1, conv_w2)
  output3 = reshape(relu(output2))
  return dense(output3, dense_w)
```

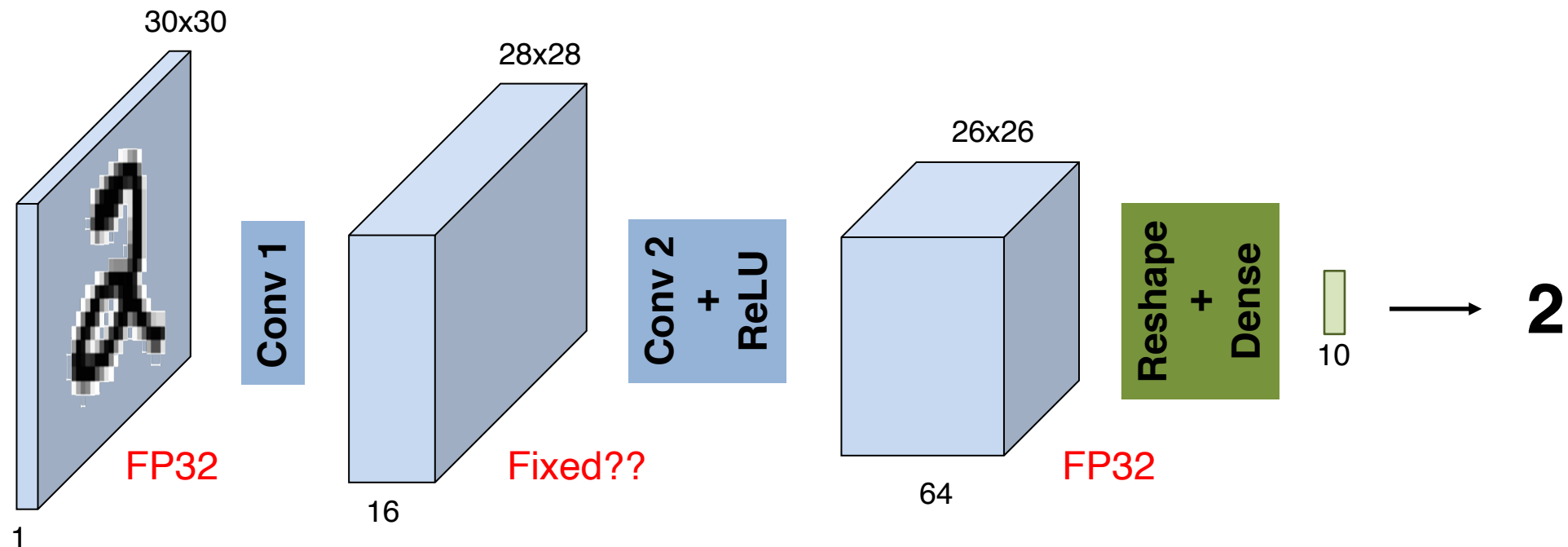
```
3 s = hcl.create_schedule(
  [img, conv_w1, conv_w2, dense_w], top)
```

```
4 f = hcl.build(s, target=p)
```

```
5 with open('convnet.npy', 'rb') as fp:
  w1 = np.load(fp)
  w2 = np.load(fp)
  w3 = np.load(fp)
  conv_w1 = ...
  conv_w2 = ...
```

Demo 1: Syntax & Data Quantization

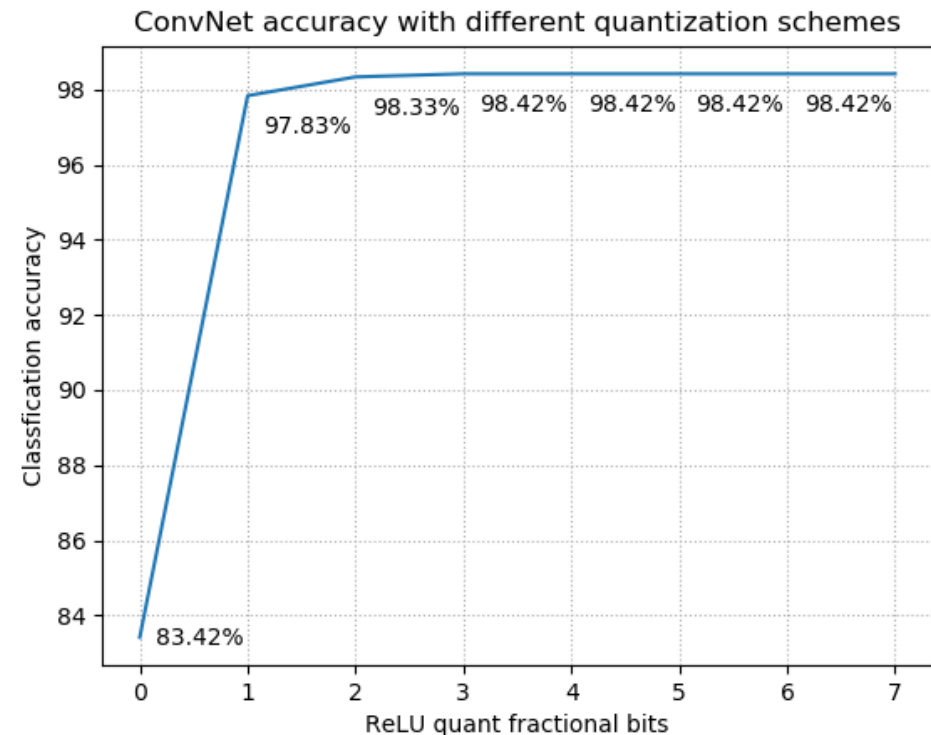
- ▶ Post-training quantization for smaller model size and higher throughput
 - Analyze the output range to determine the integer bitwidth
 - Quantize output of second layer (i.e., ReLU)
- ▶ Easily try out different quantization schemes in HeteroCL



Demo 1: Code & Results Review

- ▶ Easily explore the trade-off between accuracy & resource with **.quantize()**
 - The integer is set to 2 bits

