Building FPGA-Targeted Accelerators with HeteroCL

Zhiru Zhang

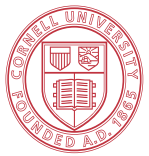
School of ECE, Cornell University

csl.cornell.edu/~zhiruz

In collaboration with

Cornell: Yi-Hsiang Lai, Shaojie Xiang, Yuwei Hu

UCLA: Yuze Chi, Jie Wang, Cody Yu, Jason Cong



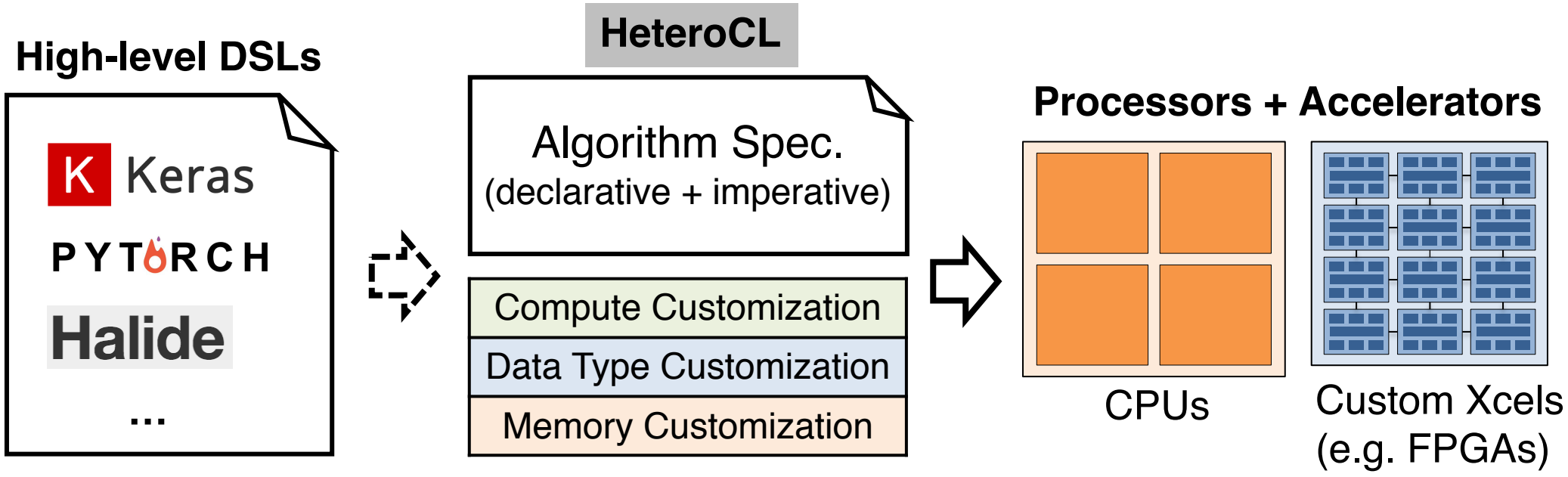
Cornell University

TVM Workshop @ UW, 12/5/2019

CSL

HeteroCL Overview

- A programming framework built with TVM for productive hardware specialization
  - **Flexible**: Mixed declarative & imperative programming
  - **Efficient**: Mapping to high-performance spatial architecture templates
  - **Portable**: Clean decoupling of algorithm & hardware customizations



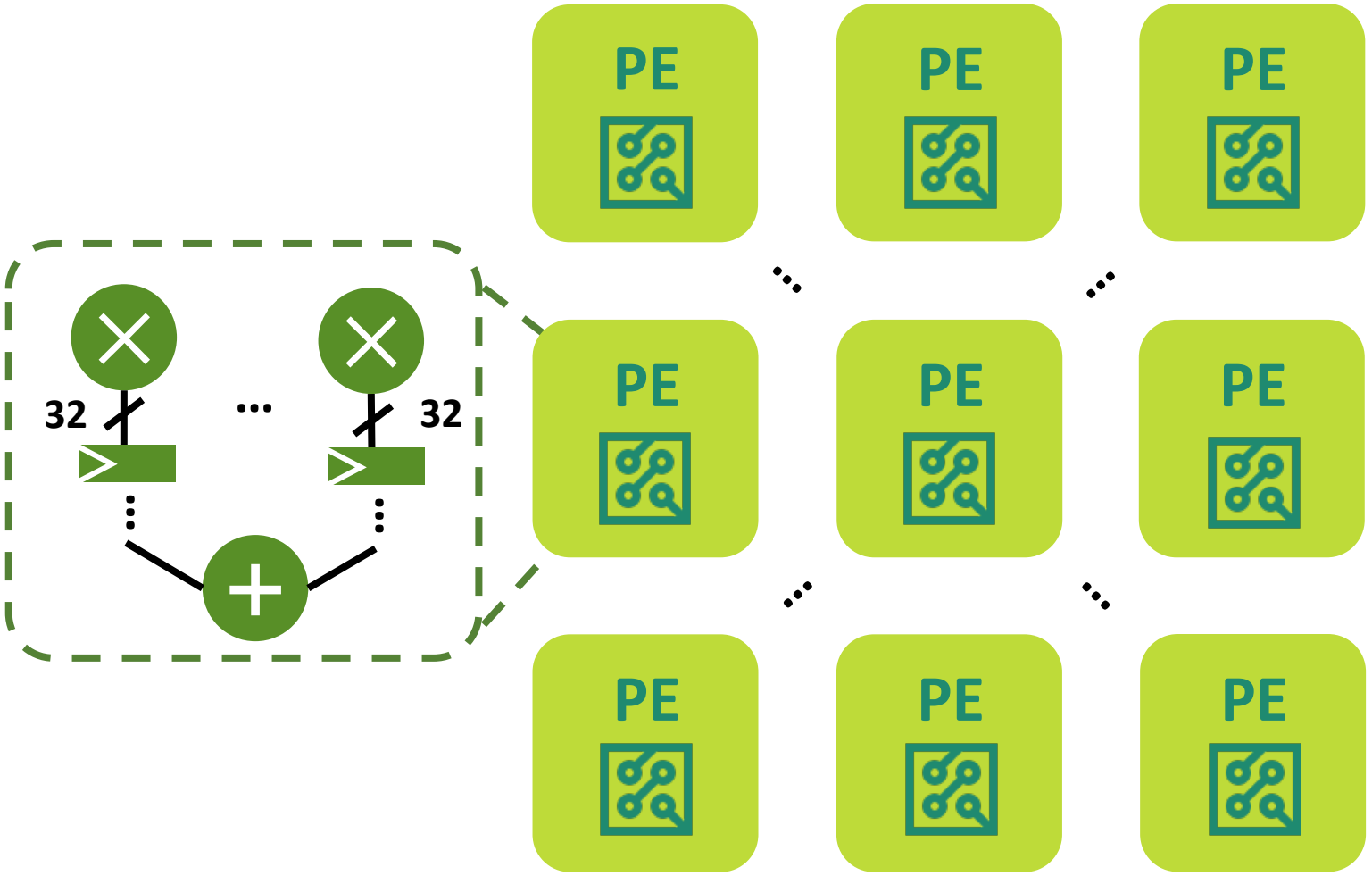
github.com/cornell-zhang/heterocl

Y.-H. Lai, et al., *HeteroCL: A Multi-Paradigm Programming Infrastructure for Software-Defined Reconfigurable Computing*, FPGA'2019 (**Best Paper Award**)

