An MLIR-based Intermediate Representation for Accelerator Design with Decoupled Customizations

Hongzheng Chen, **Niansong Zhang**, Shaojie Xiang, Zhiru Zhang

Cornell University

MLIR Open Design Meeting
08/11/2022

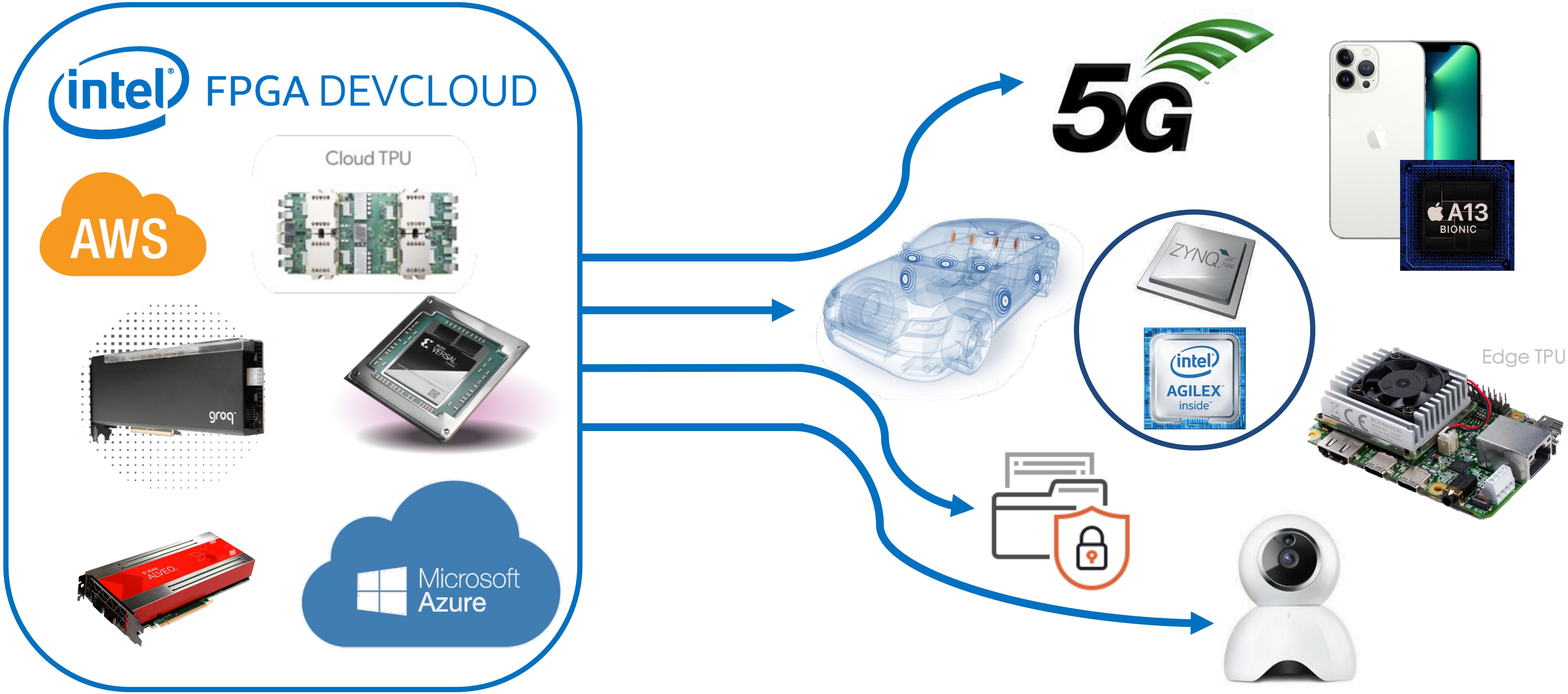
Cornell University

ZHANG

Rise of Specialized Computing

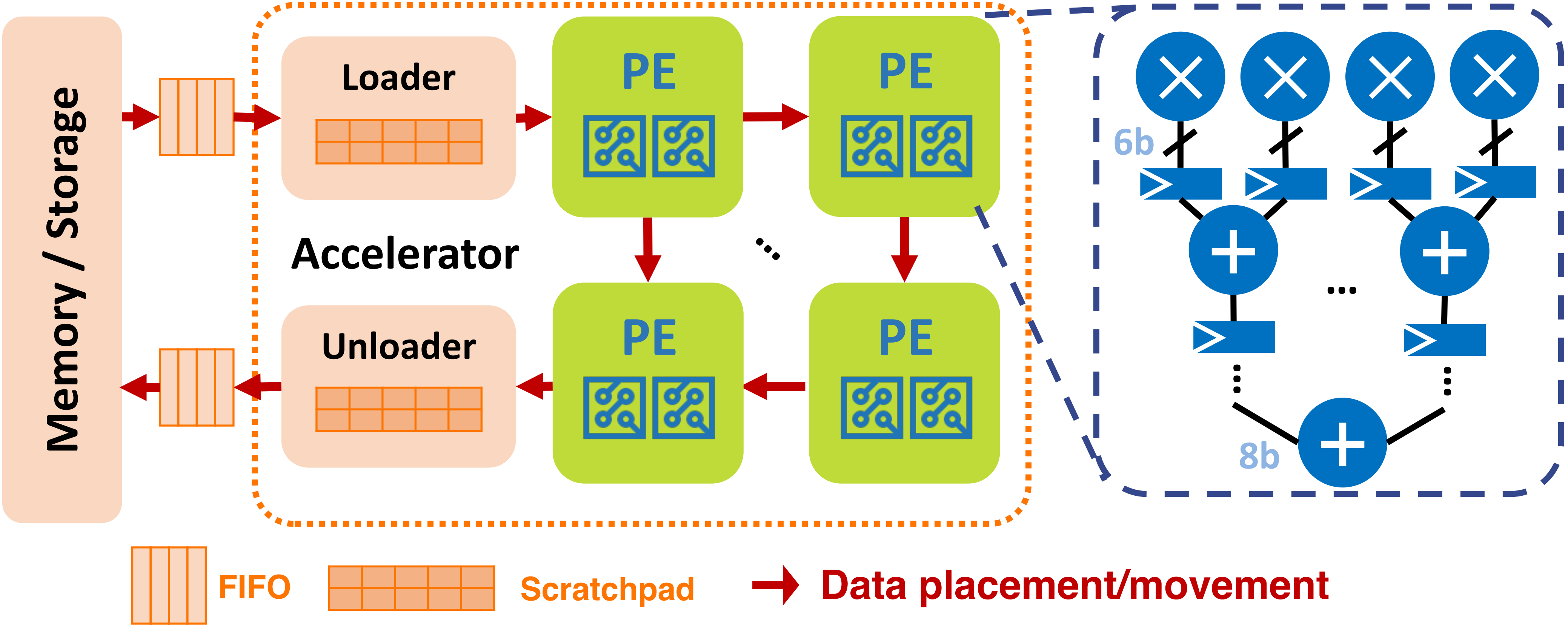
Accelerators in Data Centers

Accelerators in Edge Devices



Specialized Accelerator Design

- Accelerator design is different from programming on general processors
  - Custom processing engines (PEs)
  - Custom non-standard data type
  - Custom memory hierarchy
  - Custom data communication



High-Level Synthesis (HLS)

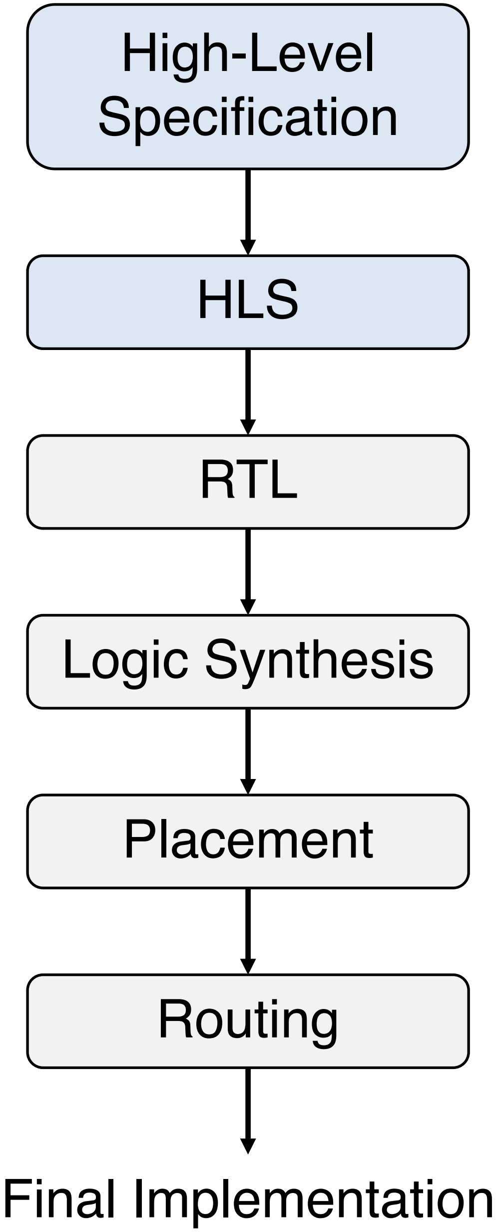
```
module dut(rst, clk, q);
  input rst;
  input clk;
  output q;
  reg [7:0] c;

  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;
}
```