```
def ConvNet(dtype_quant, quantize=True):  
    # A three layer ConvNet example  
    def top(img, conv_w1, conv_w2, dense_w):  
        # ...  
  
        # Data type customization  
        scheme = hcl.create_scheme(...)  
        if quantize:  
            scheme.quantize([top.relu], dtype_quant)  
        s = hcl.create_schedule_from_scheme(scheme)  
        # ...  
  
if __name__ == "__main__":  
    args = parser.parse_args()  
    if args.quantize:  
        if args.dse:  
            integer_bits = 2  
            for frac_bits in range(7):  
                dtype = hcl.Fixed(integer_bits+frac_bits, frac_bits)  
                ConvNet(dtype, quantize)
```



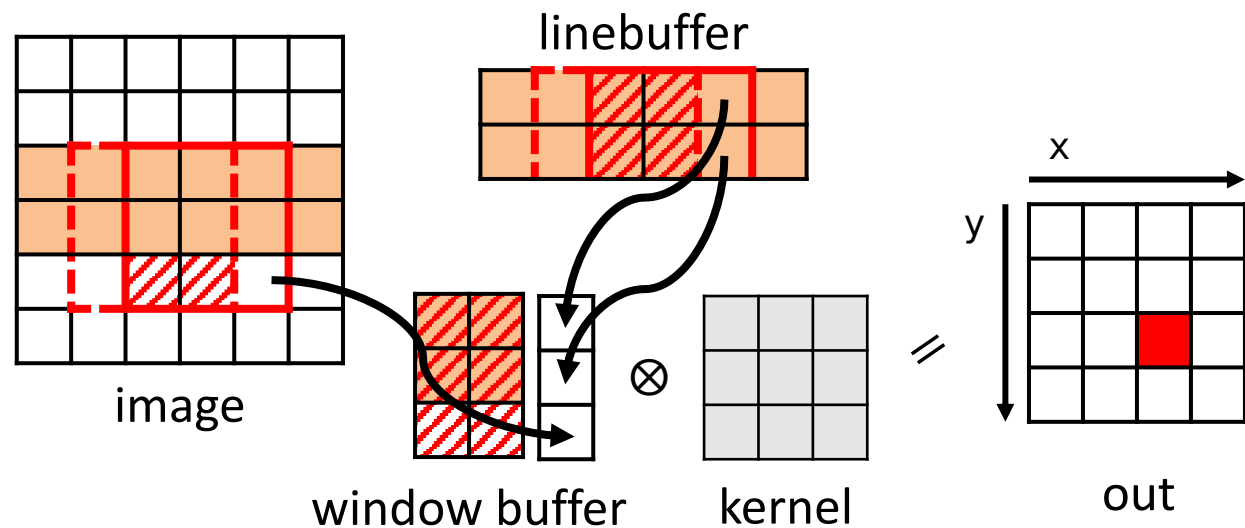
Decoupled Memory Customization

- ▶ Inferring custom on-chip storage with the **reuse_at** primitive

```
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]
```



Host-Device Data Placement

.to() : A unified programming interface for specifying data flow between

1. Host and accelerator (i.e., device)

```
from heterocl import platform
```

```
conv1 = conv(image, weight1, "conv1")  
conv2 = conv(conv1, weight2, "conv2")  
out = relu(conv2, "relu")
```