Essential Techniques for Hardware Specialization

Compute customization

- Parallelization, Pipelining …



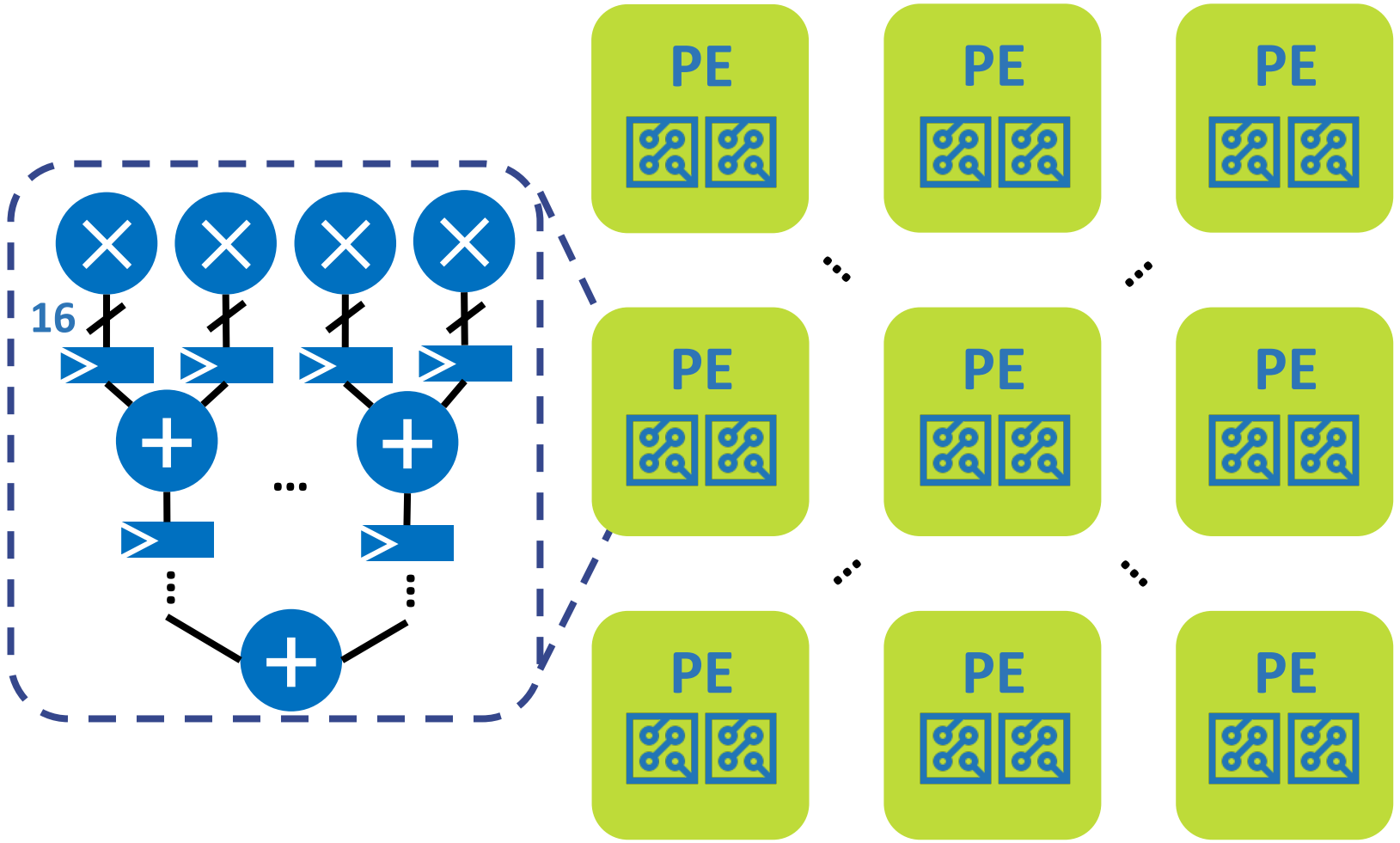
Essential Techniques for Hardware Specialization

Compute customization

- Parallelization, Pipelining …

Data type customization

- Low-bitwidth integer, Fixed point ...

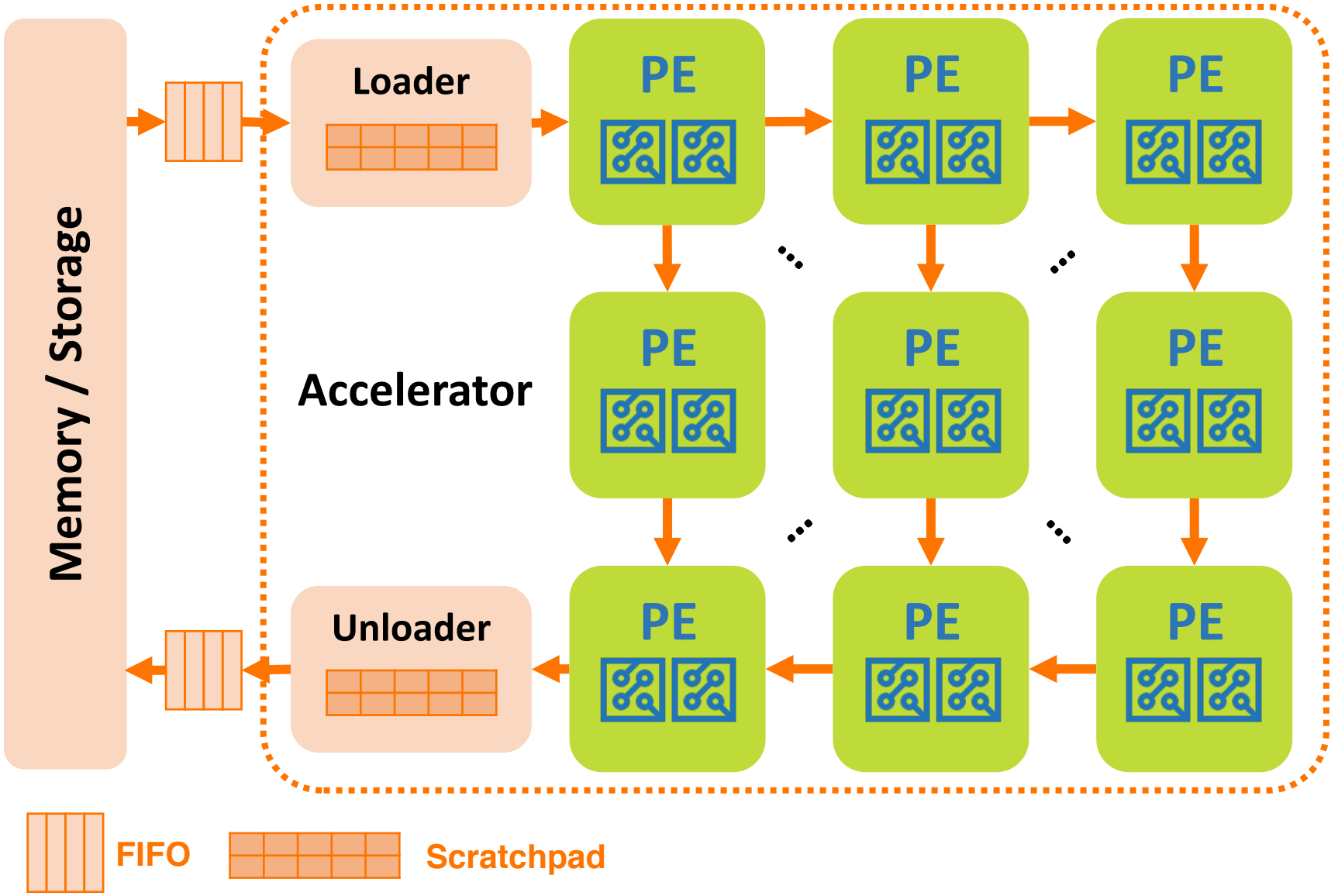


Essential Techniques for Hardware Specialization

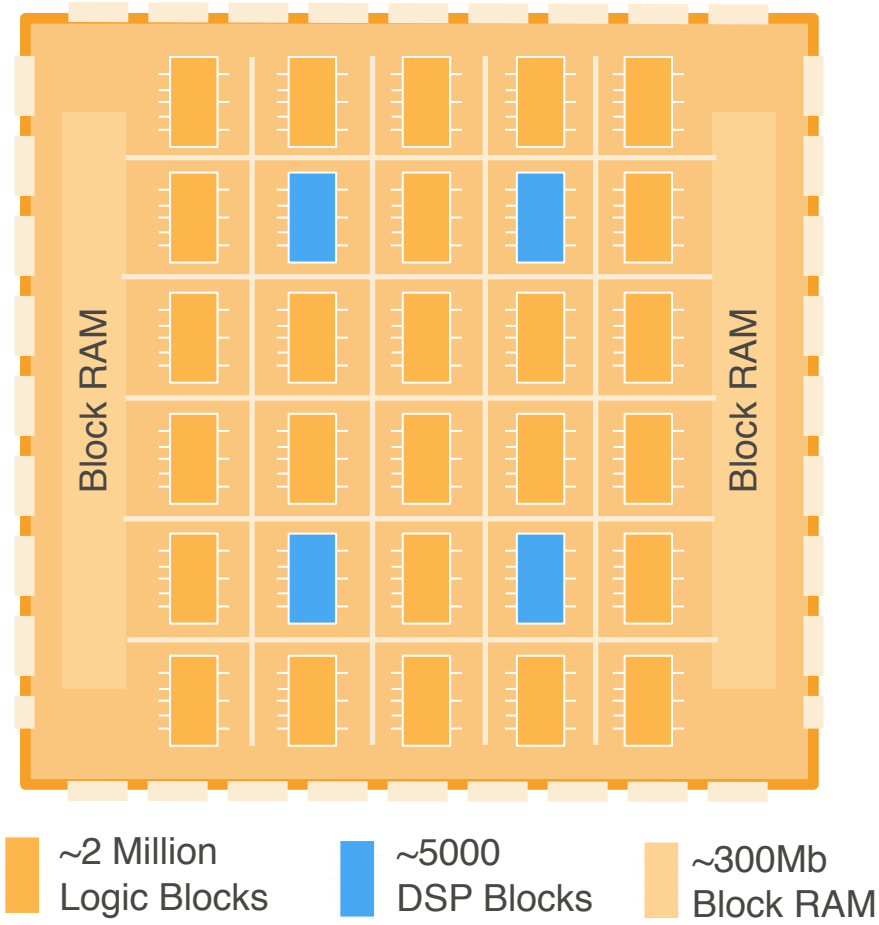
Compute customization
- Parallelization, Pipelining …

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

Memory customization
- Banking, Data reuse, Streaming ...