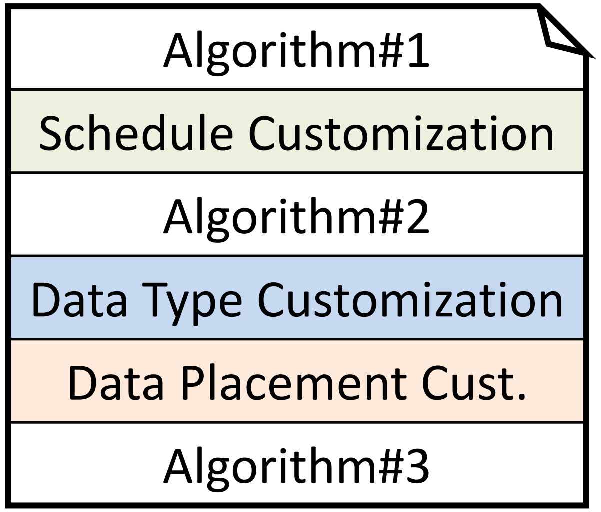
HLS C/C++



Accelerator Design with HLS: Single Kernel

- Example: convolution

```
void conv1(...) {
 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[y, x] += Input[y+r, x+c] * Filter[r, c]
}
```



Entangled hardware customization and algorithm

- Optimized HLS code:

```
void conv1(...) {
 #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 = Input[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;
}}}}}
```

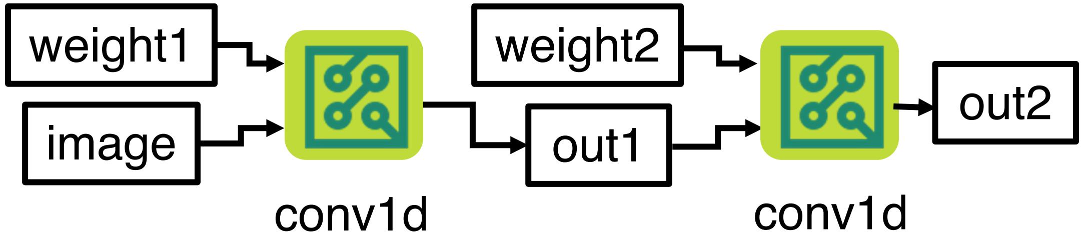
Custom schedule (Loop tiling)

Custom data type (Quantization)

Custom data placement (Reuse buffers)

Accelerator Design with HLS: Multi-Kernel

- Example: Image blur



- Optimized HLS code:

Algorithm#1
Schedule Customization
Algorithm#2
Data Type Customization
Data Placement Cust.
Algorithm#3

Entangled hardware customization and algorithm
- Less portable
- Less maintainable
- Less productive

Host-accelerator

```
void blur(DTYPE* input0, ..., DTYPE* input6,
DTYPE* output0, ..., DTYPE* output6) {
 #pragma HLS interface port=input0 bundle=g0 burst=32
 #pragma HLS interface port=input1 bundle=g1 burst=32
 stream<DTYPE> fifo_in[8], fifo_out[8];
 input_io_schedule(fifo_in, input0, ..., input6);
 #pragma HLS dataflow
 #pragma HLS stream var=fifo_inter[0] depth=32
 #pragma HLS stream var=fifo_inter[1] depth=32
 conv1(fifo_in, fifo_inter);
 conv2(fifo_inter, fifo_out);
 output_io_schedule(fifo_out, output0, ..., output6);
}
```

Inter-kernel

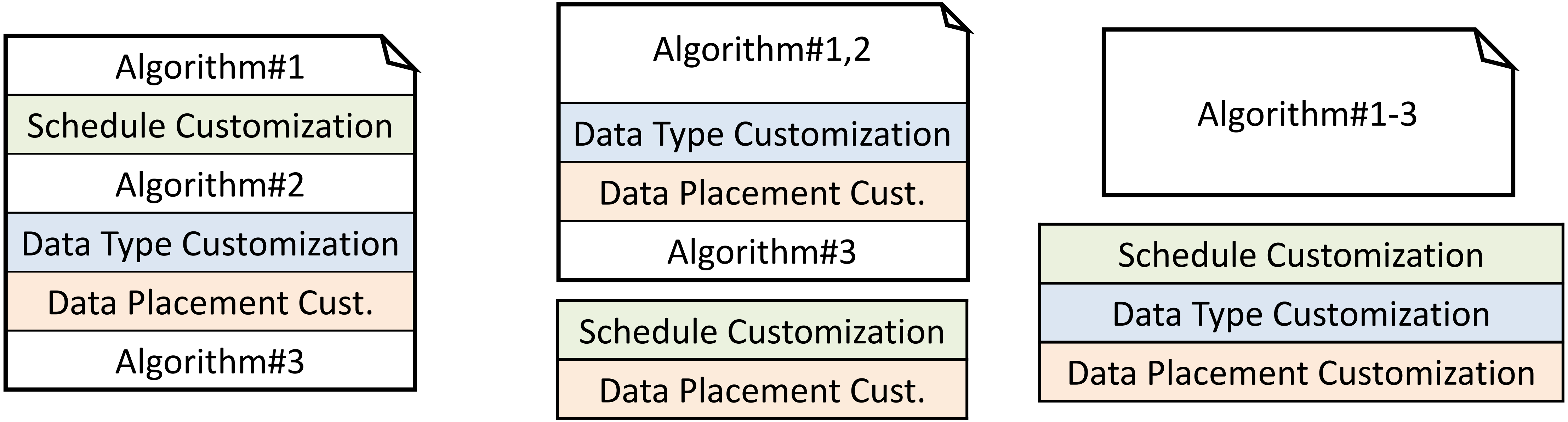
```
void conv2(stream<DTYPE> fifo_inter[8], fifo_out[8]) {
  for (yo=0; yo<128; yo++)
    for (xoo=0; xoo<16; xoo++) {
      for (xoi=0; xoi<8; xoi++) {
        #pragma HLS unroll
        stream<DTYPE> Xin[3], Yin[3], Yout[3];
        broadcast(fifo_inter, Xin[0], Xin[1], Xin[2]);
        PE(w2[0],Xin[0],Yin[0],Yout[0]); Yin[1]=Yout[0];
        PE(w2[1],Xin[1],Yin[1],Yout[1]); Yin[2]=Yout[1];
        PE(w2[2],Xin[2],Yin[2],Yout[2]);
        data_drainer(Yout[2], fifo_out);
}}}
```

Intra-kernel

No unified interface for data placement/movement in HLS

Decoupling Algorithm from Hardware Customizations

HLS C | **Halide, TVM** | **HeteroCL**



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

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

Decoupled schedules [4,5,6,7,8,9]

[4] Ragan-Kelly, et al. PLDI'13
[5] Baghdadi, et al. arXiv'18
[6] Rong, et al. arXiv'17
[7] Pu, et al. TACO'17
[8] Chen, et al. OSDI'18
[9] Ikarashi, et al. PLDI'22

Fully decoupled customization schemes [10,11,12]

[10] Lai, et al., FPGA'19
[11] Lai, et al., ICCAD'20
[12] Xiang, et al., FPGA'22

Decoupling Algorithm from Hardware Customizations

HLS C

Algorithm#1
Schedule Customization
Algorithm#2
Data Type Customization
Data Placement Cust.
Algorithm#3

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

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

Halide, TVM

Algorithm#1,2
Data Type Customization
Data Placement Cust.
Algorithm#3

Schedule Customization
Data Placement Cust.

Decoupled schedules [4,5,6,7,8,9]

[4] Ragan-Kelly, et al. PLDI'13
[5] Baghdadi, et al. arXiv'18
[6] Rong, et al. arXiv'17
[7] Pu, et al. TACO'17
[8] Chen, et al. OSDI'18
[9] Ikarashi, et al. PLDI'22

HCL-MLIR

Algorithm#1-3

Processing

Schedule Customization
Structure Customization
Expression Customization

Data

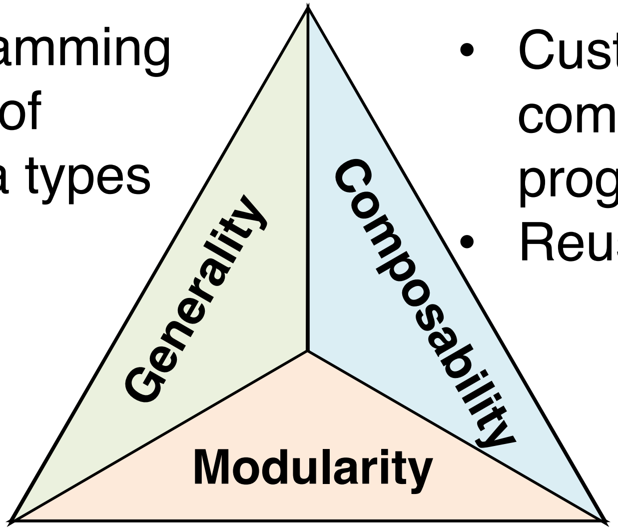
Data Type Customization
Data Placement Customization
Data Layout Customization