```
# decoupled customizations
```

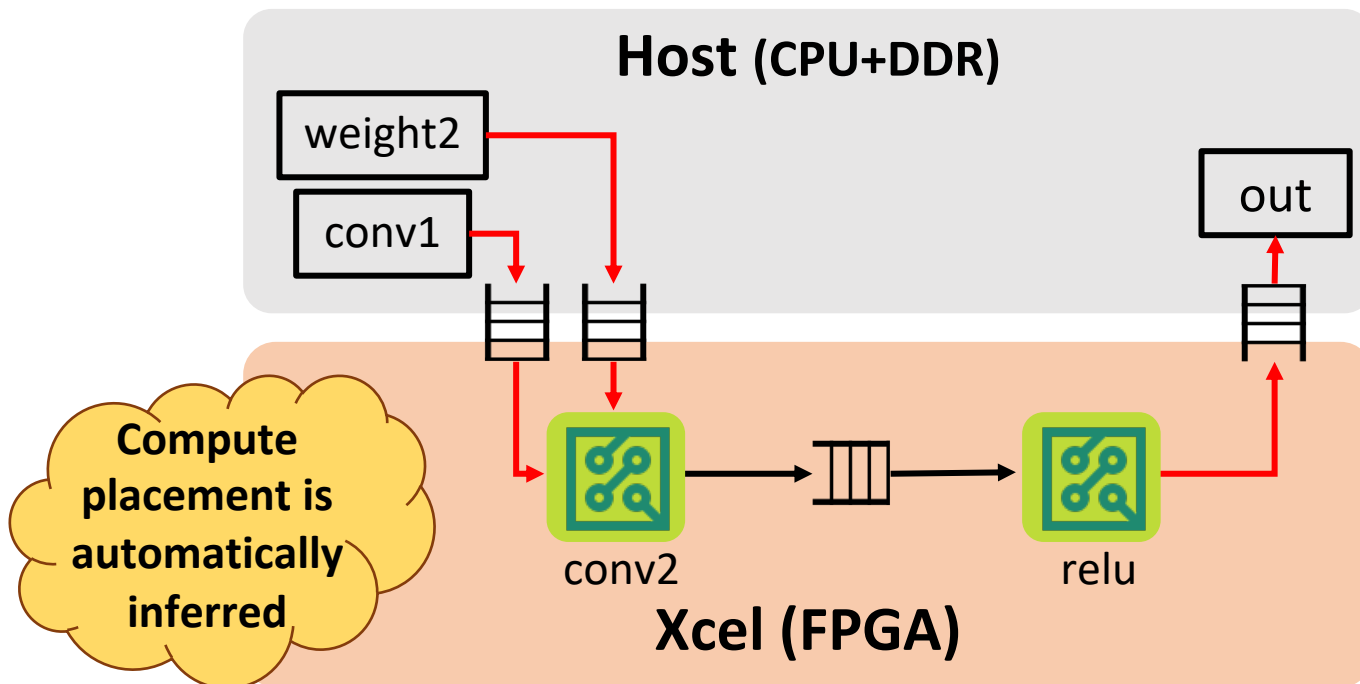
```
s = hcl.create_schedule()
```

```
p = platform.xilinx_u280
```

```
# specify data placement
```

```
s.to([conv1, weight2], p.xcel)
```

```
s.to(out, p.host)
```



Kernel-Kernel Data Placement

.to() : A unified programming interface for specifying data flow between

1. Host and accelerator (i.e., device)
2. Sub-modules of the accelerator (i.e., kernels)

```
from heteroocl import platform
```

```
conv1 = conv(image, weight1, "conv1")  
conv2 = conv(conv1, weight2, "conv2")  
out = relu(conv2, "relu")
```

```
# decoupled customizations
```

```
s = hcl.create_schedule()
```

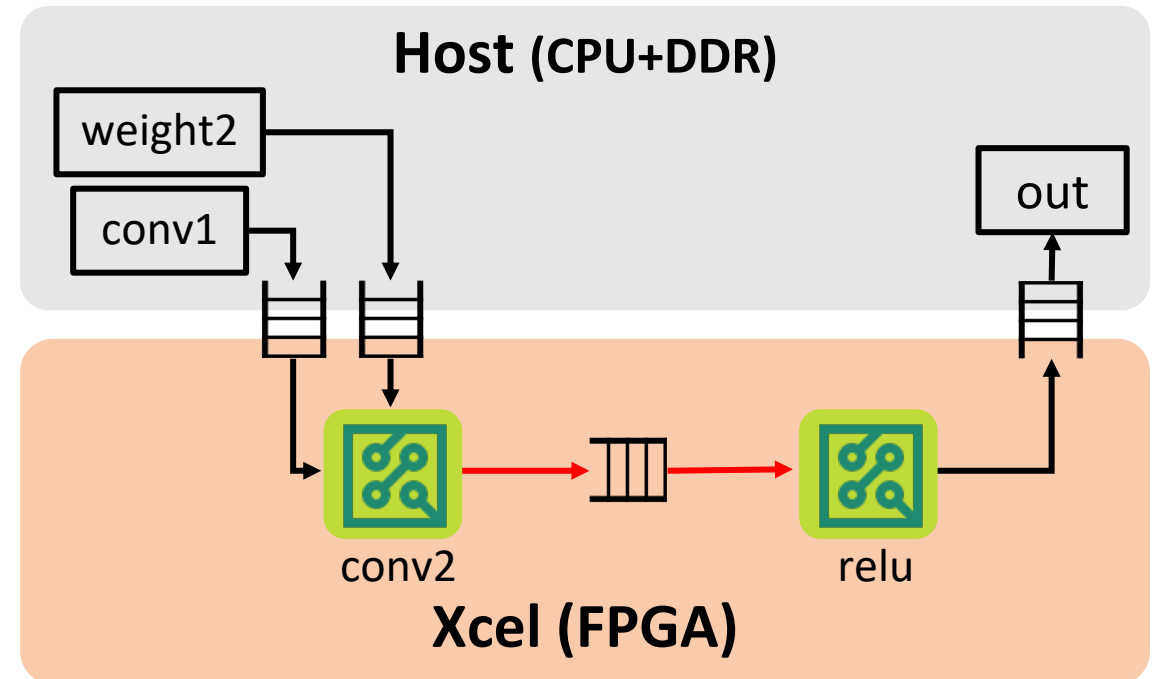
```
p = platform.xilinx_u280
```