FPGA as a Programmable Accelerator



AWS F1 FPGA instance: Xilinx UltraScale+ VU9P

[Figure source: David Pellerin, AWS]

- **Massive amount of fine-grained parallelism**
  - Highly parallel / deeply pipelined architecture
  - Distributed data/control dispatch
- **Silicon configurable to fit the application**
  - Compute at desired numerical accuracy
  - Customized memory hierarchy
- **High performance/watt**
  - Low clock speed
  - Pre-fabricated architecture blocks

But FPGAs are *really hard* to PROGRAM

Increasing Use of High-Level Synthesis (HLS)

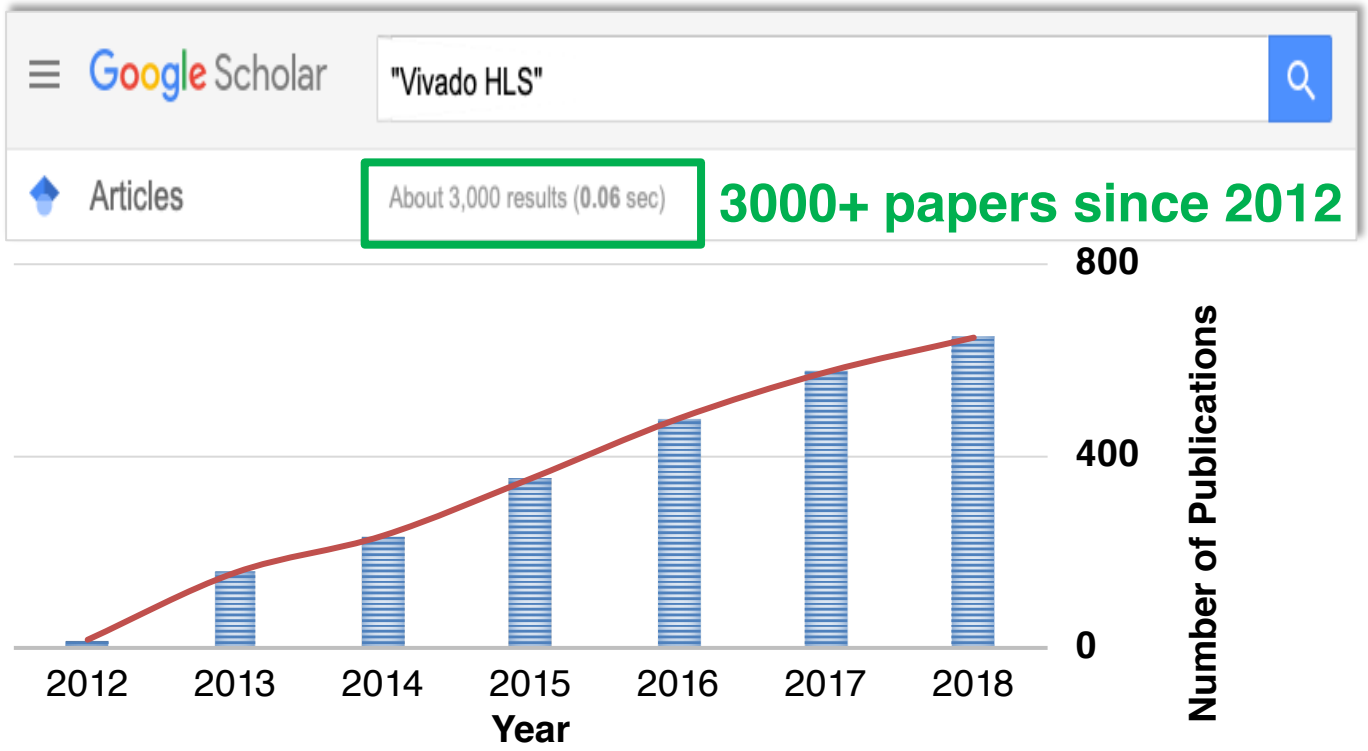
```
module dut(rst, clk, q);
  input rst;
  input clk;
  output q;
  reg [7:0] c;

  always @ (posedge clk)
  begin
    if (rst == 1b'1) begin
      c <= 8'b00000000;
  end
  else begin
    c <= c + 1;
  end

  assign q = c;
endmodule
```

RTL Verilog

vs.

```
uint8 dut() {
  static uint8 c;
  c+=1;
}
```

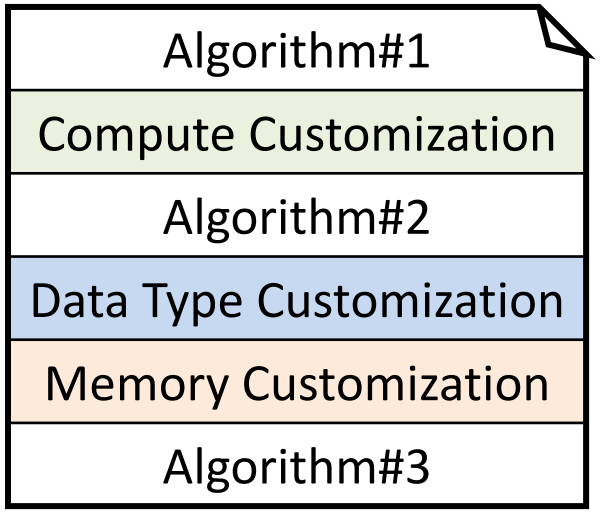
HLS C



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] * kernel[r, c]
```



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;
}}}}
```

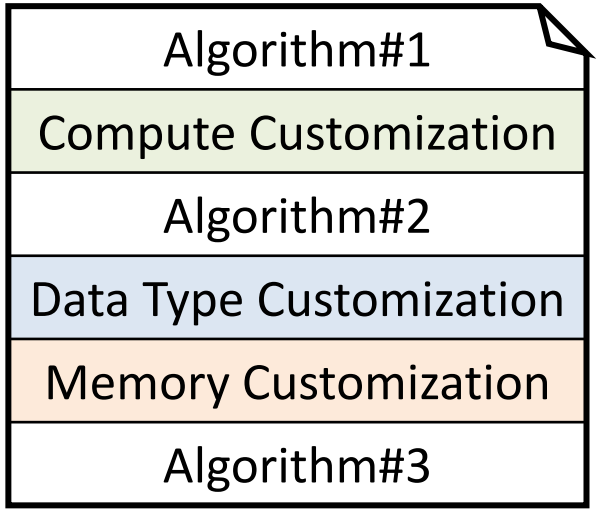
Custom compute (Loop tiling)

Custom data type (Quantization)

Custom memory (Reuse buffers)

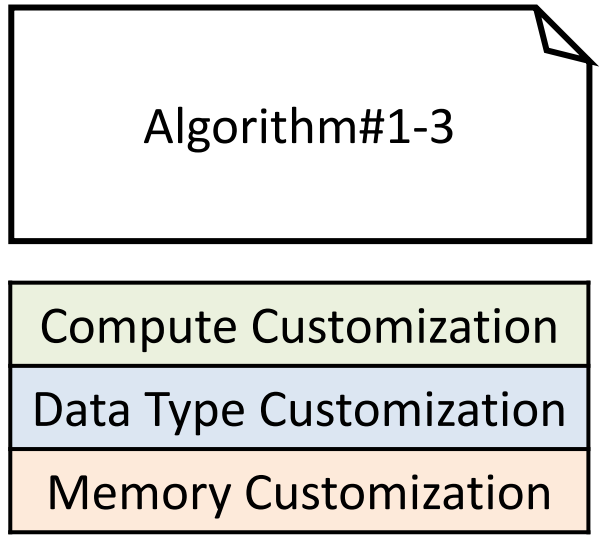
Decoupling Algorithm from Hardware Customizations

HLS C



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

HeteroCL



Fully decoupled customization schemes +
Clean abstraction capturing the interdependence

