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.
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

```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]
```

```
#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;
```

```c
for(int y = 0; y < N; y++) {
    for(int xo = 0; xo < N/M; xo++) {
        for(int xi = 0; xi < M; xi++) {
            #pragma HLS pipeline II=1
            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;
            }
        }
    }
}}}
```

Entangled hardware customization and algorithm
- Less portable
- Less maintainable
- Less productive
Decoupling Algorithm from Hardware Customization

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

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

Decoupled temporal schedules \([4,5,6,7,8]\)  

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

Fully decoupled customization schemes + Clean abstraction capturing the interdependence

[1] Intel HLS  
[2] Xilinx Vivado HLS  
Decoupled Compute Customization

HeteroCL code

```python
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])
```

HLS code

```python
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

```python
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
Decoupled Memory Customization

- Primitives can be applied with a user-defined sequence

```python
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]
```

```python
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 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)
```

```python
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]
```
Primitives can be applied with a user-defined sequence

```python
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)
```

```python
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

```python
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]))
```

```python
s = hcl.create_schedule()
linebuf = s[image].reuse_at(out, out.y)
winbuf = s[linebuf].reuse_at(out, out.x)
```
Decoupled Data Type Customization

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

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

```python
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]))
```

```python
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 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**

<table>
<thead>
<tr>
<th>Primitive</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Loop transformation</td>
<td></td>
</tr>
<tr>
<td>C.split(i, v)</td>
<td>Split loop i of operation C into a two-level nest loop with v as the factor of the inner loop.</td>
</tr>
<tr>
<td>C.fuse(i, j)</td>
<td>Fuse two sub-loops i and j of operation C in the same nest loop into one.</td>
</tr>
<tr>
<td>C.reorder(i, j)</td>
<td>Switch the order of sub-loops i and j of operation C in the same nest loop.</td>
</tr>
<tr>
<td>P.compute_at(C, i)</td>
<td>Merge loop i of the operation P to the corresponding loop level in operation C.</td>
</tr>
</tbody>
</table>

**Parallelization**

<table>
<thead>
<tr>
<th>Primitive</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>C.unroll(i, v)</td>
<td>Unroll loop i of operation C by factor v.</td>
</tr>
<tr>
<td>C.parallel(i)</td>
<td>Schedule loop i of operation C in parallel.</td>
</tr>
<tr>
<td>C.pipeline(i, v)</td>
<td>Schedule loop i of operation C in pipeline manner with a target initiation interval v.</td>
</tr>
</tbody>
</table>

**Data type customization**

<table>
<thead>
<tr>
<th>Primitive</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>quantize(t, d)</td>
<td>Quantize a list of tensors t from floating to fixed point type d in the format defined in Table 2.</td>
</tr>
<tr>
<td>downsize(t, d)</td>
<td>Downsize a list of tensors t from integers with larger bitwidth to integers d with smaller bitwidth in the format defined in Table 2.</td>
</tr>
</tbody>
</table>

**Memory customization**

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

**Macros for spatial architecture templates**

<table>
<thead>
<tr>
<th>Primitive</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>C.stencil()</td>
<td>Specify operation C to be implemented with stencil with dataflow architectures using the SODA framework.</td>
</tr>
<tr>
<td>C.systolic()</td>
<td>Specify operation C to be implemented with systolic arrays using the PolySA framework.</td>
</tr>
</tbody>
</table>
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

```python
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()
```
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()
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

```python
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))
# ...
```
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)
    s[psum].unroll(xi)
    s.quantize(local_A, hcl.Fixed(W))
    s[local_A].partition(NUM_PE)

= min(

W=8
W=16
W=32

NUM_PE

Compute throughput

= min(

W=8
W=16
W=32

NUM_PE

W

NUM_PE

Compute throughput

#Elem / I/O access
HeteroCL Compilation Flow

HeteroCL

\[ B = h.\text{compute}((10,), \lambda x: A[k] + 2) \]

\[ s = h.\text{create\_schedule}() \]

\[ s[B].\text{unroll}(B.\text{axis}[0]) \]

Extended TVM/Halide IR

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

HLS Code Gen

LLVM Code Generation

CPU

General Back End

Merlin C Compiler

Cloud FPGAs

Spatial Architectures

PolySA

Embedded FPGAs

FW: forwarding module, implements FIFO and distributes data
PE: compute module, implements the kernel function
### Evaluation with Amazon AWS f1

#### Stencil

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Application Field</th>
<th>+ stencil</th>
<th>+ unroll</th>
<th>+ quantize</th>
<th>Theoretical (limited by memory bandwidth)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Seidel</td>
<td>Image processing</td>
<td>0.2</td>
<td>2.9</td>
<td>5.9</td>
<td>6.8</td>
</tr>
<tr>
<td>Gaussian</td>
<td>Image processing</td>
<td>1.1</td>
<td>6.7</td>
<td>13.2</td>
<td>15.6</td>
</tr>
<tr>
<td>Jacobi</td>
<td>Linear algebra</td>
<td>0.4</td>
<td>2.3</td>
<td>5.0</td>
<td>5.4</td>
</tr>
</tbody>
</table>

#### Systolic

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

#### General

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Application Field</th>
<th>Speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td>KNN Digit Recognition</td>
<td>Image classification</td>
<td>12.5</td>
</tr>
<tr>
<td>K-Means</td>
<td>Clustering</td>
<td>16.0</td>
</tr>
<tr>
<td>Smith-Waterman</td>
<td>Genomic sequencing</td>
<td>20.9</td>
</tr>
</tbody>
</table>

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/
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;

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);
}
}}
}
Optimized BNN in HeteroCL

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

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

Open source release of HeteroCL is coming soon!
Questions?