```
# specify data placement
```

```
s.to([conv1, weight2], p.xcel)
```

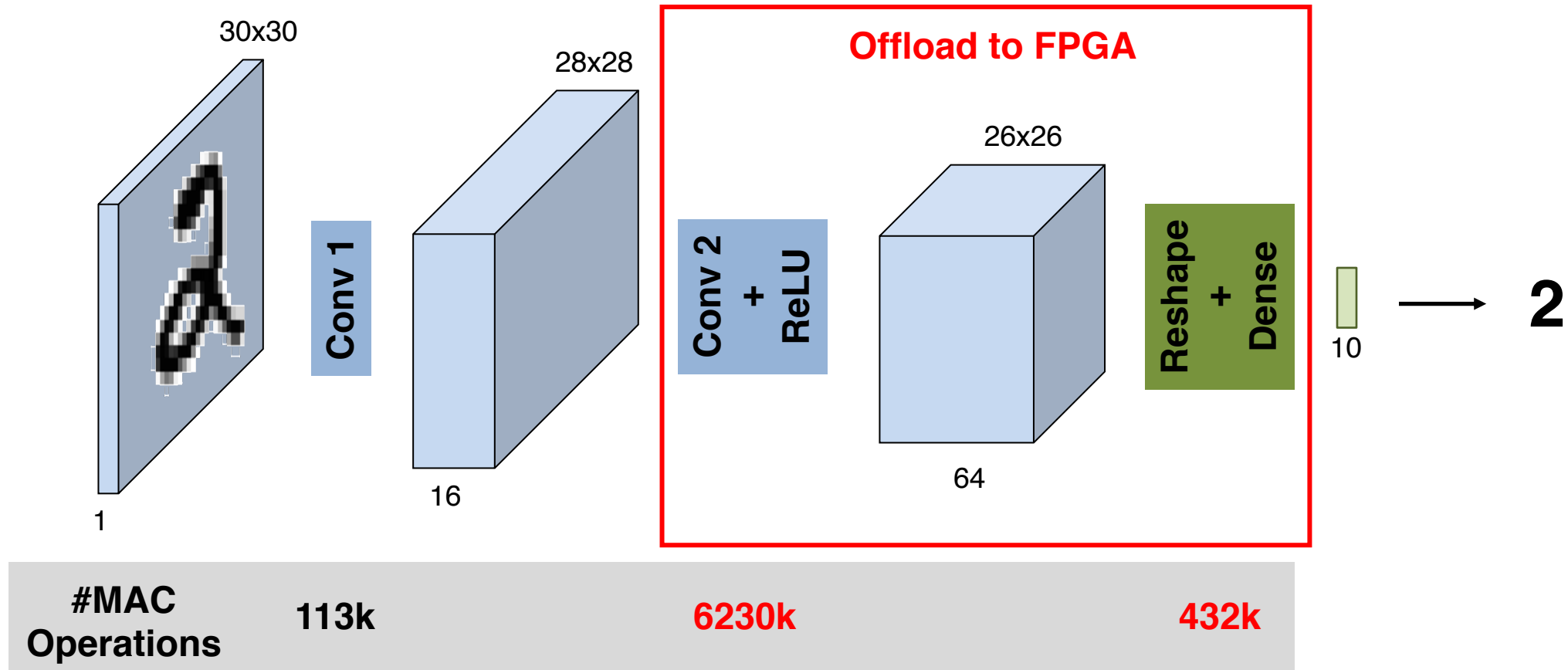
```
s.to(out, p.host)
```

```
s.to(conv2, relu)
```



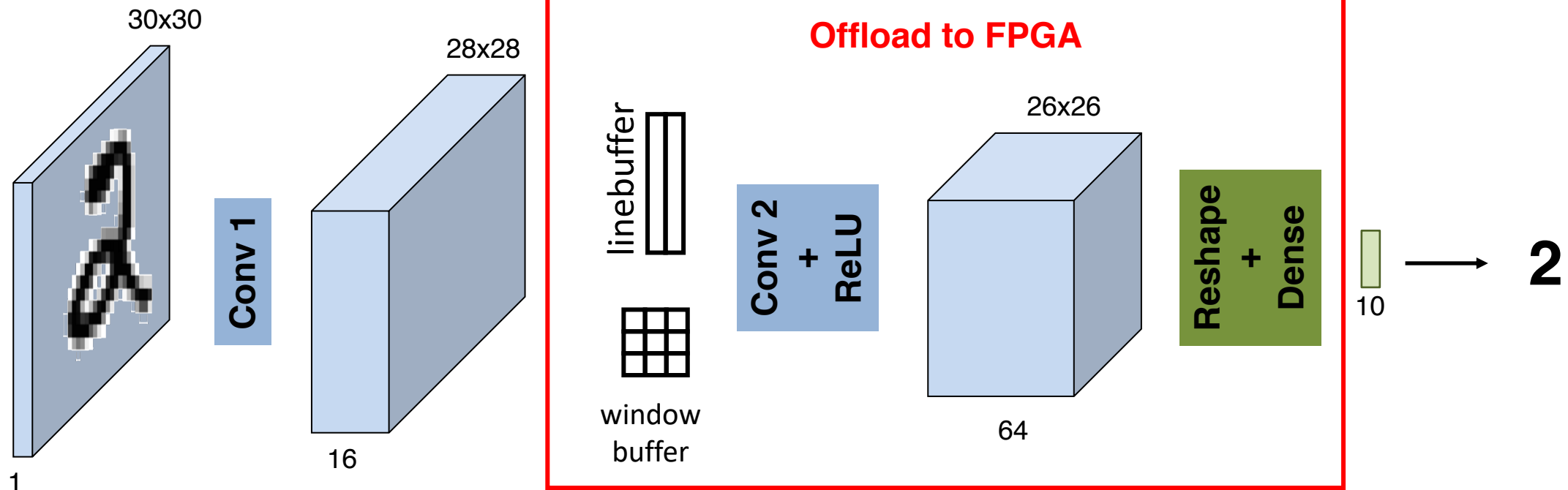
Demo 2: Data Placement & Data Reuse

- ▶ Identify compute-intensive parts and offload them to FPGA
 - Calculate the number of multiply-accumulate (MAC) operations of each layer
 - Specify data placement with `.to()`



Demo 2: Data Placement & Data Reuse

- ▶ Apply `.reuse_at()` between the first and second conv layer
 - Data can now be streamed **in serial** from host to FPGA with reuse buffers
 - Further improve the performance by streaming between Conv 2 and Dense with `.to()`



Demo 2: Code & Results Review

- ▶ Use **.to()** to specify dataflow between host and FPGA