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]))
```

Declarative programming (**TVM based**)

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

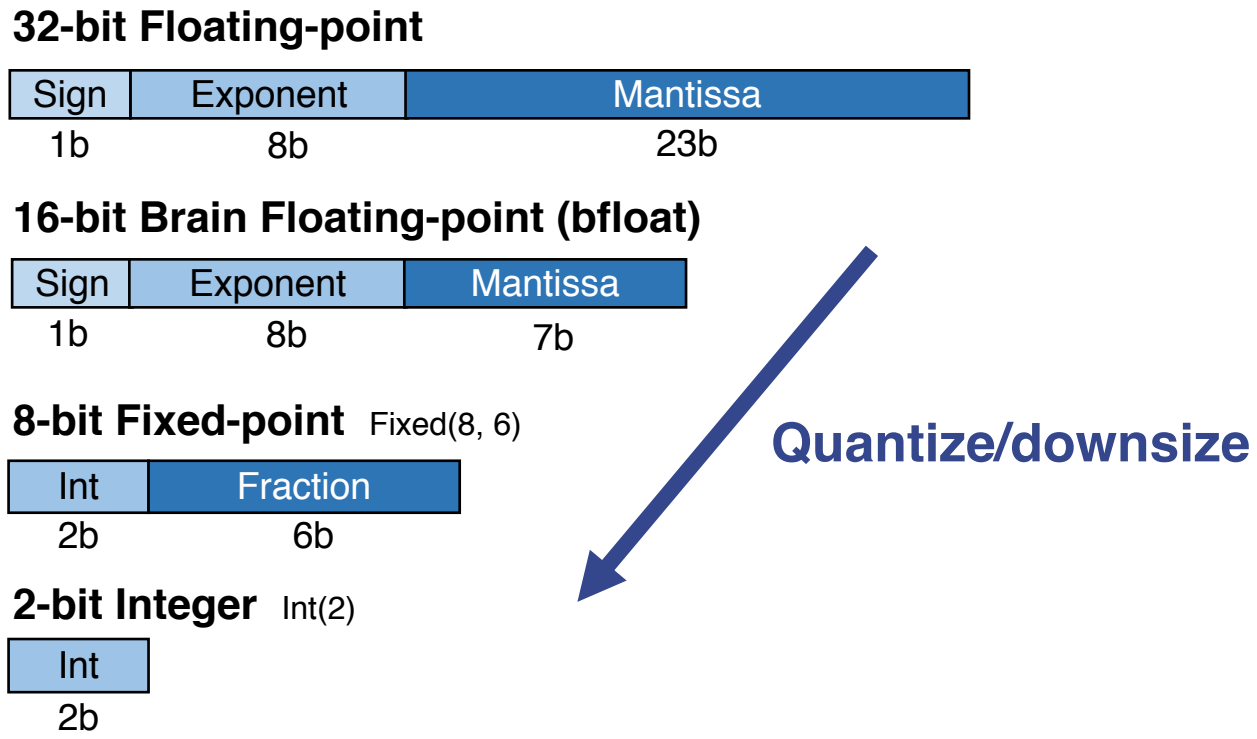
Reorder loops

Decoupled Data Type Customization

- Bit-accurate data type support (e.g., Int(15), Fixed(7,4))
  - W.I.P.: 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))
```

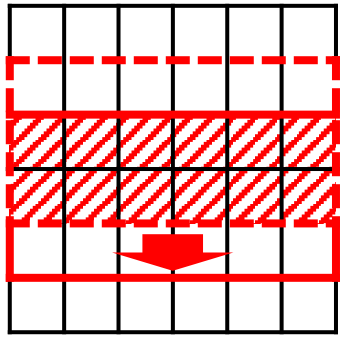


Decoupled Memory Customization

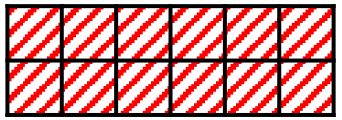
- Inferring custom on-chip storage with the **reuse_at**() primitive

```
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)
```

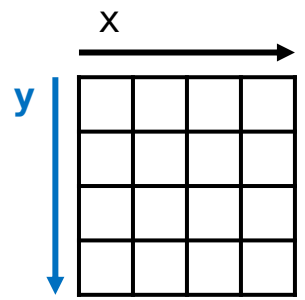
```
for (int y = 0; y < N; y++)
  for (int x = 0; x < N; x++)
    for (int r = 0; r < 3; r++)
      for (int c = 0; c < 3; c++)
        out[x, y] += image[x+r, y+c] * kernel[r, c]
```



image



linebuffer



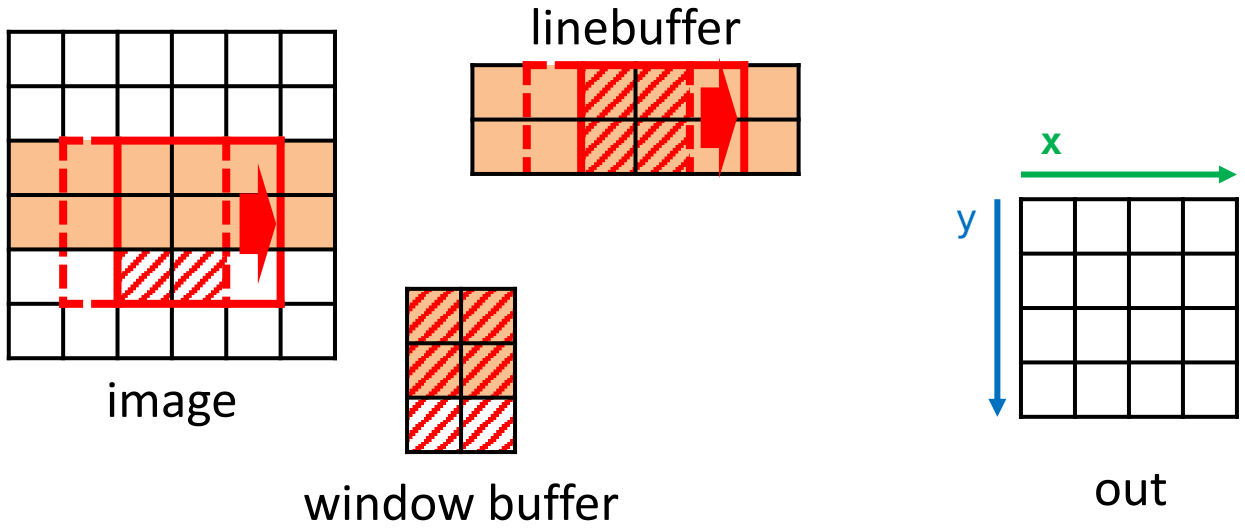
out

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]
```

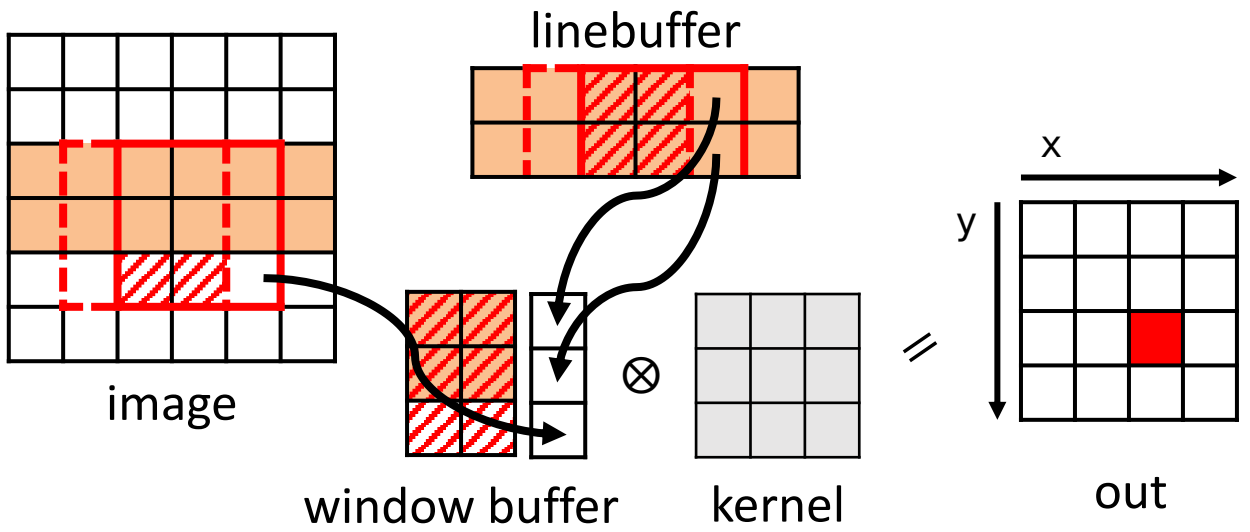


Decoupled Memory Customization

- 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)
```

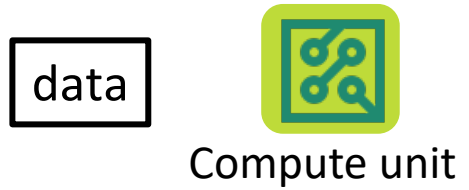
```
for (int y = 0; y < N; y++)
  for (int x = 0; x < N; x++)
    for (int r = 0; r < 3; r++)
      for (int c = 0; c < 3; c++)
        out[x, y] += image[x+r, y+c] * kernel[r, c]
```