Fully decoupled customization schemes

An MLIR-based Accelerator IR

HeteroCL — decoupling customization **at the IR level** → MLIR

- Imperative programming
- Inherent support of nonstandard data types (e.g., `i4`, `bf16`, `!dialect.Type`)



Features of MLIR

- Customizations can be composed with other programs
- Reusable passes & opts

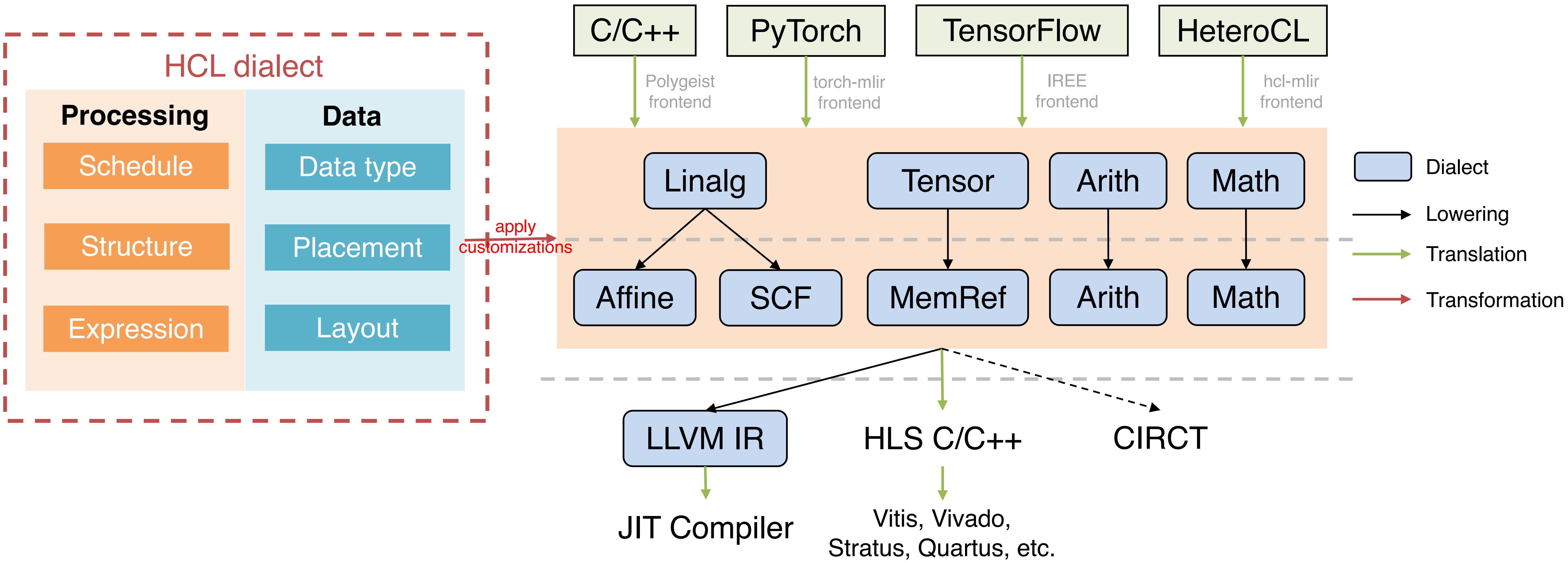
- Different levels of IRs in the same module
- Multiple modules on different devices can work together

Benefits for accelerator design

- Support a wider range of applications
- Localized customization for different parts of design
- Verify the hardware design step by step
- …

Overview of the HCL Dialect

- HCL dialect can work for versatile MLIR programs composed with different levels of IRs



A (Partial) List of Customization Primitives in HCL Dialect

(a) Processing customization

Schedule customization

hcl.split(Op, i, v)	Split loop i of operation Op into a two-level nest loop with v as the factor of the inner loop.
hcl.fuse(Op, i, j)	Fuse two sub-loops i and j of operation Op in the same nest loop into one.
hcl.reorder(Op, i, j)	Switch the order of sub-loops i and j of operation Op in the same nest loop.
hcl.compute_at(Op1, Op2, i)	Merge loop i of the operation Op1 to the corresponding loop level in operation Op2.
hcl.unroll(Op, i, v)	Unroll loop i of operation Op by factor v.
hcl.parallel(Op, i)	Schedule loop i of operation Op in parallel.
hcl.pipeline(Op, i, v)	Schedule loop i of operation Op in pipeline manner with a target initiation interval v.

Structure customization

hcl.outline(Op)	Outline operation Op as a function.
hcl.clone(Func)	Create multiple cloning of function Func.

Expression customization

hcl.rewrite(Pat, Expr)	Rewrite pattern Pat as a new expression Expr.

(b) Data customization

Data type customization

hcl.quantize(A, d)	Quantize tensor A to data type d
	FixedType: Fixed point numbers with custom width and fractional bits
	StructType: Composite data types

Data placement customization

hcl.buffer_at(A, Op, i)	Create an intermediate buffer at dimension i of operation Op to store the results of tensor A.
hcl.reuse_at(A, Op, i)	Create a reuse buffer storing the values of tensor A, where the values are reused at dimension i of operation Op.
hcl.to(A, Dst, Mode)	Move a list of tensors A to destination Dst with Mode.

Data layout customization

hcl.partition(A, i, v)	Cyclic/Block partition dimension i of tensor A with a factor v.
hcl.pack(A, i, v)	Pack dimension i of tensor a into words with a factor v.
hcl.reform(A, Map)	Change the physical data layout of tensor A based on the given affine map Map.

Decoupled Schedule Customization with HCL Dialect

- Example: 1024x1024 matrix multiplication, tiled to 8x8, with unroll and pipeline
- Customization targets are specified with operation and loop handles

```
module {
  func.func @gemm(%A: memref<1024x512xf32>, %B: memref<512x1024xf32>, %C: memref<1024x1024xf32>) {
    // (a) algorithm specification
    linalg.matmul {op_name = "C", axes = ["i", "j", "k"]}
      ins(%A, %B: memref<1024x512xf32>, memref<512x1024xf32>)
      outs(%C: memref<1024x1024xf32>)
    // (b) handle declarations
    %s = hcl.create_op_handle "C"
    %li = hcl.create_loop_handle %s, "i"
    %lj = hcl.create_loop_handle %s, "j"
    %lk = hcl.create_loop_handle %s, "k"
    // (c) customizations
    %li_outer, %li_inner = hcl.split(%li, 8)
    %lj_outer, %lj_inner = hcl.split(%lj, 8)
    hcl.reorder(%li_outer, %lj_outer, %li_inner, %lj_inner)
    hcl.unroll(%lj_inner)
    hcl.pipeline(%li_inner, 1)
    return
  }
}
```

Op and loop handle creation

Schedule customizations

Decoupled Schedule Customization with HCL Dialect

- The MLIR assembly after all schedule customizations applied
- Loops are transformed, unroll and pipeline are attached as attributes for codegen

```
#map0 = affine_map<(d0) -> (d0 * 8)>
#map1 = affine_map<(d0, d1) -> (d1 + d0)>
func.func @gemm(%arg0: memref<1024x512xf32>, %arg1: memref<512x1024xf32>) -> memref<1024x1024xf32> {
  %0 = memref.alloc() {name = "C"} : memref<1024x1024xf32>
  affine.for %arg2 = 0 to 32 {
    affine.for %arg3 = 0 to 32 {
      affine.for %arg4 = 0 to 8 {
        affine.for %arg5 = 0 to 8 {
          %1 = affine.apply #map0(%arg3)
          %2 = affine.apply #map1(%1, %arg5)
          %3 = affine.apply #map0(%arg2)
          %4 = affine.apply #map1(%3, %arg4)
          %5 = memref.alloc() {name = "sum_rv"} : memref<f32>
          %c0_f32 = arith.constant 0 : f32
          affine.store %c0_f32, %5[] {to = "sum_rv"} : memref<f32>
          affine.for %arg6 = 0 to 512 {
            // more computation
          } {loop_name = "k"}
        } {loop_name = "j.inner", unroll = 0 : f32}
      } {loop_name = "i.inner", pipeline_ii = 1 : f32}
    } {loop_name = "j.outer"}
  } {loop_name = "i.outer", op_name = "C"}
  return %0 : memref<32x32xf32>
}
```

Schedule customizations applied

Structure Customization: Function Outlining

- Outline one or multiple operations as a function
  - Essential for hardware **resource sharing**
  - Unify functions with arguments with different sizes

```
module {
 func.func @gemm(%A: memref<1024x512xf32>, %B: memref<512x1024xf32>,
                 %C: memref<1024x1024xf32>, %D: memref<1024x1024xf32>,
                 %E: memref<1024x1024xf32>)
 {
  // (a) algorithm specification
  linalg.matmul {op_name = "s1", axes = ["i1", "j1", "k1"]}
    ins(%A, %B: memref<1024x512xf32>, memref<512x1024xf32>)
    outs(%C: memref<1024x1024xf32>)
  linalg.matmul {op_name = "s2", axes = ["i2", "j2", "k2"]}
    ins(%C, %D: memref<1024x1024xf32>, memref<1024x1024xf32>)
    outs(%E: memref<1024x1024xf32>)
  // (b) handle declarations
  %s1 = hcl.create_op_handle "s1"
  %s2 = hcl.create_op_handle "s2"
  // (c) customizations
  hcl.outline(%s1, %s2) {unify}
  return
}}
```