```
p = hcl.Platform.xilinx_u280
s.to([top.conv1, conv_w2, dense_w], p.xcel)
s.to(top.dense, p.host)
```

- ▶ Use **.to()** to specify dataflow between on-chip submodules

```
s.to(top.conv2, top.relu)
s.to(top.relu, top.reshape)
s.to(top.reshape, top.dense)
```

- ▶ Use **.reuse_at()** to insert reuse buffers

```
LB = s.reuse_at(top.conv1, s[top.conv2], top.conv2.axis[1])
WB = s.reuse_at(LB, s[top.conv2], top.conv2.axis[2])
```

Demo 2: Code & Results Review

- ▶ Use **.config()** to specify HLS tool options

```
p.config(compile="vivado_hls", mode="csyn", project="hcl_prj_reuse_hls")
```

- ▶ Use **.report()** to retrieve the loop information (e.g., pipeline II)

Before Applying Data Reuse

```
top
+-----+-----+-----+-----+
|          | Trip Count | Latency | Pipeline II | Pipeline Depth |
+-----+-----+-----+-----+
| conv2_i_conv2_i1_conv2_i2 |      43264 | 13022464 |          N/A |          N/A |
| + conv2_i3                  |         16 |       298 |           18 |           29 |
| relu_args2_relu_args01_relu_args11 |      43264 |    43265 |            1 |            3 |
| reshape_i6_reshape_i7_reshape_i8 |      43264 |    43264 |            1 |            2 |
```

After Applying Data Reuse

```
+-----+-----+-----+-----+
| conv2_oc_conv2_h_reuse_conv2_w_reuse |      802816 |    802867 |            1 |           53 |
+-----+-----+-----+-----+
* Units in clock cycles
```

Current List of Customization Primitives

Compute customization

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

Data type customization

<code>downsize(t, d)</code>	Downsize a list of tensors t to type d .
<code>quantize(t, d)</code>	Quantize a list of tensors t to type d .

Memory customization

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

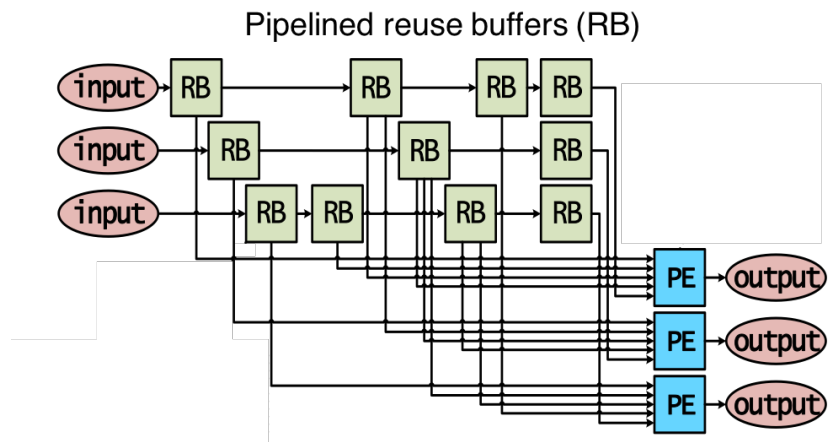


Macros for spatial architecture templates

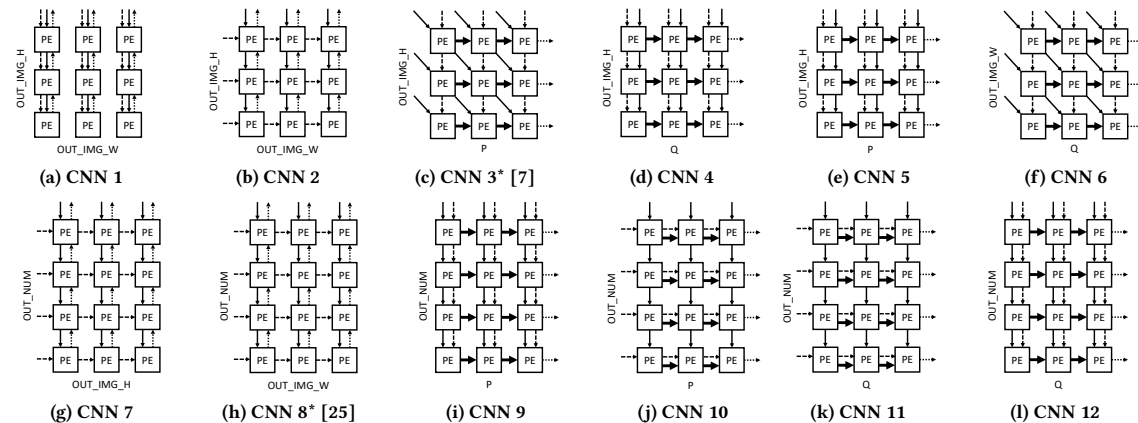
<code>C.stencil()</code>	Specify operation C to be implemented with stencil with dataflow architectures using the SODA framework.
<code>C.systolic()</code>	Specify operation C to be implemented with systolic arrays using the PolySA framework.

Targeting Spatial Architectures in HeteroCL

- ▶ HeteroCL compiler generates highly efficient spatial architectures with
 1. **SODA** for **stencil code** (data elements on a tensor accessed based on a fixed pattern)
 2. **AutoSA** for **systolic arrays** (a homogeneous array of locally connected compute units)



- SODA backend [J. Cong, et al. ICCAD'18]
 - Dataflow architecture that guarantees full data reuse without banking conflict



- AutoSA backend [J. Wang, et al. FGPA'21]
 - Produces a variety of systolic arrays with polyhedral transformation
 - Incorporates additional architecture optimizations (banking, SIMD, latency hiding, etc.)

More on AutoSA Integration

- ▶ Make it possible to connect systolic array kernels with other kernels
 - Use **.systolic()** to map a kernel to systolic arrays generated by AutoSA
 - Use **.to()** to connect the generated systolic arrays with other kernels