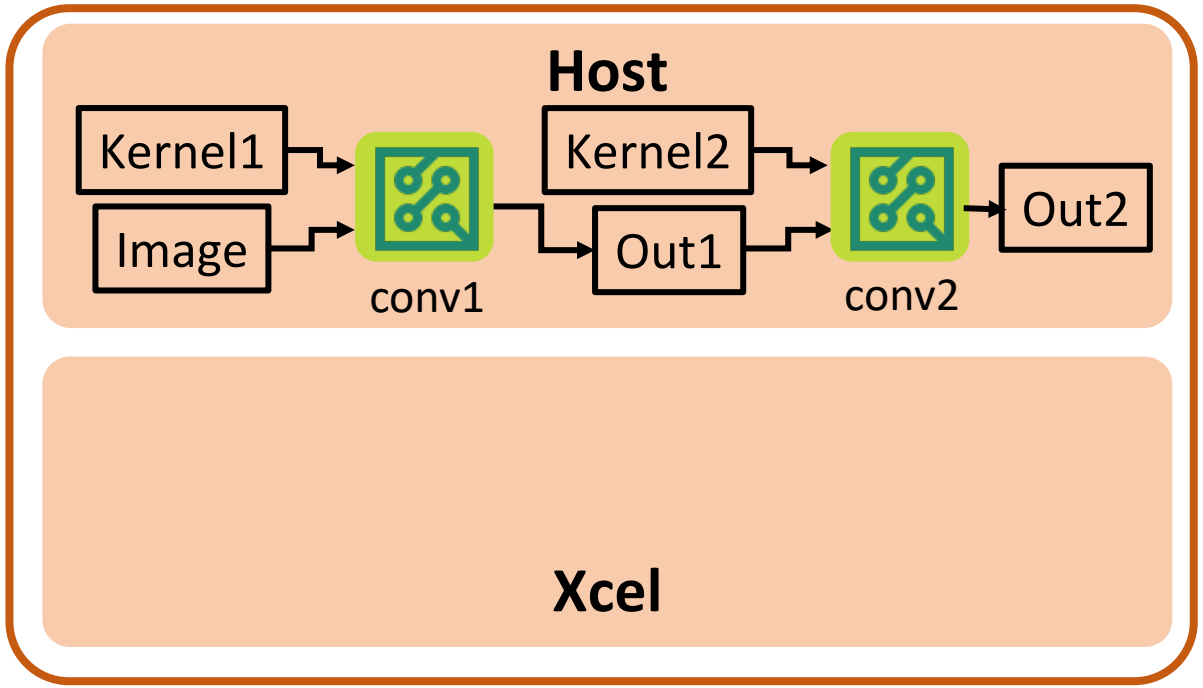


Decoupled Data Placement

- A unified interface for specifying data placement & movement

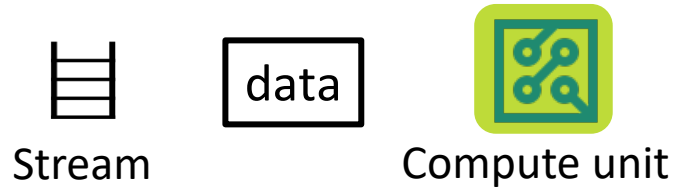
```
from heterocl import platform
@hcl.def_()
def conv(input, kernel):
    r = hcl.reduce_axis(0, 3)
    c = hcl.reduce_axis(0, 3)
    return hcl.compute(N, N), lambda y, x:
        hcl.sum(input[x+r, y+c]*kernel[r, c], axis=[r, c]))

out1 = conv(image, kernel1, "conv1")
out2 = conv(out1, kernel2, "conv2")

s = hcl.create_schedule()
p = platform.fpga_soc

f = hcl.build(p)
```



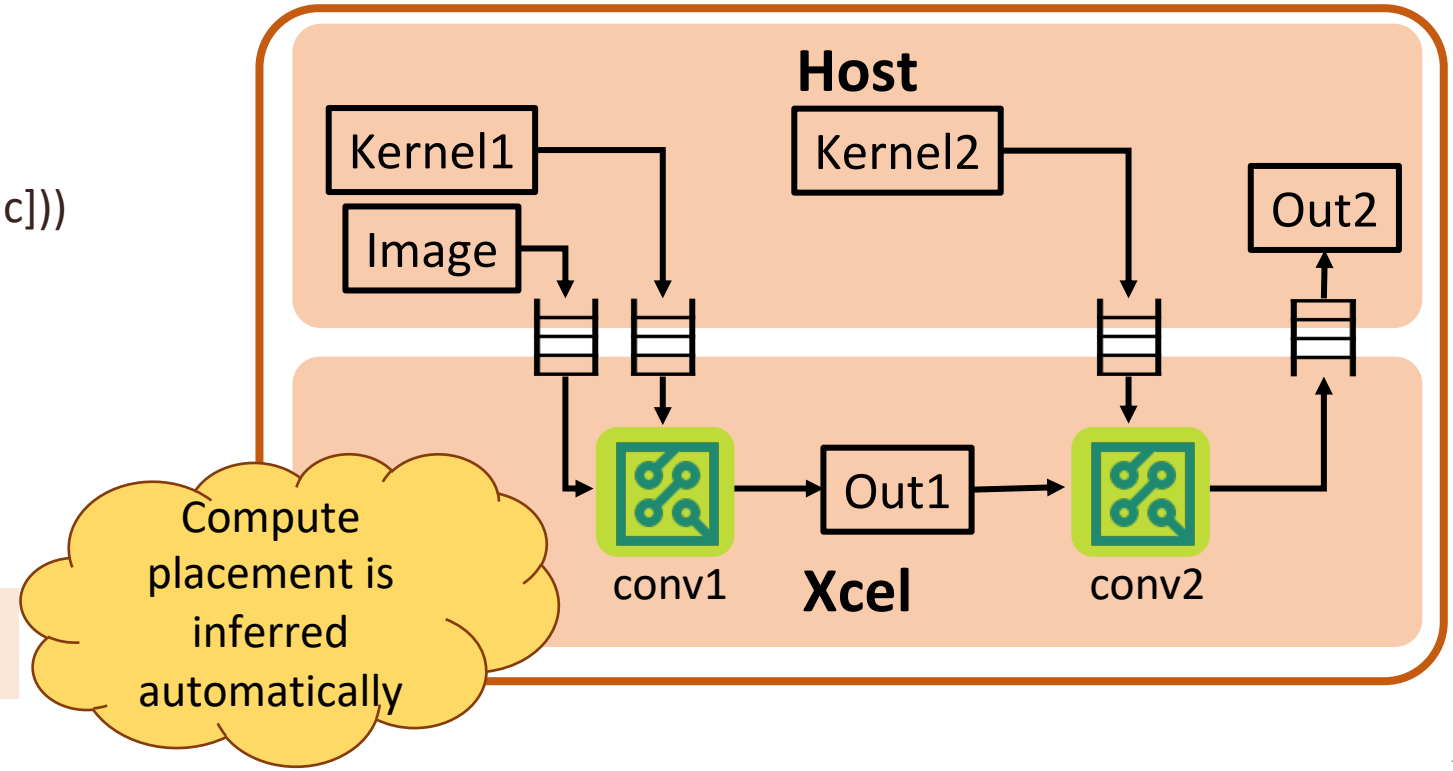


Decoupled Data Placement

- A unified interface for specifying data placement & movement between
  - Host and accelerator

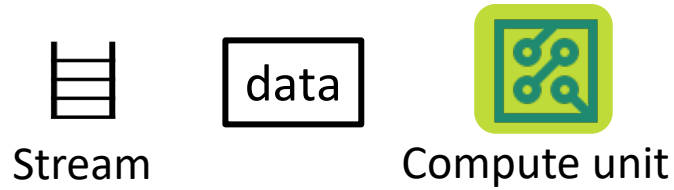



```
from heterocl import platform
@hcl.def_()
def conv(input, kernel):
    r = hcl.reduce_axis(0, 3)
    c = hcl.reduce_axis(0, 3)
    return hcl.compute(N, N), lambda y, x:
        hcl.sum(input[x+r, y+c]*kernel[r, c], axis=[r, c]))

out1 = conv(image, kernel1, "conv1")
out2 = conv(out1, kernel2, "conv2")

s = hcl.create_schedule()
p = platform.fpga_soc
s.to([image, kernel1, kernel2], p.xcel)
s.to(out2, p.host)
```