Structure Customization: Function Outlining

- Outline one or multiple operations as a function
  - ① Generate functions & call operations (1 function w/ 2 function calls)
  - ② Automatically fetch the input & output parameters and change the memref size
  - ③ Parameterize loop bounds

```
module {                 ①
 func.func private @F_s1_s2(%arg0: memref<1024x1024xf32>, ②
                           %arg1: memref<1024x1024xf32>,
                           %arg2: memref<1024x1024xf32>, %arg3: index) {
  affine.for %i = 0 to 1024 {③
   affine.for %j = 0 to %arg3 {
    affine.for %k = 0 to 1024 {
     // actual payload
  }}}}

 func.func @gemm(%A: memref<1024x1024xf32>, %B: memref<1024x1024xf32>, %C: memref<1024x1024xf32>,
                 %D: memref<1024x1024xf32>, %E: memref<1024x1024xf32>)
 {
  %c512 = arith.constant 512 : index ③
① func.call @F_s1_s2(%A, %B, %C, %c512)
  %c1024 = arith.constant 1024 : index
  func.call @F_s1_s2(%C, %D, %E, %c1024)
  return
}}
```

Data Customization Example: Binary Convolution

- 2D binary convolution (bconv)

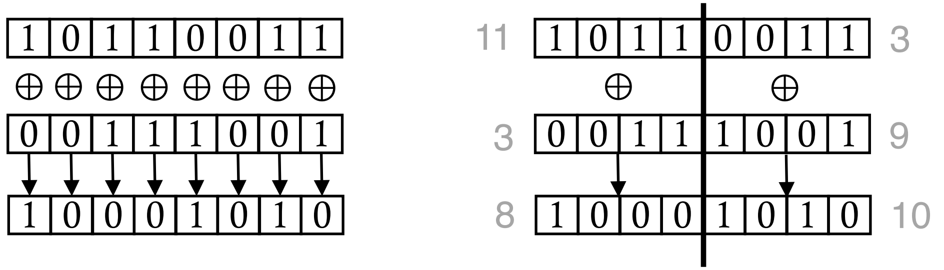
$$B_{n,c,h,w} = \sum_{rc}^{IC}\sum_{rh}^{KH}\sum_{rw}^{KW}(1 - 2 \cdot A_{n,rc,h+rh,w+rw} \oplus F_{c,rc,rh,rw})$$

Data type

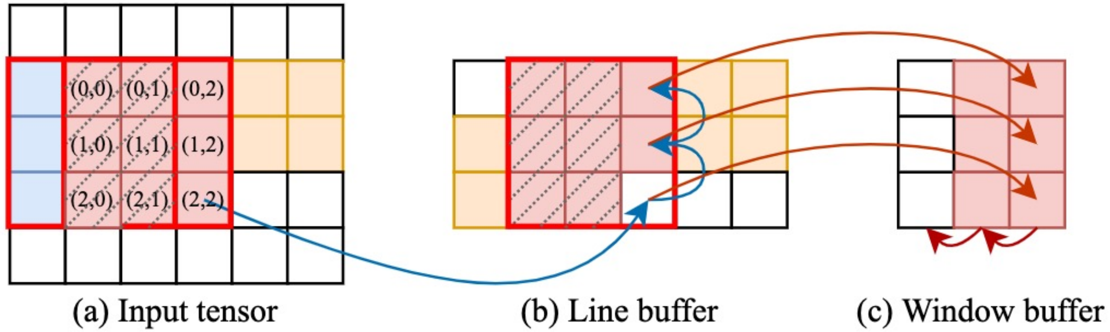
```
func.func @top(%A: memref<4x16x8x8xi1>,
               %F: memref<32x16x3x3xi1>)
  -> memref<4x32x6x6xi32> {
  // (a) algorithm specification
  %B = memref.alloc() : memref<4x32x6x6xi32>
  linalg.generic #bconv_trait
    ins(%A, %F: memref<4x16x8x8xi1>, memref<32x16x3x3xi1>)
    outs(%B: memref<4x32x6x6xi32>) { ... }
  // (b) handle declarations
  // ...
  // (c) customizations
  %pA = hcl.pack(%A, 1, 16) -> memref<4x1x8x8xi16>
  %pF = hcl.pack(%F, 1, 16) -> memref<32x1x3x3xi16>
  hcl.pipeline(%w, 1)
  %LB = hcl.reuse_at(%pA, %h) -> memref<3x8xi16>
  %WB = hcl.reuse_at(%LB, %w) -> memref<3x3xi16>
  hcl.partition(%LB : memref<3x8xi16>) {axis=[0]}
  hcl.partition(%WB : memref<3x3xi16>) {axis=[0,1]}
  hcl.partition(%pF : memref<32x1x3x3xi16>) {axis=[2,3]}
  return %B : memref<4x32x6x6xi32>
}
```

Data layout (pack); Data placement (reuse_at); Data layout (partition)

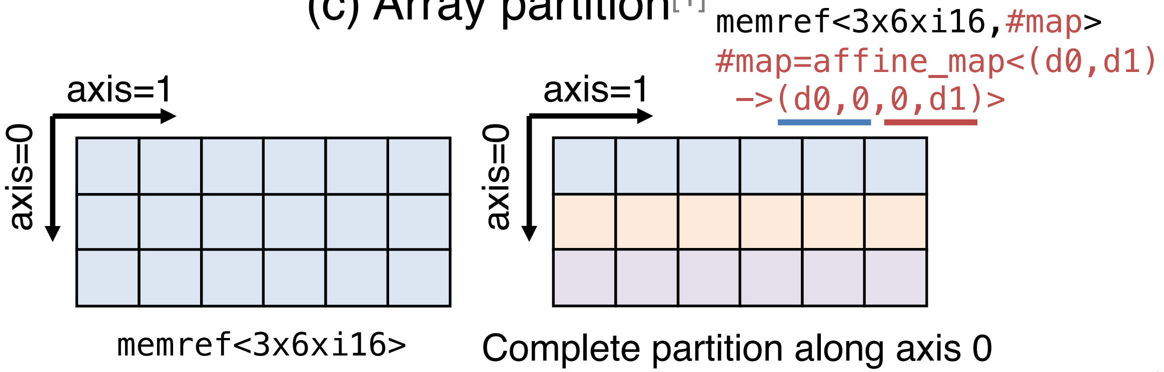
(a) Bitpacking



(b) Data reuse



(c) Array partition[1]



[1] Ye et al., ScaleHLS: A New Scalable High-Level Synthesis Framework on Multi-Level Intermediate Representation. HPCA, 2022.

Data Customization Example: Binary Convolution

```
#map0 = affine_map<(d0, d1, d2, d3)
  -> (0, 0, d2, d3, d0, d1, 0, 0)>
#map1 = affine_map<(d0, d1) -> (d0, 0, 0, d1)>
#map2 = affine_map<(d0, d1) -> (d0, d1, 0, 0)>
#set = affine_set<(d0) : (d0 - 2 >= 0)>
module {
 func.func @top(%arg0: memref<4x1x8x8xi16>, %arg1:
memref<32x1x3x3xi16, #map0>) -> memref<4x32x6x6xi32> {
  %c0 = arith.constant 0 : index
  %0 = memref.alloc() : memref<4x32x6x6xi32>
  %1 = memref.alloc() : memref<3x8xi16, #map1>
  %2 = memref.alloc() : memref<3x3xi16, #map2>
  affine.for %arg2 = 0 to 4 {
   affine.for %arg3 = 0 to 32 {
    affine.for %arg4 = 0 to 8 {
     affine.for %arg5 = 0 to 8 {
      // shift line buffer
      %3 = affine.load %1[1, %arg5] : memref<3x8xi16, #map1>
      affine.store %3, %1[0, %arg5] : memref<3x8xi16, #map1>
      %4 = affine.load %1[2, %arg5] : memref<3x8xi16, #map1>
      affine.store %4, %1[1, %arg5] : memref<3x8xi16, #map1>
      %5 = affine.load %arg0[%arg2, %c0, %arg4, %arg5] :
                memref<4x1x8x8xi16>
      affine.store %5, %1[2, %arg5] : memref<3x8xi16, #map1>
```