```
conv1 = conv(image, weight1, "conv1")
conv2 = conv(conv1, weight2, "conv2")
dense = dense(conv2, weight3, "dense")
```

```
# decoupled customizations
```

```
s = hcl.create_schedule()
```

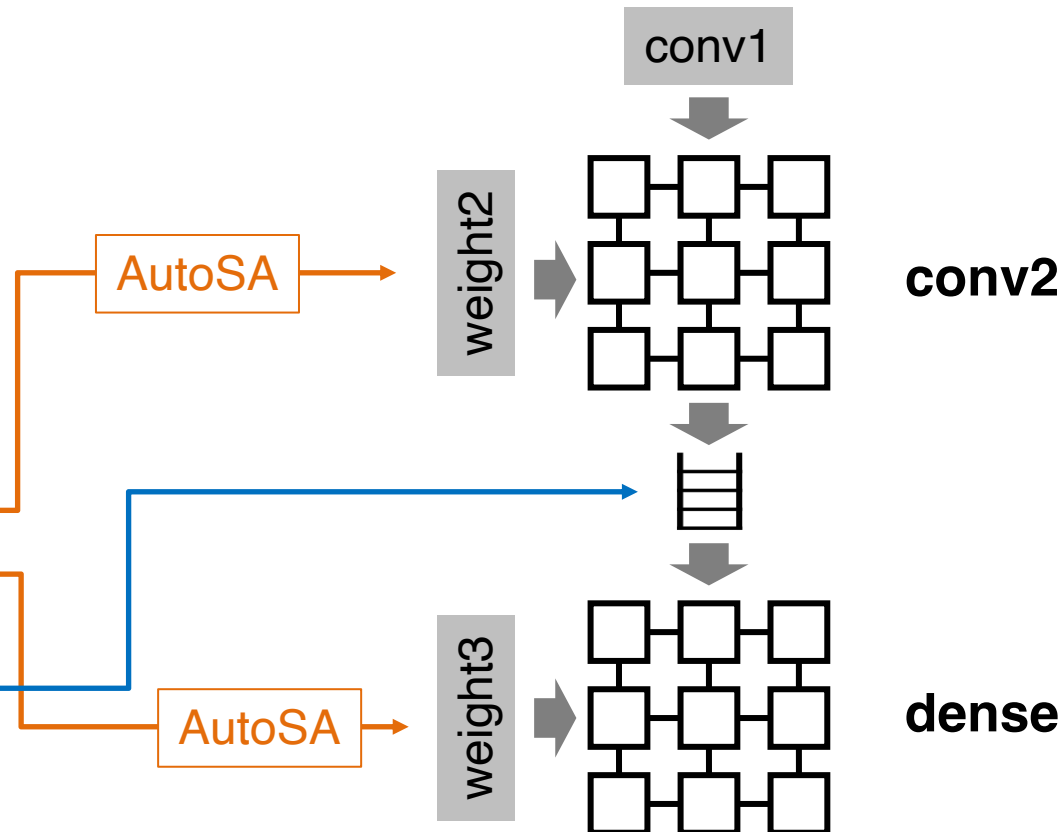
```
# map to AutoSA systolic arrays
```

```
s[conv2].systolic()
```

```
s[dense].systolic()
```

```
# specify inter-kernel data movement
```

```
s.to(conv2, dense)
```



More on AutoSA Integration

- ▶ Provide default configuration for AutoSA to generate systolic arrays
 - Size is selected according to the shape of the loop (default 8 x 8)
 - Tile/Flatten the loop if necessary
 - Can also be configured by users explicitly via **.systolic(params)**

Input Program