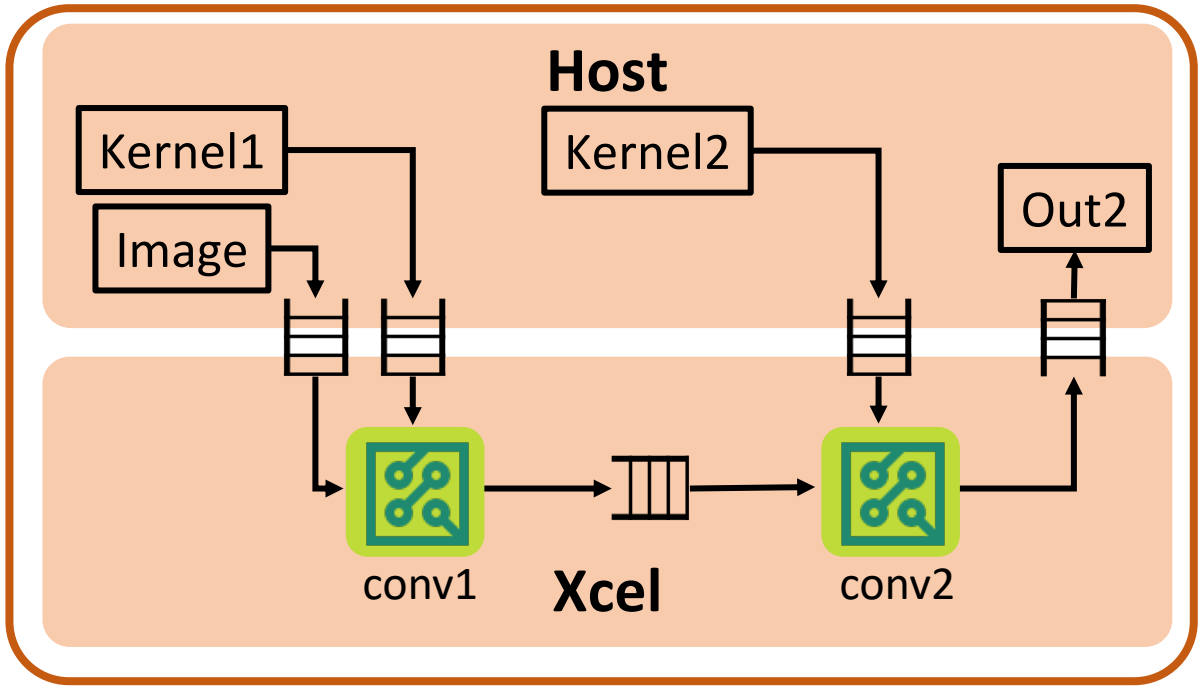


Decoupled Data Placement

- A unified interface for specifying data placement & movement between
  - Host and accelerator
  - Sub-modules in accelerator




```
from heterocl import platform
@hcl.def_()
def conv(input, kernel):
    r = hcl.reduce_axis(0, 3)
    c = hcl.reduce_axis(0, 3)
    return hcl.compute(N, N), lambda y, x:
        hcl.sum(input[x+r, y+c]*kernel[r, c], axis=[r, c]))

out1 = conv(image, kernel1, "conv1")
out2 = conv(out1, kernel2, "conv2")

s = hcl.create_schedule()
p = platform.fpga_soc
s.to([image, kernel1, kernel2], p.xcel)
s.to(out2, p.host)
s.to(out1, conv2)
```



Currently Supported Customization Primitives

Compute customization

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.
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

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.
to(t, d, m)	Move a list of tensors t to device d with mode m.

Data type customization

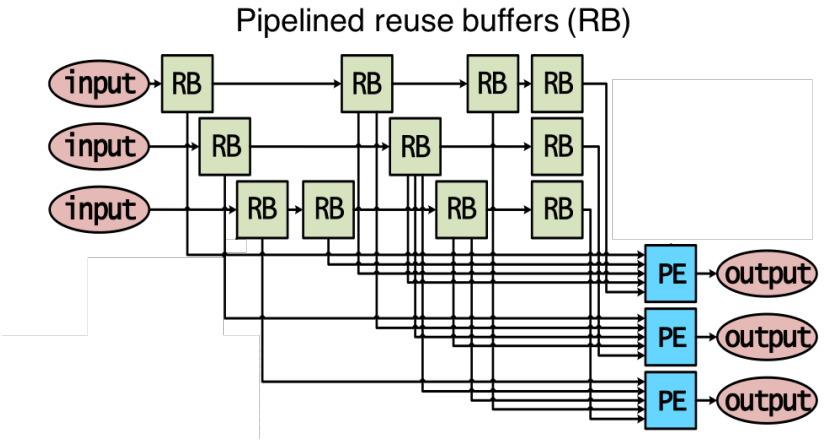
downsize(t, d)	Downsize a list of tensors t to type d.
quantize(t, d)	Quantize a list of tensors t to type d.

Macros for spatial architecture templates

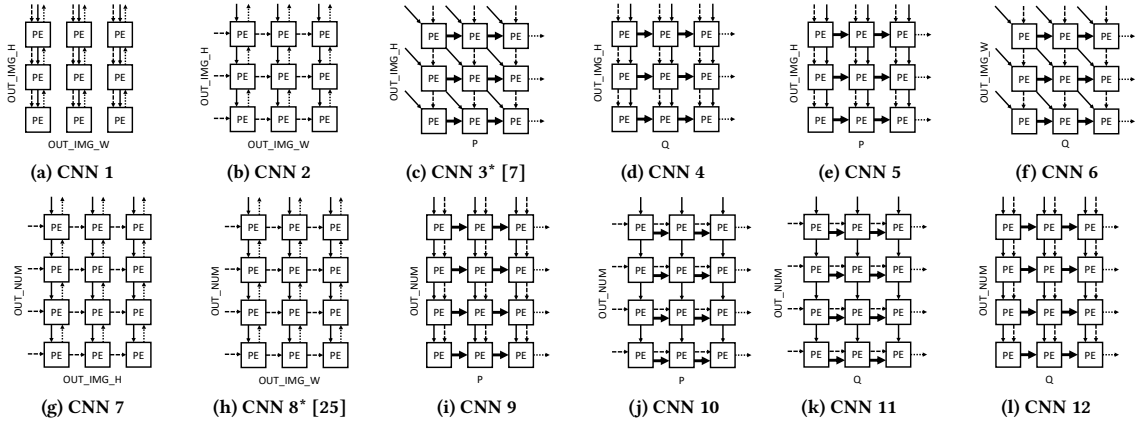
C.stencil()	Specify operation C to be implemented with stencil with dataow architectures using the SODA framework.
C.systolic()	Specify operation C to be implemented with systolic arrays using the PolySA framework.

Targeting Spatial Architectures in HeteroCL

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



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



- PolySA backend [J. Cong, et al. ICCAD'18]
  - Produces a variety of systolic arrays with polyhedral transformation
  - Incorporates additional architecture optimizations (banking, SIMD, latency hiding, etc.)

Imperative Programming in HeteroCL

- HeteroCL further provides an embedded imperative DSL
  - Not all algorithms can be described in declarative tensor-style code

```
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]
```

- Imperative & declarative programs share a unified interface for customization primitives

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

HeteroCL Compilation Flow

HeteroCL

```
B = h.compute((10,),
  lambda x: A[k] + 2)

s = h.create_schedule()
s[B].unroll(B.axis[0])
```

```
produce B {
  unrolled (x, 0, 10) {
    B[x] = (A[x] + 2)
  }
}
```

Extended TVM/Halide IR

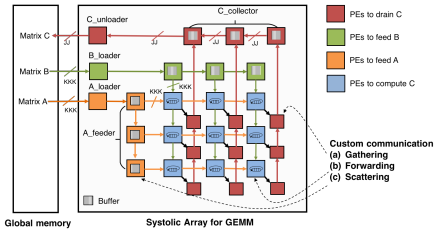
General Back End

Spatial Architecture Templates

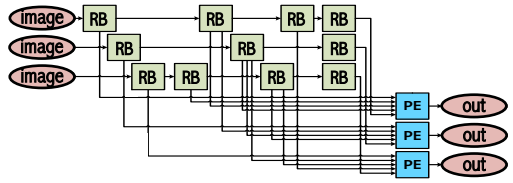
HLS Code Gen

LLVM Code Gen → CPUs

HLS Compiler
(Falcon Merlin, Intel OpenCL, Xilinx Vivado HLS)



PolySA, T2S

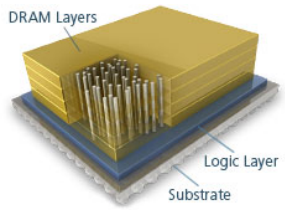


SODA

XILINX VIRTEX UltraScale HBM

intel STRATIX 10 inside

FPGAs



Near-memory accelerators

On-going work

HeteroCL Evaluation on Cloud FPGAs (Amazon AWS F1)