Data layout customization

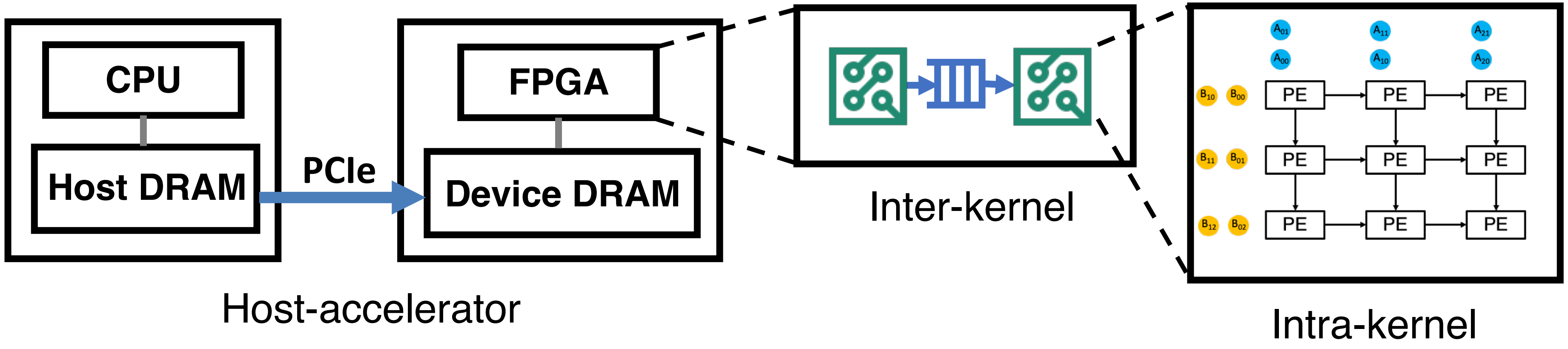
Data type customization

Data placement customization

```
      affine.if #set(%arg4) {
       // shift window buffer
       affine.for %arg6 = 0 to 3 {
        %6 = affine.load %2[%arg6, 1]
        affine.store %6, %2[%arg6, 0]
        %7 = affine.load %2[%arg6, 2]
        affine.store %7, %2[%arg6, 1]
        %8 = affine.load %1[%arg6, %arg5]
        affine.store %8, %2[%arg6, 2]
       } {spatial}
       affine.if #set(%arg5) {
        // computation
     }}} {loop_name = "w", pipeline_ii = 1 : i32}
    } {loop_name = "h"}
   } {loop_name = "c"}
  } {loop_name = "n", op_name = "B"}
  return %0 : memref<4x32x6x6xi32>
}}
```

Data Placement Customization: `.to()`

- Data placement: deliver the right data at the right moment with the right communication scheme
- Essential to performance: 3-8X ↑ with communication optimization only[1]
- Coarse-grained: host-accelerator data placement
- Medium-grained: inter-kernel data placement within an accelerator
- Fine-grained: intra-kernel data placement between processing elements



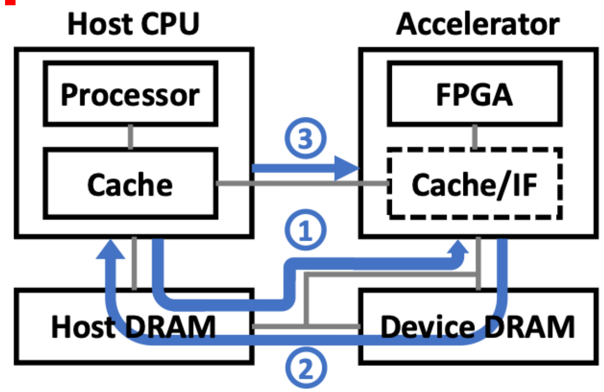
[1] A. Lu, Z.Fang, W. Liu, and L. Shannon. Demystifying the memory System of Modern Datacenter FPGAs for Software Programmers through Microbenchmarking. Int'lSymp. On Frield-Programmable Gate Arrays (FPGA), 2021

Data Placement Customization: Host-Accelerator

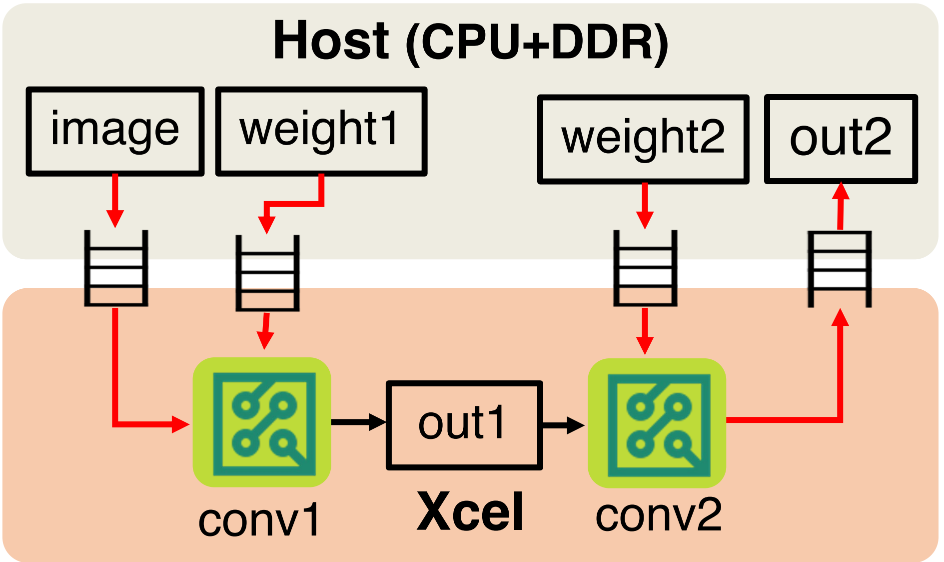
- Example: stream I/O for two convolution layers on FPGA
  - Graph partition & generate two modules for host & xcel

```
func.func @top(%image : memref<1x32x32x3f32>,
               %w1 : memref<32x3x3x3xf32>, %w2 : memref<64x3x3x32xf32>) ->
               memref<1x32x32x64xf32> {
  // (a) algorithm specification
  %out1 = memref.alloc() : memref<1x32x32x32xf32>
  linalg.conv_2d_nhwc_fhwc {op_name = "conv1"}
    ins(%image : memref<1x32x32x3xf32>, %w1 : memref<32x3x3x3xf32>)
    outs(%out1 : memref<1x32x32x32xf32>)
  %out2 = memref.alloc() : memref<1x32x32x64xf32>
  linalg.conv_2d_nhwc_fhwc {op_name = "conv2"}
    ins(%out1 : memref<1x32x32x32xf32>,
        %w2 : memref<64x3x3x32xf32>)
    outs(%out2 : memref<1x32x32x64f32>)
  // (b) handle declarations
  %conv2 = hcl.create_op_handle "conv2"

  // (c) host-xcel data placement customizations
  hcl.to([%image, %w1, %w2], Xcel) {mode="stream"}
  hcl.to([%out2], Host) {mode="stream"}
  // (d) inter-kernel data placement customizations
  hcl.to([%out1], %conv2)
  return %out2 : memref<1x32x32x64xf32>
}
```

Host-xcel data placement



Host-xcel data placement modes:
① DMA (direct streaming)
② DMA (via device DRAM)
③ Cache-coherent interface



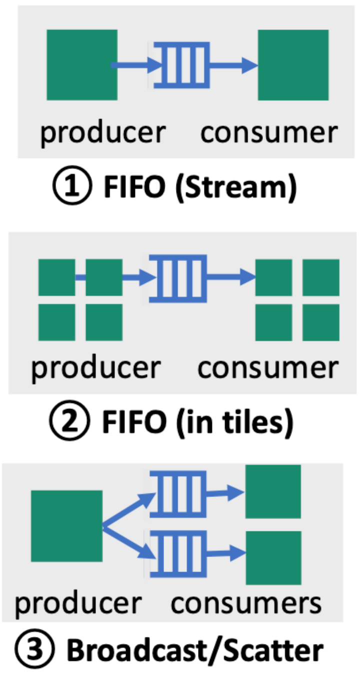
Data Placement Customization: Inter-Kernel

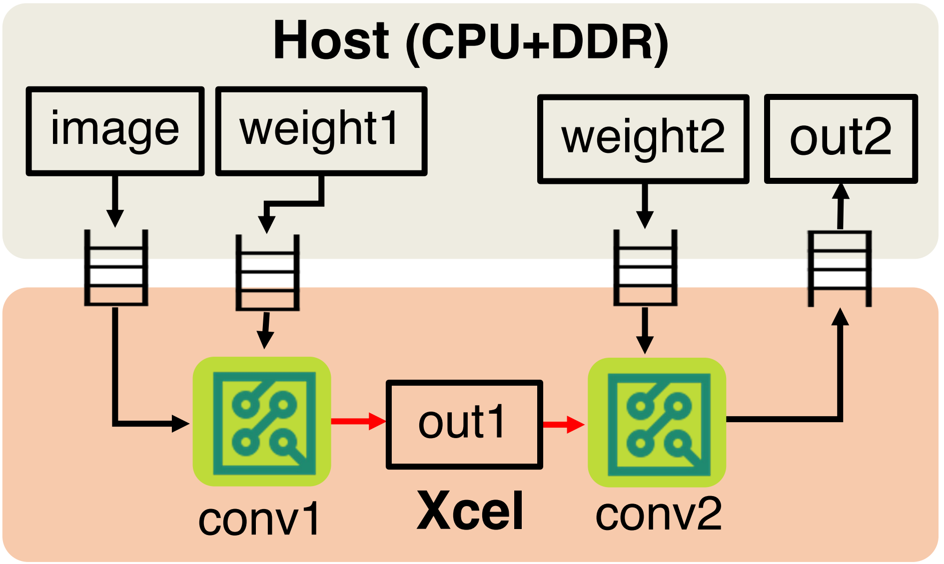
- Example: stream intermediate results between conv layers
  - Attach mode attribute to memref

```
func.func @top(%image : memref<1x32x32x3f32>,
               %w1 : memref<32x3x3x3xf32>, %w2 : memref<64x3x3x32xf32>) ->
               memref<1x32x32x64xf32> {
  // (a) algorithm specification
  %out1 = memref.alloc() : memref<1x32x32x32xf32>
  linalg.conv_2d_nhwc_fhwc {op_name = "conv1"}
    ins(%image : memref<1x32x32x3f32>, %w1 : memref<32x3x3x3xf32>)
    outs(%out1 : memref<1x32x32x32xf32>)
  %out2 = memref.alloc() : memref<1x32x32x64xf32>
  linalg.conv_2d_nhwc_fhwc {op_name = "conv2"}
    ins(%out1 : memref<1x32x32x32xf32>,
        %w2 : memref<64x3x3x32xf32>)
    outs(%out2 : memref<1x32x32x64xf32>)
  // (b) handle declarations
  %conv2 = hcl.create_op_handle "conv2”

  // (c) host-xcel data placement customizations
  hcl.to([%image, %w1, %w2], Xcel) {mode="stream"}
  hcl.to([%out2], Host) {mode="stream"}
  // (d) inter-kernel data placement customizations
  hcl.to([%out1], %conv2) {mode="stream"}
  return %out2 : memref<1x32x32x64xf32>
}
```