```
for (int y = 0; y < 26; y++)  
  for (int x = 0; x < 26; x++)  
    for (int r = 0; r < 64; r++)  
      for (int c = 0; c < 10; c++)  
        // compute ...
```



HeteroCL Transformed Program

```
for (int t = 0; t < 26*26*8*2; t++)  
  for (int s1 = 0; s1 < 8; s1++)  
    for (int s2 = 0; s2 < 5; s2++)  
      // compute ...
```

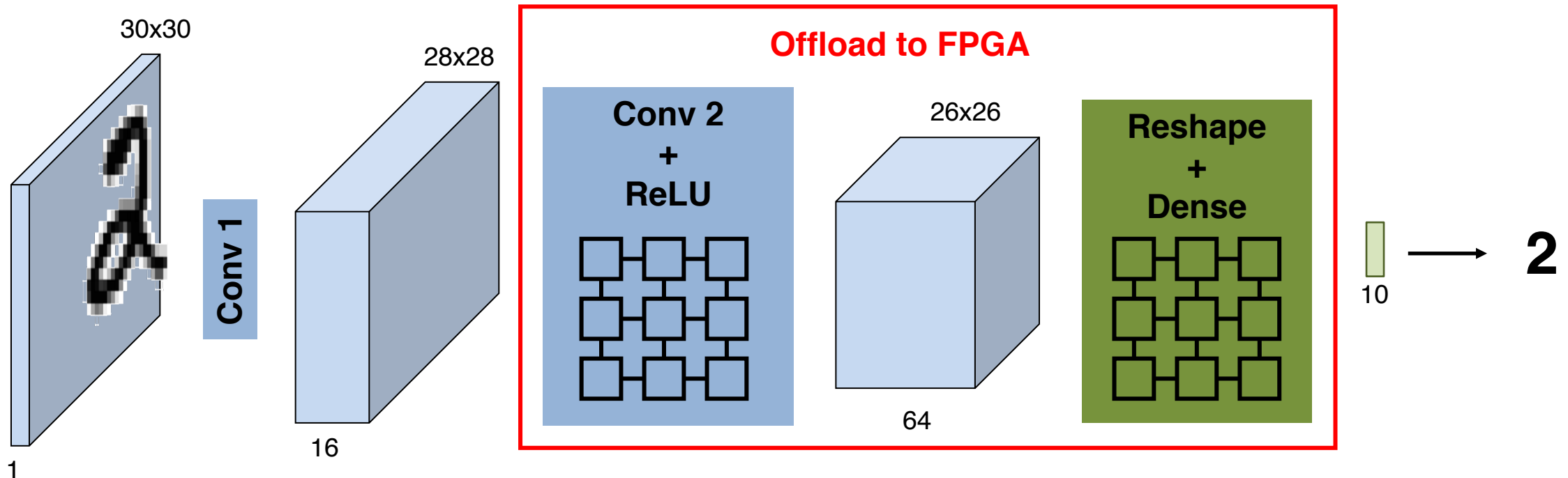


AutoSA

- ▶ Future work: Integrate T2S

Demo 3: AutoSA Integration

- ▶ Map Conv 2 and Dense to AutoSA generated systolic arrays
 - Conv 2: 8 x 13 (Loop shape: 26x26x16x64x3x3)
 - Dense: 5 x 8 (Loop shape: 26x26x64x10)
- ▶ Connect two systolic arrays with FIFO via **.to()**



Demo 3: Code & Results Review

- ▶ Use **.systolic()** to map kernels to systolic arrays generated by AutoSA

```
s[top.conv2].systolic()  
s[top.dense].systolic()
```

- ▶ Use **.to()** to specify dataflow between systolic arrays and other kernels