Stencil

Benchmark (Application)	+ 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)	Back end	Data type	Performance (GOPs)	Speedup
GEMM (Linear algebra)	CPU (Intel MKL)	float32	76.0	1.0
	FPGA (HeteroCL)	float32	245.9	3.2
		fixed16	807.6	10.6
LeNet (Deep learning)	CPU (TVM TOPI)	float32	15.4	1.0
	FPGA (HeteroCL)	float32	79.8	5.2
		fixed16	137.8	8.9

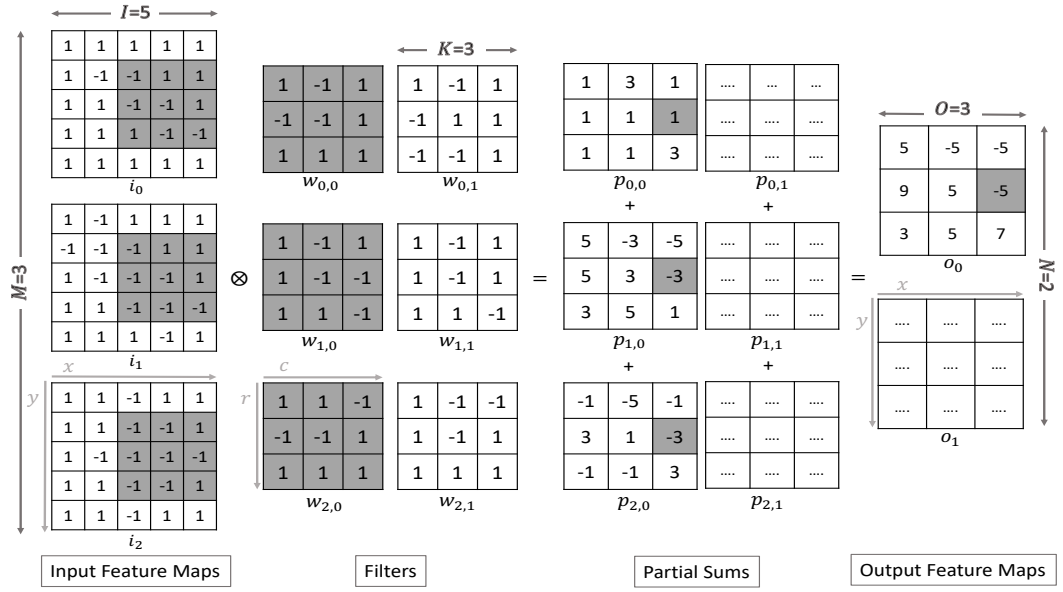
General

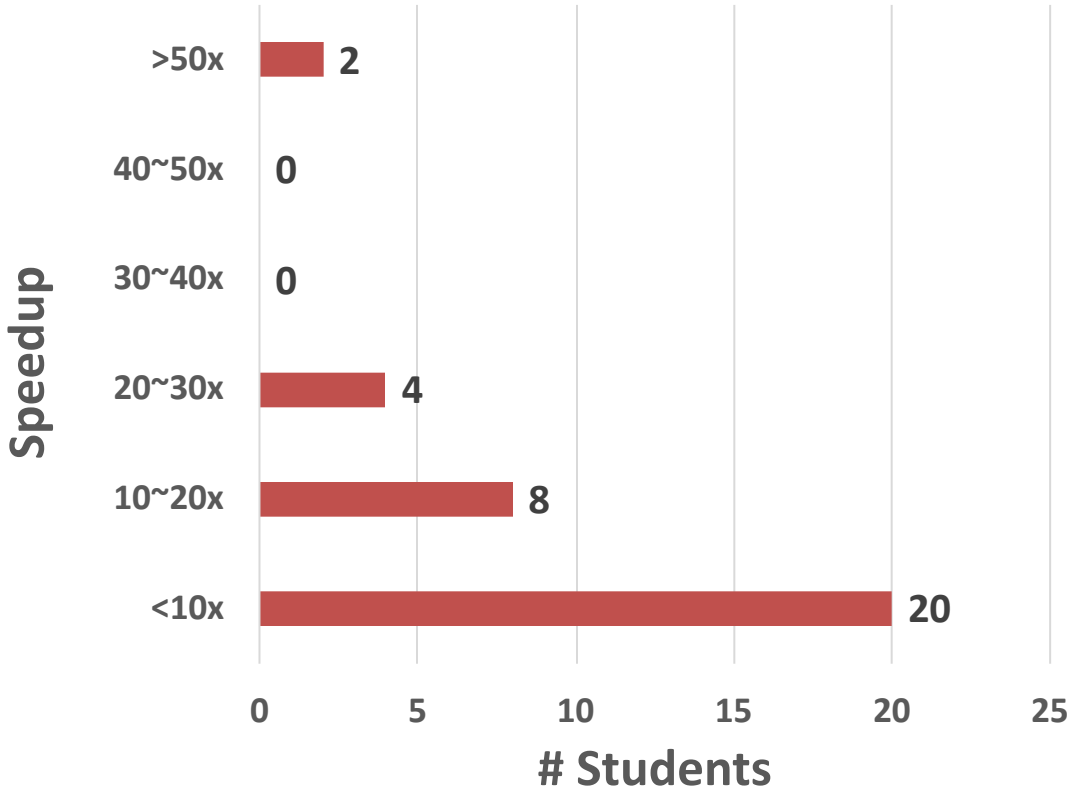
Benchmark (Application)	Speedup
KNN Digit Recognition (Image classification)	12.5
K-Means (Machine learning)	16.0
Smith-Waterman (Genomic sequencing)	20.9

Rapidly achieve good speedup for a rich set of applications

Another Case Study: Binarized Neural Network (BNN)

- Design competition in Cornell ECE 5775 [1]
  - 34 graduates & senior undergrads enrolled in recent course offering
  - Using HLS to accelerate a simple BNN on Zynq FPGA
    - Higher speedup over ARM CPU => Higher score
    - Only two students achieved a speedup higher than 50X within 2 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

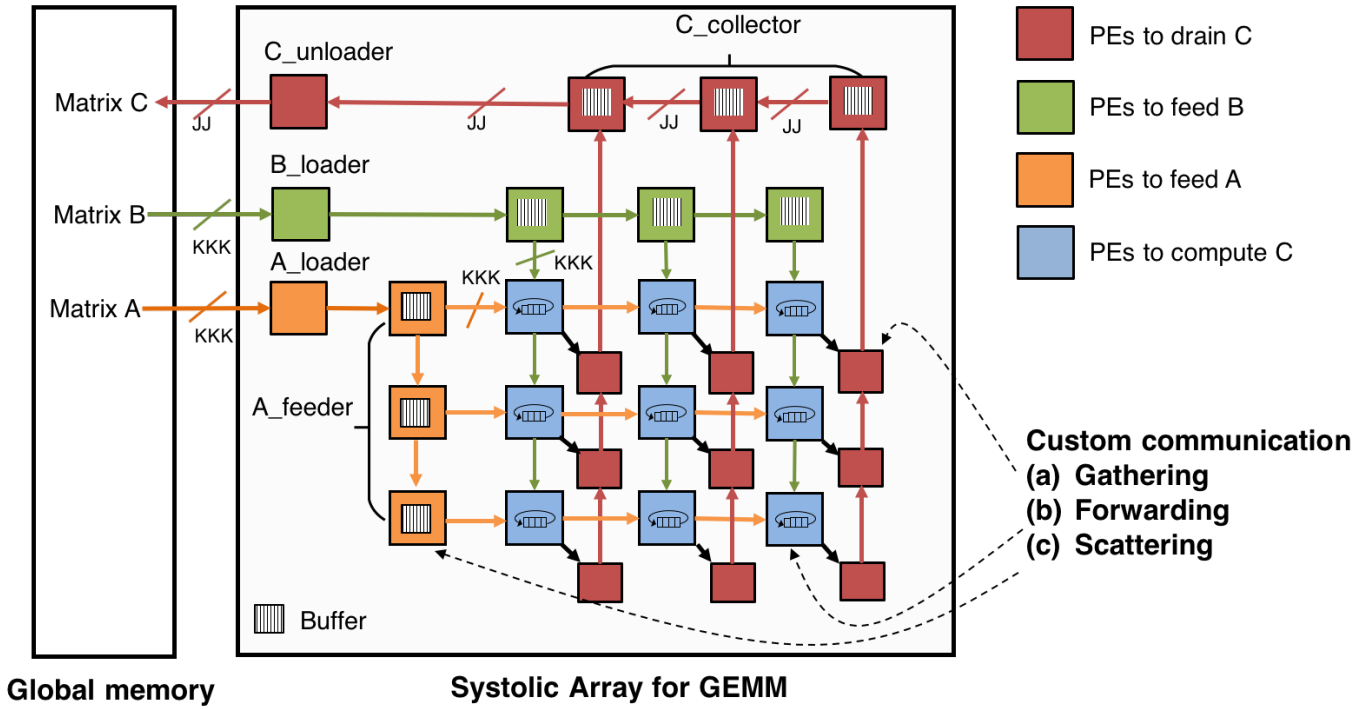
- Development time: < 3 days
- Final speedup: 63x