Host-xcel data placement

Inter-kernel data placement





Data Placement Customization: Intra-Kernel

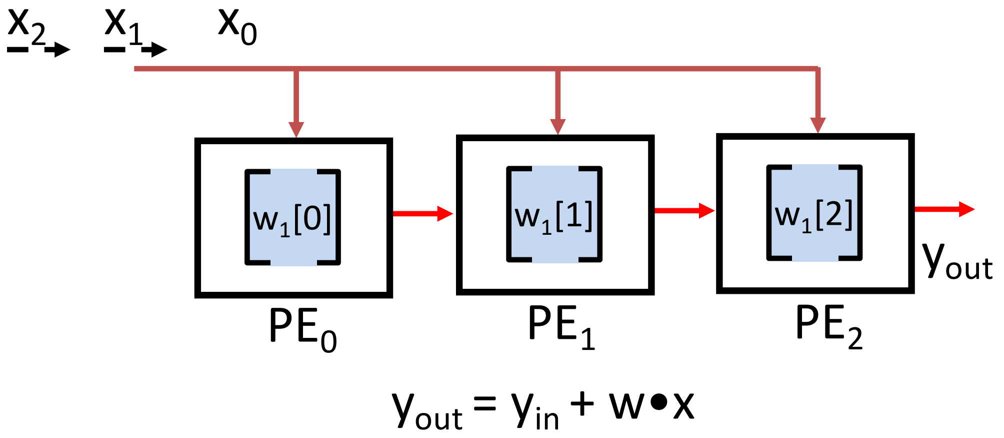
- Example: stream data between processing elements (PE) results inside kernels

```
func.func @top(%input : memref<1x32x32x3xf32>, %w1 : memref<32x3x3x3xf32>)
    -> memref<1x32x32x32xf32> {
  // (a) algorithm specification
  %out1 = memref.alloc() : memref<1x32x32x32xf32>
  linalg.conv_2d_nhwc_fhwc
    {op_name = "conv1", axes = ["n", "oc", "oh", "ow", "rc", "rh", "rw"]}
    ins(%image : memref<1x32x32x3xf32>, %w1 : memref<32x3x3x3xf32>)
    outs(%out1 : memref<1x32x32x32xf32>)
  // (b) handle declarations
  %conv1 = hcl.create_op_handle "conv1"
  %rw = hcl.create_loop_handle %conv1, "rw"
  // (c) intra-kernel data placement
  %pe = hcl.unroll(%rw, 3) {explicit}
  hcl.to(%input, [%pe:0, %pe:1, %pe:2])
  %pe0_w = hcl.to(%w1, %pe:0)
  %pe1_w = hcl.to(%pe0_w, %pe:1)
  hcl.to(%pe1_w, %pe:2)
  return %out1 : memref<1x32x32x32xf32>
}
```

Unroll innermost loop, create 3 PEs

Broadcast input

Pass on weights



Decoupled & concise specification of a weight stationary design

Parameterized Customization Template

- Reuse optimizations for kernels with different sizes
- `hcl.customization`: A sequence of optimization primitives
  - Modularity: Fully decoupled from alg spec; no need to be a monolithic design
  - Composability: Readable and parsable format; dump to file; plug in for different apps

```
hcl.customization @gemm_opt(
 %A: memref<?x?x!hcl.Type>,
 %B: memref<?x?x!hcl.Type>,
 %C: memref<?x?x!hcl.Type>,
 %s: !hcl.OpHandle,
 %i: !hcl.LoopHandle,
 %j: !hcl.LoopHandle,
 %k: !hcl.LoopHandle
) {
 hcl.pipeline(%s, %j, 1)
 hcl.partition(%A: memref<?x?x!hcl.Type>, "CompletePartition", 2)
 hcl.partition(%B: memref<?x?x!hcl.Type>, "CompletePartition", 2)
 hcl.partition(%C: memref<?x?x!hcl.Type>, "CompletePartition", 2)
}
```

Input & output arrays w/ generic shape & types (ext. partial specialization)

Op & Loop handles

Parameterized Customization Template

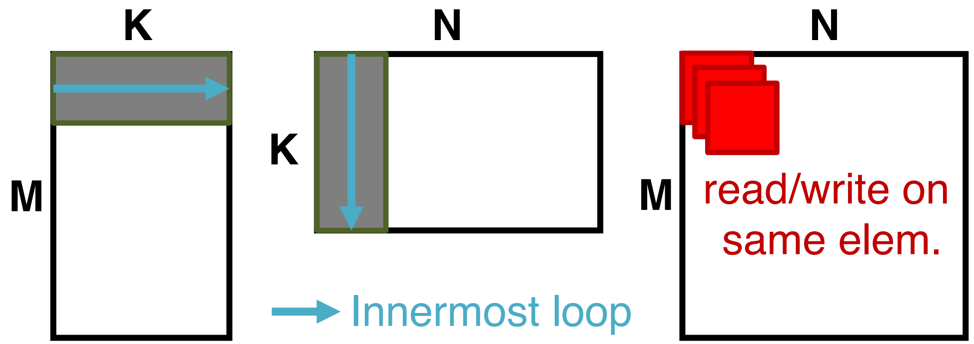
- Reuse optimizations for kernels with different sizes
- `hcl.apply`: Apply a customization to a kernel

```
module {
 func.func @top(%A: memref<1024x512xi32>, %B: memref<512x1024xi32>,
                %C: memref<1024x1024xi32>)
         -> memref<1024x1024xi32>
 {
  // loop and stage handle declaration
  // ...
  // D = A * B
  %D = memref.alloc() : memref<1024x1024xi32>
  // first kernel
  // ...
  // E = C * D
  %E = memref.alloc() : memref<1024x1024xi32>
  // second kernel
  // ...
  // apply customizations
  hcl.apply @gemm_opt(%A, %B, %D, %s1, %i1, %j1, %k1)
  hcl.apply @gemm_opt(%C, %D, %E, %s2, %i2, %j2, %k2)
  return %E : memref<1024x1024xi32>
}}
```

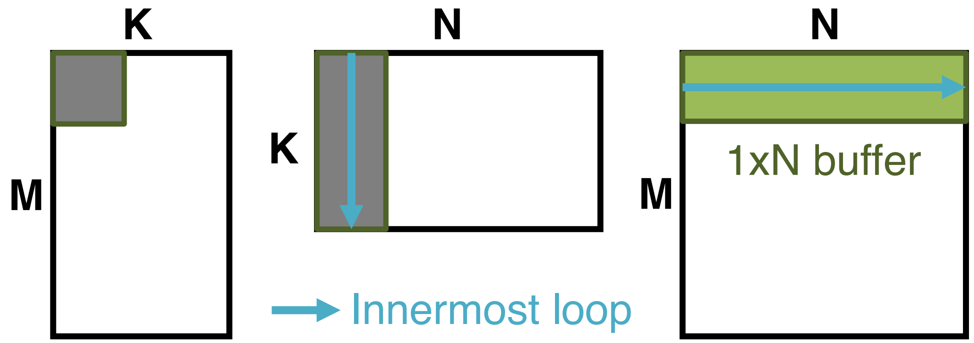
Reuse opt

Case Study: Matrix Multiplication (GEMM)

- In a vanilla GEMM implementation, **floating-point** accumulation introduces carried dependency, slowing down the pipeline (II>1)



(1) Vanilla GEMM



(2) GEMM w/ accumulation interleaving [1]

Optimized GEMM

M=1024, K=512, N=1024

```
void GEMM_v2(
 float A[1024][512],
 float B[512][1024],
 float C[1024][1024]
) {
for (int i = 0; i < 1024; i++) {
  float buf_C[1024];
  for (int j = 0; j < 1024; j++) {
    #pragma HLS pipeline II=1
    buf_C[j] = 0.0;
  }
  for (int k = 0; k < 512; k++) {
    for (int j = 0; j < 1024; j++) {
      #pragma HLS pipeline II=1
      buf_C[j] += A[i][k] * B[k][j];
    }
  }
  for (int j = 0; j < 1024; j++) {
    #pragma HLS pipeline II=1
    C[i][j] = buf_C[j];
  }
}}
```

Init buffer

Reordered compute

Write back

[1] JF Licht et al., Transformations of high-level synthesis codes for high-performance computing, TPDS'2020.