```
s.to(top.conv2, top.relu)  
s.to(top.relu, top.reshape)  
s.to(top.reshape, top.dense)
```

- ▶ Performance comparison
 - Baseline (with only pipelining):
 - Applying data reuse and data movement:
 - Mapping to AutoSA generated systolic arrays:

Portability Evaluation with Realistic Designs

Target: Xilinx Alveo U280 (Vivado HLS + Vitis)

Application	LOC (vs. HLS C++)	LUT#	FF#	DSP#	BRAM#	Fmax (MHz)	Runtime (ms)
DigitRec	58 (vs. 243)	8,914	8,850	0	2	300	1.82
3D Rendering	187 (vs. 375)	6,670	8,326	13	38	300	2.56
Optical Flow	206 (vs. 742)	23,812	32,906	182	64	300	3.62

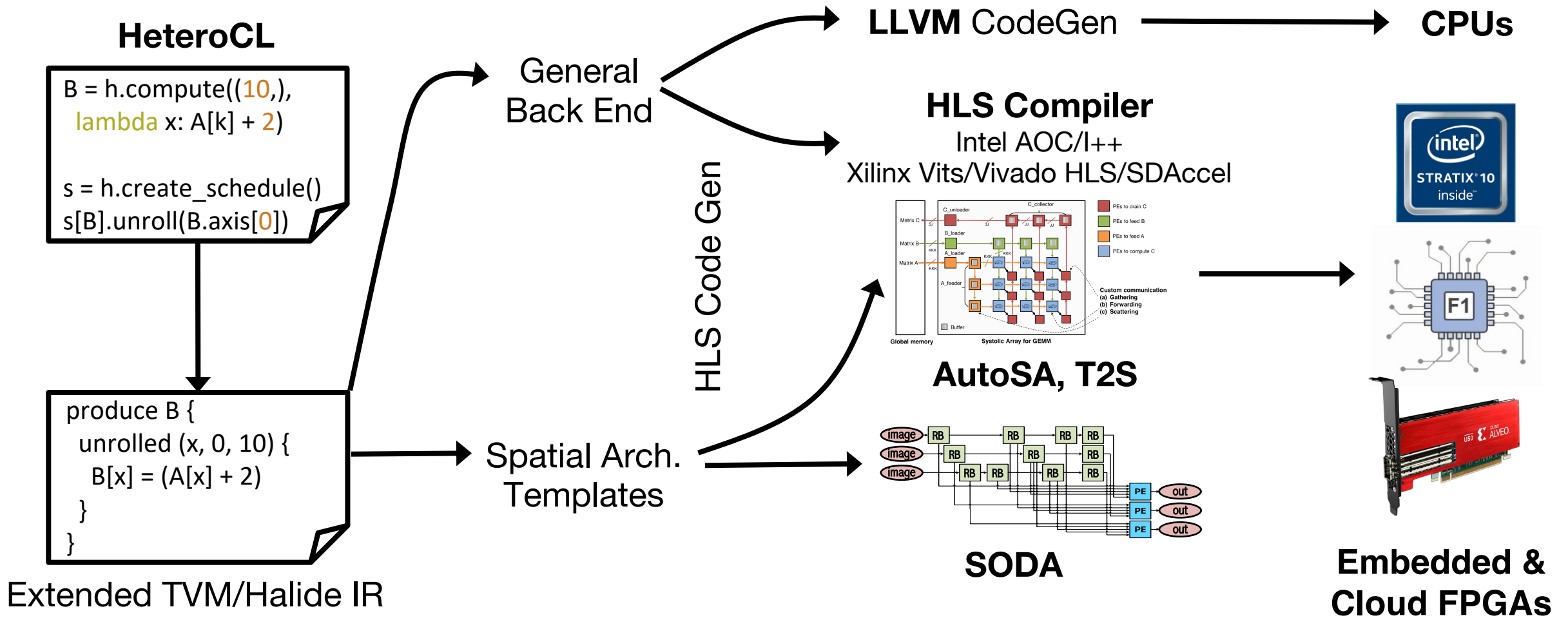
Target: Intel Stratix 10 (AOCL*, vLab)

Application	LOC (vs. HLS C++)	ALUT#	FF#	DSP#	BRAM#	Fmax (MHz)	Runtime (ms)
DigitRec	33 (vs. 422)	6,282	11,835	0	51	296.29	4.01
3D Rendering	160 (vs. 611)	11,745	22,067	22	283	329.48	7.78
Optical Flow	206 (vs. 628)	29,490	58,130	106	484	386	3.82

* Intel AOCL currently does not support fixed point type

A single HeteroCL source can be retargeted to different FPGA devices with only a few changes


Portable Compilation Flow



Flexible retargeting to diverse FPGAs from various vendors

More Examples

github.com/cornell-zhang/heterocl



HeteroCL

0.1

- Installation
- HeteroCL Tutorials
- HeteroCL Samples
 - CORDIC
 - KNN-Based Digit Recognition
 - HeteroCL Tutorial : K-Nearest-Neighbor Digit Recognition
 - Top Function Offloaded to FPGA
 - Main function
 - GEMM
 - K-Means
 - LeNet Inference
 - Smith-Waterman Genomic Sequencing

cornell-zhang / heterocl

Unwatch 18 Star 170 Fork 57

Code Issues 94 Pull requests 5 Actions Projects 1

⚠ We found potential security vulnerabilities in your dependencies.

You can see this message because you have been granted [access to Dependabot alerts for this repository.](#)

[See Dependabot alerts](#)

master Go to file Add file Code

Hecmay and Your Name [API][Backend] I... 23 days ago 352

- .circleci** [Testing] Add test cases for issues r... 24 days ago
- conda/heterocl** [Backend][Utils] IP Integration Enha... 11 months ago
- docker** [API] New Data Movement Mode w/o... 10 months ago

About

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

python fpga dsl accelerators

Readme Apache-2.0 License

More Examples

<https://github.com/cornell-zhang/heterocl/tree/master/samples>

Design	Compute	Data Type	Memory	Imperative	Macros
KNN Digit Rec.	✓	✓		✓	
K-Means	✓	✓		✓	
Smith-Waterman	✓	✓		✓	
Cordic		✓			
Sobel	✓		✓		
Optical Flow	✓		✓		
3D Rendering	✓		✓	✓	
Seidel	✓	✓	✓		✓
Gaussian	✓	✓	✓		✓
Jacobi	✓	✓	✓		✓
GEMM		✓			✓

More designs (e.g., BNN) will be added soon!

Concluding Remarks

HeteroCL offers a new high-level programming framework for building FPGA accelerators

- ▶ **Productive:** Ease-of-programming with Python-based interface and support of high-level DSL
- ▶ **Performant:** Competitive QoR against HLS expert designs with efficient mapping to spatial architectures
- ▶ **Portable:** Target-neutral algorithm specification with decoupled hardware customizations

github.com/cornell-zhang/heterocl

