Building High-Performance Systolic Arrays with HeteroCL

Yi-Hsiang Lai, Shaojie Xiang, Zhiru Zhang

05/12/2021
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

HeteroCL Overview

- **High-level DSLs**
  - Keras
  - PyTorch
  - Halide
  - ...

- **HeteroCL**
  - Algorithm Spec.
    - (declarative + imperative)
  - Compute Customization
  - Data Type Customization
  - Memory Customization

- **Processors + Accelerators**
  - CPUs
  - FPGAs


github.com/cornell-zhang/heterocl
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 ...

Diagram:
- Off-Chip Memory
  - Loader
  - Accelerator
  - Unloader
  - FIFO
  - Scratchpad
Customizations in Roofline Diagram

- Operational Intensity (#ops/byte)
- Throughput (#ops/s)
- Memory Bandwidth Roof (bytes/s)
- Compute Roof
- Memory-Bound
- Compute-Bound
Customizations in Roofline Diagram

Throughput (#ops/s) vs. Operational Intensity (#ops/byte)

- **Memory-Bound**
- **Compute-Bound**

- **Memory Bandwidth Roof (bytes/s)**
- **Compute Roof**
Customizations in Roofline Diagram

- Throughput (#ops/s)
- Operational Intensity (#ops/byte)
- Memory Bandwidth Roof (bytes/s)
- Compute Roof
- Memory-Bound
- Compute-Bound

Memory
Compute
Customizations in Roofline Diagram

- Operational Intensity (#ops/byte)
- Throughput (#ops/s)
- Compute-Bound
- Memory-Bound

Compute Roof

Custom-Precision Compute Roof

Memory Bandwidth (bytes/s)

Data Type

Roofline Diagram

Compute Roof

Custom-Precision Compute Roof
Example: convolution

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

#pragma HLS array_partition variable=filter dim=0
hls::LineBuffer<3, N, ap_fixed<8,4> > buf;

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();

            if(y >= 2 && x >= 2) {
                for(int r = 0; r < 3; r++)
                    window.insert(buf.getval(r, x), i, 2);
                window.insert(in, 2, 2);
            }

            window.insert(buf.getval(r, x), i, 2);
            for(int r = 0; r < 3; 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;
        }
    }
}}
```
Decoupling Algorithm from Hardware Customizations

HLS C/C++

<table>
<thead>
<tr>
<th>Algorithm#1</th>
<th>Compute Customization</th>
</tr>
</thead>
<tbody>
<tr>
<td>Algorithm#2</td>
<td></td>
</tr>
<tr>
<td>Data Type Customization</td>
<td></td>
</tr>
<tr>
<td>Memory Customization</td>
<td></td>
</tr>
<tr>
<td>Algorithm#3</td>
<td></td>
</tr>
</tbody>
</table>

Entangled algorithm specification and customization schemes

HeteroCL

<table>
<thead>
<tr>
<th>Algorithm#1-3</th>
<th>Compute Customization</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Data Type Customization</td>
</tr>
<tr>
<td></td>
<td>Memory Customization</td>
</tr>
</tbody>
</table>

Fully decoupled customization schemes *(More Productive & Portable)*
Hardware Customizations in HeteroCL

HeteroCL code

```
Algorithm
```

```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]))
```

```
Custom Compute
```

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

```
Custom Data Type
```

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

```
Custom Memory
```

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

Declarative code (based on TVM)

```
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]
```

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
```

```
<table>
<thead>
<tr>
<th>Sign</th>
<th>Exponent</th>
<th>Mantissa</th>
</tr>
</thead>
<tbody>
<tr>
<td>1b</td>
<td>8b</td>
<td>23b</td>
</tr>
</tbody>
</table>
```

```
8-bit Fixed-point
```

```
<table>
<thead>
<tr>
<th>Int</th>
<th>Fraction</th>
</tr>
</thead>
<tbody>
<tr>
<td>2b</td>
<td>6b</td>
</tr>
</tbody>
</table>
```

```
2-bit Integer
```

```
<table>
<thead>
<tr>
<th>Int</th>
</tr>
</thead>
<tbody>
<tr>
<td>2b</td>
</tr>
</tbody>
</table>
```

Quantize/downsize

```
image
```

```
window buffer
```

```
kernel
```

```
out
```

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

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

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

```
linebuffer
```

```
y
```

```
x
```
Exploring the Interdependence amongst Customizations

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

```python
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)
```

---

Performance

= \min(W=8, W=16, W=32)