Case Study: Matrix Multiplication (GEMM)

- Accumulation interleaving using decoupled primitives
  - `.buffer_at()` creates an intermediate buffer at a given axis
  - Algorithm code stays unchanged

	Initiation Interval (II)	Latency (cycles)	Speedup
GEMM baseline	8	4295M	1x
GEMM w/ Acc. Interleaving	**1**	539M	**7.97x**

GEMM Optimization

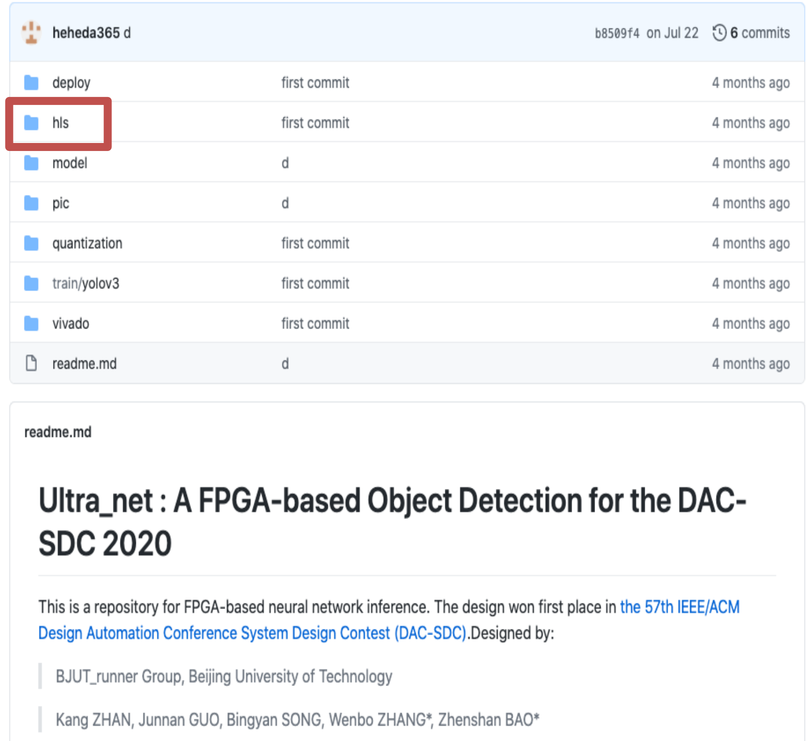
M=1024, K=512, N=1024

```
module {
 func.func @gemm(%A: memref<1024x512xf32>,
                 %B: memref<512x1024xf32>,
                 %C: memref<1024x1024xf32>)
 {
  // (a) algorithm specification
  linalg.matmul {op_name = "s",
                 axes = ["i", "j", "k"]}
    ins(%A, %B: memref<1024x512xf32>,
                memref<512x1024xf32>)
    outs(%C: memref<1024x1024xf32>)
  // (b) handle declarations
  %s = hcl.create_op_handle "s"
  %li = hcl.create_loop_handle %s, "i"
  %lj = hcl.create_loop_handle %s, "j"
  %lk = hcl.create_loop_handle %s, "k"
  // (c) customizations
  hcl.reorder(%lk, %lj)
  hcl.buffer_at(%C: memref<1024x1024xf32>, %li)
     -> memref<1024xf32>
  hcl.pipeline(%lj, 1)
  return
}}
```

[1] JF Licht et al., Transformations of high-level synthesis codes for high-performance computing, TPDS'2020.

Case Study: UltraNet

HeteroCL makes it convenient to integrate systolic kernels with other non-systolic ones



https://github.com/heheda365/ultra_net

```
func.func @ultranet(%image : memref<1x224x224x3xf32>, ...) {
  // (a) algorithm specification
  %out1 = memref.alloc() : memref<1x224x224x32xf32>
  linalg.conv_2d_nhwc_fhwc {op_name = "conv1"}
      ins(%image : memref<1x224x224x3xf32>, %w1 : memref<32x3x3x3xf32>)
      outs(%out1 : memref<1x224x224x32xf32>)
  %out2 = memref.alloc() : memref<1x224x224x64xf32>
  linalg.conv_2d_nhwc_fhwc {op_name = "conv2"}
      ins(%out1 : memref<1x224x224x32xf32>, %w2 : memref<64x3x3x32xf32>)
      outs(%out2 : memref<1x224x224x64xf32>)
  ...
  // (b) handle declarations
  ...
  // (c) customizations
  // inter-kernel data movement
  hcl.to(%image, %conv1)
  // build weight stationary systolic array
  %pe = hcl.unroll(%rw, 4) {explicit}
  hcl.to(%out2, [%pe:0, %pe:1, %pe:2, %pe:3])
  %pe0_w = hcl.to(%w1, %pe:0)
  %pe1_w = hcl.to(%w2, %pe:1)
  %pe2_w = hcl.to(%w2, %pe:2)
  hcl.to(%w2, %pe:3)
  // quantization
  hcl.quantize(%out1) : (memref<...xf32>) -> (memref<...x!hcl.Fixed<4,3>)
  return
}
```

Accelerating 3rd layer of UltraNet with a systolic array

	# LUTs	# FFs	# BRAM	# DSPs	Fmax(MHz)	RT(ms)
Baseline	60.2K	39.6K	377	508	231	2.97
+Systolic Array	69.8K	39.4K	375	594	233.8	2.27

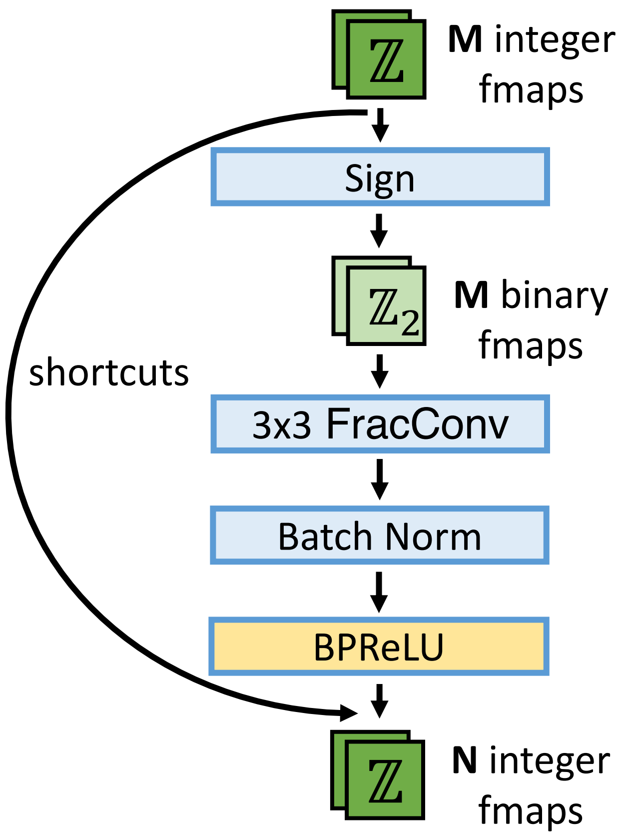
Case Study: Binary Neural Network (BNN)

- FracBNN [1]: a state-of-the-art BNN model
  - CIFAR-10: 19 convolutional (conv) layers + 1 dense layer
    - BPReLU is parametric ReLU with a shifted origin
  - All conv layers are binarized with fractional activation

Design	Param Bits	Accuracy on CIFAR-10	Frame Rate (FPS)
Vanilla BNN [2]	13.4M	88.8%	168.4
FBNA [3]	13.4M	88.6%	520.8
FracBNN in HLS C++ [1] (1575 LoC)	**0.27M**	**89.1%**	**2806.9**
FracBNN in HeteroCL (250 LoC)	**0.27M**	**89.1%**	**3530.1**

Target embedded FPGA: Xilinx Ultra96V2

FracBNN Building Block



A SoTA BNN model is *productively* implemented in HeteroCL and achieving *high performance*