✓ **More productive**

✓ **More maintainable**

```
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])
```

Quick Aside: Decoupled Spatial Mapping with T2S (joint work with Intel Labs)



Algorithm

```
Func C
C(i, j) = 0
C(i, j) += A(i, k) * B(k, j)
C.tile(i,j,k,ii,jj,kk,II,JJ,KK)
```

Spatial Mapping (+ custom communication)

```
C.isolate_producer(A, A_feeder)
C.unroll(ii, jj)
A_feeder.isolate_producer(A, A_loader)
A_loader.remove(jj)
A_feeder.buffer(ii, DOUBLE)
C.forward(A_feeder, +jj)
A_feeder.unroll(ii).scatter(A, +ii)
C.isolate_consumer(C, C_drainer)
C_drainer.isolate_consumer_chain(C,
C_collector, C_unloader)
C_drainer.unroll(ii).unroll(jj).gather(C, -ii)
C_collector.unroll(jj).gather(C, -jj)
```

N. Srivastava, et al., *T2S-Tensor: Productively Generating High-Performance Spatial Hardware for Dense Tensor Computations*, FCCM'2019

High-Performance GEMM on FPGA using T2S

- **Baseline**: NDRange-style OpenCL, tuned for Intel Arria 10
- **Ninja**: Handwritten industry design by optimized by an expert

	Baseline	T2S	Ninja
LOC	70	**20**	750
Systolic array size	--	10 x 8	10 x 8
Vector length	16 x float	16 x float	16 x float
# Logic elements	131 K	214 K	230 K
# DSPs	1,032	1,282	1,280
# RAMs	1,534	1,384	1,069
Frequency (MHz)	189	215	245
Throughput (GFLOPS)	311	**549**	626

~90% performance of ninja implementation with 3% code

Accelerating Tensor Decomposition Kernels with T2S

- Tensor decomposition kernels
  - ***MTTKRP***: $D(i,j) \quad += A(i,k,l) * B(k,j) * C(l,j)$
  - ***TTM***: $C(i,j,k) += A(i,j,l) * B(l,k)$
  - ***TTMc***: $D(i,j,k) += A(i,l,m) * B(l,j) * C(m,k)$

Evaluation on Arria-10 FPGA

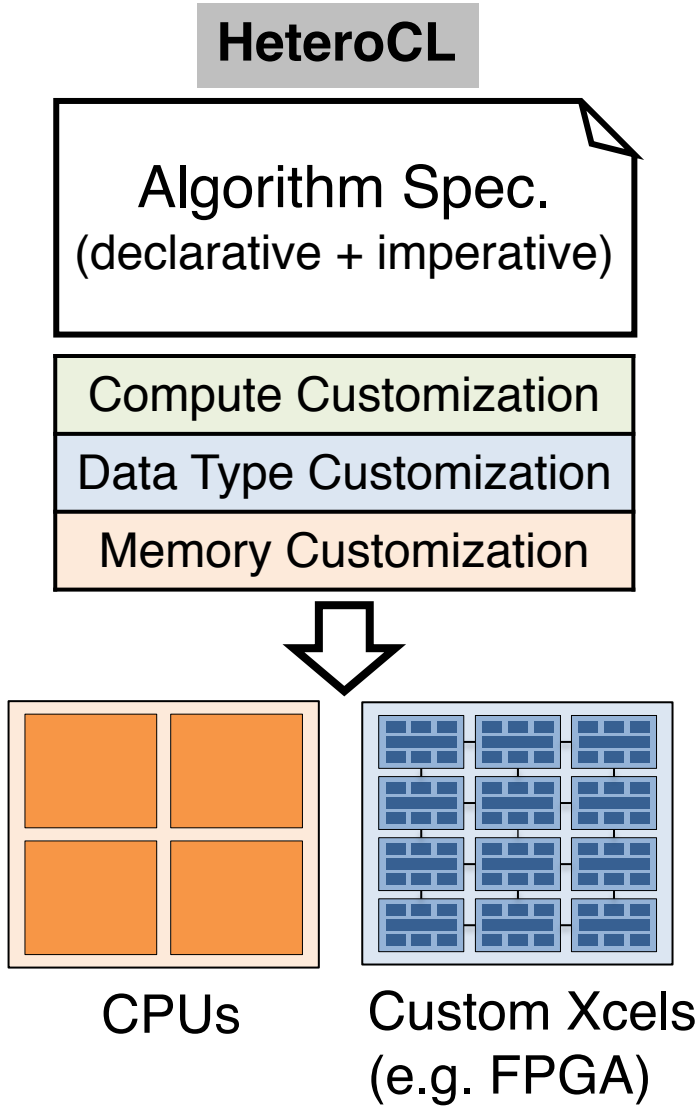
Benchmark	LOC	Systolic Array Size	Logic Usage	DSP Usage	RAM Usage	Frequency (MHz)	Throughput (GFLOPS)
MTTKRP	28	8 x 9	53 %	81 %	56 %	204	**700**
TTM	30	8 x 11	64 %	93 %	88 %	201	**562**
TTMc	37	8 x 10	54 %	90 %	62 %	205	**738**

~80-90 % DSP utilization and 560-740 GFLOPS for FPGA

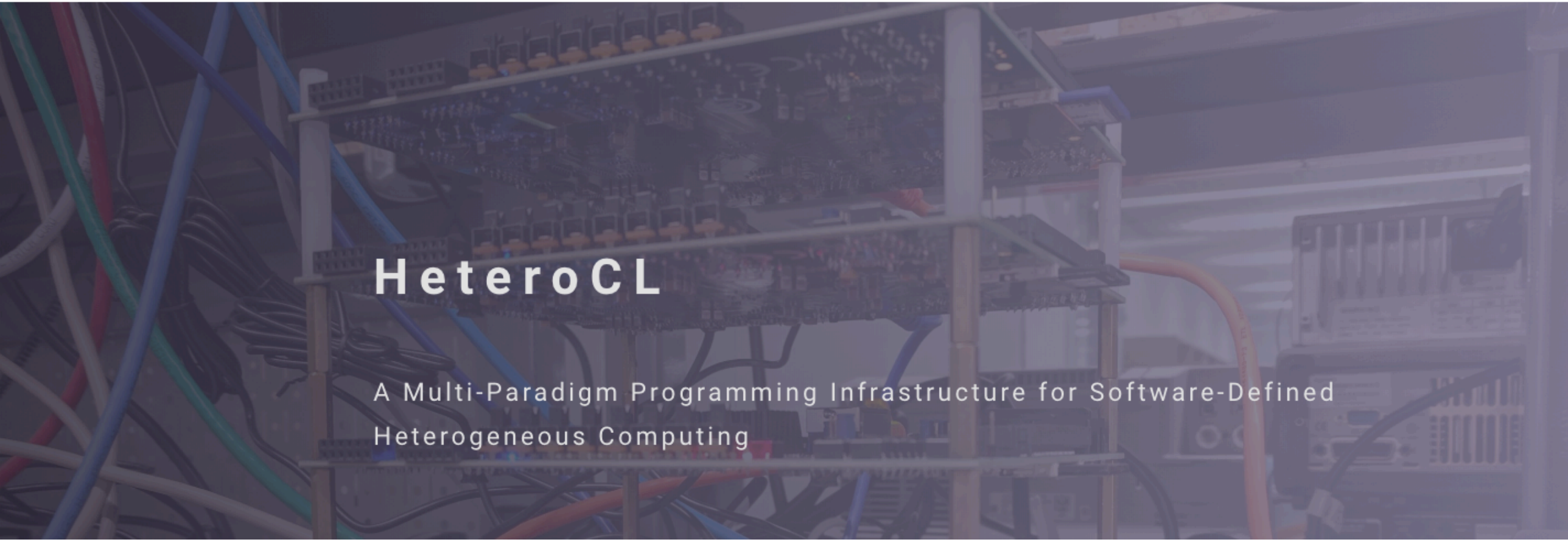
Concluding Remarks

- HeteroCL offers a new programming for building FPGA-targeted accelerators
  - *Flexible*: Mixed declarative & imperative code
  - *Efficient*: Mapping to high-perf. spatial architectures
  - *Portable*: Decoupling of algo. & HW customizations

- Ongoing efforts
  - **High-throughput data streaming**
  - **ML-assisted DSE**
  - **Near-memory sparse compute**



Thanks + Q&A



github.com/cornell-zhang/heterocl

CRISP
Center for Research on Intelligent Storage and Processing in Memory

DARPA intel NSF SRC JUMP XILINX