---

Compute throughput

\#Elem / I/O access

= \min(W=8, W=16, W=32)
Decoupled Compute Customization

HeteroCL code

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

HLS code

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

Decoupled customization

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

Customization primitives
- Portable, less error-prone

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**

```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]))

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

**32-bit Floating-point**

- Sign: 1b
- Exponent: 8b
- Mantissa: 23b

**16-bit Brain Floating-point (bfloat)**

- Sign: 1b
- Exponent: 8b
- Mantissa: 7b

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

- Int: 2b
- Fraction: 6b

**2-bit Integer**  Int(2)

- Int: 2b
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

```python
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]
```

```python
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

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

# Importing tensors from NumPy arrays
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)))

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)

s = hcl.create_schedule([img, conv_w1, conv_w2, dense_w], top)
f = hcl.build(s, target=p)

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

```python
def ConvNet(dtype_quant, quantize=True):
    # A three layer ConvNet example
    def top(img, conv_w1, conv_w2, dense_w):
        # ...

        # Data typepe 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)
```

![ConvNet accuracy with different quantization schemes](image)

- 83.42% to 97.83%
- 96.42% to 98.42%
Decoupled Memory Customization

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

```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)
```

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)

```python
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)
```

Compute placement is automatically inferred
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 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)
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()

<table>
<thead>
<tr>
<th>Conv 1</th>
<th>Conv 2 + ReLU</th>
<th>Reshape + Dense</th>
</tr>
</thead>
<tbody>
<tr>
<td>30x30</td>
<td>28x28</td>
<td>26x26</td>
</tr>
<tr>
<td>1</td>
<td>16</td>
<td>64</td>
</tr>
</tbody>
</table>

#MAC Operations

<table>
<thead>
<tr>
<th></th>
<th>113k</th>
<th>6230k</th>
<th>432k</th>
</tr>
</thead>
</table>

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
  ```python
  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
  ```python
  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
  ```python
  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

<table>
<thead>
<tr>
<th>top</th>
<th>Trip Count</th>
<th>Latency</th>
<th>Pipeline II</th>
<th>Pipeline Depth</th>
</tr>
</thead>
<tbody>
<tr>
<td>conv2_i_conv2_i1_conv2_i2 + conv2_i3 relu_args2_relu_args01_relu_args11 reshape_i6_reshape_i7_reshape_i8</td>
<td>43264 16 16 43264</td>
<td>13022464 298 43265</td>
<td>N/A 18 1 29 2</td>
<td></td>
</tr>
</tbody>
</table>

  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

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<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>
<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>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>downsize(t, d)</td>
<td>Downsize a list of tensors t to type d.</td>
</tr>
<tr>
<td>quantize(t, d)</td>
<td>Quantize a list of tensors t to type d.</td>
</tr>
</tbody>
</table>

### Memory customization

<table>
<thead>
<tr>
<th>Function</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>
<tr>
<td>to(t, d, m)</td>
<td>Move a list of tensors t to device d with mode m.</td>
</tr>
</tbody>
</table>

### Macros for spatial architecture templates

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>C.stencil()</td>
<td>Specify operation C to be implemented with stencil with dataow 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>
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]
  - Datalfow 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 ...
```

- 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)

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

### Target: Intel Stratix 10 (AOCL*, vLab)

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

* 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

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

**General Back End**

**HLS Code Gen**

**HLS Compiler**

Intel AOC/I++

Xilinx Vitis/Vivado HLS/SDAccel

**AutoSA, T2S**

**SODA**

**Spatial Arch. Templates**

**Flexible retargeting to diverse FPGAs from various vendors**
More Examples

github.com/cornell-zhang/heterocl

⚠️ 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.

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

<table>
<thead>
<tr>
<th>Design</th>
<th>Compute</th>
<th>Data Type</th>
<th>Memory</th>
<th>Imperative</th>
<th>Macros</th>
</tr>
</thead>
<tbody>
<tr>
<td>KNN Digit Rec.</td>
<td>✓</td>
<td>✓</td>
<td></td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td>K-Means</td>
<td>✓</td>
<td>✓</td>
<td></td>
<td></td>
<td>✓</td>
</tr>
<tr>
<td>Smith-Waterman</td>
<td>✓</td>
<td>✓</td>
<td></td>
<td></td>
<td>✓</td>
</tr>
<tr>
<td>Cordic</td>
<td></td>
<td></td>
<td>✓</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Sobel</td>
<td>✓</td>
<td></td>
<td></td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td>Optical Flow</td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
<td>✓</td>
</tr>
<tr>
<td>3D Rendering</td>
<td>✓</td>
<td></td>
<td>✓</td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td>Seidel</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td>Gaussian</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td></td>
<td>✓</td>
</tr>
<tr>
<td>Jacobi</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td></td>
<td>✓</td>
</tr>
<tr>
<td>GEMM</td>
<td></td>
<td>✓</td>
<td></td>
<td></td>
<td>✓</td>
</tr>
</tbody>
</table>

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]