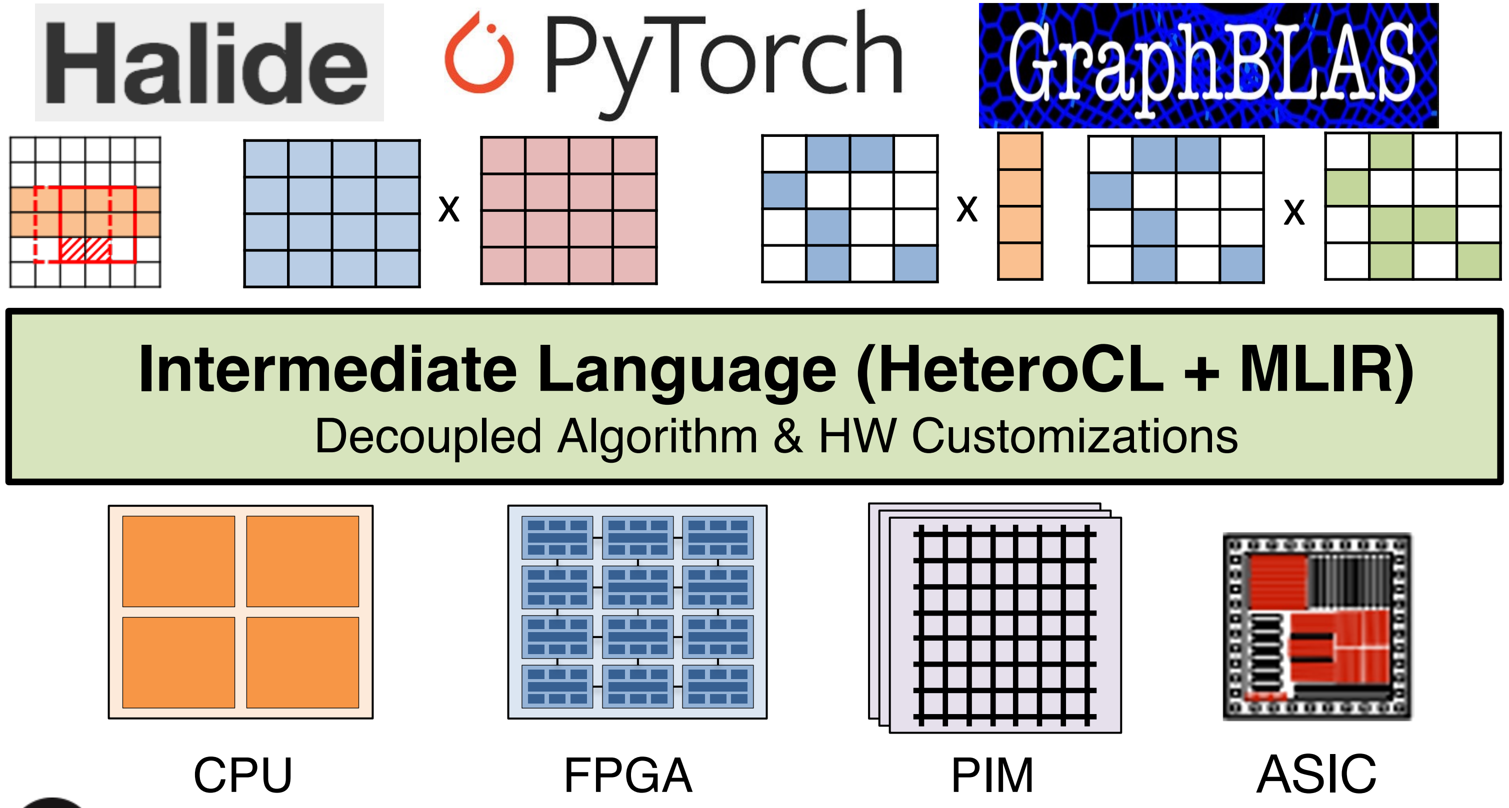
[1] Zhang et al. FracBNN: Accurate and FPGA-Efficient Binary Neural Networks with Fractional Activations. FPGA'21.
[2] Zhao et al. Accelerating Binarized Convolutional Neural Networks with Software-Programmable FPGAs. FPGA'17.
[3] Guo et al. FBNA: A Fully Binarized Neural Network Accelerator. FPL'18.

Discussion: Decoupled Customizations at the IR Level

- General Platform for high-level DSLs
  - Most of DSLs with decoupled customizations have overlaps, and only support declarative
  - Need a common platform with declarative and imperative support and decoupled customization primitives
- Performance & Productivity
  - Useful for design space exploration to search the optimal design automatically
  - Localized customizations for different constraints and scenarios
- Verification
  - Easier to verify a spec composed of customization primitives
  - Scalable to many customizations

Summary and Ongoing Efforts

- **Key Benefits of decoupled customizations in MLIR:**
  Productive, performant, and portable accelerator design
- **Ongoing efforts:**
  (1) Integration with other frontends and backend devices
  (2) Auto generation/recommendation of custom primitives
  (3) Leverage the facilities of the xform & PDL dialect



https://github.com/cornell-zhang/heterocl/tree/hcl-mlir

Related Publications

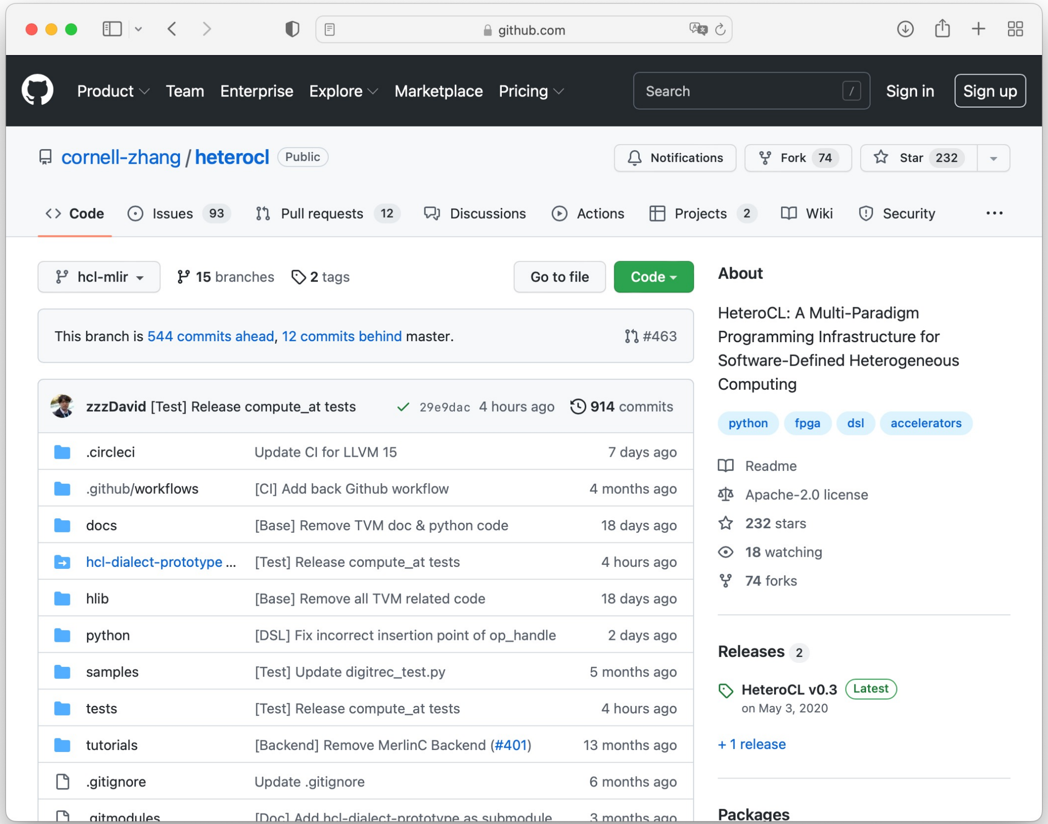
- Debjit Pal, Yi-Hsiang Lai, Shaojie Xiang, Niansong Zhang, Hongzheng Chen, Jeremy Casas, Pasquale Cocchini, Zhenkun Yang, Jin Yang, Louis-Noël Pouchet, Zhiru Zhang. **Accelerator Design with Decoupled Hardware Customizations: Benefits and Challenges**. In DAC, 2022. (Invited Paper)
- Shaojie Xiang, Yi-Hsiang Lai, Yuan Zhou, Hongzheng Chen, Niansong Zhang, Debjit Pal, Zhiru Zhang. **HeteroFlow: An Accelerator Programming Model with Decoupled Data Placement for Software-Defined FPGAs**. In FPGA, 2022.
- Yi-Hsiang Lai, Yuze Chi, Yuwei Hu, Jie Wang, Cody Hao Yu, Yuan Zhou, Jason Cong, Zhiru Zhang. **HeteroCL: A Multi-Paradigm Programming Infrastructure for Software-Defined Reconfigurable Computing**. In FPGA, 2019. (Best Paper Award)

Acknowledgements

https://github.com/cornell-zhang/heterocl/tree/hcl-mlir

Contributors & Collaborators

- **Cornell**: Shaojie Xiang, Jie Liu, Zhongyuan Zhao, Andrew Butt, Alex Na, Yassine Ghannane
- **UIUC**: Hanchen Ye
- **UCLA**: Licheng Guo, Jason Lau, Yuze Chi, Jason Cong
- **UIC**: Debjit Pal
- **CSU**: Louis-Noël Pouchet
- **AWS**: Yi-Hsiang Lai
- **Intel**: Jeremy Casas, Pasquale Cocchini, Zhenkun Yang, Jin Yang, Hongbo Rong



Sponsors

DARPA, NSF, SRC JUMP, CRISP (Center for Research on Intelligent Storage and Processing in Memory), intel, AMD XILINX